Simultaneous Floating Base and Footstep Planning in Challenging Terrain

Master Thesis
Felix Sternkopf
November 15, 2021
Gutachter: Prof. Dr. Oskar von Stryk, Fachgebiet Simulation, Systemoptimierung und Robotik

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Felix Sternkopf, die vorliegende Master-Thesis gemäß § 22 Abs.7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Datum:

Unterschrift:

Abstract

Mobile legged robots are capable to cross over complex and uneven terrain. Dealing with this challenge, the robot requires robust localization and mapping methods to achieve a good model of its environment. Additionally, it needs a suitable contact planning process with a following execution of the planned movements. If all these problems are solved, a legged robot system provides more flexibility and mobility as wheeled robot systems, especially in moving on incoherent objects like stairs or stepping stones. To solve all these problems, the planning of suitable sequences of foot contacts is needed. Already planned contact configurations can then be interpolated to continuous movements with trajectories.
Often trajectory optimization methods are used with cost functions and heuristics to generate movement. The cost functions can represent the limitation of robot joints, the observance of stability margins, or just a collision-free configuration of the whole system. The whole-body planning plans whole-body configurations, including all feet and the robot base while in contrast the classical footstep-planner instead plans only discrete footsteps without considering the robot base position. The position of the robot base in combination with the foot positions has a big influence on the solution space of the trajectory generation with respect to collisions or unstable robot postures. So the solution of the classical footstep-planner is not necessarily optimal in terms of the challenges mentioned above. The position of the robot base can be optimized after the footstep generation, but in this stage the footsteps cannot further changed to improve overall the motion plan. In this work, a new approach is presented which includes a simultaneous pre-optimization of the robot base position to the footstep planner, so the pre-optimized base positions can be used as an initialization of the trajectory generator. The pre-optimization can take care of different conditions from above, so the convergence of the trajectory generator can be massively improved.
For the evaluation of this work, the new approach is included into an existing planning framework, based on discrete graph search. Different methods are investigated, to calculate suitable base positions to the generated footsteps. These new methods enables to consider cost functions and heuristics for sampled floating base positions simultaneously to already planned footsteps guiding the solution of a graph-based search.
The challenge adressed in this work consists of finding convenient methods, which can consider the coupling of combinations of footsteps and base positions in a way the planner can efficiently use them to plan in challenging terrain. As a result, the footstep planner generates now a step plan using highly coupled footsteps and floating base solutions to the trajectory generation stage. The main goal is to improve the search-space for the trajectory optimization, with respect to collision avoidance, while avoiding singularities in joint-space caused by undesired combinations of footsteps and base positions. Furthermore, the planner has to ensure a possible movement interpolation between all states.
Another challenge is the ability of planning in soft real-time applications, to guarantee a reactive walking behaviour. Therefore, this work must also investigate methods, which provide a fast and efficient computational solution to the given problems.
The final evaluation of this work was carried out on an eight-legged robot system, developed by "Simulation, Systems Optimization and Robotics Group" at the Technical University of Darmstadt, based on data captured from real world scenarios and synthetic environments in simulation. The unique kinematics and the amount of the legs of the used robot system pose additional challenges to this work as it significantly differs from state of the art robots.

Zusammenfassung

Mobile Laufroboter bieten vielseitige Möglichkeiten schwieriges, unebenes und ungleichmäßiges Terrain effektiv zu überqueren. Dazu werden komplexe Anforderungen an die Umfelderfassung, die Kontaktplanung und die Ausführung von Bewegungen gestellt. Werden all diese Herausforderungen gemeistert, bieten lauffähige Robotersysteme wesentlich mehr Flexibilität und Mobilität als ketten- oder differentialgetriebene Plattformen, besonders was die Begehung von unzusammenhängenden Objekten, wie Treppen oder Schrittsteinen betrifft. Um all diese Probleme zu lösen, bedarf es sorgfältiger Planung von geeigneten Sequenzen von Fuß-Boden-Kontaktereignissen. Diese Kontaktkonfigurationen werden dann anschließend in Form von Trajektorien zu einer kontinuierlichen Gesamtbewegung interpoliert und integriert.
Für die Generierung der endgültigen Gesamtbewegung kommen häufig Optimierungsverfahren zum Einsatz, welche unter Einfluss von unterschiedlichen Kriterien und Heuristiken die Trajektorien generieren und optimieren. Die einzelnen Kriterien reichen von Einhaltung der Stabilitätsbedingungen, über die Limitationen von Gelenken, bishin zur Kollisionsfreiheit des Systems. Für die Lösung existieren hier zwei grundlegende Planungsansätze. Die Ganzkörperplanung plant Fußschrittpositionen zusammen mit der Position der Basis des Robotersystems, während die klassische Fußschrittplanung ausschließlich diskrete Sequenzen von Fußpositionen berechnet. Die Position der Roboterbasis, welche in Kombination mit den Fußschritten einen signifikanten Einfluss auf den Lösungsraum der Trajektoriengenerierung hat, wird in diesem Fall nicht berücksichtigt. Die Position der Basis kann zwar in jedem Fall nachoptimiert werden, allerdings können geplante Fußschritte nicht mehr angepasst werden. Wird die Basis allerdings von der Fußschrittplanung bereits voroptimiert, können die Positionen als Initialisierung für die Trajektorienoptimierung genutzt werden. Bei der Voroptimierung können unterschiedliche Nebenbedingungen bereits eingehalten werden, wodurch bei der Trajektorienoptimierung eine schnellere Konvergenz erreicht werden kann.
In dieser Arbeit soll eine bereits vorhandene auf Graphensuche basierende Fußschrittplanung um die simultane Planung der Position der Roboterbasis erweitert werden. Hierzu werden Methoden untersucht, um zu den bisher ausschließlich betrachteten Fußpositionen, geeignete Positionen der Basis zu finden. Die gefundenden Positionen sollen kostenbasiert in der Graphensuche des Systems evaluiert werden können.
Die Herausforderung des Problems besteht daraus, Methodiken zu finden, die die gegenseitige Abhängigkeit von Fußkontakten und der Roboterbasis effizient evaluieren können. Als Ergebnis soll der Fußschrittplaner erstmals eine simultane Planung der Roboterbasis für die Trajektorienoptimierung liefern. Primäres Ziel dabei ist, den Suchraum für die Trajektorienoptimierung zu verbessern, unter anderem durch Vermeidung singulärer Gelenkkonfigurationen durch ungünstige Kombinationen von Fuß- und Roboterbasispositionen. Weiterhin muss vom Planer gewährleistet werden, dass eine Bewegungsinterpolation zwischen zwei Zuständen überhaupt möglich ist.
Eine weitere Herausforderung ergibt sich durch die Laufzeitanforderungen des Planers für eine echtzeitfähige Erzeugung von Fußschritten. Nur so kann ein reaktives Laufverhalten auf einem realen System sichergestellt werden. Der Fokus der Arbeit liegt deshalb auf Methoden, die eine schnelle und effiziente Auswertung der Probleme ermöglichen.
Die abschließende Evaluation der Arbeit wurde mit einem am Fachgebiet entwickelten achtbeinigen Roboter in realitätsnaher Simulation und realen Sensordaten durchgeführt. Die besondere Kinematik sowie die hohe Anzahl der Beine des Robotersystems stellen zusätzliche Herausforderungen bei der Lösung der Probleme.

Contents

1 Introduction

The problem adressed in this work is a common problem in the field of robotic path planning. Basically, there are two ways to realize path planning for robot motions. The first type is whole-body contact planning [1], which tries to find a sequence of robot contacts with the environment representing a path from a start to a goal configuration [2][3][4]. Such planning applications are often computational intensive [5], so they need well-modeled heuristics to reduce the search-space to make them applicable in real-time. The trajectory generation stage is simplified by the application of a whole-body planner as only trajectories must found that interpolate the provided sequence of whole-body contacts. The probability of environmental collisions is reduced by decreasing the sampling distance between such whole-body contact configurations. Another way to plan paths for robot motion is discrete state planning [6], which is the initial situation in this work. For discrete state planning, the entire search-space is discretized in which a graph planning algorithm is applied to find a path between the start and goal state. Here, a state consists of contact configuration for each limb. Currently, the footstep planner does only handle the feet, whereby the computational complexity increases in a non-linear manner with the number of legs [5][7].
A yet unresolved problem of footstep planners is that despite planning footholds they do not consider any knowledge such as the legs or the rest of the robot system. The footstep planner is not able to reduce the probability of collisions in the planning stage, due to this lack auf knowledge. Additional collision detection algorithms [8] are required in the planning pipeline to cope with this issue.
These shortcomings are encountered in this work by the integration of the robot base as a new state in the discrete planner. Introducing the knowledge about the robot base in the planning problem enables the planner to determine the whole-body posture of a robot so that collisions, non-desired joint states and unstable poses can be avoided. This raises a new challenge, as the planning problem becomes harder to solve with increased state-space complexity. Simple graph planning algorithms, such as Dijkstra [9] expand all states regardless if they are goal-oriented or not. Heuristics prunes the search graph to expand the most probable branches first. In this work suitable heuristics for the foot and for the base positions are introduced.
Missing knowledge about whole-body configurations is a well-known problem in the field of path planning [10]. In many cases, the robot motion copes only with the feet, because locomotion tasks in open terrain do usually not require to incorporate further body parts such as arms. Most systems, which need to solve this issue, just use a specific heuristical approach of a whole-body contact planner. Specific means here that all heuristics are specially designed to solve specific locomotion problems with limited use outside of their intented scope. This work presents a generic approach that works on different robot systems in many different scenarios. Some approaches extend a discrete footstep planner with a floating base planner that runs as a post-footstep planning stage [11]. In this thesis, the planning of the feet and the base is performed simultaneously, so the coupling between footholds and floating base can be now considered, which is not possible using a sequential planning pipeline.
In this work, four different heuristics are introduced. Each heuristic uses the whole-body knowledge gained by including the floating base, to improve the planning results with respect to avoid collisions, non-desired joint states and unstable robot postures. Simultaneously each heuristic discards all states that are not goal-oriented to reduce the computational complexity of the whole planning pipeline.

- **Foot Distance Heuristic:** Handles the planning of foot placements in front of obstacles. Based on the observation that the ability to cross an obstacle free of collisions depends on the distance of the foot in front to it.

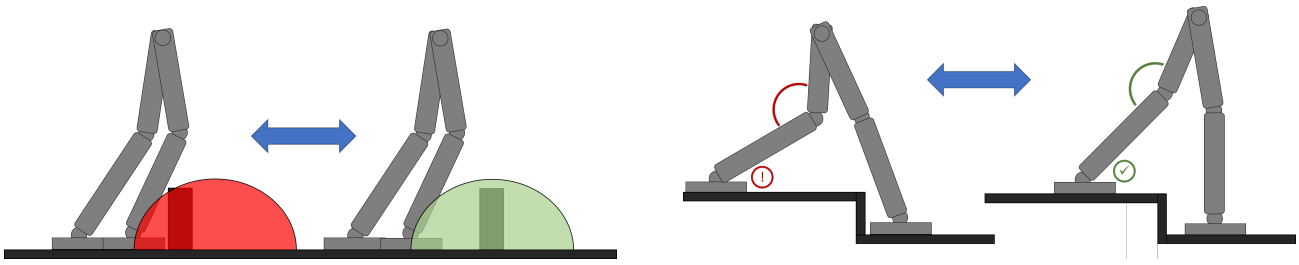
- **Foot Traversability Heuristic:** Handles the planning of foot placements on uneven terrain based on an introduced traversability value of [12].

- **Base Orientation Heuristic:** Handles the planning of base placements, based on the observation that an adapted pitch orientation of the upper-body corresponding to the terrain can improve the resulting postures with respect to the problem definition above.
- **Base Stability Heuristic:** Handles the planning of base placements, based on the introduced improved feasible support region from [13] to avoid unstable robot postures.

After explaining all heuristics in detail and showing the implementation, each heuristic is evaluated. Each heuristic has a partial contribution to the solution of the problem, which is highlighted in the evaluation. A wide-spreaded choice of different terrain models is taken for evaluation, to achieve meaningful experimental results. For each terrain model, the planner is started with ten different generated goals, at first without any heuristical approach and then with each of the introduced approaches. In the last test run all approaches are used together on all terrain models. The evaluation is done with an eight-legged robot. In the end of this work, a conclusion of all procedures and results is given. Additionally an overview for possible future work is presented.

1.1 Motivation

Mobile Robots are more frequently used in human environments [14] and they are supposed to solve increasely complex tasks in future. When robots should be deployed in real world scenarios, they have to adopt human-like skills, because their working-space is designed for human beings. Grasping effectors have to simulate human hands, exactly as the movement has to be improved to environments designed for humans. The system has to move in narrow spaces, climbing stairs and eventually surpassing small obstacles. Additionally, all these tasks have to be safely done next to people, without being a hazard for them.
For example, there are different operation areas for such systems. Maintenance robots should take over inspection tasks in dangerous environments like power plants or oil platforms. As these areas are designed for humans, it would be expensive to physically adapt the existing facilities to the needs of robot systems, so the robot systems should be able to get along with these challenges. Another use case is load-carrying systems for handicapped people. They have to transport their passengers over standard human environment, like stairs, the entry of a train or narrow spaces in their houses.
For all these tasks a legged robot outperforms wheel- and track-based ground robots. The environment for humans is made for legged individuals and for example a stair poses extreme challenges for a driving system [16]. A legged robot is more flexible in crossing over small obstacles, where a wheeled system has to move around. Though these benefits, a legged system has high requirements to the software to realize legged movement in such an environment.



(a) Foothold planned to close to a step to cross it

(b) Planning far from step is kinematically bad

Figure 1: Different problems of non optimized Footstep Planning

The Robot software has to provide a robust planning process to compute feasible foot contact configurations, to resolve these issues. After the planning of a sequence of contact configurations, a trajectory generator

has to compute feasible trajectories between each state. Like the step before, the stability of the system has to be guaranteed every time. Another issue for the trajectories is to be collision-free with the environment and the own hardware. Especially in narrow spaces or complex terrain like stairs the quality of the planned configuration can improve the search-space for trajectories. In contrast, they can also worsen the trajectory generation. For example, a foot is planned to close to a stair and has no possibility to climb on the stair without moving backwards. A more optimized planning state would have prevented such a case.
This work deals with these unsolved challenges in planning a legged locomotion. Different approaches to improve the planning results, with respect to collisions and trajectory generation are shown and evaluated on an eight-legged robot platform.

1.2 Robot System and Given Software

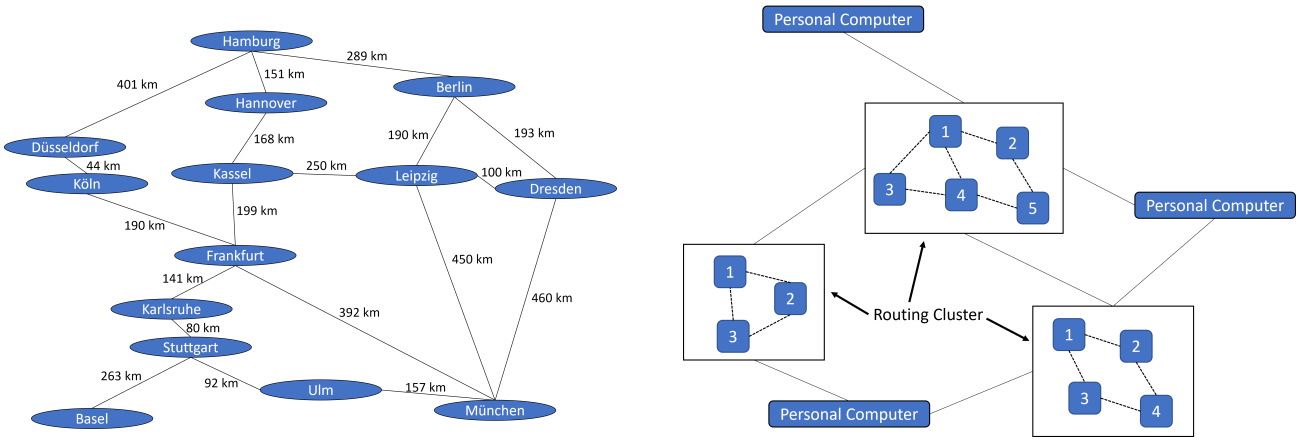
For evaluating this work, all approaches are deployed to an eight-legged robot system. The robot is designed like a classical quadruped robot [17], but for stability reasons, the quadruped infrastructure exists twice. So the robot has at least four foot contacts with the ground at any time. Each leg consists of three joints, one linear joint in the leg, passive joint in the foot with three degrees of freedom and rotational joint in the hip. In the center of feet the floating base is placed on which the main payload of the robot is acting. The payload is a challenge for the robot, because the system is used to transport handicapped people, which requires security and safetyness in motion tasks. The target of this work is to examine different methods, to extend a given footstep planning framework, with the ability to preoptimize contact configurations for the trajectory optimization. A short explanation of the used L3 Framework [18] is given in section 3.3, to provide the required insights to understand the method of implementation.
The footstep planner is part of the L3 Framework [18], which provides a plugin-based system to implement the sense-plan-act of a legged robot system. It includes a terrain model generator to compute elevation maps and terrain models, a planning system for step plans, a controlling system to manage and schedule the execution of step plans and a walking controller to compute trajectories and execute them. The footstep planner provides a planning algorithm pipeline to plan multiple states with different foot contact positions and a position for the robot base.
For the evaluation in this work, the L3 footstep planner is deployed on an eight-legged robot, developed at the "Simulation, Systems Optimization and Robotics Group" at the Technical University of Darmstadt. In the current version of the footstep planner, the robot base is planned individually by placing it in the projected center of the feet.
The approach of this work is to intervene in the planning process to optimize the footsteps in combination with a computed robot base position to build a pre-optimized solution for trajectory generation. The existing footstep planning framework is described in detail in section 2. The pre-optimization is part of section 4.

2 Foundations

In the following chapter all important technical foundations are explained, which were used for the research, the investigation of methods or the final implementation approaches. At first, there will be a short summary of different graph search algorithms [19], followed by the basics of mechanics. Then the most important data structures used for this work are explained. Finally, the basics of numerical optimization are illuminated.

2.1 Graph Based Path Planning

The planning of paths from a start position to a goal position is a common problem not only in robotics. Navigation systems in cars have to find the best path between two cities. The quality of the path is defined by the user and can be expressed with the shortest distance or shortest travel time. Furthermore, the path planner should avoid traffic jams or construction sites. Another application for pathfinding is the routing algorithm in physical networks. All over the planet routing stations are connected with physical connections [20]. The data has to be sent on the shortest physical routing connection, to guaratee a fast data transfer in the internet. In video games, characters have to find paths in a virtual environment [21] or in behavioural research pathfinding algorithms create behaviour chains and medical assistance systems use them to take over surgery issues.



(a) An example graph for jam navigation.

(b) An example graph for a routing system.

Figure 2: Different possible graph Representations.

In robotics, the field of path planning is one important part of robot motion. Before the robot can execute motions or real steps in a real environment all the motions have to be planned out first. This is not restricted to walking tasks, but also grasping or manipulation tasks depend on an accurate planning [22]. The robot has to avoid obstacles to ensure a secure motion, the path has to be short as possible to not waste time and energy, in every state of motion the path has to guarantee stability so the robot does not fall over. In some applications the motion has to be natural and especially on humanoid systems, this point is a common one. All these examples for path planning have a common foundation: They can be expressed as discrete scenarios solvable by performant and reliable planning algorithms. The discretization constitute a graph where each state in the system is represented by a graph node and every transition between two states is an edge in the graph. For traffic navigation tasks the cities are the graph nodes and the streets are the edges. In network routing, the routing stations are the nodes and the edges are the physical connections. In robotics motion

planning the environment is mostly discretized with a grid where each node is reachable over its direct neighbours. Each transition between two states is marked with a transition cost. The costs are calculated with heuristics, depending on what is important for the resulting path. If the algorithm has to find the shortest path, the cost could be calculated with a distance function. After the environment of the robot is discretized, obstacles are detected and included in the graph, different path planning algorithms can be used to find the best path from a start position to a goal position.
Based on the use case the shortest path must not be the best solution. In robotics, the runtime of an algorithm is important, because during locomotion tasks online replanning is required reflecting sudden changes in the environment. In such cases, the algorithm has to find a new solution as fast as possible in order to prevent collisions with objects in the environments. Often an increased calculation time is preferred, no matter if the solution path is the shortest or not. In many robotic tasks, this trade-off between calculation time and quality of results is crucial. Another important criterion for the algorithm is completeness, which means that if a solution exists it has to be found in a finite amount of time.
For discrete graphs several algorithms exist which fulfil these conditions. The individual use case determines the usage of a single algorithm. In the following, the two most important graph search algorithms and their application to robotic motion are presented.

2.1.1 Dijkstra

Dijkstra [9] is a path planning algorithm to find the shortest path between a fixed source node and all other nodes in a weighted and undirected graph. In special variants, these algorithms can also be used to find a path between two specified nodes or to find paths in directed or unweighted graphs. The algorithms use positive integers or real numbers as node labels and a data structure for storing and querying partial solutions sorted by distance. The distance between two nodes is a value computed by a heuristic, that represents a real distance between two cities or the amount of needed petrol to drive from one city to another. The original algorithm uses a min-priority queue and runs in time $\theta((|V|+|E|)log(|V|))$ where $|V|$ is the number of nodes and $|E|$ is the number of edges.
In the following example the algorithm is only used to find the shortest path between two nodes, due to the close relation to the footstep planning. In the initialization the dijkstra algorithm will assign initial distance values to each node, commonly infinity or the integer maximum value. In the following steps the algorithm will try to improve the values step by step.

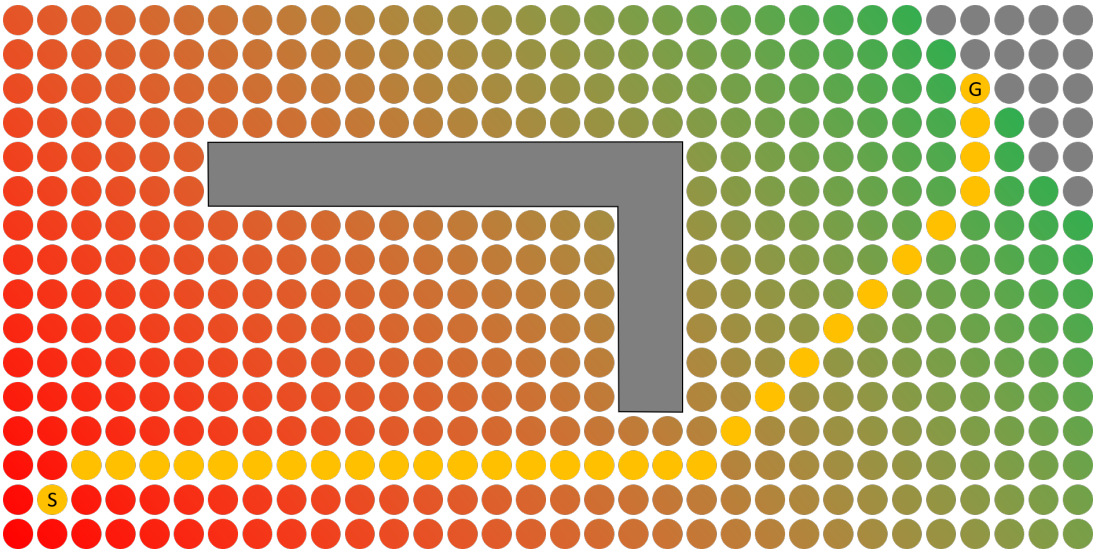


Figure 3: Expanded Search Graph with Dijkstra Algorithm

In figure 3 the result of a Dijkstra path planning algorithm is shown. All expanded nodes in the graph are marked in the colours red to green where red indicates a state far from the goal and the more green a state is, the closer it is to the goal. The resulting path is marked in yellow with the start node S and the goal node G. The assumption in this figure requires, that each node is connected with its eight neighbours and every edge has the same cost value. In the center of the map is an obstacle that has to be avoided. All nodes marked in grey were not expanded by the algorithm. Overall there are 455 expanded states and the resulting plan is optimal with 31 states.
In the presentation of the other graph search algorithms, the same graph map is used to have valuable results to compare.
The Dijkstra algorithm can be separated into six steps, where steps three, four and five are repeated until the goal state is expanded. This is only valid for a path planning issue between two nodes, because the classical algorithm always expands every node in the graph.

1. At the beginning all nodes are unvisited and are saved in a set called the unvisited set.
2. An initial value is assigned to each node. The start node has the value zero and the rest in the graph has the value infinity. The start node is marked as the current node.
3. Calculation of the tentative distance to all unvisited neighbours through the current node.
4. Comparison of newly calculated value and the old assigned value of the node. The smaller one will be the new value of the node.
5. Current node is marked as visited. If the goal node is visited the algorithm terminates.
6. Otherwise the unvisited node with the smallest value is marked as the current node and the algorithm jumps back to step three.

When planning a path between two nodes, it is not necessary to wait until the goal node is marked as visited. The algorithm can be stopped if the goal node is the one with the smallest value in the unvisited set. The algorithm always finds the most optimal path with respect to the given heuristic. The most valuable problem for footstep planning on robot systems is the performance. The algorithm does not converge fast to the goal

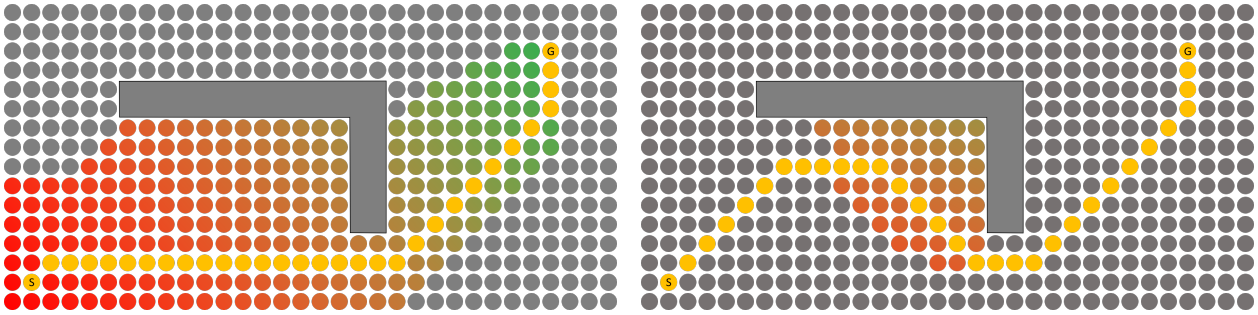
node, so it may be possible to plan step plans in real-time. Especially in large graphs with a big amount of nodes, this will become a problem.

2.1.2 A-Star Algorithm

The classical A-Star algorithm [23] is an informed graph search algorithm analogous as the Dijkstra algorithm before. In contrast, its core issue is to find an optimal path between two nodes in a graph with positive path costs. The A-Star algorithm can be seen as a generalization or an extension of the Dijkstra algorithm. The algorithm uses heuristics, which leads to faster convergence to the goal states, o improve the performance. During the search the nodes are divided into three different categories.

- Unknown Nodes: Unknown nodes are not yet expanded and the algorithm has no information about these node. In the beginning every node is an unknown node.
- Known Nodes: A path to these nodes is known. It could be a suboptimal path.
- Investigated Nodes: The shortest path to these nodes is known

Every known or investigated node has a relation to all of its predecessors to find the path to the start node. At the beginning of the algorithm, every node in the graph is an unknown node and every node has a heuristic value to the goal node. The start node is taken as the current state. In each step, the current node is investigated, which means that every successor node gets a new cost value. It is computed with its heuristic value and the path cost between the current node and the successor. After this step, the current node becomes an investigated node. The algorithm repeats this step with the successor with the smallest new cost value, which becomes the next current node. This loop is repeated until the goal node becomes an investigated node.
In figure 4 it can be seen that the A-Star Algorithm needs quite fewer state expansions than the Dijsktra algorithm. The reason for this is the heuristic value. The quality of the result and the run time of the algorithm depends on the used heuristic. If a bad heuristic is chosen the performance becomes worse or the result becomes suboptimal.



(a) Non weighted A-Star Result (b) Weighted A-Star Result

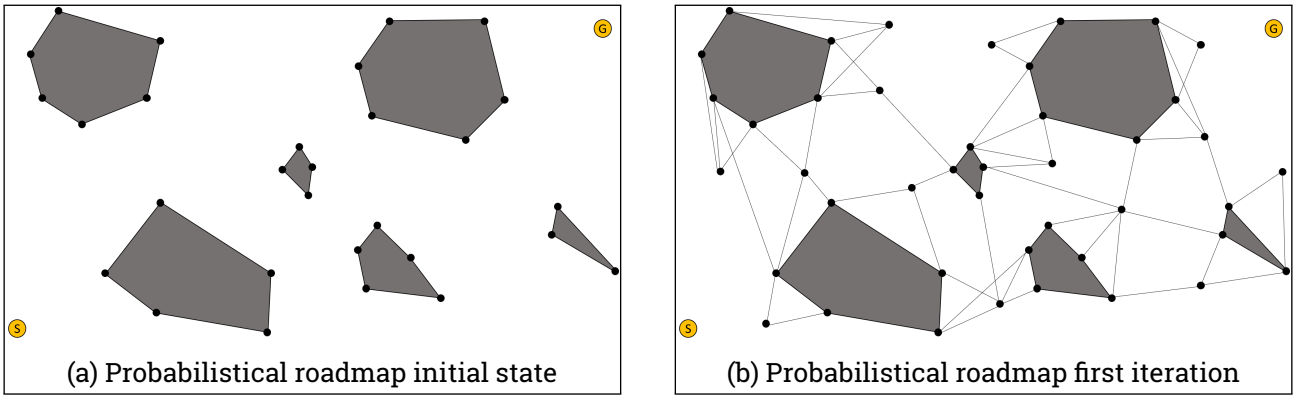
Figure 4: Different results for A-Star Graph algorithm

Besides the classical A-Star algorithm different variations and adaptions of this algorithm exist. Generally, the heuristic functions can be weighted for each node and situation, so the algorithm achieves a faster convergence. The trade-off in such variations is that the quality of the path becomes worse.
In Figure4 it can be seen, that the amount of expanded states is essentially smaller than with the classics algorithm. In this case, the user takes into account the suboptimal result. Which algorithm version should be used depends on the situation and the goals of the application. In robotics, the performance and runtime

of an algorithm are often more important than the quality of the result. Especially in the planning case, the real-time possibility is desirable.

2.1.3 Probabilistical Roadmap Planner

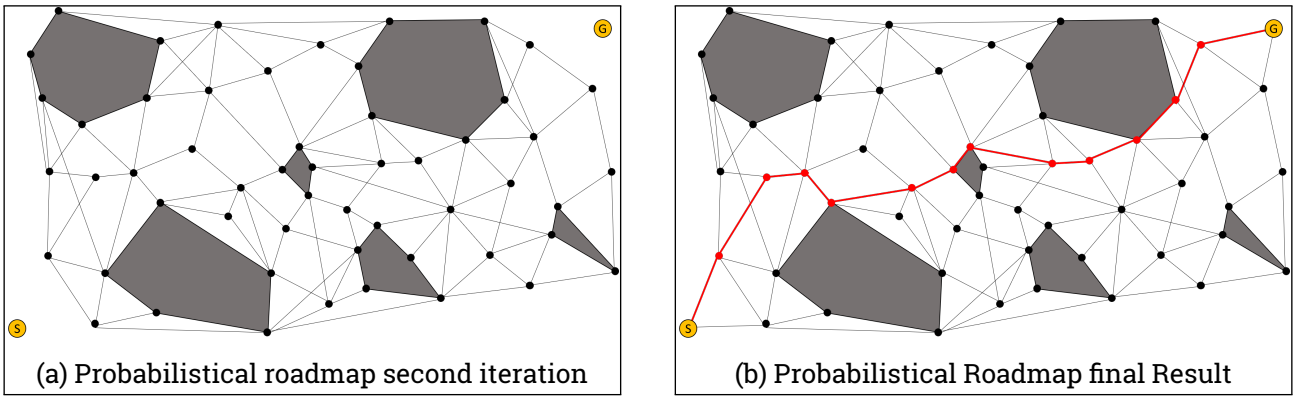
The probabilistic roadmap planner [24] is a motion planning algorithm in robotics, which computes a path between a start robot configuration and a goal robot configuration with respect to avoiding obstacles in the environment.



(a) Probabilistical roadmap initial state (b) Probabilistical roadmap first iteration

Figure 5: Probabilistic Roadmap Planner: Iteration one and two

The basic approach behind the probabilistical roadmap planner is to take randomly generated samples out of the configuration space of the robot system. After the random sampling, the configuration is tested on collisions. If the configuration is collision-free, the planner tries to connect the configuration with a nearby configuration in space. In the end, the start and the goal configuration are added to the configuration graph and a graph search algorithm is used to find a path from the start configuration to the goal configuration.



(a) Probabilistical roadmap second iteration (b) Probabilistical Roadmap final Result

Figure 6: Probabilistic Roadmap Planner: Iteration three and four

Basically, the planner consists of two planning phases. First, in the construction phase, the roadmap is built. From a set of precomputed robot configurations for motion primitives, a random configuration is sampled. Then the configuration is connected to some neighbour states, typically the k-nearest neightbors [25]. Configurations and connections are added to the graph until it is dense enough to compute a path as

shown in figure 5 and 6. Now in the second phase or the query phase, the start and goal configurations are added to the graph and with a graph-based algorithm, the shortest path between the start and the goal is computed. This can be done with the Dijkstra algorithm, shown in the section above, for example.

2.2 Important Data Structures

In this work, several specialized data structures are used. So in this section, these data structures are explained. All used and presented structures are mainly used in a mapping or planning environment, but they also can be used otherwise. Due to this work is moving in the planning environment, the explanations and examples will focus on this theme. In addition to a short explanation, a short relation to the usage in this work is given.

2.2.1 Point Cloud

An important prerequisite to applying graph planning algorithms onto discretized environment graphs is to recognize the environment and organize the detected data into a structure that is easy to use for algorithmic applications. The common way to scan the environment in robotics is the usage of a laser scanner. A three-dimensional laser scanner scans the environment by measuring the time of flight of single rays of light. Each ray is fired in the space, hits a surface and is reflected back to the scanner. The sensor can assign each measured point to a coordinate in the space. Due to the knowledge of geometrical relationships between the sensor, the robot and the world, the system can map a height value z to each point.
This collection of measured points which gets completed with each scanning iteration is called the point cloud [26]. Point clouds are a common basic data structure in robotics to work with environmental data. Mathematically a point cloud is a set of points in a vector space that represents an unorganized structure. The structure is described with the space coordinates of these points. Additionally, the measured points can contain attributes like surface normals, color values and record time stamps.
Due to the massive amount of points in an environment point cloud, one main task in working with them is the memory issue. If the data is needed in real-time applications it is necessary that the data structure provides an efficient access method. A common way to solve this problem is data processing with octrees. Another issue is the visualization of the point cloud data. Specialized algorithms are needed, to get an efficient real-time visualization of point cloud. Generally, the point cloud is not the data structure that is used in high-level algorithms. In robotics, the point cloud is more an interface between the sensor for environment recognition and optimized data structures which can be used with fast access to apply algorithms on them.
Other use cases for point clouds are architecture applications that calculate specialized morphological environment data in building sites, medical computer vision-based approaches, video games and all planning tasks which use computer-aided design to plan three-dimensional structures. In statistics, point clouds are used for graphical representations of datasets.
In robotics, the point cloud provides the basics for data structures like the octree, three-dimensional voxel grids and grid maps which are shown in the following sections.

2.2.2 Octree

An octree is a special case of a tree in computer science and it is an important data structure in computer graphics. Generally, it is a rooted tree in which nodes have eight child nodes or zero child nodes. Like in a classical tree these nodes are called leaves. Octrees are mainly used to divide three-dimensional data sets into hierarchical structures. The root represents the whole dataset and each node is an octand of the data of its predecessor. The trees are suitable for the divide and conquer principle.

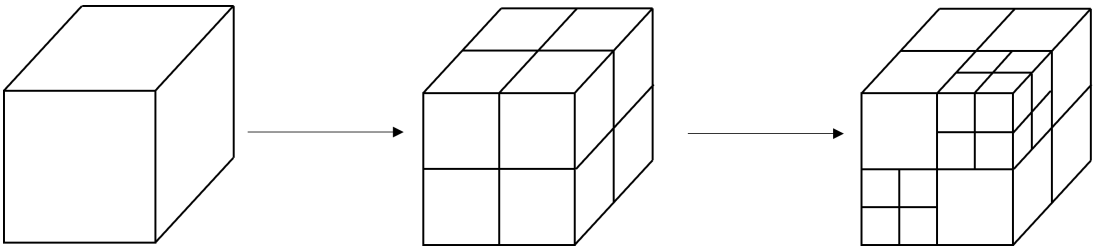


Figure 7: Octree Generation Example

The main application of octrees is the division of a three-dimensional data space. For example, the environment sensors of a robot system scan the environment in a finite cube. This cube is divided into an octree, to simplify the access to single data points. The root is the whole environment cube. The first tree level is built from the eight subcubes of the root. This scheme is repeated until every subcube contains a maximum amount of data points. the smaller this maximum the more precise the octree is. Now the access to an approximated position in the environment is access in a tree structure which is more performant than an iteration through the unorganized point cloud. In robotics, a common application is the usage as an occupancy grid [27].

2.2.3 3D Voxel Grid

A voxel is an important element from 3D computer graphics [28]. It represents a value on a regular grid in the three-dimensional space. A voxel grid can be used to discretize a 3D environment. In 2D it can be compared with a pixel. Voxels are not described with their fixed position encoded with their values. Instead, rendering systems compute the voxel position based on the position relative to other voxels in the grid.
A voxel represents a single data point in the three-dimensional grid and can consist of a single piece of data like a color value or more complex structures. Spaces between the voxels are not represented in the voxels dataset and become reconstructed with approximation applications like interpolation. This depends on the situation and the usability.
Common uses of voxel grids are terrain visualization in computer games or in medicine visualization issues. Related to this work, voxel grids can be used instead of height maps. In height maps, only the top layer of the environment is saved in the map. In the case of overhangs or caves, the height map cannot represent more than one height data. Another use case is 3D occupancy maps. The voxel dataset can hold occupancy data, like costs or opacity of the environment. Especially in planning with respect to obstacles this data structure can be useful to avoid collisions with the environment.

2.2.4 Multilayer Grid Map

Basically, the grid map is a two-dimensional discretization of the world [29]. The environment becomes split into multiple grids. The size depends on the resolution of the grid map. The smaller the resolution, the more exact the map will represent the world. Each grid holds a data value with information about the environment. The kind of data depends on the situation and the use case. For example, the grid map can be used as a heightmap where each grid holds a height value for the height on a given position.
The multilayer variant of the grid map is just an extension of the simple grid map which extends the simple map with multiple data layers. Each grid in the map has multiple data layers with different kinds of data. For example, a grid can hold the height values in one layer and a planning cost for this position in another layer. With this extension, the user does not have to use multiple maps with their own initialization costs. The user simply can add different layers to the map, with all the data which is needed. Just a simple lookup

on the defined layer and the index is necessary, to read the map.
The multilayer grid map is readily implemented and simplifies the work with the terrain. In this work, the grid map is used for different data types, including the height of the terrain or the planning costs in the terrain.

2.2.5 Bounding Volumes

Bounding volumes [30] or bounding boxes are used to approximate complex three-dimensional objects. In the task of collision avoidance or terrain dependent motion planning it is necessary to calculate overlap tests or intersections. The more complex the three-dimensional structures are, the more complex are such calculations. With multiple bounding boxes in combination, it is possible to approximate each structure. Only simple volumes are used, to reduce computation time.

- Bounding Rectangle
- Bounding Capsule
- Bounding Cylinder
- Bounding Ellipsoid

Bounding volumes are most often used to accelerate different geometrical tests. In ray tracing, they are used for intersection tests. In robotics, bounding boxes are mainly used for collision tests with the environment. Therefore volumes are defined for the robot structure and for the environment.

2.3 Basics of Mechanics

An important issue in motion planning and motion execution is the stability of the system. No matter what motion is planned, in every case, the system should fulfil the stability conditions. What these conditions are and how they can be guaranteed is explained in this section [31].
In mechanics are descriped three different kinds of systems:

- Rigid Bodies
- Deformable Bodies
- Fluids

In this work and in the main part of current research, robots are considered as rigid bodies, so at this point, deformable and fluids can be ignored.
Basically, a system is in mechanical equilibrium if it is not exposed to any accelerations. So it is static or it is moving with constant velocity. This is the case when all forces and torques acting on the system are in equilibrium.

$$\sum F_i = 0\text{: Sum of all Forces is zero} \tag{1}$$

$$\sum M_i = 0\text{: Sum of all Torques is zero} \tag{2}$$

With these equations, every acting force or torque in every point has to be observed and if the sum is zero there is a mechanical equilibrium.
These two constraints could be enough if frictionless point contacts are assumed. In practice, every contact has a friction term, which cannot be ignored in the stability criteria. Friction arises between each pair of bodies in contact and increases the necessary force two move bodies in contact. The friction force can be computed with the tangential and the normal part of the contact force and a friction coefficient μ. The friction coefficient $\mu$is a constant and depends on the body's material. All three parameters define a cone in direction of the contact point normal. The contact force must be described by a vector that lies completely inside the friction cone, to guarantee stability. Every contact point has to fulfil this criterion.
Another common way to ensure static equilibrium for motion systems is the support polygon criterion. All feet build with their contact points a polygon on the ground, considering a contact plane. With this polygon, the center of mass of the system can be controlled. As long as the projection of the center of mass on the contact plane lies in the support polygon, stability is ensured. At the moment the projected center of mass is leaving the polygon the system becomes unstable. The support polygon criterion is directly related to the point above because the position of the center of mass changes the forces which are exerted in the contacts. So if the center of mass projection is leaving the support polygon, the contact forces no longer lie in the friction cone. The stability criteria are violated.
Related to this work, it is important to ensure that the center of mass projection never leaves the support polygon for every step in the step plan. When a foot is swinging the polygon loses one corner and is shrunk, so the base position of the robot must be adapted to the new situation.

2.4 Basics of Optimization

Mathematical optimization is the computational selection of the best element, with regard to some conditions. There are different sorts of optimization problems that arise in computer science, robotics or engineering. The solution of an optimization problem is strongly dependent on the sort of optimization problem. In the simplest case, the problem consists of maximizing or minimizing a function by testing an allowed set of values and computing the results of the function. The generalization of an optimization problem is an own field in mathematical research and there are multiple techniques and formulations for each type of optimization problem. Each optimization problem can be represented in the following way:

Given a function: $f : A \to \mathbb{R}$

Sought an element: $x_0 \in A : f(x_0) \leq f(x) | x \in A$*(minimization)*

Sought an element: $x_0 \in A : f(x_0) \geq f(x) | x \in A$*(maximization)*

A subfield of mathematical optimization is convex optimization. Convex optimization problems deal with the problem of minimizing convex functions over convex sets. Many classes of convex optimization are solvable with polynomial time algorithms. In this work, optimization is used to compute a feasible robot base position set that fulfils all stability criteria. This issue can be solved with a convex optimization problem, and due to this fact, this is the only subfield of optimization which is presented in this work, to understand the basic functionality.
A function is called convex if the line segment between each pair of points on the function graph lies above the graph between these two points. A convex set is a subset of the euclidean space and for every two points in the set, the set contains the whole line segment between these two points. In convex optimization the target function f is convex and maps a subset $\mathbb{R}^n$ into $\mathbb{R} \cup \{\pm\infty\}$. The standard form of a convex optimization can

be written as

$$\begin{aligned} \underset{x}{minimize} \quad & f(x) \\ subject\,to \quad & g_i(x) \leq 0, i = 1, ..., m \\ & h_i(x) = 0, i = 1, ..., p \end{aligned}$$

where $x \in \mathbb{R}^n$ is the optimization value, the function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ is convex and$g_i(x)$ and $h_i(x)$ are the optimization constraints. This notation describes the problem to find x that minimizes $f(x)$ and all x satisfy the constraints. A feasible set C of the optimization problem contains all points x satisfying the constraints. The convex optimization can be subdivided into more specialized optimization problems and three of them are important for this work so they are explained in the following list.

- **Linear Programming:** Linear programming, also called linear optimization describes an optimization problem which deals with achieving the best result in a mathematical model which is based on linear conditions. In the field of linear programming, the objective function is linear and this function is optimized subject to linear equality and linear inequality constraints. The feasible set of a linear program is a convex polytope. A linear programming algorithm finds a point in the polytope which maximizes or minimizes the objective function. Linear programs can be expressed as

  $$\begin{aligned} Find\,a\,vector \quad & x \\ that\,maximizes \quad & c^T x \\ subject\,to \quad & Ax \leq b \\ and \quad & x \geq 0 \end{aligned}$$

  where the components of x are the variables to be searched, c and b are given vectors and A is a given matrix.
  Linear programming can be used in multiple fields of science, like mathematics, computer engineering or economics with linear data models.

- **Quadratic Programming:** Quadratic programming is the process of solving mathematical optimization problems with quadratic functions. Mainly there are two types of quadratic programming. On one side there is the optimization of a quadratic function with linear constraints, on the other side, the optimization of a quadratic function with quadratic constraints. Both are methods of nonlinear programming and for both types exist multiple iterative and numerical solution methods.

- **SOCP:** Second-order cone programming [32] is a specialized type of nonlinear programming of the form

  $$\begin{aligned} minimize \quad & f^T x \\ subject\,to \quad & \parallel A_i x + b_i \parallel_2 \leq c_i^T + d_i, \quad i = 1, ..., m \\ & Fx = g \end{aligned}$$

  where the parameters are $f \in \mathbb{R}^n$, $A_i \in \mathbb{R}^{n_i \times n}$, $b_i \in \mathbb{R}^{n_i}$, $c_i \in \mathbb{R}^n$, $d_i \in \mathbb{R}$, $F \in \mathbb{R}^{p \times n}$ and $g \in \mathbb{R}^n$. The optimization value is $x \in \mathbb{R}^n$ and $\parallel x \parallel_2$ is the euclidean norm. The second order cone in the optimization problem comes from the constraints. Second order cone programs can be solved efficiently by interior point methods and the are especially used in force optimization in robotics.

The convex optimization with the three explained subfields is just a small part of the research field, optimization. Due to in this work only the *second order cone program* is practically used the technical foundations around optimization are kept shorter to avoid confusion.

2.4.1 Simplex Algorithm

The simplex algorithm [33] is a numerical application to solve linear optimization problems. A simplex algorithm has two possible results. Either it solves the linear program with a finite amount of steps or it can be said that the problem is not solvable. Simplex algorithms are the most important methods to solve linear optimization problems in practice.
The method is stable, which means that for each variant of the algorithm an example can be found which can be solved in exponential runtime, but in practice the algorithm is often faster. An advantage is that the method can react to system changes, like adding constraints. A simplex algorithm can start from the last known optimum and reduce the solution of the changed system. Most of the known optimization algorithms have to start from the beginning if the system has changed.
The algorithm can be explained with geometrics. A classical linear optimization problem describes a convex polyhedron as a feasible set of results. The optimum can be found in one of the vertices of the polyhedron. The simplex algorithm iterates over the edges of this polyhedron until it finds the optimal result. From a mathematical point of view, the system can be described in the following way

$$\underset{x}{maximize}\{c^T x | Ax \leq b, x \geq 0\}$$

where $A \in \mathbb{R}^{m \times n}$ is a matrix with real numbers and $c \in \mathbb{R}^n$ the target vector and $b \in \mathbb{R}^m$ a non negative vector. The target of the optimization is to find a point x which fulfils all the constraints and provides a maximum value of $F(x) = c^T x$. Every linear optimization problem can be rewritten in this form.
The algorithm consists of two phases. In phase one a start configuration is calculated, or the algorithm comes to the result, that the system is not solvable. Phase two is the real optimization which converges iteratively to the optimum value. In every step of phase two, the algorithm tries to find a new solution from the current one, which has a more optimal target value. If this is no more possible, the algorithm stops and returns the last known point x.

2.4.2 Interior Point Method

Another group of algorithms to solve optimization problems are the interior point methods [34]. The main application is linear or quadratic programs, but also in some non-linear programs, they can be used. In comparison to classical active set methods like the simplex algorithm, the interior point methods have a faster convergence. One disadvantage is that they cannot react to system changes and are forced to restart from the beginning if the system has changed. In the easiest case, an interior point method is used to solve a linear problem like:

$$\underset{x}{minimize}\{c^T x | Ax = b, x \geq 0\}$$

A is a $m \times n$ matrix and c and b are vectors. The feasible set $X = \{x : Ax = b, x \geq 0\}$ has the form of an polyhedron. An optimal solution of the problem can be found in one of the vertices of the polyhedron. The simplex algorithm tries to find a solution iterating over the edges of the polyhedron, the interior point method searches a path in the inner of the polyhedron to find an optimum.
The interior-point methods are the most reliable methods to solve quadratic optimization problems. They have a global convergence and in practice, the methods have a complexity like $O(\log n)$.

3 Related Work

The main problem addressed in this work, is to investigate new methods for footstep planning with the ability to pre-optimize the robot base position with respect to the foot positions. An optimal base position changes the whole posture of the robot to avoid collisions with the environment and to avoid unwanted joint state configurations and unstable postures.
In discrete footstep planning the base position can be simply assumed as the projected center of the feet or it must be post-optimized in the trajectory generator. Both solutions are usually not optimal with respect to problem formulation in the trajectory optimization. Trajectory optimizers can optimize the floating base as part of the problem formulation, which is computational expensive. Using the neutral stance pose as initial guess can partially counteract this problem. However, significant improvement is expected, when the provided initial guess is close to the optimal trajectory. A desirable solution would be a sequence of contact configurations including a position of the base. Both pre-optimized with respect to the target position, the system stability and the avoidance of collisions with the environment or the own body.
Another very important aspect is the performance of the planning task of the robot system. A reactive walking behaviour require to make all computations in real-time. Planning time can be reduced by two main approaches. The first method is to reduce the search-space for whole-body configurations, for which different strategies are described in the following sections. The second method is to outsource complex parts of the computation to an offline stage of the planning. This can effect different parts of the planning pipeline which are not dependent on dynamic data. Some solutions pr-ecalculate different basic whole-body configurations and another paper defines a database with default poses on different terrain models. This work shows that there are many application areas for offline computations.
The remaining chapter is organized as followed. First, the most relevant papers for this work are summarized to give an overview of the current state of research. The literature research is concluded by a conclusion of all these results to get parallels to this work and what is different and new in this solution. Finally, a brief overview of the existing footstep planner used in this work is provided.

3.1 Extern related work papers

An Efficient Acyclic Contact Planner for Multiped Robots

The work from Tonneau et al. [35] focuses on the planning of complex movement behaviours like standing up from a chair, climbing stairs or getting out of a car. They reduce the search tree with their own reachability condition, to avoid a bad performance through searching in the complete configuration tree of the robot. This condition provides a reachability constraint ignoring collisions and reduces the dimension of the solution space of possible configurations. The contact planning in their work is divided into two phases. The first phase describes the planning of a root path, of the robot's base without whole-body configurations and the second phase is the planning of whole-body configurations along this root path with respect to the stability constraints. The planning pipeline can be seen in figure 8

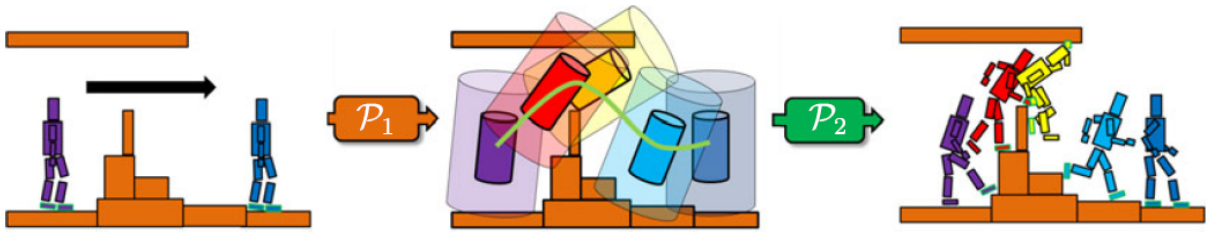


Figure 8: Planning Pipeline with both planning phases P1 andP2 [35]

For planning of contact configuration, they use an acyclic planner due to more flexibility in crossing obstacles and the sequence of moved feet independent of a fixed foot pattern, which makes finding solutions in complex terrain much easier. The main contribution of their work is the definition of a reachability condition to reduce the solution space for feasible contact configurations, while the acyclic planning and the division into different planning phases are state of the art solutions already. The introduced reachability condition has to check for valid root positions in phase one. At first bounding boxes are defined around all environmental obstacles. In the next step different volumes are defined around the robot. The first one includes the torso of the system. Another one includes the reachable working space of the robot. With the help of the bounding volumes they define three constraints for the reachability condition:

- The first constraint guarantees that at least one limb is in range to the environment, which is checked by at least one intersection between the working space of the robot and the environment.
- The second one is a necessary constraint, which ensures that the robot's torso never collides with the environment. The torso bounding volume is not allowed to have any intersect with the environment.
- The last constraint is a sufficient one. It tells that the whole robot bounding box, except of the end-effectors, should not intersect with any obstacles. The end-effectors need to intersect, to guarantee a possible contact planning.

After searching a root path a sequence of discrete whole-body configurations for this path is computed. Every configuration in the plan has to fulfil the stability constraints of the system. The computation of the whole second phase is formalized as an algorithm with following steps:

- Discretization of the root path to a sequence of root configurations
- Extension of each root configuration to a whole-body contact configuration, where every configuration has to be collision-free and in static equilibrium
- Link contacts are computed with a *First-In-First-Out* queue. The link with the longest time without being in contact is used first in the computation.

All contact configurations have to guarantee stability conditions. For achieving this the first criterion is that every configuration should have only one contact break with the environment. If the next configuration has more than one contact break, the algorithm creates more configurations in between those states to mitigate this problem. Another important criterion for stability is the mechanical stability constraint. Their work is based on the basic principles of mechanical stability, as the balance of forces and the rules of the friction cone. With a check on how much a contact force deviates from a friction cone, all configurations can be ranked with a quality value.
The main challenge is dimensionality reduction of the search-space for root configurations using a reachability condition. In this thesis, the reachability condition method is interesting, due to the similarity between the computation of a combination of footsteps and base positions and the computation of a root configuration.

A Reachability Based Planner for Sequences of Acyclic Contacts

Another approach from Tonneau et al. [36] is an extension of the first paper. The whole planning is similar to the first approach, with a division into two planning phases, but in this work, the focus does not lie on the main planning problem, it proposes theoretical contributions to the two planning subproblems. The first point is an introduction of characterization for the guide trajectory, called "True Feasibility". If a guide trajectory is truly feasible it is guaranteed, that a mapping to the contact manifold is possible. Through this characterization it is possible to assert the relevance of a guide root path without computing contact configurations which is a great improvement for the planning performance. The second focus of this work is a consequence of the first one. Due to it is possible to characterize the root path with its relevance, more time can be spent on selecting a particular root path sequence with desirable properties, such as robustness, efficiency and naturalness for an acyclic contact planner. The paper proposes a large variety of movements with different types of legged robots.

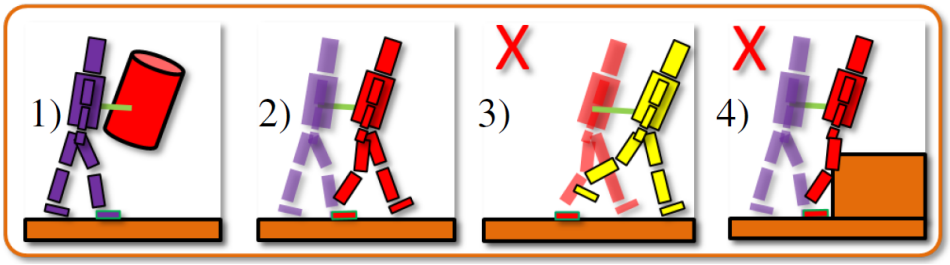


Figure 9: Planning Configurations with EFORT-Criterion [36]

The introduce the *true feasibility* characterization, to reduce the search-space for the guide trajectory. It says, that single root placements have to be close to the environment, but not too close, to avoid torso collisions. Formally this means the definition of a reaching space as the set of interesting root placements. The guide trajectory is computed only in this space. The "True Feasibility" characterization has two conditions.

- The necessary condition for a contact to be possible: An environment volume intersects with the range of motion, during the trunk of the robot is not colliding with the environment volume.
- The sufficient condition extends the necessary one: The bounding box for the robot trunk is replaced with a volume, which encompasses the whole robot in a given pose, except for its effector surfaces, to be in contact.

In practice, the sufficient condition is not interesting, because it leads the solver to lose many interesting trajectories, so a compromise condition is used, where the volume from the sufficient condition can be scaled dynamically. Then the trajectory is computed with a random based probabilistic roadmap algorithm.
In the next stage of the planner, the guide trajectory is transformed into a sequence of discrete contact configurations. Like in the first approach of Tonneau et al. [35] the guide trajectory is discretized into a sequence of discrete placements. Then for each root placement, a contact configuration is computed which satisfies the stability constraints and is collision-free. The main contribution of this work in this stage of the planner is the definition of the "EFORT - Criterion". Through the contact computation is based on a random sampling algorithm, the planner could produce inefficient or unnatural postures. Efficiency here means the frequency of breaking contacts and the probability to move partially backwards. The "EFORT - Criterion" is used when new contacts are created and it only allows new contacts to exert contact forces with the direction of motion. So it is guaranteed that no new contacts against the movement direction are chosen. Additionally,

the contact is maintained longer, because it cannot be broken until all other effectors have moved in the same direction, without breaking the stability constraints.
The main strength of this work is the efficiency of different robot shapes. So especially the "EFORT - Criterion" and the characterization of the root position could be very interesting for this thesis because it is directly transferable to different planning problems.

Whole Body End Pose Planning for Legged Robots

Ferrolho et al. [37] present a work for whole-body end pose planning for legged robots on inclined support surfaces in complex environments. The goal is to develop a robust planning algorithm for humanoid robots, to plan a movement path to a specific target position with contact informations and afterwards computing a final stance depending on the robot's operation task. The path planning should be collision-free and should provide a suitable solution to cross complex environments with obstacles and uneven ground. The approach divides the kinematic structure into a lower-body part and an upper-body part. For both parts, a solution is calculated independently. Different sections of the algorithm are outsourced to offline applications, such as the discretization of reachable workspace or computations of pre-defined whole-body configurations, to reduce computation times. Their work focuses on the planning task with humanoid robots, but the theory can be transferred to all legged robot systems. They demonstrate a high success rate in the planning tasks, but although the performance is improved to previous approaches
Their work includes the discretization of the reachable workspace of the robot into a bounded 3D voxel grid. For every voxel, they create two distinct lists, an occupation list and a reach list [38]. The upper-body planning is a sampling method over the upper-body inverse dynamic reachability map. If the upper-body planning is unconstrained the sample space contains all upper-body configurations, including such with arms behind its back. As this is a non-natural pose for humanoid robots, the constraint sample variant excludes all arm positions behind the robot. The lower-body sampling extends the pseudo-random method from Yang et al. [38] with a static equilibrium robustness measurement. After creating samples for both body parts, based on the current planning problem, the pipeline consists of three steps which are shown in figure 10. At first an inverse kinematic is calculated to the configuration sample. Then a check on static equilibrium and a check for collisions is made. If one step fails, the algorithm is repeated with the next configuration sample.

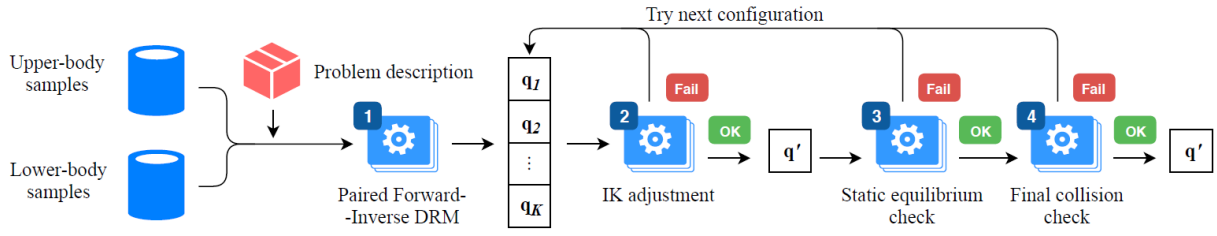


Figure 10: Planning Pipeline: Planning for both robot paths, Computation of a Configuration Space, Checking all Conditions for valid Configurations [37]

The measurement for the robustness of the static equilibrium is described as an optimization problem with a linear program, depending on the center of mass. Due to the splitting of the kinematic structure there is no center of mass in the lower-body configuration, so a lumped point mass was added above the pelvis level to approximate the center of mass.
The context of this thesis lies in the splitting of the kinematic structure, which is related to the splitting between the footsteps and the base position. The difference to [37] is that the order of the calculation is not

final. In this thesis all limbs should be planned simultaneously, while Ferrolho et al. have a fixed order of calculating samples from the lower-body and the upper-body because they are handled independent of each other. In this thesis it is intended that the base position and the foot positions are interdependend due to the position of the robot base affects the options to place the feet.

Locomotion Planning for a Humanoid Robot with Whole Body Collision Avoidance

Another approach from Kumagai et al. [11] proposes a locomotion planning framework for a humanoid robot with an efficient footstep and whole-body collision avoidance planning. The key method is to reduce large computational costs for the whole-body planning by executing a footstep planning first and then performing a sequential whole-body posture planning, using the footsteps center trajectory as a guide. The main motivation is the movement in narrow spaces where using a fixed whole-body pattern is impossible. State of the art approaches are searching global solutions for the whole planning problem, which results in bad performance, especially on robot systems with many degrees of freedom. In [11], only the footstep planner is used as global planning stage and then body poses for each footstep pairs are computed separately, resulting in significantly planning speed up.
The global footstep planner is based on an A-Star algorithm using a LiDAR-based environment modeling. The robot body and the environment are approximated with bounding boxes. The robot upper-body bounding box has a dynamic size, depending on the sway motion of the robot.

After the footstep planning process, a complete path of whole-body configurations is determined. The whole-body planning uses a quadratic program with prioritized conditions for collisions and link- and velocity limits. Planning through a narrow passage or a bundle of obstacles can result in an unnatural pose, for which reason every step ends up in a collision-free configuration with a given default pose of the system. A limitation of this approach is the fixed footsteps in the planning of whole-body configurations. If a more optimal combination of footsteps and a whole-body pose theoretically exists, the whole-body planner is not able to switch over the already planned footsteps. This thesis is confronted with the identical problem, so a simultaneous approach of the planning could solve this limitation. There is no immense deterioration of the computation time because the footstep planner plans a step, and then a body pose is searched corresponding to this step. Only if there are better combinations of feet and base pairs the planner selects an alternative solution. So the body pose is not planned globally, which is a main target in this thesis. It is discussed in detail in section 4.

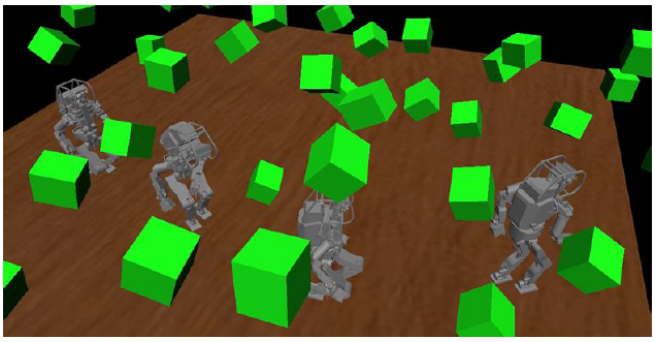


Figure 11: Path Planning through terrain with Obstacles [11]

Humanoid Robot Collision Free Footstep Planning

The next paper which is referenced here [39] is focused on the avoidance of 3D obstacles and the approach works only on flat ground. Offline computed swept volume approximations are used during the planning phase, to reduce the computation time. A rapidly exploring random tree variant is used to find collision-free sequences of half steps. The computation of the root pose of the system is based on the ZMP criterion. Though

this is only considered with maximal three-legged robots in practice, the computation of the root is nearly irrelevant for this thesis.

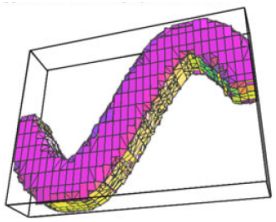


Figure 12: Planning with approximated Swept Volumes [39]

Interesting is the approach with the swept volumes for collision avoidance. In a grid of possible footsteps for each footstep, a volume is computed for the lower part of the leg. These volumes are computed offline to achieve better performance. Now the planner tries to find a sequence of these swept volumes. With intersections with the environment, it is possible to detect collisions in the sequence and change the choice. With respect to this thesis, these swept volumes could be used to avoid collisions with stairs or stepping stones on the ground. Due to the independence of the volumes to the planner, it is possible to compute them offline, which improves the performance of the algorithm. In the evaluation, it is shown that the rapidly exploring random tree achieves very satisfying results in finding paths. But due to the approach being limited to two-legged humanoid robots, only the swept volume approximation is an interesting procedure.

Rough Terrain Motion Planning for Actuated Tracked Robots

Another really interesting scientific work is this one from Octavio Villareal et al. [12]. This work deals with a dynamic approach to adapt foothold positions during the walking process. With each step in the lift-off phase of the walking process, the algorithm computes the expecting landing foothold. In the same step, a cutout of the heightmap in this region is made. The region cutout and the expected foothold target are used as input for diverse foothold heuristics. Based on these heuristics the algorithm searches for the best possible position in the region cutout and generates an adaption for the expected foothold position. If a better target position is found, the adaption algorithm corrects the planning and changes the new target.
Due to performance issues in the complex computation of the terrain heuristics a convolutional neural network learns each possible adaption. In the evaluation of this work the usage of a well trained neural network achieves computation times better by a factor of ten, compared to the heuristic-based algorithms, so it is possible to compute adaptions within situations of disturbances in real-time. One example shows the robot is pushed away from its original step plan. The real-time adaption generates new foothold positions instantly and lets the robot return on its original path.

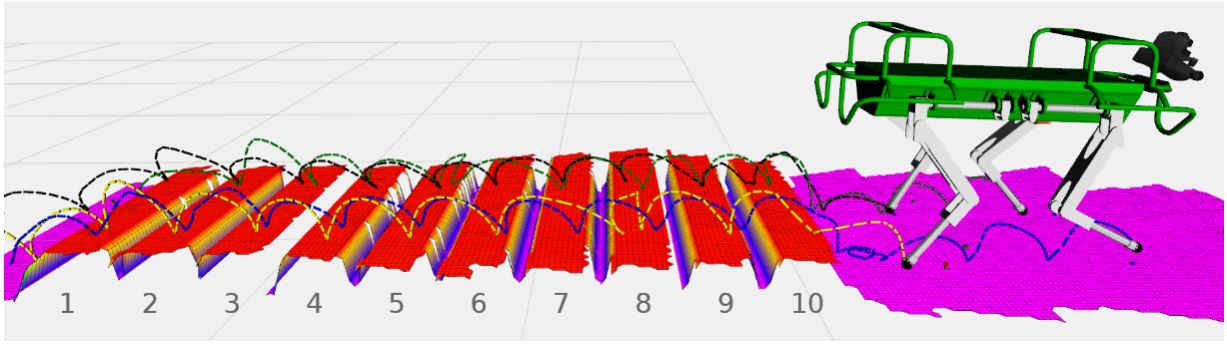
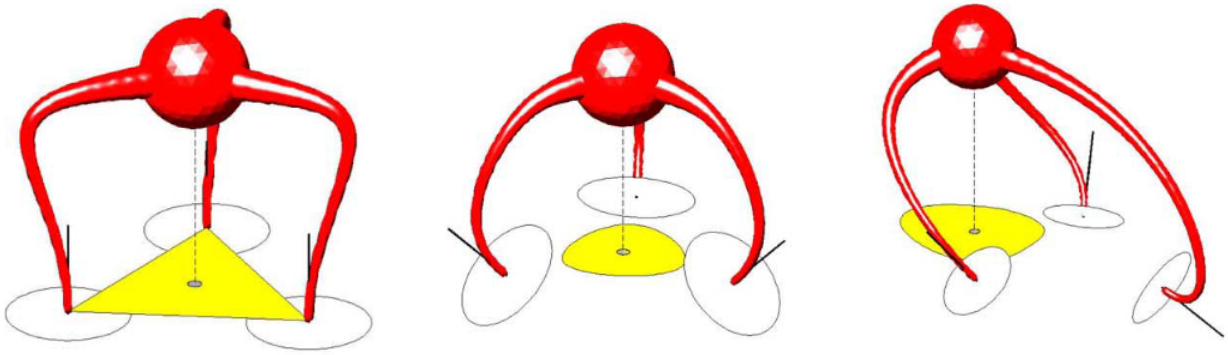


Figure 13: A gap crossing scenario with the original plan and the adapted step plan [12]

The most interesting knowledge from this paper with respect to the planning field are the foothold heuristics on which the algorithms compute the foothold adaption. These heuristics can be overtaken in the planning field directly. At first, every position in the cutout which is kinematically impossible is discarded. Then the roughness of each pixel in the region is computed based on the elevation mean and the standard slope deviation to its neighbourhood. All pixels above a defined roughness threshold are discarded. Additionally, an uncertainty margin is included which discards each pixel around such rough pixels and also pixels near detected obstacle edges. The last point is the avoidance of collisions with the environment. A foothold can be reachable and kinematically possible, but in some situation, another part than the foot can end up in a collision. This heuristic calculates these cases and discards such positions. All these heuristics can also be used in the planning scenario.

Testing Static Equilibrium for Legged Robots

The content in this section consists of three different papers which build on each other. At first, there is the work around Bretl et al. [40] and it treats the static equilibrium for legged robots with respect to the position of the robot base at fixed foot positions.
For static equilibrium, the robot system has to compensate all gravity forces on its center of mass with its feet contact forces. Due to the forces at the feet contacts being fixed on different robot base positions there is an enclosed set of possible positions for the center of mass. Until now it was a good approximation to use the convex support polygon of the feet in contact as this set of positions. Bretl shows that this assumption is only a good one on flat and smooth terrain. If the robot is moving on rough and steep terrain this approximation does not build a good set for base positions. As soon as the feet in contact can no longer exert forces parallel to the gravity vector, the support region is no longer a polygon. In general cases on natural terrains, it is never a polygon and in some cases, it does not even lie between the feet positions.



(a) Support Polygon on Flat Terrain

(b) Support Region on uneven Terrain

(c) Support Region outside the Feet

Figure 14: Different results for Support regions, depending on the Terrain. [40]

In the refined version of the support region, Bretl uses three conditions to the robot base position. The first and second condition describes the force and torque balance of the system with respect to the foot contacts and the gravity. The third condition requires each foot reaction force to lie inside a friction cone according to a Coulomb Friction Model. With these conditions, Bretl formulates a second-order cone program and a cutting plane algorithm to compute the refined support region. A strong advantage of this algorithm is that it is not always necessary to compute the whole support region. In some cases, only some positions for the center of mass must be checked on lying in the support region. If the amount of checked query points is low this query check is much more performant than computing the whole support region. In the normal case, the algorithm runtime depends on the desired accuracy of the approximated support region. The more precise the result should be the more calculation time is needed.
The work from Bretl et. al is continued from Orsolino et al. [13]. In this work, an actuation aware extension of the support region is handled. Here another fourth condition is added to refine the support region. The new condition is based on the torque limits of the robot limbs. Orsoline defines so-called force polytopes around each foot. A force polytope describes the three dimensional set of possible reaction forces which can be exerted by the feet without exceeding the torque limits. Such a volume A_i can be calculated with the following relations

$$w^{lim} = J(q)^{-T}\tau^{lim}$$

$$A_i = \{w_i \in \mathbb{R}^m | \exists \tau_i \in \mathbb{R}^n\}$$

where w is the reaction force vector, $J(q)$ the jacobian matrix of the legs and τ^{lim} the torque limits in the limbs. Additionally to the torque limits the reaction forces have to fulfil the mechanical equilibrium presented by Bretl et. al. Especially in narrow passages, this condition provides helpful results because the robot has not much configuration possibilities in such situations.
The findings from both of these works are especially in the planning scenario an interesting contribution because in planning it is all about planning single limbs, while all other limbs are fixed. This assumption simplifies the calculation from both papers enormously, because all dynamic parameters can be ignored.

3.2 Conclusion of the current State of Research

As shown above, there are many different approaches to solve the footstep and floating base planning problem on legged robot systems. All of them deal with the problem of whole-body configurations and try to find a

global solution. But it is also clear, that there is no universal solution for this problem. In every single approach, the trade-off between performance and robustness is an issue because the search-space for a whole-body configuration is nearly infinite. One strategy is to divide the global problem into different less complex problems, like in [35], to deal with this tradeoff. Another solution is to outsource as many computations as possible in offline procedures like in [37].
The target of this work is to extend a global footstep planner with a planning strategy for a pre-optimized robot base position. If the planner only computes a position path for the steps, the trajectory generator has a nearly infinite search space for the leg trajectories. If the planner provides not only the step positions but a sequence of optimized whole-body configurations, the trajectory generator has a substantially smaller and more optimal search space for its results. Optimized means, that the search space is reduced to a constrained set of sensible possibilities. The conditions for the optimization are dependent on the use case, but generally, they are collision avoidance, stability margins and sometimes natural movements. The more specified the operation area of the robot, the more specified become the conditions. So the target is to reduce this search space for trajectories to an optimized set, by computing a robot base position that is optimized based on given constraints. For robots without any limbs except their legs, like an insectoid robot from [36], this means a whole-body configuration. But for humanoid robots with further effectors, it is a weakened version of a whole-body configuration because the arms are not integrated into the planned configurations.
This work does not investigate methods to plan whole-body configurations. Instead, it tries to solve what a whole-body configuration does, in another way. Due to only concentrating on the robot base position, the performance issue of whole-body planning is not such an issue. Another advantage is, that an upper-body part if it exists, is independent of the walking planner and can be handled on its own. The idea is, that an intelligent positioning of the robot base can be a great benefit for the search space of the trajectory optimization.
Additionally, the papers shown before have approaches to solve atomic problems. First, the feet are planned and then the base or vice versa. Simultaneous planning to model the motion system as one system in real-time applications is not solved until today.

3.3 L3 - Footstep Planning Framework

For the evaluation of the results of this work, the footstep planning from the Legged Locomotion Library (L3) Framework is used. L3 is a locomotion library for legged robots. Actually, it includes a footstep planning framework, a step controller for scheduling step plans, a walking controller for the execution and a basic terrain model generator. Each framework in the library is programmed based on a plugin structure and each issue can be represented as a plugin. Farther the library offers a whole data infrastructure to realize walking and planning issues, like data structures, functional structs and all the needed methods. The framework provides the perfect environment for evaluating footstep planning extensions because it is possible to simply add new functionalities as different plugins. There is no intervention into the planning pipeline needed, because the pipeline ensures, that all plugins are called in the right order.

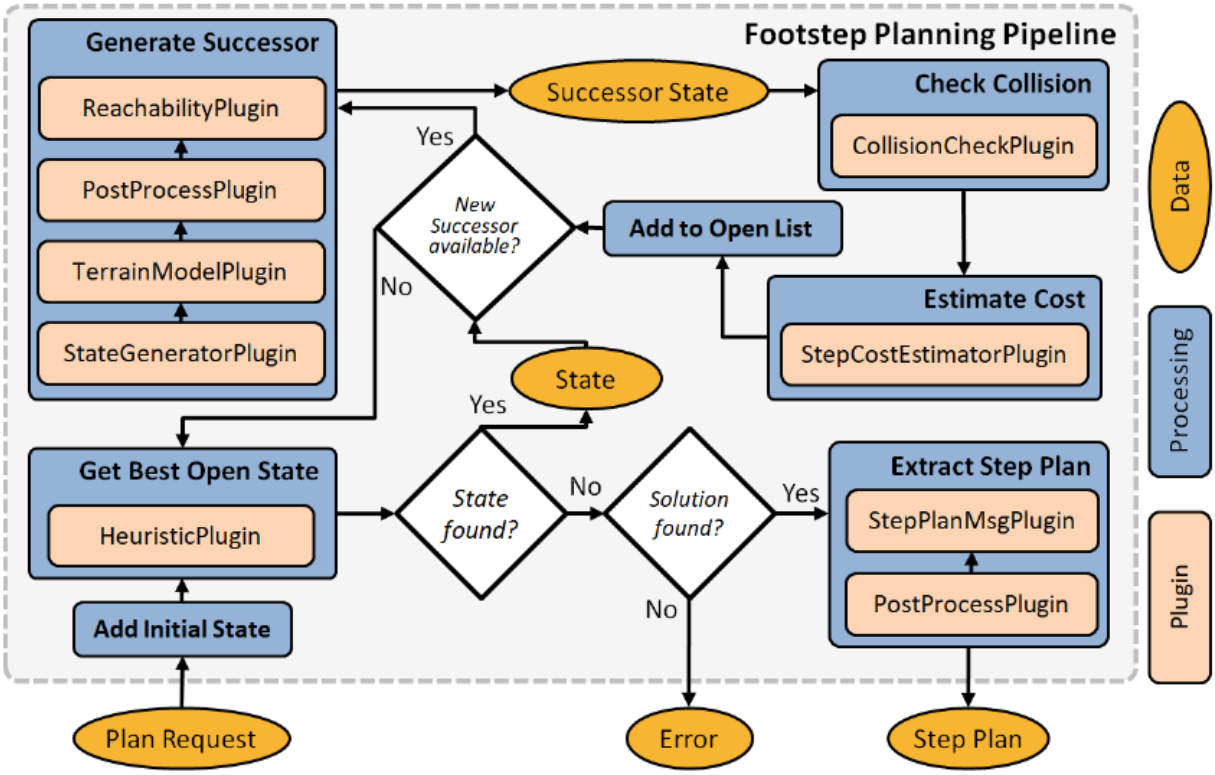


Figure 15: Planning pipeline in the L3 framework [18]

The infrastructure of the L3 library provides important data structures for all planning tasks. There are ready to use structs like steps, foot and leg descriptions, poses, transforms, footholds and many more useful tools. The geometrical structs are based on the Eigen library and can be used like all Eigen tools. For all elements, the basic geometrical functions like transform methods or arithmetic operations are implemented and tested with unit tests.
As mentioned above the planner is designed as a plugin-based structure. A central controller schedules the planning process and the plugin types. A plugin represents a single task of the planning process and the user of the library can implement his own methods as plugins or he can use a default implementation, provided by L3. In the end, the user only calls the central footstep planning controller and the planning should work out of the box. Here a short summary of the plugin types is given:

- Heuristic: Here the user can implement all the heuristics he wants to use for the graph search algorithm in the planning.
- Post Process: Provides the possibility to the user to give a post-process to every single structure of the planning process, like single states or final step plans.
- Reachability: Plugin which computes the reachability of given step poses.
- State Generator: Generates a complete planning state
- Step Cost Estimator: Estimation to get the costs for one step
- Terrain Model: Interface to the terrain model. The default implementation uses the L3 Terrain Model, but the user can use any model he wants.

- Collision Check: Collision checks for the environment and own body

Instead of implementing an own infrastructure for this work, the L3 framework is used. The advantage is, that the investigated methods can be implemented as new plugins and the planner schedules them and includes them into the planning process without any further work to do.
More detailed information about the L3 library and the planner itself, can be seen at [18] or [41]. These works present the whole library with all its functionalities in detail.

4 Method

In this chapter, the approaches and methods of this thesis are presented. The main content is about the contribution of this work to the current state and why these methods provide an advantage in the field of robotic motion planning. The main focus lies here on the legged locomotion of robots, especially the motion with an eight-legged robot system which is also used in the evaluation of the approaches.
In the related work section, many different approaches are shown. Every single solution targets one single problem in motion planning. In every case, there is a tradeoff between the optimality of the found path and the performance of the computation. Especially the performance is an important aspect with respect to a real-time application like motion planning and in nearly every solution a non-optimal path is taken into account for a better computation performance.
A promising planning strategy is whole-body planning. In this approach, the planner generates a sequence of whole-body configurations of the robot system. The advantage is that the motion application of the robot only has to find trajectories between these whole-body configurations. The goal configurations guarantee a collision-free state which brings the robot closer to the goal state of the plan. It is obvious that the planning of whole-body configurations is computational complex because the combination of every robot joint generates a tall set of whole-body samples. On the other side, the planning of only footholds is performant because the planner only has to find feasible placements for the feet. The rest of the robot is not taken into account. These approaches have the problem that especially in narrow passages the upper-body of the robot can run into collisions or the planner generates unstable robot postures. A compromise of these two strategies would be a planning of the base of the robot and a planning of the feet. The base represents a central link of the robot system and analogue to a human body it can be the hip or the breast. The planned floating base simplifies the whole upper-body planning of the robot and reduces the number of joints to be planned. This idea of floating base planning is not new and some approaches from above use this simplification, but each approach separates the planning of the base from the foothold planning. The feet and the base can never be planned as coupled system due to this separation of planning phases. Based on the order of planning, one phase depends on the other but never vice versa. Depending on the environment, problems with the trajectory generation can occur as introduced in chapter 1. The main contribution of this work is the simultaneous planning of the floating base and the feet, to handle these challenges. With simultaneous planning, the resulting goal configuration can be pre-optimized with respect to obstacle collision avoidance in the trajectory generation, unstable postures and non-desired joint-states.
Different methods and approaches are investigated, implemented and evaluated in this work, to achieve an improvement in the trajectory generation. The following section gives a detailed explanation of these approaches.

4.1 Hardest first approach for simultaneous planning

The main problem in the planning step is the size of the search-space. With a probabilistic sampling approach, the search-space of possible configurations is nearly endless. Using a discrete state-based planning algorithm like graph search brings a massive reduction of the search-space. But even here, the amount of possible state combinations just with a foothold planning is enormously large, depending on the discretization resolution. Including the floating base planning, this problem is not applicable in real-time.
Many states in the set of possible states are not optimal with respect to reaching the goal on the shortest path, additionally, there are some states which lead to physical or mechanical limits of the robot system. For example, in front of a stair, there are possible foot placements that are too close to the stair to move the foot over the obstacle without collision. So the approach to realize simultaneous feet and base planning in this work is to reduce the discrete state search-space to achieve planning usable in real-time applications. The

single placements for the feet or the base, which lead to problematic planning behaviour like in the example, have to be detected and discarded from the search-space, to reduce it.
In this work, the hardest first approach from [42] is used, to achieve this goal. In the standard planning behaviour, the continuous environment of the robot is discretized into discrete planning states. These states consist of possible foot positions and a floating base position for the succeeding state and the current position of the robot limbs. In this work, a polygonal discretization around the feet and the base is used, so each foot and the base have a limited set of reachable positions and orientations. Without the approach from this work a standard A-Star algorithm would expand each possible combination of states, resulting in a massive amount of states. The hardest first heuristic computes a value for each limb which says how hard it is to move this limb. The fewer possibilities a limb has the harder it is. Actually, the hardest first heuristic is a collection of multiple different heuristics, each of them calculates a value of how much sense a single position has to move to. If a possible foot position has multiple bad heuristical values, the position will be discarded from the planning algorithm. After checking all limbs, each limb has a rest amount of possible placements. The one with the fewest is the hardest limb. Due to this limb could lose the rest of its possible positions through bad placements of the other ones, it will be prioritized by the planning.
In this work different heuristics to discard single foot or base placements related to the robots environment and physical limitations are investigated and presented in the following sections.

4.1.1 Hardest first Gait Generation

The standard way to plan a step plan with the A-Star algorithm is to expand all possible state combinations and their transitions. If this procedure will be executed in each step with all feet and the robot base, an enormous amount of data points would be generated, depending on the size of the discretization polygons. In the case of homogenous feet polygons, this would be

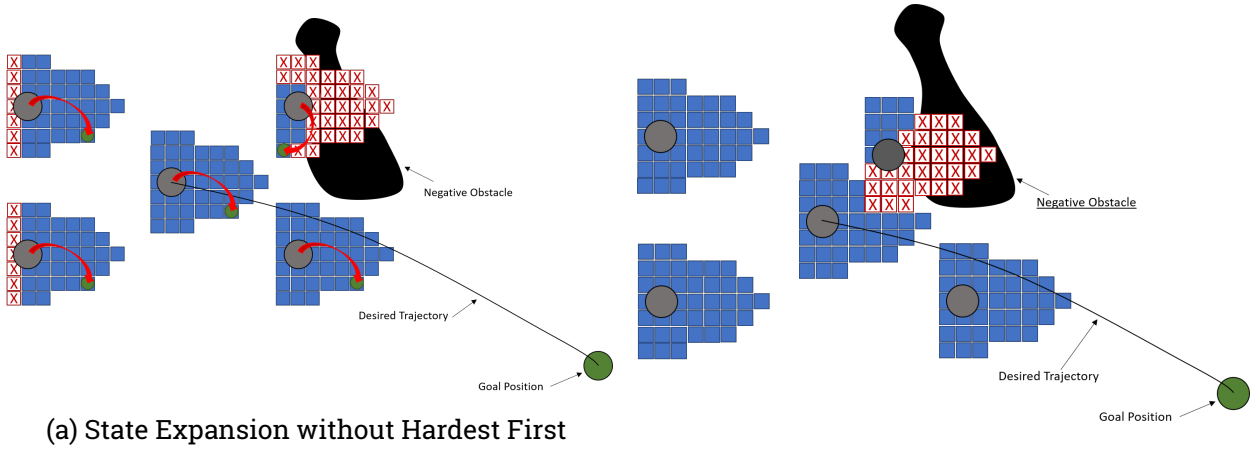
$$N = p_f^n * p_b \tag{3}$$

where N is the number of expanded states, p_f is the size of one foot polygon and p_b is the size of the discretization polygon of the robot base and n is the number of feet. As mentioned in chapter two this would lead to an immense workload on the memory, because all graph nodes are kept in the memory, by using an A-Star algorithm. With already four planning feet in one iteration this would lead to an overload in the memory system, as shown in the example where $p_f = 320$ and $p_b = 680$ which are common values for the discretization polygons.

$$N = 320^4 * 680 = 7.130.316.800.000 \tag{4}$$

A gait generator is used to manage the input of the state generation, to avoid this problem. Such a gait generator is just a tool to not lose control over the state generation. The configured gait determine the order of visited states. In the special case of footstep planning, it provides the feet to handle next in the state generator. In the standard footstep planning, non-weighted gait generators are used. For example, a simple gait generator for bipedal robots provides an alternating list of the right and the left foot. So the state generator expands only the states for one of the two feet in one iteration. In the case of a quadruped, a common non-weighted solution is the cyclic gait generator. The user can configure different cycle orders for the feet and the gait generator generates feet pattern for the state generator. However, it doesn't matter how many feet are passed to the state generator. Without special commands, it won't discard positions that are not optimal with respect to collision avoidance or the robot kinematic. So the state generator would expand all possible positions for a foot given by the discretization, no matter if a position brings the robot's body into a collision or dangerously close to an obstacle. Later in the reachability or the collision checks, these positions could be discarded but the A-Star algorithm has already expanded all states. This state expansion could be

reduced significantly, using the gait generator to build a pre-order of footholds, where all feet with at least the possibilities are prioritized. Such a gait generator has still a cyclic pattern but this pattern builds only a suggestion of possible following footholds. This heuristic version of a cyclic gait generator iterates through all foothold suggestions from the pattern and decides which foot has to be passed to the state generator by using heuristics. Each heuristic investigates the whole state-space of a foot and computes a cost value for each possibility. In the end, the mean of all costs is computed which represents the heuristic value for the foot for one individual heuristic. Then the mean of all heuristics on the foot is calculated whereby the single heuristics are weighted to differentiate between more important heuristics and less important ones.
Only the foot with the highest cost value is passed to the state generator. So it is guaranteed that the state generator expands the minimum amount of necessary states. If the state generator expands the states for each limb in an arbitrarily order it can happen, that a heuristically restricted foot has a too large search-space reduction due to the virtual target position of another limb.



(a) State Expansion without Hardest First Heuristics

(b) Deadlock in the front left Foot

Figure 16: State Expansion with and without the Hardest First Approach

In figure 16a a quadruped robot with one floating base is shown. The blue polygons represent the discretization around each foot and each square in a polygon is a possible target position for the foot or the floating base. The green circle is the target position for the whole step plan and due to the shortest distance heuristic, the robot has a desired trajectory that connects the robots center and the goal position. The front left foot is restricted in its possibilities due to a negative obstacle in front of it. The state generator now has an arbitrary order for its state expansion and the restricted foot is in this example the last one. If the state generator gets a foot it always expands all possible states but for better comprehensibility, only one combination of states is shown. Due to the distance heuristic of the planner, the planner would choose the positions which bring the robot as close to the target as possible. But with this choice, the restricted foot is forced to move to the right as the already set new footholds restrict the remaining search-space and so it is no more possible to cross the obstacle. A better solution would have been to first plan straight over the obstacle, against the distance heuristic of the planner. As shown in figure 17 the desired trajectory gets changed after the step, but with this step, the restricted foot has more possibilities to move to the goal position. If the gait generator had chosen this solution at first all the other states would never have been expanded in the state generator, because there are much fewer combinations to build. In the best case, the complexity of the expansion is reduced from an exponential to a linear problem.

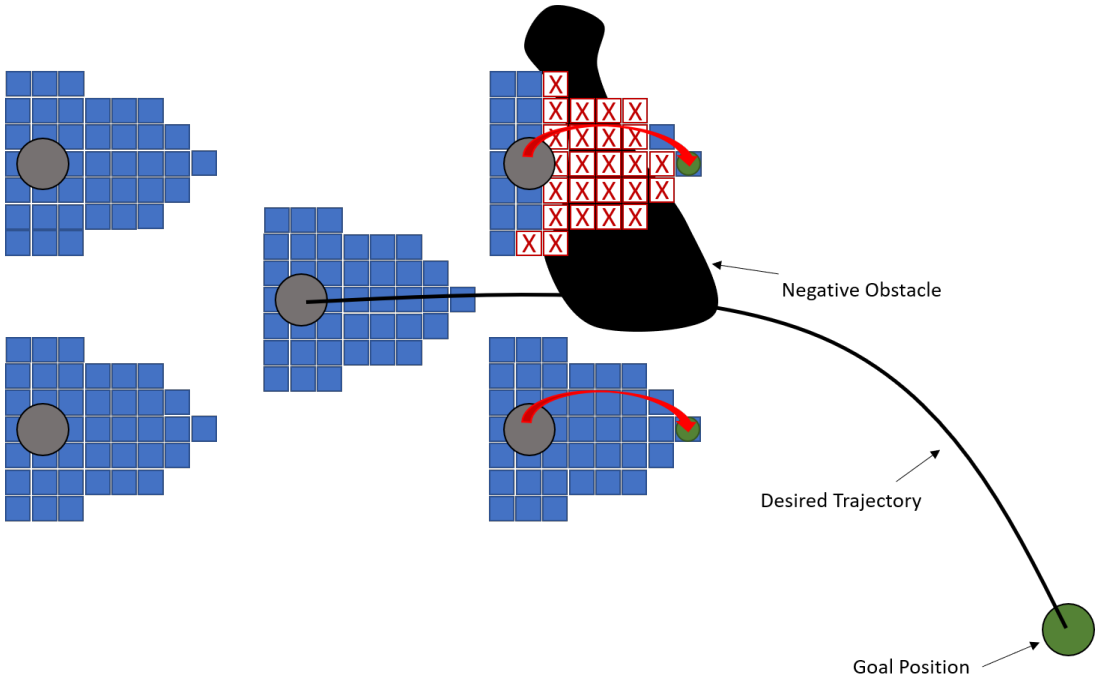


Figure 17: State Expansion with Hardest First Heuristics

Now the advantages of a heuristical approach of building a gait pattern are shown. Due to the benefit of complexity reduction in state expansion, it is now possible to include the floating base planning, simultaneously with the footstep planning.
With an additional floating base position in the step plan, it is possible to improve the postures of the robot with respect to its environment. If this improvement is still part of the footstep planning the trajectory generation has a more detailed input of the whole target configuration and can be executed with improved performance.
The following sections deal with the individual heuristics for foot positions and floating base positions.

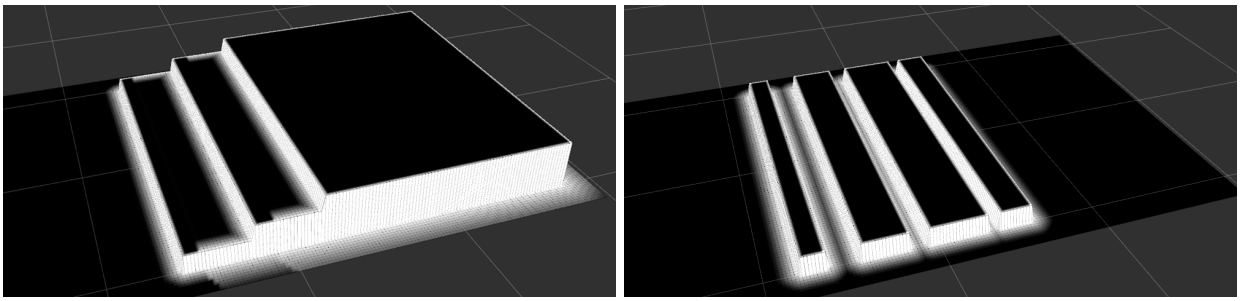
4.2 Hardest first heuristics feet

In the classic footstep planning pipeline without any heuristical optimization, every foot has a discrete amount of successor states, described with a polygon around the foot. If any gait generator provides a pattern that require the state expansion of a foot, all these discrete states would be expanded. The target of a foot heuristic is to decide whether a state should be expanded or not. For every reason to discard footholds, one heuristic is designed and applied. In this work explicitly two heuristics are investigated, implemented and evaluated. Both investigated heuristics are related to the terrain around the foot and require a model of the surrounding environment. The first heuristic computes a decision based on the distance of the position to the nearest obstacle. The second one is based on the roughness of the terrain.

4.2.1 Heuristic: Distance to nearest obstacle

Like mentioned above the first heuristic for the foot state expansion is based on the distance to the nearest obstacle of a position. The problem is that the possibility to cross an obstacle depends on the last position of the robot in front of the obstacle. In the case of positive obstacles, the front feet should not be too close to the obstacle, in contrast to negative obstacles, where the front feet should be as close as possible to the obstacle.

First, the distance heuristic for positive obstacles is presented. When a foot is placed close to a positive object the foot may not be able to move forward across the object due to the robot kinematic constraints. Depending on the length of the leg, the robot needs a minimum swing space for crossing an obstacle. In front of a positive obstacle it can occur the case that the final foot position on the obstacle is kinematically possible, but no feasible swing motion can be found. This results in several parameters with which the problem can be described. At first of course the minimal swing radius of a leg, which is given through the robot kinematics. Next, the height of the obstacle is important to find the optimal distance to it.
With these two parameters, a cost map of the surrounding area of the robot can be computed. Every cell in the cost map grid has a cost value to place a foot on it. The algorithm takes the common elevation grid map and computes a new layer with the cost map. The swing radius parameter is given statically in a robot description file, while the obstacle height has to be detected in the grid map. Now the algorithm iterates over a detected grid map and computes the cost value for each pixel. Additionally to the minimum swing radius around obstacles, a buffer distance around each obstacle is taken into account to securely avoid collisions. The algorithm has three possibilities to compute the cost values of a grid cell. In every case, the algorithm discards the cell within the swing radius around an obstacle, with a maximum cost value. The first way is the binary cost map. A grid cell can only assume two values. Zero means a minimum cost value and the position is accessible, while one is the maximum cost value and means that a position is not accessible. All cells around an obstacle within the minimal swing radius and the buffer distance assume the value one and are discarded from the planner. All remaining cells are assumed to have the value zero and are accessible.



(a) Distance Cost Map Simulated Staircase (b) Distance Cost Map Simulated Step Stones

Figure 18: Distance Cost Map for the Heuristic where white means not traversable

The disadvantage of a binary cost map is that the buffer zone has to be chosen carefully because if it is too big, it can happen that crossing an obstacle is no more possible because there remains a small valid area for the foot to step on. This is solved by introducing a buffer zone, defined by a cost function. All cells within the minimal swing radius have still the maximum cost value. All cells within the buffer distance zone have decreasing cost values, depending on the distance to the obstacle and the cost function. So the algorithm tries to find a position as close as possible to the obstacle with the minimum amount of costs. This results in a position outside the minimum swing range and the buffer zone.
Another aspect is the positioning of the feet in front of negative obstacles. Many insights from the positive obstacle problem can be adapted for this problem. The only difference is that the foot needs to be as close as possible to the object to be able to cross it. If a foot is placed too far from it, it can happen that the leg intersects the obstacle after stepping it down. Another problem is, that the leg joints can reach kinematic limits if a support foot is paced too far from the obstacle. Like in the case with the positive obstacles a cost map is computed, in which all cells near negative obstacles are taken into account. In contrast to the positive obstacles, there is no minimum swing space between the foot and the obstacle. In every case, the foot should be placed as close as possible to the negative obstacle to ensure a problem-free crossing motion.

The algorithm just adds a security buffer zone to avoid that footholds being placed on an obstacle edge.
The two cost maps can be combined into a single map layer, where the cells in front of positive obstacles are decreasing linearly in their cost values and where the negative obstacle edges have a safety margin to avoid accidents. The heuristic just has to perform a map lookup at a possible target position and can provide the heuristical value for a given position. The innovation of that approach is the large amount of computational reduction. The cost map filter only has to be applied once when the map is loaded. Otherwise the computation had to be repeated in every state check request.

4.2.2 Heuristic: Terrain traversability

Another heuristic in this work introduces three terrain properties. The *roughness* of a position describes the mean height difference to its neighbourhood.

$$r_p = h_{mean} - h \tag{5}$$

The roughness at the position p is r_p and h_{mean} is the mean height in a given neighbourhood, where h is the current height at the position p. The roughness represents how easy it is to traverse a position, related to the surrounding terrain. The next important property is the *slope* value, which indicates the inclination of a position p related to its neighbourhood and is computed using the terrain normal vectors,

$$s_p = \arccos(n_p) \tag{6}$$

where s_p is the slope value at the position p and n is the normal vector at the position p. The normal vector is computed, based on the neighbourhood of the position, so the slope value is related to the neighbourhood too.
The combination of the *roughness* and the *slope* value builds the basic property for the heuristic computation, the *traversability*. The traversability should indicate how easy a terrain position is to move to, with respect to the balance of the system. The traversability value is computed as a normalized and weighted sum of roughness and slope of a position.

$$t_p = 0.5 * (1.0 - w_s * s_p) + 0.5 * (1.0 - w_r * r_p) \tag{7}$$

The weights w_s and w_r are parameters and have to be optimized empirically with experiments. The slope value s_p and the roughness value r_p are defined above and in combination, they provide the traversability value t_p.
The heuristic takes the traversability value directly as the heuristical value for the possible foot position. The higher the value is, the harder is the position to visit. The heuristic iterates over the whole foot polygon and computes the mean value of all traversability values. The higher the traversability mean is, the harder it is to move the foot. If the traversability value at a position exceeds a specified threshold, the position is discarded from the polygon and will not be visited by the planner. This threshold has to be determined experimentally. This is shown later in the evaluation of the heuristics.
The traversability heuristic is also realized with a new grid map layer that contains all traversability values for each position. So the heuristic algorithm has only to look up the values in the map.

4.3 Hardest first heuristics floating base

Now the two heuristics for the feet are described, this work also introduces two heuristics to place the floating base of the robot system. The floating base is described with a central link of the robot system and is an approximation of a whole-body configuration. If the planner is able to build floating base positions related

to each foot configuration, the trajectory generator has much effort to optimize the resulting trajectories. But planning including a floating base can be hard with respect to the performance due to the amount of expanded states becomes exponential large with the number of used legs. These heuristics ensure that unnecessary states are pruned of the search-space to reduce the planning time.
In this work two heuristics for floating base positions are introduced, both depending on the terrain surrounding the robot system. So they require a suitable model for the terrain to perform their tasks. The used terrain model in this work uses the terrain model from the L3 library in combination with the grid map model from *ETH Zürich*.
The first heuristic tries to control the pitch of the floating base, such that the base matches the inclination of the ground. The second heuristic is more complex and is divided into multiple stages. Generally, the heuristic computes a feasible region for possible floating base positions with respect to friction values, the contact normals and the torque limits in the leg joints.

4.3.1 Heuristic: Floating base orientation

The optimal positioning of the floating base can improve the trajectory generation immensely. Additionally, well-chosen heuristics can reduce the search-space significantly. The number of possible state combinations increases with the number of degrees of freedom. In some situations, especially walking on rough terrain, a specified floating base position can improve the whole walking process in terms of the avoidance of unstable positions or non-desired joint-states. When the robot's position is in front of a stair the orientation or the position of the floating base impacts the reachability region for the legs as shown in figure 19. With respect to the collision avoidance and the avoidance of non-desired joint-states, a backshift of the floating base can also shift the minimum swing radius of the front legs.
Generally, it can be said that the swing radius is shifted in the same direction as the floating base moved to. So if the floating is moved backwards, the swing radius of the leg is shifted backwards too. The same applies to a forward movement. Another possibility to influence the reachability property of the feet is to change the orientation of the floating base, especially in the pitch of the link. Increasing the pitch can be seen like leaning back the upper-body. This will also shift the swing radius of the leg backwards. The same applies to bending over or decreasing the pitch value. The leg swing radius is shifted forwards.

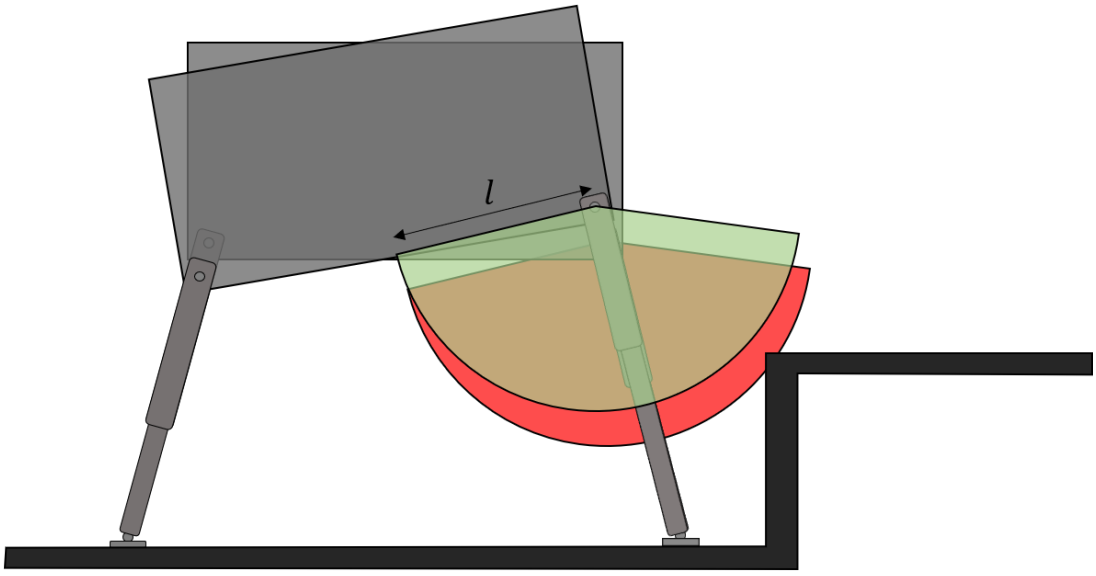


Figure 19: Motion Area Shift through Base Pitch Rotation

The heuristic works in different steps. In the first step, the terrain in front of the robot is scanned. The terrain is approximated with a slope in the planning direction. In the next step, the slope inclination is compared with the current floating base orientation. If the slope reaches a minimum inclination threshold between the current orientation and the slope's inclination the heuristic is triggered. Otherwise, the heuristic returns a default value which means there are no specific restrictions for the floating base.
A detailed terrain model is required, to build a virtual slope in front of the robot. At first, the height values are needed. Then an obstacle detection is performed on the given terrain map. If there are obstacles in front of the robot with edges nearly orthogonal to the planning direction, the virtual slope is built with the current height of the feet center and a point on the edge in the planning direction of the robot. The slope is only built if the height of the obstacle in front of the robot exceeds a minimum height. This should avoid that a rough ground triggers the heuristic algorithm.
After building the virtual slope the inclination of the slope is compared with the current pitch value of the floating base.

$$d' = \sqrt{d^2 + h^2} \tag{8}$$

$$i = \arccos(\frac{d}{d'}) \tag{9}$$

The inclination i of the slope can be computed with the set of catheds. So at first the distance from the feet center to the edge is computed as the hypotenuse d', from the distance on the ground d and the height value h in this point.

$$H = \begin{cases} 1 & |i - \beta_b| > t \\ 0 & |i - \beta_b| \leq t \end{cases} \tag{10}$$

The heuristic value H is normalized between 0 and 1 for each heuristic. If the difference between the inclination of the slope i and the current floating base pitch β_b diverge over a given threshold t, the pitch of the floating base should be adapted. In the default case, the heuristic tries to place the floating base parallel to the ground inclination. With a parameter, this behaviour can be adapted in each direction. Every time the algorithm decides to adapt the floating base orientation the maximum heuristical value is chosen to trigger expansion of the floating base in the next iteration.
If the floating base is not expanded first, the planner would expand all possible position and orientation combinations for the floating base. This would lead to a large overhead of expanded states. Expanding first the floating base, the planner does not need to expand all unnecessary combinations.

4.3.2 Heuristic: Feasible region

The last heuristic to find a floating base position is the most complex one. It is based on the static equilibrium constraints of the robot system. Additionally, friction conditions and torque limits of the joints are taken into account. The main target of the heuristical computation is to reduce the area of possible floating base positions to the optimal one. At first, it is described what an optimal floating base area means.
The simplest possibility to achieve static equilibrium in multi-legged systems is to build a support polygon based on the foot contacts of the system. The only constraint to the floating base of such a system is, that the projected position of the robot floating base never leaves this support polygon. For most of the applications, this approximation of the supporting base area is enough, but especially in rough terrains the model for static equilibrium has to be defined more precisely.

At first, the position of the floating base has to fulfil the conditions of a mechanical static equilibrium. As mentioned above this is defined with the balance of forces and the momentum balance. Based on this definition, three constraints to the feasible support region can be derived.

$$\sum_{i=1}^{n} x_i + mg = 0 \tag{11}$$

The balance of forces defines that the sum of all contact forces x_i added to the gravity force has to be zero, where n is the number of contact points, m the mass of the system and g the gravity acceleration. Additionally, the sum of all torques has to be zero too, based on the same contact forces.

$$\sum_{i=1}^{n} r_i \times x_i + c \times mg = 0 \tag{12}$$

Here r_i are n frictional contact points and c is the 2D projected position of the center of mass of the system. The center of mass corresponds to the floating base of the system, to simplify the problem. Otherwise, a transformation from the floating base to the center of mass has to be added in every computation step. The last constraint given by the mechanical static equilibrium is the compensation of the friction forces at the contact points. On perfectly flat terrain they are zero, but in a real application, this cannot be assumed.

$$\|(I - \eta_i \eta_i^T) x_i\| \leq \mu_i \eta_i^T x_i \tag{13}$$

I is the identity matrix, $\eta_i \eta_i^T$ the decomposed contact normal matrix in point i and μ_i the friction coefficient at the point i. The friction coefficient is a constant value depending on the contact materials. In further definitions and implementations, the friction coefficient between rubber and asphalt is assumed. It is a relative neutral value that is frequently used and it covers most cases.
All three constraints are convex constraints on the reaction forces x_i and the position of the center of mass c. The first and the second ones ensure the force and torque balance. The third one ensures, that all reaction forces lie inside the friction cone defined by the contact points. After denoting c into c_1, c_2, $c_3 \in \mathbb{R}$ it can be shown that

$$c \times mg = m\|g\| \begin{bmatrix} -c_2 \\ c_1 \\ 0 \end{bmatrix} \tag{14}$$

the first and the second constraint only depend on the horizontal position of the center of mass. This position is defined as

$$y = Pc \qquad \text{where} \quad P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{is a selection matrix} \tag{15}$$

Now for any $z \in \mathbb{R}^3$ with coordinates z_1, z_2, $z_3 \in \mathbb{R}$ the skew symmetric matrix T can be defined as

$$T(z) = \begin{bmatrix} 0 & -z_3 & z_2 \\ z_3 & 0 & -z_1 \\ -z_2 & z_1 & 0 \end{bmatrix} \tag{16}$$

so that $T(z)x = z \times x$ for all $x \in \mathbb{R}^3$. So the first and the second equilibrium constraints become

$$\sum_{i=1}^{n} x_i + mg = 0 \tag{17}$$

$$\sum_{i=1}^{n} T(r_i)x_i - T(mg)P^T y = 0 \tag{18}$$

with

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}, \quad y = Pc \quad A_1 = \begin{bmatrix} I & \dots & I \\ T(r_1) & \dots & T(r_N) \end{bmatrix}, \quad A_2 = \begin{bmatrix} 0 \\ -T(mg)P^T \end{bmatrix}, \quad t = \begin{bmatrix} -mg \\ 0 \end{bmatrix} \tag{19}$$

$$B = \text{diag}(I - \eta_1\eta_1^T, \dots, I - \eta_N\eta_N^T), \quad u = \begin{bmatrix} \mu_1\eta_1 \\ \vdots \\ \mu_N\eta_N \end{bmatrix} \tag{20}$$

the equilibrium constraints can be rewritten as the two conditions

$$A_1 x + A_2 y = t \tag{21}$$

$$\|Bx\| \leq u^T x \tag{22}$$

These conditions define the set of feasible reaction forces x and positions of the center of mass y at fixed contact positions.

$$\chi = \{y \in \mathbb{R}^2 | A_1 x + A_2 y = t, \|Bx\| \leq u^T x\} \tag{23}$$

Now the support region is defined with all feasible positions of the center of mass of the system. That is the projection of χ onto the y-textitspace.

$$\Upsilon = \{y \in \mathbb{R}^2 | \exists x \in \mathbb{R}^{3n} \text{such that} (x, y) \in \chi\} \tag{24}$$

The support region can be represented as the projection of a nonlinear convex set, bounded by linear and second-order cone conditions onto a 2D linear subspace. Now the computation of the shape of this projection is shown.
The shape of the feasible support region is approximated with a so-called cutting plane algorithm. An advantage for the computation is that the support region is a convex set that can be approximated with an inner polygon.

$$\Upsilon_{inner} = \{y \in \mathbb{R}^2 | a_i^T y \leq b_i \text{ for all } i = 1, \dots, N\} \tag{25}$$

Assuming that Υ is bounded and has non empty interior, then for any $\epsilon > 0$, an a_i and b_i can be found for $i = 1, \dots, N$ such that

$$\Upsilon_{inner} \subseteq \Upsilon \quad \text{and area}(\Upsilon) - \text{area}(\Upsilon_{inner}) \leq \epsilon \tag{26}$$

The parameter ϵ specifies a desired bound on the error of the approximation. By decreasing ϵ the polygon can be approximated to any desired precision.
In the beginning, an initialization of the inner polygon has to be found. This work assumes that the support

region lies within the support polygon defined by the contact feet. This assumption can be made because the kinematic of the robot does not allow support regions outside the support polygon. For the initialization, a sequence of points on the bound Υ is generated around the current center of mass. The amount of points is related to the number of vertices of the support polygon, defined by the contact points and they are uniformly distributed around the center of mass projection. Each point $v \in \mathbb{R}^2$, extremal in some direction $a \in \mathbb{R}^2$, is found by solving a second-order cone program of the form

$$\begin{aligned} \underset{z}{maximize} \quad & a^T z \\ subject\,to \quad & A_1 x + A_2 y = t \\ & \|Bx\| \leq u^T x \end{aligned} \tag{27}$$

and taking $v = z_{opt}$. The direction a is generated uniformly distributed around the center of mass. Depending on the number of vertices of the support polygon defined by the contact points m, m directions are generated. The circle around the center of mass is distributed into $2\pi/m$ direction bins. In every direction, a boundary point is searched. Every point v also defines a halfspace

$$\{z \in \mathbb{R}^2 | a^T z \leq a^T v\} \supseteq \Upsilon \tag{28}$$

So in addition to Υ_{inner} the algorithm constructs another polygon outside the support region $\Upsilon_{outer} \supseteq \Upsilon$ as an intersection of these halfspaces. After initialization of both approximation polygons is made, the approximation of the support region can start.
The algorithm is repeated until the difference between the area of the outer polygon and the area of the inner approximation is smaller than the desired error bound. In each iteration step, all edges of the inner polygon are investigated. The edge which would cut the biggest area of the outer polygon is taken as the next start point for the optimization. The optimization direction is given through the normal of the edge to the outside of the polygon. With one step of the optimization problem, the next point on the bound of Υ is found. The point is added as a vertex to the inner polygon. The outer polygon is redefined through the new halfspace. The inner polygon builds the approximation of the support region.
The planning of the floating base works similar to the planning of footholds. A discrete reachability polygon is used to discretize the reachable space around the floating base. The heuristic takes every grid cell in the polygon and checks if it lies in the support region for the given foot contacts. Every grid cell which lies outside the polygon is discarded from the discretization and gets the heuristical value zero. Points within the polygon have a heuristical value of one and positions on the boundary of the polygon are marked as a "*bad*" choice, which means a heuristical value of 0.5. The heuristical grid values are summarized and divided with the amount of all grids. The result is the "*how hard*" heuristic value for the floating base with respect to the support region.
This is the basic variant of the support region heuristic. The set of feasible positions of the center of mass can be extended as desired with arbitrary conditions. In the following section, an approach based on the torque limits of the robot joints is presented. The basic structure of the method and the approximation algorithm are still the same, but the optimization problem has to be redefined for the new constraints.

4.4 Improved floating base support region

In the following section, a refinement of the approach from above is presented. To have a clean summary of the different definitions of support regions, they are listed shortly.

- **Support Polygon**: The simplest approach to model the feasible region for the floating base. Support polygons are defined by the contact points of the system and are built from the convex hull of them.

- **Feasible Support Region**: Approximated refined approach of the support polygon based on the reaction forces and the friction values of the system. Defined by the second-order cone program from above.
- **Improved Feasible Support Region**: Refined approach to the feasible support region. The optimization problem is extended with a condition that checks the torque limits in the joints.

In the following approach, the second-order cone program from above is extended with a condition, checking on the torque limits of the leg joints. The approach is based on the assumption, that the torques in the joints are dynamically changed with a dynamic repositioning of the floating base. If the floating base position diverges from the feet center more and more the torques in at least one joint could exceed their limits. With the extended second-order cone program, this region should be subtracted from the feasible support region approximation.
The approach is based on the technique of so-called wrench ellipsoids. Wrench ellipsoids can be obtained as a consequence of the kinetic energy theorem. It says that the work done by all forces acting on a particle is equal to the rate of change of the particle's kinetic energy, which leads to

$$\tau = J(q)^T w \tag{29}$$

This represents the static relationship between the generalized task-space wrenches $w \in \mathbb{R}^m$ and the generalized joint-space forces $\tau \in \mathbb{R}^n$. $J \in \mathbb{R}^{m \times n}$ is the jacobian matrix of the end-effector of a mechanical chain. If a unit hypersphere S_τ in joint-space is considered as

$$S_\tau = \{\tau \in \mathbb{R}^n | \tau^T \tau \leq 1\} \tag{30}$$

it can be obtained an ellipsoid as a new set ϵ_w that describes how S_τ is mapped into the task-space:

$$\epsilon_w = \{w \in \mathbb{R}^m | w^T J J^T w \leq 1\} \tag{31}$$

By definition, the ellipsoid ϵ_w is a pre-image of the unit hypersphere S_τ, under the mapping $J(q)^T$. The singular values of the Moore-Penrose pseudoinverse of $J(q)$ correspond to the semiaxes of ϵ_w and the ratio between the greatest and the smallest eigenvalue is used as a measurement of the anisotropy of the ellipsoid and the force amplification of the mechanical chain.
In a similar way, these facts can be adopted to analyze the pre-image of the joint-torque hypercube ζ_τ, the set of all joint-torques τ within the actuator's limits τ^{lim}:

$$\zeta_\tau = \{\tau \in \mathbb{R}^n | \tau^{min} \leq \tau \leq \tau^{max}\} \tag{32}$$

The joint-torque limits represent the hardware-dependent limits of the torques of the actuator's joints. The task-space wrench polytope P_w, pre-image of ζ_τ, can be written as follows:

$$P_w = \{w \in \mathbb{R}^m | \tau^{min} \leq J(q)^T w \leq \tau^{max}\} \tag{33}$$

While the force ellipsoid ϵ_w can be used as a qualitative property to measure the robot's force amplification capabilities, the wrench polytope P_w also includes the information about the amplitude of wrench that the robot can perform at the end-effector.
Originally wrench polytopes and wrench ellipsoids were introduced for system with mechanical chains with $m = n$, where m is the dimension of the end-effectors force and n is the number of actuated joints in the chain. In such a case the Jacobian $J(q$ would be a square matrix and its transpose could be inverted except for singular configuration, to achieve the vertex representation of the wrench polytope P_w:

$$w^{lim} = J(q)^{-T} \tau^{lim} \tag{34}$$

where $\tau^{lim} \in \mathbb{R}^n$ is a vertex of ζ_τ and $w^{lim} \in \mathbb{R}^m$ is a vertex of P_w.
In the next step, it is shown how the floating base motion can influence the joint torques in single configurations and how it is representable that some configurations exceed the limits of the joint-torques. It is started with the computation of dynamic wrench polytopes because all other states can be derived from this case. For this work, the static wrench polytopes are sufficient because in the planning phase no robot motion is happening. The dynamic wrench polytope describes the set of feasible contact wrenches that a floating base robot can perform at its contact points while moving. At the beginning, the motion equation of a floating base system is considered as

$$M(q)\dot{s} + C(q,s) + g(q) = S\tau + T(q)^T f \tag{35}$$

where $q = [q_b^T q_j^t] \in SE(3) \times \mathbb{R}^n$ is the vector of the system's coordinates composed from the floating base position $q_b \in SE(3)$ and the coordinates $q_j \in \mathbb{R}^n$ describing the positions of the n actuated joints. The system velocities are described with the vector $s = [\nu_b^T \dot{q}_j^T] \in \mathbb{R}^{6+n}$ and $\tau \in \mathbb{R}^n$ is the vector of actuated joint-torques.$C(q,s)$ and $g(q) \in \mathbb{R}^{6+n}$ are the Coriolis and the gravity terms and $M(q) \in \mathbb{R}^{(6+n)\times(6+n)}$ is the joint-space inertia matrix of the system, while $S \in \mathbb{R}^{(6+n)\times n}$ is the selector matrix whose rows corresponding to the actuated joints are set to ones during the others are set to zero. The contact forces are described with the vector $f \in \mathbb{R}^{mn_c}$ which is mapped into joint torques with the stacked jacobians in $T(q) \in \mathbb{R}^{mn_c \times (6+n)}$. If the equation is splitted up into a floating base part and a joint part it looks like it follows:

$$\begin{bmatrix} M_b & M_{bj} \\ M_{bj}^T & M_j \end{bmatrix} \begin{bmatrix} \dot{\nu}_b \\ \ddot{q}_j \end{bmatrix} + \begin{bmatrix} c_b \\ c_j \end{bmatrix} + \begin{bmatrix} g_b \\ g_j \end{bmatrix} = \begin{bmatrix} 0_{6\times n} \\ I_{n\times n} \end{bmatrix} \tau + \begin{bmatrix} J_b^T \\ J_q^T \end{bmatrix} f \tag{36}$$

The joint Jacobians $J_q \in \mathbb{R}^{(mn_c)\times n}$ are the bottom n rows of $T(q)$ and they can map joint-torques into contact forces for each limb individually:

$$J_q = \text{diag}(J(q_1), J(q_2), \ldots, J(q_{n_c})) \tag{37}$$

where $J(q_i)$ are the Jacobians of the limbs in contact with the ground. With these basics it is possible to define a dynamic wrench polytope for each limb of the robot which is in contact with the environment:

$$A_i = \{f_i \in \mathbb{R}^m | \exists \tau_i \in \mathbb{R}^{n_l} \text{s.t.} M_{bi}^T \dot{\nu} + M_i \ddot{q}_i + c(q_i, \dot{q}_i) + g(q_i) = \tau_i + J(q_i)^T f_i, \quad -\tau^{lim} \leq \tau_i \leq \tau^{lim}\} \tag{38}$$

where $i = 1, \ldots, n_c$ is the index of the contact points and n_c is the amount of active contacts between the system and the ground. The vectors $q_i \in \mathbb{R}^{n_l}$ and $\tau_i \in \mathbb{R}^{n_l}$ are the joint positions and torques in joint-space of the i^{th} limb, while n_l is the number of actuated degrees of freedom of this limb.
As a final consideration, it can be observed that in a static behaviour ($\dot{q} = 0, \ddot{q} = 0$) the definition of the polytope can be rewritten to

$$A_i = \{f_i \in \mathbb{R}^m | \exists \tau_i \in \mathbb{R}^{n_l} \text{s.t.} g(q_i) = \tau_i + J(q_i)^T f_i, \quad \tau^{min} \leq \tau_i \leq \tau^{max}\} \tag{39}$$

The term $g(q_i)$ is the effect of gravity acting on the limb i. With respect to the planning issue the static wrench polytope is a sufficient model because during the planning phase no motion effects occur.
To extend the optimization problem with the possibility to check on the torque limits of the joints the definition of the static wrench polytope is rewritten as

$$A_i = \{f_i \in \mathbb{R}^m | \exists \tau_i \in \mathbb{R}^{n_l} \text{s.t.} g(q_i) - \tau_i = J(q_i)^T f_i, \quad \tau^{min} \leq \tau_i \leq \tau^{max}\} \tag{40}$$

and considering the joint torque τ_i is limited through its minimum and maximum boundary, the equation can be reformulated as follows to highlight the dependency on the torque limits:

$$A_i = \{f_i \in \mathbb{R}^m | g(q_i) + \tau^{min} \leq J(q_i)^T f_i \leq g(q_i) + \tau^{max}\} \tag{41}$$

For further simplification the term $g(q_i) + \tau^{min}$ is shortcutted with d_i^{min} and the term $g(q_i) + \tau^{max}$ with d_i^{max}. Now the matrix G and the vector d can be defined to extend the optimization block from above. The matrix $G \in \mathbb{R}^{d\times()mn_c}$ is built from the concatenation of all jacobian of the actuated system and the vector d is defined through the concatenation of all torque limits of the joints combined with the gravity effect.

$$G = \text{diag}(\begin{bmatrix} J(q_1)^T \\ -J(q_1)^T \end{bmatrix}, \ldots, \begin{bmatrix} J(q_{n_c})^T \\ -J(q_{n_c})^T \end{bmatrix}) \tag{42}$$

$$d = [d_1^T ... d_{n_c}^T] \in \mathbb{R}^{2n_c n_l} \tag{43}$$

G and d can now be used to extend the definition of the feasible support region and the optimization problem which is used for the cutting plane algorithm.

$$\chi = \{y \in \mathbb{R}^2 | A_1 x + A_2 y = t, \|Bx\| \leq u^T x, Gx \leq d\} \tag{44}$$

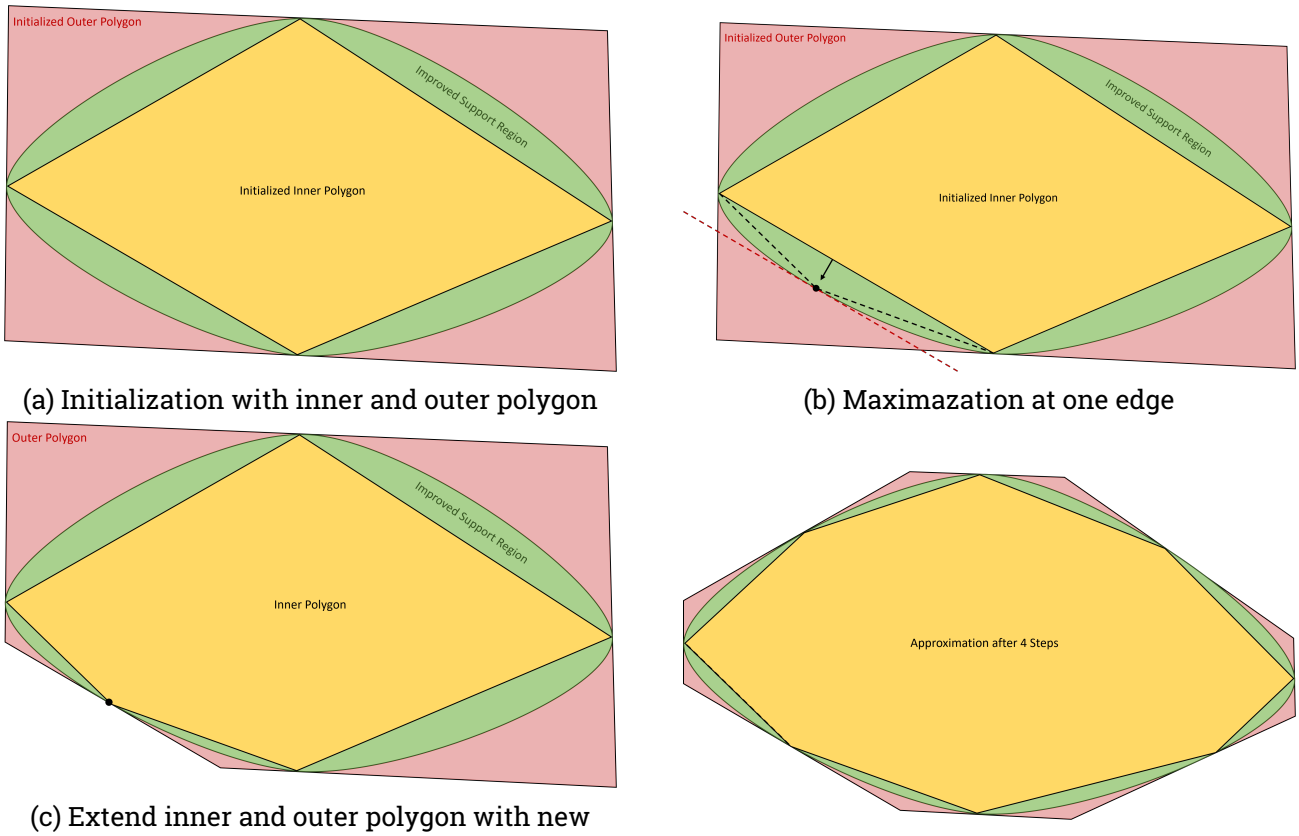
where x is the reaction force instead of f. The 2D projection now looks like

$$\Upsilon = \{y \in \mathbb{R}^2 | \exists x \in \mathbb{R}^{3n} \text{such that} (x, y) \in \chi\} \tag{45}$$

and the resulting optimization problem for the approximation algorithm is defined through

$$\begin{aligned} \underset{z}{maximize} \quad & a^T z \\ subject\,to \quad & A_1 x + A_2 y = t \\ & \|Bx\| \leq u^T x \\ & Gx \leq d \end{aligned} \tag{46}$$

Introducing the third inequality corresponds to an intersection of the friction cones with the wrench polytopes of the system. The reaction forces which lie in this intersection satisfy the friction conditions and the torque limits of the actuating leg.



(a) Initialization with inner and outer polygon

(b) Maximazation at one edge

(c) Extend inner and outer polygon with new point

(d) Planning path across stepping blocks

Figure 20: Approximation result after four steps

Now the algorithm continues like above with the cutting plane algorithm shown in figure 20, to approximate the feasible region and the heuristic checks if given floating base positions lie inside the improved feasible support region. Especially in rough terrain, where the robot has to climb stairs or rocks, this can be useful because the legs can end up in configurations where the torques exceed their limits.

4.5 Combination for simultaneous planning

Finally, four incoherent heuristic approaches has been proposed in this work. Two heuristics for the footstep planning and two heuristics for the floating base planning. In this section, it is explained how these heuristic approaches work together to realize the first approach for simultaneous footstep and floating base planning. The original problem was that the discrete footstep planner cannot avoid collisions with the environment, unstable robot postures or non-desired joint-states in robot limbs expect the feet. An A-Star planner expands in every step every successor state from the current state, regardless if a state is accessible for the individual task or whether it makes heuristically no sense to visit that state. The main contribution of this work is to investigate heuristics that decide if such a state should not be expanded and also how to prioritize state expansion.
Looking at the standard pipeline of a footstep planner shown in figure 15, the heuristics are needed twice. Once at the gait generation, which has to find the limb that has the fewest following states and then later in the pipeline where a state is checked for its reachability. Both areas of operation have small differences. In the gait generation step has to be computed how many successor states of a current state are pruned. In the reachability check, all heuristical bad states have to be pruned from the discretization so that the planner

does not expand these states.
At first, a gait generator decides which foot or whether the base limb is planned next. In this work, the gait generator makes its decision based on the results of all used hardest first heuristics, by calling all of them and combining the results to a single hardest first heuristic value. The combination is computed by the mean of all results. The limb with the highest heuristical value is passed to the state generator. Now the state generation would expand every following state of a given foot or a base, no matter the hardest first heuristic would prune one of them. The same heuristics used in the gait generation step, have to be used here too, to handle this. This time the state generator iterates through the discretization of the following states and computes the heuristical value of each state. If the heuristical value of a state exceeds a configurable threshold the state is discarded by the state generation. The hardest first logic has to be the same as in the step before to ensure consistent behaviour.
The threshold is given as a dynamic parameter to the system and is use case dependent. The higher the threshold is chosen, the fewer states are pruned. The probability that a path can be found increases, but the system could generate infeasible plans. The lower the threshold, the more conservative the system acts. In extremely rough terrains it could happen that the planner would not find a path if the system is too cautious, but a found path is likely more feasible to execute.

5 Implementation

It is necessary to implement all the functionalities on a robot system, to evaluate the single methods and the combination of them. In the following chapter, all the implementations based on the approaches of this work are described. It is possible to use all the methods on different robot systems, due to the modularity of the used L3 framework.

5.1 Terrain Model

The locomotion task of a legged robot is addicted to the environment terrain of the robot system, so it is necessary to have a precise model of the terrain around the robot. The pipeline of terrain recognition and modelling starts with visual sensors like a laser scanner or a camera system. The sensors provide a point cloud of the environment and the terrain model generator has to generate an environment model out of the data, the robot can deal with.
In this work, the basic terrain model generator from the L3 library is used. The L3 terrain model generator saves the point cloud in a database to store it for other applications. On the other side, it generates a multilayer grid map, using the grid map implementation from ETH Zurich [29]. It provides a conversion method for point clouds to grid maps. In the standard framework, the generated grid map has four layers. The elevation layer stores the height values and the three normal layers store the x, y and z values for the surface normals.
The L3 terrain model generator is built as a plugin pipeline, so after a point cloud subscription, every implemented plugin is called to process the point cloud. So it is easily possible for the user to implement additional plugins to generate their own layers in the grid map or manipulate the values in existing layers. After the plugin pipeline is finished, the L3 terrain model generator publishes the processed grid map to provide it to the whole robot system. In the following section, the extensions needed for this work are shown and described in detail.

5.1.1 Grid map generation from image files

It is recommended not to start with real environment data, to test the raw functionality and the correctness of the algorithms. So the first tests should be done on synthetic and clean data without environmental inaccuracies or sensor noises. The most accurate data can be provided by discrete graphics, so a possibility to generate grid maps from grayscaled images is needed.
The grid map implementation of ETH Zurich [29] provides such a conversion method, which converges images to grid maps with a height layer. The grayscale value of the pixel describes the height value, where white is the maximum height and black is the minimum height. The conversion algorithm expects three configuration values. The height boundaries to interpolate between black and white and the resolution to compute the size of the map. The map has the resolution value r, so that

$$l_x = n_x * r \tag{47}$$

$$l_y = n_y * r \tag{48}$$

$$h_i = \frac{h_{max} - h_{min}}{255} * g_i + h_{min} \text{ with } g_i \in [0, 255] \tag{49}$$

where $l_{x/y}$ is the length of the map edge in meters, $n_{x/y}$ the number of pixels at the map border. h_i is the height at the pixel i and g the grayscale value at pixel i. The whole processing is implemented as a sensor plugin, so from the grayscaled image a gridmap is generated with the given conversion method. Another method converts the grid map to a point cloud with the height values. In the last step the virtual sensor provides the point cloud to the terrain model processing pipeline, so it is processed like a point cloud from any sensor, including the normals generation.

5.1.2 Automatic map resizing

The terrain model is not built in one step, but it is created iteratively. The robot records its environment continuously with its sensors. In regular time intervalls, the sensors send these records in form of point clouds to the terrain model generator. In the terrain model pipeline, the point cloud is processed and each point of the cloud is copied to the grid map in the model. Some points are updated because new values overwrite the old ones. Some points are completely new and fill an empty grid cell. The whole grid map generation is an additive application and the map is continuously updated by the model generator.
The resulting map grows in its size with every update, due to this iterative process. One solution is to initialize the grid map with a size which will never bee exceeded. This approach is not recommended because the map is not dynamically extentable and the initial map would constist of many unused grid cells, so a dynamic resizing algorithm is needed. In the first iteration, the grid map has no size. With every received point cloud the boundaries of the cloud are calculated and the new size of the grid map can be computed. The dynamic resize method is only called when the new boundaries will result in an enlargement of the map. If the map becomes smaller, already recorded information can be lost. With the dynamic resizing, the map allocates only the memory which is really needed by the map. This solution improves the performance especially in the following applications where iterations over the whole map are necessary.

5.1.3 Variable grid map filtering

Most heuristics in this work can be realized with a cost map and they are directly related to the robot's environment. This means that every grid in the map has a heuristic value for the foot or the base to reach. The heuristic value or the cost value can be computed based on the neighbourhood of the grid cell. It is affected by obstacles or the terrain roughness. For the heuristics which are dependent on the terrain, it is obvious to realize a new grid map layer with the cost values. The only issue for the heuristic method is then to use a lookup in the cost map to calculate the heuristic value of a given grid cell in the map.
The grid map library of the ETH Zurich [29] provides a filter framework based on the ROS filter framework. With this framework, it is easily possible to generate new map layers with the filtered content of another map layer. The filter framework is built as a plugin pipeline, known from the terrain model pipeline. In a central config file, all filters are listed with their input and output layers and different parameters. Each filter is a single plugin with an initialization step and an update step. In the initialization, the parameters of the plugin are loaded. In the update step, the specified filter logic is provided, where the new content of the output layer is computed. The whole pipeline works as a so-called filter chain. When the chain is initialized, it calls the initialization method of every listed filter plugin. If the filters are needed, for example when new map information from the sensors are subscribed, the update method of the chain calls the update for each plugin. With the filter chain and the single filter plugins, it is easy to implement and test new filters. The only limitation is the performance. The filter chain always calls the updates for the whole map, but in one sensor iteration only single points in the map are updated and many grid cells are computed redundant. This application is, especially for tall maps, not applicable in real-time issues.
In the terrain model pipeline explained above, there are multiple terrain model plugins. These plugins have

the task to process the map before it is published to the rest of the system. The pipeline is triggered with every point cloud update from the sensors, and only the updated points are really transferred. The solution now is to implement the grid map filters from above for single points directly in the plugin pipeline of the terrain model generator. The plugins have access to the current grid map so they can compute all filters based on the last known neighbourhood of one grid cell and they can write it to the related layer in the current map. Both solutions are implemented in this work. The first approach provides an easy possibility to design new filters and debug them. But for real-time applications, only the second approach is usable. For the targets of this work, seven filter plugins are implemented.

- The first filter uses the elevation layer as input and is a simple *smoothing filter*. It works with the mean over the neighbourhood of a cell and is just used as a preprocessing for other filterings.

  $$f_j = \frac{\sum_{i=0}^{n} r_i}{n} \tag{50}$$

  where f_j is the grid cell to compute in the map, r_i is on the height value in each neighbourhood cell and n is the amount of neighbourhood grid cells.

- The *slope filter* computes slopes in the environment, based on the normal z vectors.

  $$f_j = \arccos\left(a_j^z\right) \tag{51}$$

  where a_z describes the normal vector in z-direction.

- One of the most important filter logics is the *edge detection* to find all edges in the map. It works with the standard deviation over the slope layer. The steeper the slope is the stronger is the resulting edge.

- Another useful filter based on the *edge detection* is the *binary edge filter*, which just creates a binary depiction of all edges, over a given threshold.

  $$f_j = \begin{cases} 1 & r_i > t \\ 0 & r_i \leq t \end{cases} \tag{52}$$

  The threshold is described by t and r_j is the raw edge value in the grid map cell.

- The *obstacle distancing filter* is a directly heuristic logistic. It generates a cost margin around obstacles, based on their height. The *edge detection* and the *obstacle distancing* are explained more detailed in the related heuristics section.

- The *roughness* generally describes the difference between the elevation and the smoothed elevation map. This is just a preprocessing for the *traversability filter*.

  $$f_j = h_j - \bar{h_j} \tag{53}$$

  where h_j is the height at the grid cell j, and $\bar{h_j}$ is the mean of the height value.

- This filter describes the traversability in a given environment. The traversability is defined by the sum of weighted roughness and slopes. It is a value that says how uneven a terrain region is.

  $$f_j = 0.5 * (1.0 - (s_j * \omega_s)) + 0.5 * (1.0 - (r_j * \omega_r)) \tag{54}$$

  where s_j is the slope value at cell j and r_j is the roughness value at cell j. The ω's are the weights for the slope and roughness part.

Each filter is implemented as a single plugin and is called in the terrain model pipeline in the order as above. The order of the filters is important because the computed result values are based on each other. In the plugin config file, the call order of the plugins can be configured and the plugins can be enabled or disabled on runtime.

5.2 Floating Base Planning

In this work, the Footstep Planner from the L3 Library is used. The L3 Footstep Planner provides a generic plugin-based pipeline to generate a footstep plan by using a discrete graph planning A-Star algorithm. The pipeline includes the discretization of the environment to a discrete graph, heuristics for the planning algorithm, reachability conditions for the feet and a state generation stage to expand all possible states. The L3 basic library provides all the data structures which are important for the planning issue, like steps, footholds and planning states. Details about the L3 Footstep Planning can be found in the related work chapter 3 or in [18].
In this work, the simultaneous floating base planning beside the planning of footsteps is added to the L3 framework. Various adaptions have to be made to the existing platform, to achieve this goal. All the data structures specifically made only for footsteps must be extended with a floating base model. Fortunately, another thesis [42] works on the floating base planning parallel to this work, so the basic data structures for a floating base and the integration to the system was already done. Only the adaptations for the planning task has to be made. In the following subsections, this integration of the floating base to the system parts is shown.

5.2.1 L3 Floating Base Compatibility

One data structure which is needed to be changed is the result of the state generation. At the beginning of the planning pipeline, a gait generator decides which feet can be planned in the next step. This information is passed to the state generator, which expands all states for each given foot. These states are saved in a specified data structure, the `StateGenResult`. In the old version, the `StateGenResult` contains only a list of expanded footholds. In order to expand also floating base states a possibility to save an expanded floating base in the state generator result is added.

```
struct StateGenResult
{
  // Contains a list of feet and one floating base for the state result
  FootholdPtrArray footholds;
  FloatingBase::ConstPtr floating_base;
}
```

Listing 1: StateGenResult Structure

The gait generator has passed the planning patterns to the state generator as a list of lists of foot indices, a `MultiFootIndexArray`. Each array contains a pattern to expand. With this solution, the state generator never would expand states including the floating base, because the information is simply missing in the pattern. So a data structure is added, `ExpandStateIdx`, which holds all necessary information to generate a feet pattern including a floating base. The `ExpandStateIdx` is a structure that holds a list of foot indices and floating base indices, so a pattern can also include the floating base. The `ExpandStateIdx` provides additionally the possibility to plan a list of floating base indices in the case of a multi-base system, but a multi-base planning approach is not part of this work.

```
struct ExpandStateIdx
{
  // Provides a list for feet and for a multi base system
  FootIndexArray foot_idx;
  BaseIndexArray floating_base_idx;
}
```

Listing 2: ExpandStateIdx Structure

Another important data structure in the footstep planner framework is the footstep. A footstep represents all possibilities for a foothold to move to, based on a given discretization. An analog basestep class is provided, because a floating base can have additional requirements to its motion options.
Additionally, to the functional backend of the floating base planning, the visualization has to be extended. The planning framework provides a full visualization part for the RViz tool. Every step in the planning pipeline creates feedback that is published. The visualization stage subscribes to this feedback and with the feedback information, it generates a visualization of the planning process. The feedback contains the start and the goal configuration, all expanded states, all visited states and the final step plan. For each data structure a visualization method is provided, which creates a related RViz marker, so after the planner finished the whole step plan is visualized in RViz as a sequence of markers. The main use case of the visualization tool is the debugging issue and to have this possibility for the floating base implementation, a visualization plugin for the floating base is implemented. In addition to the footholds, the feedback now contains the final position of the floating base and the expanded and visited floating base states. The visualization tool is extended with a method to create a sequence of markers for the floating base position. Like the foothold visualization, the marker form and color can be configured by the user in a given configuration file.

5.2.2 Polygonal Discretization

The L3 Footstep Planner library is a search-based path planner. Search-based planning is a motion planning method that uses graph search methods to compute paths over a discrete representation of the planning problem, so it is necessary to convert the continuous configuration-space into a discrete state-space. In the standard solution, provided by the L3 Footstep Planner, a discrete reachability polygon is defined around each foot to represent the reachable space in a discrete environment. These polygons can be configured by the user, depending to the used robot system.
The polygons are defined by their vertices related to the foot origin in a two-dimensional plane. Additionally, an orientation interval is given for each position in the polygon. With the global planning resolution the discretization of the polygon is computed, so for every foot, a finite set of possible states exists.
The same discretization step is needed for the floating base, so the discretization is extended with a floating base reachability polygon. The difference to the foot discretization is, that the floating base needs a three dimensional discretization, with respect to the pitch orientation, so in addition to the yaw orientation of the positions a pitch value is added. In the course of this work, the whole logic around the reachability polygons is extended with complete three-dimensionality. The feet polygons are further defined in the two-dimensional space, but the floating base can be configured as a six degrees of freedom system and the system automatically creates a reachability polytope instead of a two-dimensional polynomial.

5.2.3 Hardest First Heuristic Plugins

As explained in the foundation section 2 the L3 Footstep Planner is built as a pipeline that divides the planning issues into different single problems. All these tasks are implemented as one or multiple plugins and the planner schedules these plugins in a predefined order to solve the complete planning problem. Some plugins

are so-called singleton plugins, some plugins can exist more than once. If a plugin is unique the plugin method can simply be called in the planning pipeline. For plugins that can exist in multiple forms, there are related aggregator classes. The aggregator has the task to manage multiple plugins of one class and combining all their results. As an example, there can be different heuristic plugins. Each heuristic plugin has a heuristic method that computes a heuristic value and each plugin has a weight, which declares the importance of the plugin. The aggregator calls each heuristic plugin and summarizes all weighted results normed by the amount of heuristics to the resulting heuristic value. The plugin pipeline has only to call the aggregator instead of every single plugin.
In this work, a new plugin type with a related aggregator is defined and implemented, to realize the hardest first planning approach from section 4. The base type of the plugin is the `HFSHeuristicPlugin` and it is called in the planning pipeline by the *Gait Generator*. As explained above the *Gait Generator* generates patterns in which feet or bases have to be expanded to the state generator. The hardest first approach uses heuristics to find the hardest foot or the hardest base to plan it next. Like the normal planning heuristic, there can be more than one hardest first heuristic, so a plugin aggregator is implemented which combines all the plugins results.

```
class HFSHeuristicPlugin : public virtual FootstepPlanningPlugin
{
  HFSHeuristicPlugin(const std::string& name);

  bool isUnique() const final { return false; }

  virtual bool getFeetHeuristic(const l3::FootIndex idx, Step::ConstPtr step, double& cost);

  virtual bool getBaseHeuristic(const l3::BaseIndex idx, Step::ConstPtr step, double& cost);
}
```

Listing 3: Hardest First Plugin Type

The hardest first heuristic consists of two main methods. The `getFeetHeuristic(...)` computes the heuristic value of a given foot and the `getBaseHeuristic(...)` computes the value of a given base. In both methods, the value is calculated related to the given current step. Even if in this work only single base planning is handled, the interfaces and plugin types are designed for multi-base planning. Each plugin has a constructor where its name is initialized and the method *isUnique* declares whether there can be multiple plugins of this type that can be loaded. In this work, four hardest first implementations are made. A polygonal-based base heuristic class is implemented, due to all approaches are based on the polygonal discretization. All implementations are shown in detail in the following list.

- **Polygonal HFS Heuristic:** The *Polygonal HFS Heuristic* inherits directly from the *HFS Heuristic Plugin*. The only method in this plugin is the loading method of the parameters, which are necessary for all the heuristics. It is the base plugin of three of the presented algorithms. The *Foot Distance HFS Heuristic*, the *Foot Traversability HFS Heuristic* and the *Base Orientation HFS Heuristic* are based on polygonal reachability.
  The base plugin is primarily responsible for the discretization of the environment around the feet or the base in form of reachability polygons. For the whole planning application, the same polygons are used as for the state generator and the reachability processing.
  Another important instance is the subscriber for the multilayer grid map. All heuristics use the grid map to calculate their heuristic values, so it makes sense to implement the subscriber and the global map variable in the base class of the plugins.

- **Foot Distance HFS Heuristic:** This plugin represents the first heuristic for the feet, based on the distance of the feet to close obstacles. Only the method `getFeetHeuristic(...)` is implemented. The whole method depends on the environment so at first a check is made, whether the terrain model is available or not. If it is not available a default value for the heuristic value is returned. Here it is the inverse index of the foot, normed by the number of feet. This is just a method to guarantee unique values for the feet and to get a deterministic order of planned feet.
  If the terrain model is available in form of a grid map, the plugin iterates through the reachability polygon of the given step and calculates for each polygon cell the position in the grid map. The actual logic of the distance heuristic is implemented in the grid map filter, which generates a cost map, based on the distance of cells to obstacles and the height of the obstacles. The plugin only has to make a look up in the grid map on the related layer for every position in the reachability polygon. The cell contains a cost value between zero and one. A cost value of one means a bad step position and zero stands for full traversability. If the cell contains a high value an obstacle is close. This does not mean, that the robot wants to cross this obstacle, in some cases, it only passes the obstacle on one side. In this case, the global planning direction is important. The algorithm computes the motion direction and a direction vector for the obstacle. The cost value of a cell is only taken into account if the robot system plans directly to the obstacle. If the obstacle lies beside the path, so the robot only passes the obstacle the cell cost value is discarded.
  The heuristic calculates the mean of all cells as the heuristic value for the foot. The higher the value the harder the foot is to plan.

- **Foot Traversability HFS Heuristic:** The traversability heuristic is analogue to the distance heuristic implementation. The only difference is the layer of the grid map where the cost values are looked up.

- **Base Orientation HFS Heuristic:** The *Base Orientation HFS Heuristic* is the last of the three polygonal hardest first approaches and this plugin uses the introduced reachability polygon of the floating base. The implementation is based on the pitch adaption approach in section 4. The plugin overwrites to methods, the `loadParams(...)` method, which loads all needed parameters from the parameter server and the `getBaseHeuristic(...)` method, which calculates the heuristic value of a given base index. As mentioned above the algorithm is based on the pitch adaption approach. This approach tries to adapt the pitch orientation of the floating base in parallel to the terrain. An additional parameter which is loaded in this implementation is the inclination deviation factor. This factor indicates how much the orientation deviates from parallelism.
  Like in the other polygonal heuristics at first the availability of the terrain model is checked. Then the algorithm iterates over the whole reachability polygon and computes for each cell in the polygon the position in the grid map. From each cell, the closest obstacle is searched. Here only obstacles with a minimum height are counted as obstacles. The minimum height is another configurable parameter. When an obstacle is found in front of the robot the algorithm investigates the edges of the obstacles and calculates the edge normal vectors to check whether the obstacle is in the planning direction of the robot. If this is the case the terrain ground is approximated with a slope from the projected polygon cell of the floating base to the closest point on the obstacle edge. The pitch of this approximated slope should be the new pitch of the floating base, to simplify the crossing of the obstacle.
  This routine is made for each cell in the polygon and the cells with a pitch adaption needed are counted. In the end the heuristic of the base is calculated by the amount of cells which need a pitch adaption normed by the amount of all cells.

- **Base Stability HFS Heuristic:** The last heuristic plugin is the only one that is not based on a polygonal discretization. This implementation is based on the stability margins of the robot. The plugin improves

the support region of the floating base based on different stability criteria. These criteria can be dynamically enabled and disabled. This provides an easy way to evaluate all single criteria and the user can decide itself which criteria are needed and which are not. The criteria are iteratively dependent on each other, so they have to be enabled in a descending order. How the single criteria work or the whole pipeline of the approach is running can be seen above in the method section under the construction of the improved feasible region. The beginning of the computation is the support polygon defined by the feet in contact with the ground. The support polygon is built by the convex hull of all feet which are in contact with the ground. The ground is approximated as a flat plane to define a polygon on it.
The support polygon of the contact feet is computed with an algorithm based on the counterclockwise order of all contact points. At first, the algorithm searches for the most bottom left point of all contact positions. Next, all points are stored in a list, sorted in counterclockwise order beginning at the bottom left start point. Now the algorithm iterates in counterclockwise direction through all the points and discards every point, lying inside the polygon. This is done by computing the orientation of the point with respect to the start point and the next point in the counterclockwise order. After discarding all inner points the algorithm checks if there are still three points. If not there is no convex support polygon. If there are three or more points, the algorithm returns the points in a list, sorted counterclockwise beginning with the bottom left start point.

```
vector<Point> computeConvexSupportPolygon(FootholdArray feet)
{
  Point bottom_left = min({feet.x, feet.y});

  sortCounterClockWise(bottom_left, feet);

  vector<Point> result;
  for(Point p in feet)
  {
    if(orientation(bottom_left, p, p.next) == inside)
      continue;

    result.push_back(p);
  }

  if(result.size() >= 3)
    return result;

  return null;
}
```

Listing 4: Compute convex Support Polygon(Pseudocode

After computing the support polygon of the feet, the inner and the outer polygon of the feasible support region have to be initialized. It is always possible to build a convex polygon with exactly four edges around the region, due to the convex constraints. The convex set of the feasible region is given by a second-order cone optimization problem as shown in chapter 4. The algorithm generates four sequential points on the boundary of the feasible region by solving the optimization problem in four different directions, to generate an initial inner polygon. The directions are generated with $\frac{\pi}{4}$ steps around the current floating base position, to achieve a uniform distribution of the four points on the boundary. For the robot system, evaluated in this work, this is a valid assumption, because the floating base and the feasible region always lie in the contact support polygon. The four points on the boundary are the four vertices of the initial inner polygon in the feasible region, they are connected with four edges. The outer polygon is built with four tangents, defined by the feasible region and the inner poylgon vertices. The

intersections of the tangents are the vertices and the line segments between the vertices are the edges of the outer polygon.

```
ConvexPolygon initializeInnerPolygon(Step step)
{
  vector<Point> initial_points;
  for(i = 0; i < 4; i++)
        initial_points.push_back(solve_socp(i * pi/4, step.feet, step.base);

  vector<Edge> initial_edges;
  for(Point p in initial_points)
    initial_edges.push_back(Edge(p, p.next));

  return ConvexPolygon(initial_points, initial_edges);
}
```

Listing 5: Initialization inner Polygon(Pseudocode)

When the initialization of the approximation of the feasible support region is done, the cutting plane algorithm starts to approximate it. In a while-loop is checked if the difference of the area of inner and outer polygon becomes smaller than a threshold ϵ. If this is not the case, the algorithm iterates through the inner edges to find the edge that cuts the biggest area of the outer polygon. The cut algorithm, explained in the section for helper classes is used to cut the outer polygon with an edge. The cut algorithm is an in place algorithm which cuts a polygon with a given line and a cutting direction. The cutting direction is a normal vector of the cutting line and specifies which side of the original polygon is discarded and which one is the result of the cut algorithm. The method cuts the outer polygon with every inner edge and a cutting direction which points outside the inner polygon, to find the right edge. All resulting polygons are compared with respect to their area. The edge with results in the smallest polygon is the right edge for the optimization because it cuts the biggest area of the outer polygon.
When the right inner edge is found the second-order cone program is started, beginning on the edge center with the edge's normal outside the inner polygon as the optimization direction. The result of the optimization problem is a new point on the boundary of the feasible support region. The new point is added to the inner polygon, the algorithm to add a point to a polygon is presented in the section for the helper classes. Additionally, a parallel line to the old inner edge is created which intersects the boundary point of the feasible region. With this line, the outer polygon is cut. This is repeated until the difference of the inner and the outer polygon area is less than the parameter ϵ. The inner polygon is used as the approximation of the feasible region.

5.2.4 State Generator Plugins

The main step in the planning pipeline is the state generation. In this step, the continuous environment of the robot is discretized into a set of discrete states for the planning algorithm. For this work, the whole discretization is realized with a polygonal approach. Around each foothold, a reachability polygon is defined which depends on the physical possibilities of the robot. With a discrete resolution, this polygon is discretized into a set of cells that can be reached by the foothold. The complete reachable set of new states is implemented in the `Footstep` class. A footstep represents all possible foot placements related to the robot center for one foot. In the section above the *Basestep* class is presented which is the analoge implementation for the floating base of the robot.
The state generation step of the planning pipeline is realized with a footstep planning plugin called the *State Generator*. The state generator iterates through all possible discrete following states of a foothold

and permutates every possibility of each foot. The result is organized in a list of `StateGenResult` structures which are explained above. In the standard state generator of the L3 footstep planning, only the foothold discretization is noted. So for this work, the state generator is extended with the state expansion of floating base states.

```
class PolygonalBaseStateGenerator : public PolygonalStateGenerator
{
protected:
  typedef std::map<BaseIndex, std::vector<Basestep>> BasestepSetMap;

public:
  PolygonalBaseStateGenerator();

  bool loadParams(const ParameterSet& params) override;

  std::list<StateGenResult> generatePredStateResults(const PlanningState& state, const State
      & start, const State& goal, const ExpandStatesIdx& state_expansion_idx) const override;
  std::list<StateGenResult> generateSuccStateResults(const PlanningState& state, const State
      & start, const State& goal, const ExpandStatesIdx& state_expansion_idx) const override;

protected:
  std::list<StateGenResult> generatePredFloatingBases(const PlanningState& state, const
      ExpandStatesIdx& state_expansion_idx, const BasestepSetMap& basesteps) const;
  std::list<StateGenResult> generateSuccFloatingBases(const PlanningState& state, const
      ExpandStatesIdx& state_expansion_idx, const BasestepSetMap& basesteps) const;

  BasestepSetMap floating_base_set_;
};
```

Listing 6: Extended Polygonal State Generator

The `generateStateResults` method is the main method in the state generation. For both planning strategies, backward and forward planning, an interface method is provided. In this work, only the forward planning is shown, but the implementation for the backward planning is analogous to the forward planning strategy. So here only the methods `generateSuccStateResults` and `generateSuccFloatingBases` are explained.

```
std::list<StateGenResult> PolygonalBaseStateGenerator::generateSuccStateResults(const
    PlanningState& state, const State& start, const State& goal, const ExpandStatesIdx&
    state_expansion_idx) const
{
  std::list<StateGenResult> feet_result = PolygonalStateGenerator::generateSuccStateResults(
      state, start, goal, state_expansion_idx);
  std::list<StateGenResult> base_results = generateSuccFloatingBases(state,
      state_expansion_idx, floating_base_set_);

  std::list<StateGenResult> result;
  for (StateGenResult fr : feet_result)
  {
    if(base_results.size() > 0)
    {
      for (StateGenResult br : base_results)
      {
        StateGenResult tmp(fr.footholds, br.floating_base);
        result.push_back(tmp);
      }
    } else {
      StateGenResult tmp(fr.footholds);
```

```
      result.push_back(tmp);
    }
  }
  return result;
}
```

Listing 7: Generation of succeeding States

The extended method `generateSuccStateResults` gets a pattern with all feet which can be expanded in this state and retrieves at first all possible states for the feet from the parent method. Then all possible placements for the floating base are computed with the new method `generateSuccFloatingBases`. For the complete result, both partial results are permuted with each other.

```
std::list<StateGenResult> PolygonalBaseStateGenerator::generateSuccFloatingBases(const
    PlanningState& state, const ExpandStatesIdx& state_expansion_idx, const BasestepSetMap&
    basesteps) const
{
  std::list<StateGenResult> result;
  for (const BaseIndex& idx : state_expansion_idx.floating_base_idx)
  {
    BasestepSetMap::const_iterator itr = basesteps.find(idx);
    if (itr == basesteps.end())
      continue;

    FloatingBasePtrArray floating_bases;
    for (const Basestep& basestep : itr->second)
      floating_bases.push_back(basestep.getSuccFloatingBase(state.getState()));

    if (result.empty())
    {
      for (FloatingBase::Ptr floating_base : floating_bases)
        result.push_back(StateGenResult(FootholdPtrArray(), floating_base));
      break;
    }
  }
  return result;
}
```

Listing 8: Generation of succeeding Floating Bases

The method `generateSuccFloatingBases` iterates through all floating bases in the planning pattern from above and generates a result for each combination of possible floating base states. In this work, the possibility of multi-base planning is not dealed with, so here the list of results will only contain the possibilities for one floating base. This list of all possible following states is passed to the planning algorithm which searches for the best following state with diverse heuristics.

5.2.5 Reachability Plugin

Before a position can be marked as a possible placement for a foot or a floating base it has to be ensured, that the position is reachable. In the standard solution, the state generator iterates through the cells of the reachability polygon and asks the reachability plugin if it is possible to reach this cell position for the robot. Due to the standard L3 reachability plugin and the standard L3 state generator using the same discretization polygon every position which is asked by the state generator, is reachable.
In this work, the hardest first approach computes for every cell position how safe it is to reach a single

position based on the presented heuristics. It is important that this information is not only used in the pattern generation, but also in the reachability step. The reason for this is simple. The hardest first pattern generation generates a pattern based on the information, how hard a foot is to plan. The fewer possibilities a foot has for its placement, the harder it is. So it can happen that the hardest first approach prefers a foot, but the heuristical discarded positions are not known by the reachability plugin and the state generator can expand all the discarded placements for this foot.
The original L3 reachability plugin is extended with a heuristical reachability approach, to handle this issue. For each hardest first heuristic a reachability plugin derived from the original one is implemented where the plugin interface is extended by two methods.

```
class HeuristicalPolygonalReachability : public ReachabilityPlugin
{
public:
  HeuristicalPolygonalReachability(const std::string& name = std::string());

  bool loadParams(const ParameterSet& params) override;

  bool isReachable(const PlanningState& state) const override;
  bool isReachable(const State& state) const override;

protected:
  bool isFootReachable(const FootIndex& foot_idx, double dx, double dy, double dyaw) const;

  bool isBaseReachable(const BaseIndex& base_idx, double dx, double dy, double dyaw, double
      dpitch) const;

  virtual bool isFootHeuristicalReachable(const double& x, const double& y, const double&
      yaw) const {return true;}

  virtual bool isBaseHeuristicalReachable(const double& x, const double& y, const double&
      yaw, const double& pitch) const {return true;}

  void mapCallback(const grid_map_msgs::GridMap& map_msg);

  double heuristical_threshold_;
};
```

Listing 9: Heuristical Reachability Base Class

Here only the method for the feet is explained due to the method for the floating base works analogously. The method `isFootReachable` just checks the reachability based on the given discretization. The method `isFootHeuristicalReachable` checks whether the placement is reachable based on the current heuristic. For each hardest first heuristic, there is one plugin that implements this interface. If it is a heuristic for the floating base it overrides the method `isBaseHeuristicalReachable` and if it is a foot heuristic then it overrides the `isFootHeuristicalReachable` method. The used method takes the position of the queried placement and checks the heuristical value on this position. With a given parameter, the heuristical threshold the plugin decides whether the position is reachable or not. Due to the most heuristics being implemented as grid map layers the new reachability plugin has a callback method to subscribe to the current grid map of the environment. For this work, the following plugins are implemented.

- `BaseOrientationHFSReachability`
- `BaseStabilityHFSReachability`
- `FootDistanceHFSReachability`

- `FootTraversabilityHFSReachability`

The new state generator asks for each foot or floating base placement of all reachability plugin implementations. If at least one discards a position, the position is not in the set of possible placements of the next state.

5.3 Helper Classes

Two helper classes are introduced in the following section, to allow easy implementation of the approximation algorithm for the improved feasible support region. The first one represents the structure of a polygon edge, as a line segment. The other one is a convex polygon class which provides all necessary algorithm methods to work with a convex polygon.
An edge can be seen as a line segment and is defined by two $2D$ endpoints. The edge is implemented as a struct in the header files of the heuristics package and provides three constructors, an empty one, and two methods to create an edge with two endpoints. Furthermore, the struct has two global variables for the endpoints, which are $2D$ float vectors of the *Eigen* library. In the following the two endpoints are written as `Point` variables to simplify the explanation. Additionally, the struct provides the following methods:

- **double m()**: Computes the inclination of the line, which is defined by the two endpoints `p1` and `p2` with the formular
  `(p2.y()-p1.y())/(p2.x()-p1.x())`.
- **double b()**: Computes the axis section on the y-axis of the line, which is defined by the two endpoints `p1` and `p2` with the formular `p1.y()-(m()*p1.x())`.
- **bool equals(Edge other)**: Returns true if two edges have the same endpoints.
- **bool contains(Point p)**: Checks if a given point lies on the edge and between the two endpoints.
- **bool intersects(Edge other, Point intersection, bool in_extern)**: Checks if two edges have an intersection point. If an intersection is found, it is stored in the passed `Point` variable. If `in_extern` is set to `true`, it is furthermore checked if the intersection lies between the endpoints of the given edge. If the intersection lies outside the current edge, the intersection is set, but the method returns `false`.
- **Point getOpponentPoint(Point p)** Requires one of the endpoints of an edge and returns the other endpoint.
- **bool getMatchingEndpoint(Edge other, Point p, vector<Point> opps**: Checks if two edges are connected at two endpoints. If they are connected it returns `true` and the matching endpoint is stored in the passed `Point` variable. The remaining endpoints are stored in the passed `vector` variable.
- **Vector getNormal()**: Computes a unitized normal vector of the edge, the direction of the normal is random.

Another helper type is the struct which represents a convex polygon. A polygon is defined by a list of points and a list of edges. There is no order for the two lists, but there are some methods for the struct which change the order in place. For example after the `computeArea()` the list of points is sorted in counter clockwise order. Furthermore there is no information about the connection of the edges, but for the algorithms this is not necessary. The polygon struct has also three constructor methods, one for the empty polygon and two which initializes the polygon global variables. The global variables are a list of `Point` variables and a list of `Edge` variables like mentioned above.

- **void deletePoint(Point p)**: Deletes a points from the polygon and the two related edges. The algorithm creates a new edge which connect the two remaining points.
- **bool contains(Point p)**: Checks if a point lies inside the polygon. This algorithm uses the property of convex polygons. A random directed ray starting at the given point should have exactly on intersection with one of the edges of the polygon. If is has two or zero intersections it lies outside the polygon.
- **bool extendEdge(Point new_point, Point old_point**: If the new and the old point are different and lie on the same edge of a polygon, the old point is replaced by the new point. All related edges in the polygon are changed. Returns `false` if both given points don't share the same edge.
- **bool addVertex(Point p1, Point p2, Point new_point**: Adds a new point to the polygon and reconnects all related edges. The points `p1` and `p2` are the two neighbors of the new point.
- **bool cut(Edge cutter, Vector normal)**: Cuts a polygon with the cutter edge. The discarded part of the polygon is defined by the given normal vector of the cutter edge, it is directed into the discarded part. This algorithm is specified for the use case of the used cutting plane algorithm from above. At first all intersections of the cutter with polygon edges are computed. In the issue of this work, three cutting cases are possible. The first case is the normal one with two intersections. In this case the cutting edge intersects two edges anywhere between their endpoints, but not directly at the points. In this case two new points are created at the intersections and the cutter is the new connecting edge. The two cutted edges and their matching endpoint are deleted from the polygon.

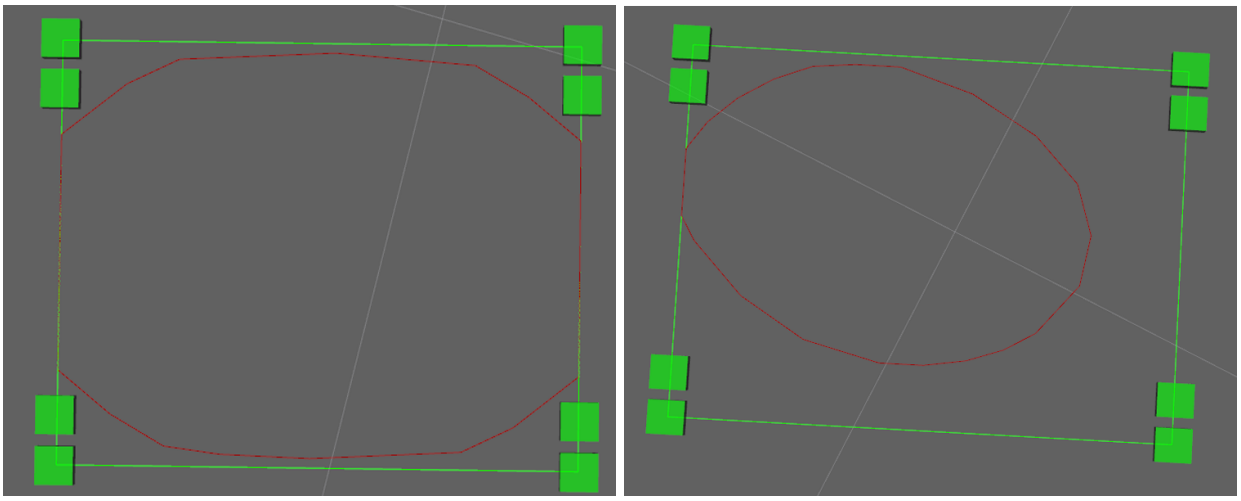
```
bool cut(Edge cutter, Vector normal)
{
  list<Point> intersections;
  for(Edge e : edges)
  {
    Point i;
    if(cutter.intersects(e, i))
      intersections.push_back(i);
  }

  switch(intersections.size())
  {
    case 2:
      edges.push_back(cutter);
      edges.delete(cutted_edge_1);
      edges.delete(cutted_edge_2);
      vertices.push_back(intersections(0));
      vertices.push_back(intersections(1));
      vertices.delete(outlying_point);
      return true;
    case 3:
      edges.push_back(cutter);
      edges.delete(cutted_edge);
      edges.delete(outlying_edge);
      vertices.push_back(intersection_with_cutted_edge));
      vertices.delete(outlying_point);
      return true;
    case 4:
      edges.push_back(cutter);
      edges.delete(outlying_edge_1);
      edges.delete(outlying_edge_2);
      vertices.delete(outlying_point);
      return true;
```

```
    default:
      return false
  }
}
```

Listing 10: Convex Polygon Cutting Algorithm (Pseudocode)

The second case is the first special case with three cutted edges. In this case, the cutting edge cuts one edge between its endpoints and two edges at their matching endpoint. The matching endpoint remains in the polygon and the cutting edge is a new edge. The intersection with the unique edge is a new point. All remaining related elements are deleted from the polygon. In the last case, the cutting edge has four intersections with the polygon, due to both intersections being at vertices of the polygon. Again the cutting edge is the new edge, only the outlying point and both outlying edges have to be deleted.



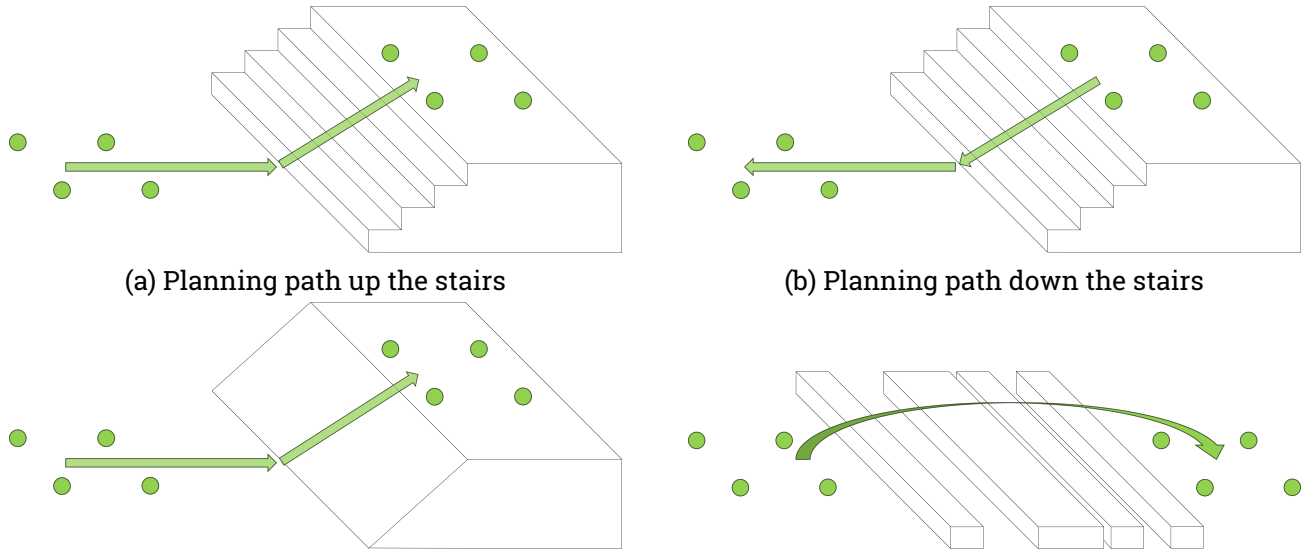
(a) Feasible Region on Flat Terrain (b) Feasible Region on a concave terrain

Figure 21: Approximations of feasible support regions in Simulation

- **double computeArea()**: Computes the area of a convex polygon with the sum of integrals method [43]. For this method, the list of vertices needs to be sorted in a counterclockwise order. The method does this in place with the list of vertices, so after computing the area of the polygon the list of points is sorted. Due to performance issues, the ordering is not changed after each operation with the polygon.

6 Evaluation

In this section, the evaluation of this work is presented. The evaluation should confirm the claims of this work and it should show how these features can target all the challenges, defined in the problem definition. At first, the procedure of the evaluation task is explained and justified. In the next four sections, the results of the experiments are shown, each section handles the results of one implemented approach. The approaches are evaluated isolated because each approach targets one part of the problem. In the last step of evaluation, different combinations of approaches are tested and evaluated. All choices of tested combinations is done in the context of the problem definition. For the evaluation, the following plan of experiments is designed.
Each heuristic is evaluated isolated using the identical planning setup. A default working set of planning parameters is taken and the amount of threads for multithreaded planning is set to four, which is the standard value. The *Gait Generator* which is used for the evaluation is the *HFS Gait Generator* from [42]. This *Gait Generator* works similar to the standard cyclic *Gait Generator*, but additionally, to the order of the cycles, the *HFS Value* is checked and influences the order of expanded limb states. The *HFS Gait Generator* always passes exactly one limb id to the *State Generator*, either one foot id or the floating base id.
The goal of the hardest first approach is to optimize the trajectory generation during passing rough and uneven terrain. To achieve this, the approaches are targetting to plan floating base positions in a way to create optimized whole body postures. The simultanous planning of footsteps and a floating base results in an exponential growth of the state-space. Another issue is, that the greedy behavior of the A-Star algorithm prefers floating base states close to the goal, which is not always the best choice, which must be counteracted by good heuristics. The evaluation is done in three phases, which are explained in the following bullet list.



(a) Planning path up the stairs

(b) Planning path down the stairs

(c) Planning path up a ramp

(d) Planning path across stepping blocks

Figure 22: Different evaluation scenarios for a staircase

- **Phase 1:** Ten pre-seeded random goals on a flat terrain without any obstacles. Evaluation metrics are the deviations of the mean of the expanded states, the visited states and the planning time between the planning without the heuristics and the heuristical planning.
- **Phase 2:** Ten pre-seeded random goals on a terrain with four different obstacle types. Evaluation metrics are the mean ratio of collisions with the environment, the ratio ob unstable postures and the ratio of non-desired joint-states.

- **Phase 3:** Ten pre-seeded random goals on a terrain with a staircase, captured by real sensor data. The evaluation metrics are the same as in phase two.

6.1 Foot Distance HFS Evaluation

Evaluation Phase One (Flat Terrain)		
Min Deviation expanded States	Max Deviation exp. States	Mean Deviation exp. States
4	18	9,7(1,1%)
Min Deviation visited States	Max Deviation visited States	Mean Deviation visited States
1979	8220	5233,1 (0,2%)
Min Deviation Time	Max Deviation Time	Mean Deviation Time
0,004	0,09	0,046 (1,6%)
Path found		✓

Table 1: Evaluation plan Foot Distance Heuristic Phase 1

Evaluation Phase Two (Synthetic Terrain)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	32%	19%	14%
HFS Planning	1%	17%	16%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	31%	13%	12%
HFS Planning	0%	14%	12%
Path found			✓

Ramp			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	0%	11%	9%
HFS Planning	0%	12%	8%
Path found			✓

Step Blocks			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	9%	11%
HFS Planning	1%	10%	8%
Path found			✓

Table 2: Evaluation plan Foot Distance Heuristic Phase 2

Evaluation Phase Three (Real Staircase)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	39%	21%	19%
HFS Planning	1%	22%	21%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	18%	11%
HFS Planning	1%	19%	10%
Path found			✓

Table 3: Evaluation plan Foot Distance Heuristic Phase 3

The first approach which is evaluated is the foot distance heuristic. To get a better relation to the implementation, the approaches are called by their class names in the following evaluation.
The `FootDistanceHFSHeuristic` is based on the idea that footholds should never be set too close to obstacles that will be crossed by the system. With placing footholds on the distance to the obstacle the possibility to move the foot over the obstacle in the next step is improved because rigid robot systems have a minimal motion radius of their legs. Another issue is that the planner never places the feet in narrow gaps which could prevent collisions in the following states. The results are shown in the tables above and are shortly analyzed in the following bucket list

- **Phase 1:** As shown in table 1 the evaluation shows, that the heuristic does not have a large impact to the planner's performance. This behaviour is repeated for each of the heuristics so in the next sections the results of phase one are not discussed any more.
- **Phase 2:** The `FootDistanceHFSHeuristic` is expected to avoid collisions in the planning. As shown in table 2, this can be confirmed. Additionally the approach has no impact to the remaining metrics.
- **Phase 3:** Also on real sensor data terrain the heuristic has an expected behaviour, as shown in table 3

6.2 Foot Traversability HFS Evaluation

Evaluation Phase One (Flat Terrain)		
Min Deviation expanded States	Max Deviation exp. States	Mean Deviation exp. States
2	30	12,2(1,3%)
Min Deviation visited States	Max Deviation visited States	Mean Deviation visited States
1344	7836	5404,3(0,3%)
Min Deviation Time	Max Deviation Time	Mean Deviation Time
0,008	0,139	0,055(1,6%)
Path found		✓

Table 4: Evaluation plan Foot Traversability Heuristic Phase 1

Evaluation Phase Two (Synthetic Terrain)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	32%	19%	14%
HFS Planning	32%	19%	14%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	31%	13%	12%
HFS Planning	31%	13%	12%
Path found			✓

Ramp			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	0%	11%	9%
HFS Planning	0%	11%	9%
Path found			✓

Step Blocks			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	9%	11%
HFS Planning	33%	9%	11%
Path found			✓

Table 5: Evaluation plan Foot Traversability Heuristic Phase 2

Evaluation Phase Three (Real Staircase)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	39%	21%	19%
HFS Planning	38%	14%	17%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	18%	11%
HFS Planning	33%	11%	7%
Path found			✓

Table 6: Evaluation plan Foot Traversability Heuristic Phase 3

The expected outcome was here, that the `FootTraversabilityHFSHeuristic` approach has an impact on the ratio of unstable robot postures. Also for this approach the analysis of phase 1 is discarded, because there is no decreasing performance of the planner, using this heuristic.

- **Phase 2:** As it can be seen in table 5, the approach has no impact to the results. This is reasoned by the perfect surface properties of synthetic terrain.
- **Phase 3:** The exptected behaviour from the `FootTraversabilityHFSHeuristic` is to reduce the ratio of unstable postures and non-desired joint-states. As shown in table 6, this can be confirmed.

6.3 Floating Base Orientation HFS Evaluation

Evaluation Phase One (Flat Terrain)		
Min Deviation expanded States	Max Deviation exp. States	Mean Deviation exp. States
1	26	8,5(0,9%)
Min Deviation visited States	Max Deviation visited States	Mean Deviation visited States
365	5107	4782,8(0,1%)
Min Deviation Time	Max Deviation Time	Mean Deviation Time
0,012	0,08	0,039(1,3%)
Path found		✓

Table 7: Evaluation plan Floating Base Orientation Heuristic Phase 1

Evaluation Phase Two (Synthetic Terrain)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	32%	19%	14%
HFS Planning	24%	16%	3%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	31%	13%	12%
HFS Planning	22%	11%	5%
Path found			✓

Ramp			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	0%	11%	9%
HFS Planning	0%	8%	4%
Path found			✓

Step Blocks			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	9%	11%
HFS Planning	26%	8%	4%
Path found			✓

Table 8: Evaluation plan Base Orientation Heuristic Phase 2

Evaluation Phase Three (Real Staircase)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	39%	21%	19%
HFS Planning	33%	18%	8%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	18%	11%
HFS Planning	28%	16%	3%
Path found			✓

Table 9: Evaluation plan Base Orientation Heuristic Phase 3

All climbing scenarios, where the robot has to overcome a height difference are targeted by the following approach. The `BaseOrientationHFSHeuristic` tries to detect obstacle edges in front of the system and approximate the inclination of the following motion with a plane between the current ground and the closest edge.

- **Phase 2:** The main goal of this approach is to reduce the ratio of non-desired joint-states. This goal can be confirmed with table 8 for all synthetic obstacle types. Furthermore the heuristic has a positive effect to the remaining evaluation metrics.
- **Phase 3:** The same behaviour is observable on real terrain data, as shown in table 9

6.4 Floating Base Stability HFS Evaluation

Evaluation Phase One (Flat Terrain)		
Min Deviation expanded States	Max Deviation exp. States	Mean Deviation exp. States
1	31	9,8(1,1%)
Min Deviation visited States	Max Deviation visited States	Mean Deviation visited States
681	8325	5620,9(0,4%)
Min Deviation Time	Max Deviation Time	Mean Deviation Time
0,006	0,089	0,038(1,3%)
Path found		✓

Table 10: Evaluation plan Floating Base Stability Heuristic Phase 1

Evaluation Phase Two (Simulated Terrain)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	32%	19%	14%
HFS Planning	33%	2%	12%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	31%	13%	12%
HFS Planning	30%	1%	8%
Path found			✓

Ramp			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	0%	11%	9%
HFS Planning	0%	0%	1%
Path found			✓

Step Blocks			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	9%	11%
HFS Planning	32%	2%	4%
Path found			✓

Table 11: Evaluation plan Base Stability Heuristic Phase 2

Evaluation Phase Three (Real Staircase)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	39%	21%	19%
HFS Planning	36%	3%	14%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	18%	11%
HFS Planning	35%	4%	7%
Path found			✓

Table 12: Evaluation plan Base Stability Heuristic Phase 3

In the last approach the floating base position is restricted with the feasible support region.
The `BaseStabilityHFSHeuristic` computes an area in which the floating base can be placed and the stability of the system is guaranteed.

- **Phase 2:** For the `BaseStabilityHFSHeuristic` it was expected that it has an impact to the ratio of unstable postures. In each scenario this can be confirmed, as shown in table 11 for all synthetic obstacle types. Furthermore the heuristic has a positive effect to the remaining evaluation metrics.
- **Phase 3:** As shown in table 12 the same can be said to the evalution on real terrain data.

6.5 Combination of all Heuristics

Evaluation Phase One (Simulated Terrain)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	32%	19%	14%
HFS Planning	1%	0%	1%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	31%	13%	12%
HFS Planning	0%	2%	2%
Path found			✓

Ramp			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	0%	11%	9%
HFS Planning	0%	1%	1%
Path found			✓

Step Blocks			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	9%	11%
HFS Planning	0%	2%	1%
Path found			✓

Table 13: Evaluation plan all Heuristic Phase 1

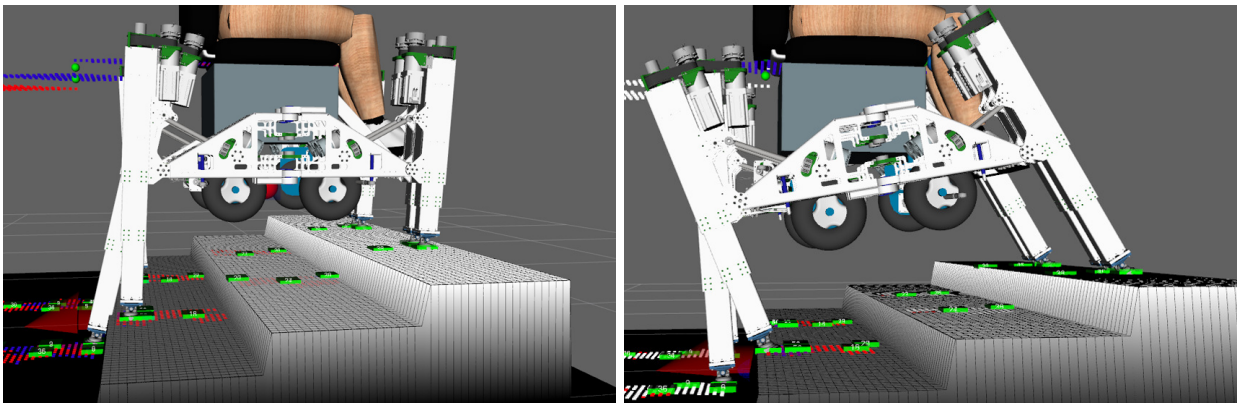
Evaluation Phase Two (Real Staircase)

Stairs upwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	39%	21%	19%
HFS Planning	1%	3%	1%
Path found			✓

Stairs downwards			
	Collision ratio	Unstable Postures Ratio	Non-desired Joint States Ratio
Standard Planning	33%	18%	11%
HFS Planning	1%	4%	3%
Path found			✓

Table 14: Evaluation plan all Heuristic Phase 2

In the last step of the evaluation, all heuristics are combined together. In the cases above it is shown that every heuristic has at least one positive effect on the planning, with respect to the related metric of each heuristic. Now it is expectable that all heuristics used together with a tuned set of parameters, achieve the best result with respect to the isolated tests. This expectation has been fulfilled, which is shown by the planning results in the tables above. In each case on every terrain, an improvement of all metrics can be observed.



(a) Collisions without the HFS Heuristics (b) No Collisions with HFS Heuristics

Figure 23: Example for Collision Avoidance with HFS Heuristics

Additionally, for each case, a path was found and the evaluation metrics for each test case could be improved massively in synthetic terrain and in the terrain built from real sensor data. In figure 23 the cooperation of all heuristics working together and how they avoid a collision is shown.
In the following all parameters of the approaches are presented and which parameter choice was taken for the evaluation results.

- **FootDistanceHFSHeuristic - Positive Obstacle Distance:** Describes the minimal distance which has to be held in front of positive obstacles. The optimal value depends on the kinematic structure of the robot legs and the motion radius of each foot. The tradeoff for this parameter is to cross an obstacle without collisions and to generally find a solution over the obstacle. For this evaluation, the best choice was a distance of $20cm$.
- **FootDistanceHFSHeuristic - Negative Obstacle Distance:** Analogue to the value above, with the difference that it describes a maximal distance to negative obstacles. The tradeoff is a security margin to edges and a minimum distance to find a solution down an obstacle. Here it is $1cm$.
- **FootTraversabilityHFSHeuristic - Traversability Threshold:** Each grid cell in a map has a traversability value, defined by the roughness and the slope value at this position. The value lies between one and zero, where one means fully traversable and zero means not traversable. The lower the threshold the more rough the ground can be to traverse it. After multiple test runs the best value was 0.7.
- **BaseOrientationHFSHeuristic - Observation Radius:** Describes the radius in which the algorithm searches for obstacle edges to approximate the base inclination. The bigger the radius is, the earlier the base pitch orientation is adapted. The base adaption is only necessary for the last step before the first leg has to cross an obstacle. This depends on the kinematic structure of the robot and the motion radius of the legs. Here it is $30cm$.
- **BaseStabilityHFSHeuristic - Approximation Area:** Describes the area difference of the outer and the inner approximation polygon. The smaller the value, the more precise the approximated support region, but also the computational effort increases. For the stability region of an eight-legged robot, a difference of $10cm^2$ is enough. With fewer contact feet a smaller difference is recommended.

All these parameters directly influence the planning behaviour in front or behind obstacles. They have to be tuned for each robot system individually and dependent on the field of operation. A robot, which acts directly with human beings should have more secure settings than a robot that only works in a harmless environment.

7 Conclusion

In this last section, a recap of this work is given and an outlook for future work is presented. The original problem that gives the reason for this work was missing whole-body knowledge of a discrete footstep planner in constrast to a whole-body configuration planner. With a sequence of whole-body configurations, collisions with the environment, unstable robot postures and non-desired joint state ocnfigurations could be avoided [1]. A discrete footstep planner runs into these issues due to it only considers a sequence of feasible footholds. Discrete footstep planning is a frequently used approach in robotic path planning, due to its low computational complexity [5][14]. In this work the L3 library [18] was used, which provides a discrete footstep planner and an integrated solution for terrain modelling. The existing library had to be extended with a simultaneous floating base planning, to integrate the approaches and evaluations of this work.
Four heuristical approaches were developed, to integrate a simultaneous floating base planning to the discrete footstep planner. All heuristics presented in section 4 contribute an improvement to footstep planning with respect to collision avoidance, avoidance of unstable postures and the avoidance of non-desired joint states. The first heuristic is focused on the placement of footholds, based on the distance to obstacles to be crossed. The innovation of this approach lies in the distinction of the behaviour in front of negative and positive obstacles. The second heuristic is based on irregularities in the terrain. The combination of the slope and the roughness value [12] of a position, enables the computation of a feasible traversability value on inclinated terrain, such as traversable ramps.
Additionally to the foot heuristics, also heuristical approaches for the floating base were developed in this work. The first approach for the floating base is based on the observation, that an inclined upper-body, related to the terrain, results in more natural postures. The innovated inclination pre-computation, based on the obstacle detection, enables an proactive adaption of the upper-body orientation before the obstacle is crossed. This approach can avoid collisions with the terrain, before a trajectory generation runs into them. The last presented heuristic computes a feasible support region for floating base placements, based on the friction at contact points and the torque limits in the leg joints. In addition the heuristic always place the floating base in the center of the support region in order to avoid unstable postures and non-desired joint state results. A generalized version of the proposed approach determines the support region based on the force limits, that can be applied to robots using linear joints in the legs, so that the heuristic is applicable on multiple types of robots.
All these approaches were evaluated on an eight-legged robot in simulation. For the evaluation synthetic and real terrain data were used to show whether the expected outcomes were reached. For each heuristic, the evaluation shows, that the expected improvements of the planning behaviour were achieved. Also the combination of all heuristics was evaluated with the same results, so the heuristics do not influence each other in a undesired way. The next section gives a brief overview how future work can leverage the presented results.

Future Work

For the future, there are several possibilities and tasks which can be done to extend this work. The most valuable point is the approximation of a whole-body configuration with the robot base. In this work, it is assumed that the base is a central link of the robot system which can be seen as a point mass of the upper body. To refine the whole-body configuration, the approaches could be extended with respect to multi-base planning. This means that more than a single upper-body link of the robot is taken into account to achieve a more detailed behaviour of the upper-robot body.
Single contributions of this work can be extended. The foot distance heuristic and the base pitch orientation heuristic could be extended with a direction detection improving the purposefulness of the algorithms. Cur-

rently, the approaches investigate all obstacles in range for the sensors. It would make sense to only check the obstacles which lie in the current planning direction. On one side this would reduce the computational cost of the algorithms, on the other side the robot can pass obstacles sideways, without heuristical restrictions which are not necessary.
Another extension possibility is the evaluation for additional robot systems. In this work, only the presented eight-legged robot is used for evaluation. All approaches are designed for a generic application on different legged robot systems. To refine the evaluation results it could be possible to rerun all tests with a bipedal or quadrupedal robot system. Additionally, further evaluation scenarios could be added. For example, the ramp or the step stones, captured with real sensor data.

List of Figures

List of Tables

Listings

References

[1] M. Gienger M. Toussaint and C. Goerick. *Whole-body Motion Planning – Building Blocks for Intelligent Systems*. 2001.

[2] Maurice Fallon Scott Kuindersma Sisir Karumanchi Matthew Antone. *An Architecture for Online Affordance-based Perception and Whole-body Planning*. 2013.

[3] Antonio El Khoury Florent Lamiraux Sébastien Dalibard. *Dynamic walking and whole-body motion planning for humanoid robots: an integrated approach*. 2013.

[4] Hongkai Dai Andrés Valenzuela Russ Tedrake. *Whole-body motion planning with centroidal dynamics and full kinematics*. 2014.

[5] Kiril Solovey. *Complexity of Planning*. 2020.

[6] Steven M. LaValle. *Planning Algorithms*. 2006.

[7] Jingjin Yu Steven M. LaValle. *Optimal Multi-Robot Path Planning on Graphs: Structure and Computational Complexity*. 2015.

[8] Ming C. Lin Dinesh Manocha Job Cohen Stefan Gottschalk. *Collision Detection: Algorithms and Applications*. 1996.

[9] Huijoan Wang Yuan Yu Quanbo Yuan. *Application of Dijkstra algorithm in robot path-planning*. 2011.

[10] Nathan Ratcliff. *Controlling Floating-Based Robots*. 2014.

[11] Robotis. *Efficient Locomotion Planning for a Humanoid Robot with Whole-Body Collision Avoidance Guided by Footsteps and Centroidal Sway Motion*. 2018.

[12] O. Villareal V. Barasuol M. Camurri L. Franceschi M. Focchi D. G. Caldwell M. Pontil C. Semini. *Fast and Continous Foothold Adaption for Dynamic Locomotion through CNNs*. 2018.

[13] R. Orsolino M. Focchi S. Caron G. Raiola V. Barasuol D. G. Caldwell C. Semini. *Feasible Region: an Actuation-Aware Extension of the Support Region*. 2020.

[14] O.Khatib K.Yokoi O.Brock. *Robots in Human Environments: Basic Autonomous Capabilities*. 1999.

[15] Jose A. Tenreiro Machado Manuel F. Silva. *An Overview of Legged Robots*. 2006.

[16] Jose A. Tenreiro Machado Manuel F. Silva. *Stair-climbing gait for a four-wheeled vehicle*. 2020.

[17] Marc Raibert Kevin Blankespoor Gabriel Nelson Rob Playter. *BigDog the Rough-Terrain Quadruped Robot*. 2008.

[18] Alexander Stumpf. *An Integrated Concept for Footstep Planning and Navigation for Different Types of Multi-Legged Robots in Challenging Environments*. 2020.

[19] Hiroshi Imai Takao Asano. *Efficient Algorithms for Geometric Graph Search Problems*. 1986.

[20] Sasa Klampfer Joze Mohorko Zarko Cucej. *Graph's theory approach for searching the shortest routing path in RIP protocol: A case study*. 2000.

[21] Daniel Jallov. *Graph Algorithms for AI in Games*. 2017.

[22] Yun Lin Yu Sun. *Robot grasp planning based on demonstrated grasp strategies*. 2014.

[23] Akshay Kumar Guruji Himansh Agarwal D.K. Parsediya. *Time-Efficient A* Algorithm for Robot Path Planning*. 2016.

[24] Roland Geraerts. *A comparative study of probabilistic roadmap planners*. 2004.

[25] Shiliang Sun Rongqing Huang. *An adaptive k-nearest neighbor algorithm*. 2010.

[26] Ming Liu. *Robotic Online Path Planning on Point Cloud*. 2016.

[27] J.Jessup S.N.Givigi A.Beaulieu. *Robust and efficient multi-robot 3D mapping with octree based occupancy grids*. 2014.

[28] Wolfram Burgard Maren Bennewitz Diego Tipaldi Luciano Spinello. *Techniques for 3D Mapping*.

[29] Peter Frankhauser Marco Hutter. *A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation*. 2016.

[30] Ulrich Schwesinger Roland Siegwart Paul Furgale. *Fast collision detection through bounding volume hierarchies in workspace-time space for sampling-based motion planners*.

[31] Frederick Reif. *Understanding Basic Mechanics*. 1995.

[32] Miguel Sousa Lobo Lieben Vandenberghe Stephen Boyd Hervé Lebret. *Applications of second-order cone programming*. 1998.

[33] Konstantinos Paparrizos Nikolaos Samaras George Stephanides. *A new efficient primal dual simplex algorithm*. 2003.

[34] Floran A. Potra Stephen J. Wright. *Interior Point Methods*. 2000.

[35] Tonneau et al. *An Efficient Acyclic Contact Planner for Multiped Robots*. 2018.

[36] Tonneau et. al. *A reachability-based planner for sequences of acyclic contacts in cluttered environments*. 2015.

[37] Ferrolho et al. *Whole-Body End-Pose Planning for Legged Robots on Inclined Support Surfaces in Complex Environments*. 2018.

[38] Yang et. al. *Humanoid motion planning with realtime end-pose selection in complex environments*. 2016.

[39] Perrin et. al. *Fast Humanoid Robot Collision-Free Footstep Planning Using Swept Volume Approximations*. 2012.

[40] Bretl et. al. *Testing Static Equilibrium for Legged Robots*. 2008.

[41] Felix Sternkopf. *First Steps with the L3 Framework: An Agile Framework for Humanoid Walking*. 2020.

[42] Filip Bjelonic. *Efficient Floating Base Optimization during Footstep Planning*. 2021.

[43] Peter Bender. *Einfache Anwendung zur Berechnung des Flächeninhaltes von Polygonen*.