Real-Time Navigation for Bipedal Robots in Dynamic Environments

Thesis

by Octavian Alexandru Donca

Advisor: Dr. Ayonga Hereid

The Ohio State University

Department of Computer Science and Engineering

2022

Thesis Committee:

Ayonga Hereid, advisor

Wei-Lun Chao

Presented in Partial Fulfillment of the Requirements for the Degree Bachelor of Science in Computer Science and Engineering with Honors Research Distinction in Mechanical Engineering of The Ohio State University

ABSTRACT

The popularity of mobile robots has been steadily growing, with these robots being increasingly utilized to execute tasks previously completed by human workers. Bipedal robots, a subset of mobile robots, have been a popular field of research due to the large range of tasks for which they can be utilized. For bipedal robots to see a similarly successful integration into society, robust autonomous navigation systems need to be designed. These autonomous navigation systems can generally be divided into three components: perception, planning, and control. A holistic navigation system for bipedal robots must successfully integrate all three components of the autonomous navigation problem to enable robust real-world navigation. Many works expand on fundamental planning algorithms such as A*, RRT, and PRM to address the unique problems of bipedal motion planning. However, many of these works lack several components required for autonomous navigation systems such as real-time perception, mapping, and localization processes. Thus, the goal of this research is to develop a real-time navigation system for bipedal robots in dynamic environments which addresses all components of the navigation problem. To achieve this: a depth-based sensor suite was used for obstacle segmentation, mapping, and localization. Additionally, a two-stage planning system generates collision-free and kinematically feasible trajectories robust to unknown and dynamic environments. Finally, the Digit bipedal robot's default low-level controller is used to execute these feasible trajectories. The navigation system was first validated on a differential drive robot in simulation. Next, the system was adapted for bipedal robots and validated in hardware on the Digit bipedal robot. In both simulation and in hardware experiments, the implemented navigation system facilitated successful navigation in unknown environments and in environments with both static and dynamic obstacles.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Ayonga Hereid, for taking me in as an undergraduate student and introducing me to the world of robotics research.

Table of Contents

ABSTRACT . . . . **ii**

ACKNOWLEDGEMENTS . . . . **iii**

LIST OF FIGURES . . . . **vii**

LIST OF ALGORITHMS . . . . **viii**

1 INTRODUCTION **1**

1.1 Related Works 2

1.1.1 Path Planning 3

1.1.2 Bipedal Robot Motion Planning 4

2 HARDWARE AND SIMULATION **7**

2.1 Hardware 7

2.1.1 Digit 7

2.1.2 Depth Cameras 7

2.1.3 LiDAR Sensor 8

2.2 Simulation 9

2.2.1 Gazebo 9

2.2.2 Turtlebot3 9

3 SYSTEM ARCHITECTURE **10**

3.1 Robot Operating System 10

3.2 Move Base Flex . . . 11
3.2.1 Move Base Flex Actions . . . 12
3.2.2 Move Base Flex Behavior Tree . . . 12

4 PERCEPTION, LOCALIZATION, AND MAPPING . . . 14
4.1 Perception . . . 14
4.2 Point Cloud Pre-Processing . . . 15
4.2.1 Voxel Grid Filtering . . . 15
4.2.2 Pass-Through Filtering . . . 16
4.3 Obstacle Segmentation . . . 16
4.4 Localization . . . 17
4.4.1 Feature Point Extraction . . . 19
4.4.2 Feature Point Correspondence . . . 19
4.4.3 Motion Estimation . . . 20
4.4.4 LiDAR Mapping . . . 22
4.5 Mapping . . . 23
4.5.1 Global Map . . . 23
4.5.2 Local Map . . . 25

5 REAL-TIME MOTION PLANNING . . . 26
5.1 D* Lite Global Planner . . . 26
5.2 Timed-Elastic-Band Local Planner . . . 32

6 RESULTS . . . 37
6.1 Simulation Experiments . . . 37
6.1.1 Static Obstacles, Unknown Environment . . . 38
6.1.2 Dynamic Obstacles, Unknown Environment . . . 39
6.1.3 Dynamic Obstacles, Large Unknown Environment . . . 40
6.2 Hardware Experiments . . . 41
6.2.1 Static environment . . . 42

6.2.2 Dynamic environment 42

7 CONCLUSION 44

BIBLIOGRAPHY 50

List of Figures

2.1 Digit joint and sensor configuration 8

3.1 Navigation system architecture overview. 11

4.1 Depth camera point cloud pre-processing. 16
4.2 LOAM system overview. 18
4.3 Global and local mapping results. 24

6.1 Static simulation experiment results. 38
6.2 Dynamic simulation experiment results. 39
6.3 Long distance navigation simulation experiment results. 40
6.4 Static hardware experiment results. 41
6.5 Dynamic hardware experiment results. 42

List of Algorithms

1 LOAM motion estimation algorithm [41] 21
2 Optimized D* Lite initialization algorithm [20] 28
3 Optimized D* Lite shortest path generation algorithm [20] 29
4 Optimized D* Lite shortest path execution algorithm [20] 31

CHAPTER 1

INTRODUCTION

Bipedal robots have been a popular field of research, due to the large range of tasks they can be utilized in. The general humanoid shape of bipedal robots should allow them to be better integrated into a society designed for humans. However, for bipedal robots to be fully integrated into society, robust autonomous navigation systems need to be designed. These systems can generally be divided into three stages: perception, planning, and control, where it is only with the combination of these three stages into a single navigation system in which bipedal robots can truly integrate into human society.

Thus, autonomous bipedal navigation requires the implementation of a holistic navigation system to enable the exploration of unknown and dynamic environments. This navigation system must combine the three components of autonomous navigation – perception, planning, and control – into a single navigation system. A robust perception system is required that is capable of extracting useful obstacle and environment structure information. This environment information must then be used to generate useable representations of the global and local environment for planning. Using the captured environment information, a planning system must generate global paths and local trajectories that are robust to unknown environments and dynamic obstacles. Furthermore, planning must respect the kinematic constraints of the robot while avoiding obstacles to ensure safe and feasible navigation. Finally, these trajectories must be executed with a low-level controller that maintains safe and stable walking gaits. The combination of these capabilities will enable the safe navigation of bipedal robots in complex environments.

This thesis focuses on the implementation of a real-time navigation system for the Digit bipedal robot with the goal of achieving robust autonomous navigation in unknown and dynamic environments. The system utilizes two Depth cameras and a LiDAR sensor for perception. These sensors are used to map the environment using global and local cost maps which capture large-scale environment structure and local dynamic obstacles. During navigation, odometry is calculated for robot localization using LiDAR Odometry and Mapping to enable accurate, long distance operation. Using the estimated odometry, planning within the global and local maps is achieved through a two-stage planner which generates collision-free global paths and obstacle avoiding local trajectories. In particular, a D* Lite global planner, which is capable of fast re-planning, is used to generate high-level paths in the global costmap. Next, a Timed-Elastic-Band local planner then follows the global path through optimization-based trajectory generation that respects kinematic, velocity, acceleration, and obstacle avoidance constraints. This local trajectory is executed by generating a sequence of velocity commands sent to Digit's walking controller. The combination of these components addresses the three parts of the autonomous navigation problem, and culminates in a real-time navigation system which can navigate bipedal robots through unknown and dynamic environments.

1.1 Related Works

Over time, many works have expanded on the path planning methods of A* [4, 15, 14, 10, 16], Rapidly-exploring Random Tree (RRT) [5, 1, 30], and Probabilistic Roadmap (PRM) [34] to address the unique problems of bipedal motion planning and footstep planning. However, on their own, these motion planning approaches lack several components required to create a holistic autonomous navigation system such as real-time perception, mapping, and localization processes. Furthermore, only few works expand further to integrate these bipedal motion planning methods into robust bipedal navigation systems. However, many of these approaches still are unable to address components of the autonomous navigation problem required to create holistic autonomous bipedal systems such as on-robot perception systems [28], localization methods [13], robustness to dynamic obstacles [31, 23], or validation in hardware. Before beginning to introduce the navigation system described in this thesis, some previous works in the field of path planning, motion planning, and bipedal navigation are discussed.

1.1.1 Path Planning

Two common approaches to the path planning problem are through heuristic-based algorithms and sampling-based algorithms. Of the heuristic-based algorithms, one of the most foundational path planning approaches is that of A* [11], with many robot motion planning approaches using A* directly or one of its many successors. A* is a graph search algorithm that generates a least-cost path in a weighted graph. The A* path planner applies a heuristic, an estimate of the path cost to the goal state, to the planning problem, which allows it to direct the search towards the most relevant states. When this heuristic is admissible, it guarantees the found path cost is optimal. While A* works well for generating a path in a known, static graph, many real-world scenarios often provide a graph that is either incomplete or changes over time. The authors of [7] discuss the limitations of the initial A* path planner and describe several successor approaches that attempt to address the limitations of A*. First, it is often the case that the initial graph becomes invalid during navigation, to address this incremental replanning algorithms are introduced. Instead of completely replanning from scratch once the initial graph becomes invalidated, incremental replanning methods such as Focussed Dynamic A* (D*) [40], D* Lite [20], and Delayed D* [8] attempt to modify the existing path to account for graph changes. On the other hand, in many cases a quick reaction from the robot is necessary which can be infeasible when the planning space is large and complex. To address this issue, anytime planning methods such as Anytime Repairing A* (ARA*) [24] first generate a quick, suboptimal solution which is refined over time. The separate benefits of incremental replanning methods and anytime methods led to the creation of anytime replanning algorithms such as Anytime Dynamic A* (AD*) [25] which combines the benefits of D* Lite with those of ARA*.

While these graph-based search algorithms generate optimal paths, they require discretization of the environment and tend to degrade when utilized in high dimensions. Sampling-based planning is another family of path planning algorithms that sample the configuration space for planning. One of the most popular sampling-based planners is Rapidly-exploring Random Trees (RRT) [22]. RRT is a single-query planner, which incrementally grows a tree from a start position to a goal position. To grow this tree, a random configuration is sampled from the configuration space. If the sampled configuration is in free space, a connection to the tree is made from the sampled configuration to the closest vertex of the tree. The simplicity and properties of RRT have made it a popular basis for further path planning algorithms. For example, RRT*, which is introduced in [17], improves on the original RRT algorithm by introducing a near neighbor search and tree rewiring operation which allows the guarantee of asymptotic optimality. Several approches build off RRT and RRT* to tackle additional problems in planning such as Kinodynamic RRT* to apply kinodynamic constraints to planning; execution extended RRT (ERRT) [3] and Dynamic RRT (DRRT) [6] which adapt RRT for real-time and dynamic scenarios; and many others. Another group of popular sampling-based planners is those based on Probabilistic Roadmap (PRM) [18]. While RRT is considered a single-query planner, PRM is a multi-query planner that is comprised of two processes. First, a learning phase samples the configuration space to generate a roadmap, or forest, of the space, then a query phase connects the start and goal positions to the roadmap. Like A* and RRT, PRM is used as the foundation for a plethora of other planning algorithms such as PRM* [17], Visibility PRM [32], Flexible PRM [2], and more.

1.1.2 Bipedal Robot Motion Planning

At the root of the autonomous navigation problem, path planning exists as a core component to achieving robust navigation systems for bipedal robots. The ability to navigate through the environment in a safe and collision-free manner is essential. The fundamental path planning algorithms of A*, RRT, PRM and their derivatives have laid the foundation of path planning for robots. Many existing works use these path planning algorithms to design motion planning methods for bipedal robots. For example, the authors of [4] adapt A* for indoor, goal directed navigation for legged robots. Using A* in a 2D environment representation, they perform a search over the space of valid footsteps that the robot can perform. Following this sequence of footsteps results in successful navigation of legged robots over elevated and rough terrains. Additionally, the method introduced in [15] take on a similar approach but also consider the energy cost associated with navigation when planning footsteps to achieve a lowest energy cost path. Furthermore, generating optimal paths when planning over long distances can take large amounts of time. However, it is often required to place a limit on the planning time. To address this, [14] introduces a method to use anytime path planners to generate initial suboptimal paths that are bound with restricted planning times. In other cases, replanning of a generated path is often necessary if the robot deviates from the path during navigation, the environment changes over time, or terrain is poorly observed during navigation. The authors of [10] address this by using a D* Lite based planner to replan global paths and planned footsteps when deviations occur. However, all of these methods operate in a 2D representation of the environment, which can be insufficient in representing complex environments. The method introduced in [16] addresses this by using a 3D map of the environment which is able to sufficiently represent complex terrains.

While these motion planning methods, among many others, address several essential problems in bipedal robot navigation, they still miss several key features that are necessary for a real-time autonomous navigation system such as real-time planning, robustness to dynamic environments, and the ability to plan in unknown environments. Furthermore, these motion planning methods alone are not sufficient to enable robust navigation, bipedal motion planning methods must be integrated with perception, mapping, localization, and control methods to create holistic navigation systems. Several works expand on these initial motion planning approaches to integrate more of these components of the navigation problem. The authors of [28] address this by incorporating an external vision system to capture obstacle and robot information for environment mapping and robot localization. However, requiring an external vision system makes applying the motion planning approach infeasible in many scenarios. The work shown in [31] addresses this by including an onboard laser range sensor for perception. However, the motion planning method implemented is only shown to be capable of planning in static environments. Another approach by [13] uses a forward facing

RGB-Depth camera and shows successful navigation results on the Lola robot navigating in both static and dynamic environments. However, no localization method is considered which may cause problems for long distance navigation. A different approach by [23] also uses a forward facing RGB-Depth camera and shows impressive motion planning and control capabilities in static, height-constrained environments. However, this work also does not consider localization and does not consider environments with dynamic obstacles.

In summary, while there exists a large basis for bipedal motion planning, many existing approaches to creating navigation systems lack one or more essential components of the autonomous navigation problem. Thus, this thesis introduces work with the goal of addressing all components of the autonomous navigation problem. The available hardware and simulation tools used during development and validation are first discussed in Chapter 2. Next, initial system architecture considerations are presented in Chapter 3. The perception processes used are then explored in Chapter 4, starting with the introduction of what sensor data is available for navigation and which pre-processing methods are applied to this data. Then, the chapter introduces how the available sensor data is used for obstacle segmentation, robot localization, and environment mapping. Afterwards, Chapter 5 explains how the perception results are used to facilitate real-time motion planning. Finally, experimental results in both simulation and hardware are discussed in Chapter 6 and concluding remarks are presented in Chapter 7.

CHAPTER 2

HARDWARE AND SIMULATION

In this chapter, we introduce the bipedal robot used in this research called Digit, which is developed by Agility Robotics, along with the simulation configuration and the Turtlebot3 differential drive robot. Additionally, Digit's equipped sensor suite is explored and the simulated sensor suite is explained.

2.1 Hardware

2.1.1 Digit

Digit is a bipedal robot developed by Agility Robotics. Designed as the successor to the legged robot Cassie, Digit comes with 20 actuated joints and 30 degrees of freedom. Additionally, Digit comes with a robust sensor suite for perception and state estimation. Digit comes equipped with one RGB camera, one RGB-Depth camera, three Depth cameras, one LiDAR sensor, one IMU (inertial measurement unit), and one GPS, with the configuration of the perception sensors shown in Fig. 2.1. Out of this sensor suite, the current navigation system only makes use of two depth cameras and the LiDAR sensor.

2.1.2 Depth Cameras

Digit comes with four cameras with depth capabilities: one Intel RealSense D435 RGB-Depth camera and three Intel RealSense D430 Depth cameras. The RGB-Depth camera is located at the top of Digit's torso and is placed at a downward angle of 45^o. This camera is capable of collecting both visual RGB images and 3D depth point clouds from in front

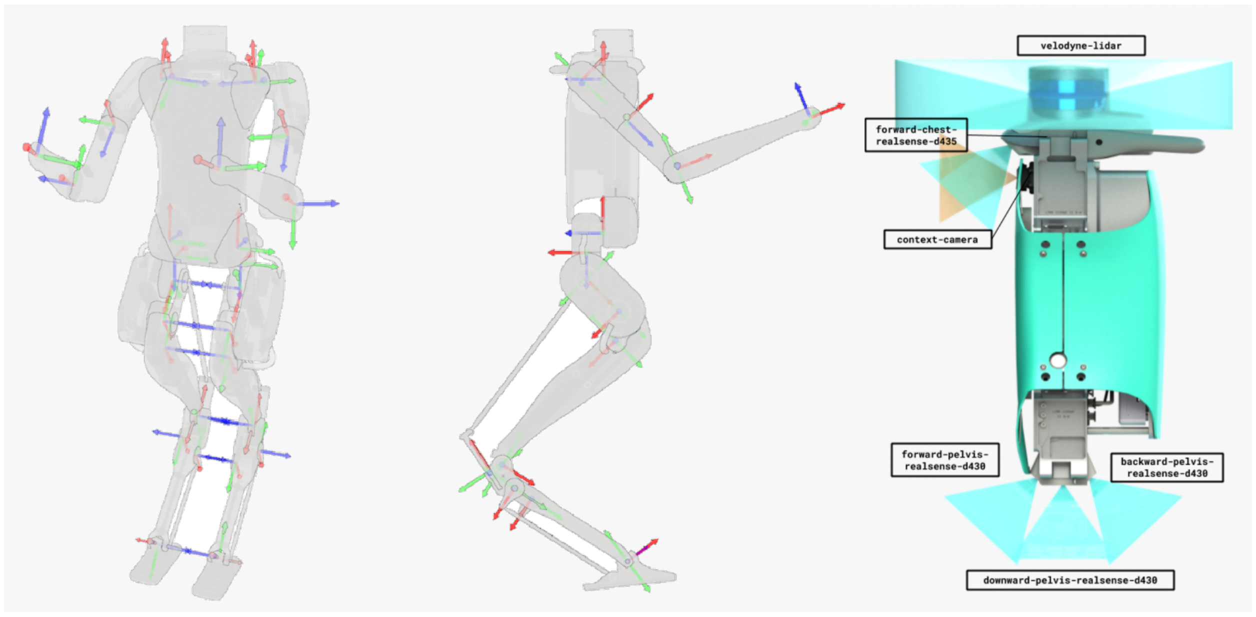
velodyne-lidar
forward-chest-realsense-d435
context-camera
forward-pelvis-realsense-d430
backward-pelvis-realsense-d430
downward-pelvis-realsense-d430

Figure 2.1: The Digit bipedal robot from Agility Robotics. The 20 joints of the Digit bipedal robot (left and center). Coverage of the perception sensors provided in Digit's sensor suite (right).

of the robot. Additionally, there are three depth-only cameras that are placed along Digit's pelvis. These three depth cameras are configured as a forward, downward, and backward depth camera each oriented at 45^o, 90^o, and 135^o, respectively. These cameras capture depth information from directly in front, below, and behind the robot with a field of view of 85^o x 58^o in the horizontal and vertical axes and a reported operating range of 0.2 m to 10.0 m. Due to the placement of these depth cameras, blind spots exist outwards from the right and left sides of Digit. Out of these four depth capable cameras, the forward and backward pelvis depth cameras are utilized in the navigation system.

2.1.3 LiDAR Sensor

In addition to the depth cameras available on Digit, a Velodyne VLP-16 LiDAR sensor attached on top of Digit's torso is used to capture depth information from the environment. The LiDAR sensor uses 16 laser channels that scan in a circular pattern to achieve a field of view of 360^o x 30^o in the horizontal and vertical axes. The LiDAR sensor is used for accurate depth measurements over long distances with a reported operating range of up to 100 m. Due to the LiDAR's placement on top of Digit's torso and the small vertical field of view, the LiDAR sensor is not able to receive ground detections until 5.5 m away from the robot in all directions.

2.2 Simulation

During development the navigation system is not tested directly on the Digit robot, but rather first validated in simulation. Simulated environments are generated using Gazebo, where the navigation system is executed on a simulated Turtlebot3 differential drive robot.

2.2.1 Gazebo

Gazebo is an open-source 3D robotics simulator that provides high fidelity real-world physics simulation. Gazebo supports several physics engines such as ODE (default), Bullet, DART, and more that allow the simulation of friction, gravity, and other forces. Simulated environments can be constructed as Gazebo worlds allowing the simulation of many situations such as indoor and outdoor robotic navigation. Additionally, these simulated environments can be both static and dynamic, allowing for better representation of complex navigation environments. Furthermore, robot and sensor models can be directly inserted into these simulated worlds, allowing sensor measurements and real-time navigation reactions to be tested.

2.2.2 Turtlebot3

During simulated testing, the Turtlebot3 differential drive robot was simulated in Gazebo environments. The Turtlebot3 robot is a single-axle differential drive robot that can be readily simulated in Gazebo. Turtlebot3 comes equipped with a Laser Distance Sensor (LDS) and 6-DOF IMU. However, for validation on Turtlebot3 to better reflect navigation performance on Digit, two Intel RealSense D430 depth cameras and a Velodyne VLP-16 LiDAR sensor are added to the robots simulated configuration. This setup copies the utilized sensors on Digit's sensor suite and mimics the configuration of Digit's sensors in simulation.

CHAPTER 3

SYSTEM ARCHITECTURE

In this chapter we introduce system architecture considerations. As previously mentioned, to enable a truly autonomous navigation system many separate components must integrate seamlessly together into one navigation system. The perception components - obstacle segmentation, localization, and mapping - rely on a reliable stream of information from the sensor suite. Mapping relies on obstacle segmentation for continuous updates of obstacle positions in the environment and on localization for an accurate estimate of the robot position within the mapped environment. The planning system must receive constant estimates of the robot position from localization and requires the most up-to-date environment maps to generate safe and optimal trajectories. Finally, the walking gait controller then relies on planning to receive the latest trajectory to execute. In summary, a constant and reliable communication framework is necessary for an autonomous navigation system to be feasible. In this work, the Robot Operating System (ROS) is leveraged as the backbone middleware to facilitate stable communication and Move Base Flex [35] is used as the communication framework to orchestrate actions between system components.

3.1 Robot Operating System

The Robot Operating System is an open-source middleware that provides several software packages and tools for robotics applications. ROS provides three main services for robotics development: plumbing, tools, and community [39]. ROS provides the plumbing for a message-passing system for inter-process communication. The general ROS communica-

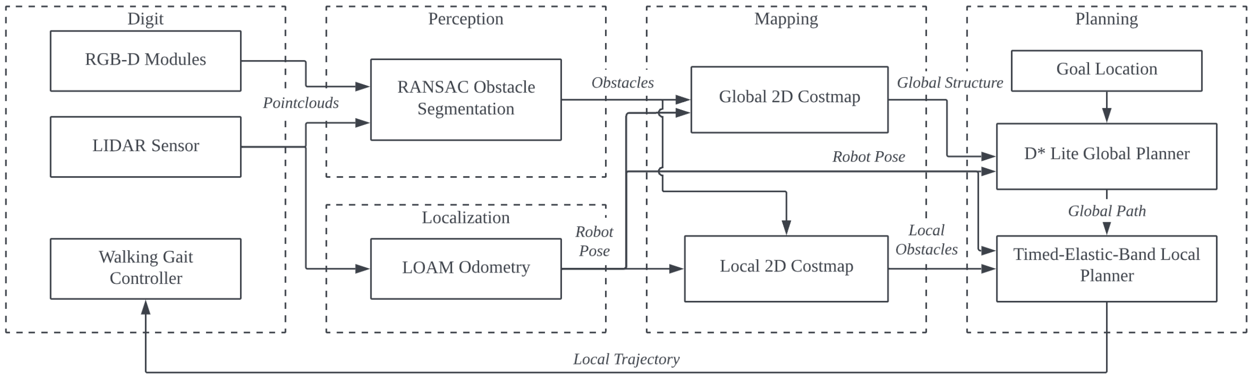

Figure 3.1: The proposed navigation system, an architecture built on top of Move Base Flex [35]. Point cloud detections are used for obstacle segmentation by Random Sample Consensus [9]. Global and local costmaps [26] are generated from the obstacle segmentations. LiDAR Odometry and Mapping [41] localizes the robot. A D* Lite global planner [20] uses the global costmap to generate an optimal, collision-free path, which is used by the Timed-Elastic-Band local planner [36] to generate local obstacle-avoiding trajectories, which are executed through velocity commands sent to Digit's gait controller.

tion framework consists of nodes (processes) and topics (data streams). Processes running as nodes can receive and and send data to other processes by publishing or subscribing messages to named data streams in the form of topics. Alongside this plumbing, ROS provides several tools for debugging, logging, visualization, and others that aid in development. Finally, the large open-source community provides a large selection of ready to use tools and packages for simulation, perception, mapping, planning, control, and more. Building on the provided capabilities of ROS, the described navigation system utilizes Move Base Flex to organize action structure and inter-process communication.

3.2 Move Base Flex

Move Base Flex is an extension to the popular `move_base` [27] used in the ROS navigation stack. Move Base Flex is designed to maintain high levels of flexibility and modularity through abstraction. Built upon Move Base Flex, a high-level overview of the implemented navigation system is shown in Fig. 3.1. First, Move Base Flex maintains an abstraction for environment representation independence by splitting the framework into an abstract

Move Base Flex level and a map implementation level. The abstract Move Base Flex level defines the core functionality of the map actions, while representation specific details are implemented in the map implementation level. Furthermore, Move Base Flex extends this abstract implementation to planning, control, and recovery actions by implementating AbstractPlanner, AbstractController, and AbstractRecovery interfaces. These implement interactions between the path planner, low-level controller and recovery behavior reactions with the Abstract Navigation Server, while planner, controller, and recovery plugins handle implementation specific details.

3.2.1 Move Base Flex Actions

This Abstract Navigation Server defines several actions: `get_path`, `exe_path`, `recovery`, and `move_base` that define the core operation of the Move Base Flex framework.

The `get_path` action computes the path from a given start position to a given goal position. The exact planner plugin to use can be specified and a final path, associated path cost, outcome return code, and a return message are returned.

Using an input path, `exe_path` obtains the controller plugin, starts the controller process, and sends the input path to the controller. The controller executes at a set frequency, generating velocity commands to follow the input path, directly publishing the commands to execute on the robot. Execution returns the current robot pose, velocity, distance to the goal position, angle to the goal target, an outcome return code, and a return message.

The `recovery` action executes the specified recovery strategies and returns an outcome return code and return message.

The `move_base` action is a compound action that combines the execution of `get_path`, `exe_path`, and `recovery` actions into one action. Given a goal position, a planner, controller, and recovery actions, the final output is the same as the `exe_path` action.

3.2.2 Move Base Flex Behavior Tree

For deployment, Move Base Flex uses a flexible behavior tree representation. The Move Base Flex behavior tree uses three different node types: leaf nodes, inner nodes, and sub-trees.

Leaf Nodes: Leaf nodes are terminal nodes that represent either a condition to be checked or a simple action to execute.

Inner Nodes: The inner nodes of the behavior tree connect the condition and action nodes to create more complex workflows. Composite inner nodes can contain several children nodes and can call them all in a particular sequence such as in parallel or sequentially. Decorator inner nodes instead modify the function of a single child node.

Sub-Tree Nodes: Sub-tree nodes, as the name implies, execute another behavior tree instead of a single node. In the Move Base Flex behavior tree there are three different sub-trees used: the Navigation sub-tree, Planning sub-tree, and Control sub-tree.

1) Navigation Sub-Tree: The Navigation sub-tree first checks the validity of the start and goal positions, ensuring both poses are not in collision. If either of the positions are in collision, the recovery behaviors are executed and the validity of the positions is rechecked. If all recovery behaviors are exhausted and the start or goal positions are still not valid, navigation terminates. Otherwise, the Planning and then the Control sub-trees are executed.

2) Planning Sub-Tree: Once valid start and goal positions are received, the Planning sub-tree is executed. The sub-tree attempts to find a valid initial path between the start and goal positions. If no valid path exists, recovery behaviors are executed such as clearing the map, relaxing planning constraints, or navigating away from close obstacles.

3) Control Sub-Tree: Once a valid path is received, the Control sub-tree starts the controller and begins to execute the received path. Again, if any issues prevent execution of the provided path, recovery behaviors are executed in an attempt to resolve controller issues. If all recovery behaviors are exhausted and the path is still not executable, the Planning sub-tree is re-executed. In parallel to the controller execution, a second planner is also started to modify the initial path as the environment is explored to account for discovered obstacles.

CHAPTER 4

PERCEPTION, LOCALIZATION, AND MAPPING

In this chapter, we describe the Perception system and related perception, localization, and mapping processes applied in the proposed navigation system. The Digit bipedal robot is equipped with a suite of depth cameras and a LiDAR sensor used for environment perception. The resultant sensor readings are first pre-processed before being used for obstacle segmentation and mapping due to sensor range limitations and erroneous sensor readings. Knowledge of the pose of Digit relative to the world is essential for successful navigation. A LiDAR-based localization method is described which provides odometry estimation. For planning, an environment representation is required which balances the need for sufficient environment information with computational load. Using the pose estimation from localization, a 2D mapping method is explored which combines raw sensor measurements with segmented obstacles.

4.1 Perception

The perception component of this system relies on the sensor suite provided in the Digit bipedal robot. Out of Digit's equipped sensors, the proposed navigation system utilizes the forward and backward depth cameras placed at Digit's pelvis and the LiDAR sensor placed on top of Digit's torso.

4.2 Point Cloud Pre-Processing

The two depth cameras collect depth information from directly in front and behind Digit. The obtained point clouds are used to capture information about obstacles in the environment local to Digit. These depth readings are accurate within 2.0-3.0 m of Digit, while detections farther than 3.0 m from the forward and backward depth cameras become unreliable, with detection accuracy dropping significantly and point cloud distortion greatly affecting height readings. Each depth camera provides a dense 3D point cloud, where due to the orientation of the cameras the majority of detected points lie within 1.0 m of Digit. This results in an uneven density throughout each point cloud, with a large density of points being focused close to Digit's feet and a low density of points farther away from Digit. These properties of the raw sensor measurements make directly using the obtained point clouds for perception tasks unsatisfactory. Therefore, several pre-processing steps are applied to the sensor measurements before they are used in obstacle segmentation, localization, and mapping.

4.2.1 Voxel Grid Filtering

First, as previously mentioned the raw sensor measurements from the forward and backward depth cameras return dense point clouds with uneven density. Therefore, the depth camera readings are first downsampled using Voxel Grid filtering as implemented in [38]. The 3D environment of the point cloud is discretized into voxels, or cubes, of space. Each of these voxels may contain a number of points from the point cloud within its space. The original point cloud is downsampled by replacing all points within a voxel with the centroid of those points. The downsampling factor using this method is determined by the voxel size used. Voxel Grid filtering of the point cloud results in both the overall point cloud size and the density difference between point cloud regions to be reduced, resulting in increased computational efficiency. In processing the point clouds from Digit's depth cameras, we use voxels of size 0.02 m per voxel dimension, which sufficiently retains environment information while reducing the size of the depth camera point clouds by 91.07% on average.

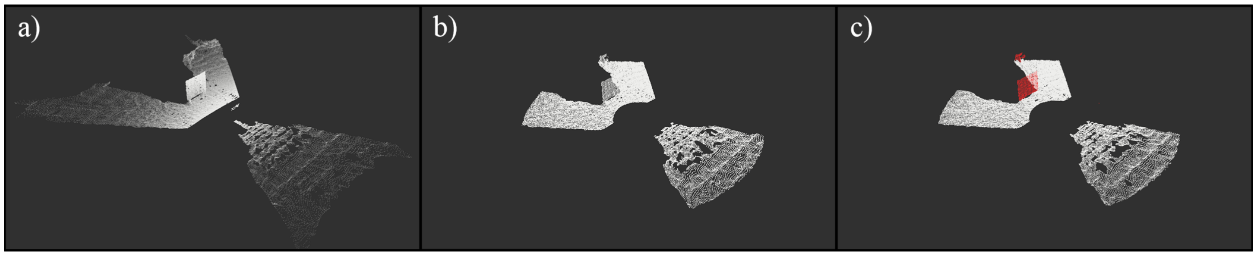

Figure 4.1: Pre-processing and obstacle segmentation results. a) Original point cloud, b) Filtered cloud, and c) Filtered point cloud with segmented obstacles in red.

4.2.2 Pass-Through Filtering

In addition to the size and inconsistent density of the raw sensor measurements, measurement accuracy decreases with increasing distance from the depth cameras and outlier detections are sometimes captured due to object reflections. To prevent inaccurate detection due to measurement distance, the point cloud is filtered to retain only points within 2.9 meters of each depth camera. Furthermore, to prevent reflections causing outlier points to be detected underground and outside of the range of the sensor, a pass-through filter is applied to the point clouds to remove NaN values and points with a height value below the estimated ground plane. The final result of point cloud pre-processing can be seen in Fig. 4.1.

4.3 Obstacle Segmentation

After filtering the point cloud from all depth cameras, the resulting clouds are then fused with the LiDAR point cloud before being used for obstacle segmentation. The Random Sample Consensus (RANSAC) method proposed in [9] is used to segment obstacles from this fused point cloud. Where, for a given point cloud, P, RANSAC randomly samples 3 points to solve a unique plane model:

$$ax + by + cz + d = 0, \tag{4.1}$$

where $a, b, c, d \in \mathbb{R}$ are the fitted coefficients, and $(x, y, z) \in \mathbb{R}^3$ represents the Cartesian coordinates of a point. Then, the absolute distance, D_i, of each point i in the cloud is calculated for the fitted model:

$$D_i = \left| \frac{ax_i + by_i + cz_i + d}{\sqrt{a^2 + b^2 + c^2}} \right|, \text{ for } i \in \{1, 2, .., n\}, \tag{4.2}$$

where n, is the total number of points in the point cloud P. Points within a given threshold distance, $D_{threshold}$, to the plane model are labeled as inliers of the model, denoted P_I,

$$P_I = \{p_i \in P \mid D_i < D_{\text{threshold}}\}. \tag{4.3}$$

This process is repeated iteratively for a specified number of iterations, N, determined statistically as:

$$N = \text{round}\left(\frac{\log(1-\alpha)}{\log(1-(1-\varepsilon)^3)} \right), \tag{4.4}$$

where α is the desired minimum probability of finding at least one good plane from P, usually within $[0.90 - 0.99]$, and $\varepsilon = (1-u)$ where u is the probability that any selected point is an inlier. The resulting plane model is selected as the one which generates the largest number of inliers with the smallest standard deviation of distances.

We use this resulting plane model to represent the estimated ground plane of the navigation environment, given the assumption Digit is navigating in an environment with a flat ground. Points from the fused point cloud can then be classified as either ground points, inliers to the RANSAC plane model, or obstacle points, outliers to the RANSAC plane model. Then, two separate point clouds are created, a ground point cloud and an obstacle point cloud comprised of ground points and obstacle points, respectively.

4.4 Localization

Obtaining sensor readings of the environment from the depth cameras and LiDAR sensor are not directly sufficient for navigation. For successful navigation, applying the obstacle segmentation and sensor measurements for environment mapping requires an accurate odometry source. Direct sources of odometry for pose estimation such as Inertial Measurement Units (IMUs) and Global Navigation Satellite Systems (GNSS) are often used for mobile robot

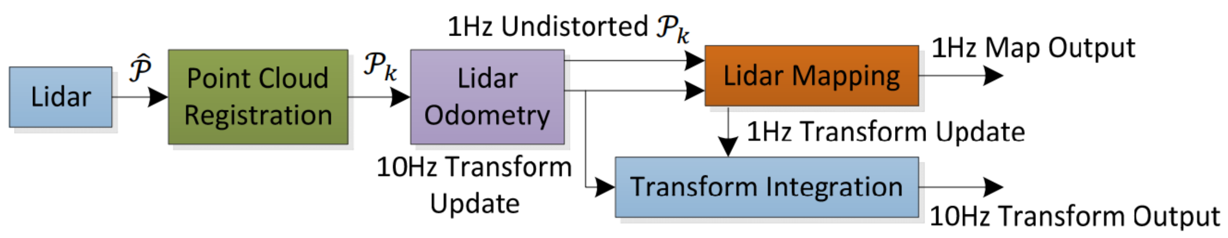

Figure 4.2: Overview of the LiDAR Odometry and Mapping (LOAM) system as shown in [41].

navigation. However, when these direct sources of odometry are not available, Simultaneous Localization and Mapping (SLAM) methods can be used for pose estimation.

Many SLAM methods exist that utilize different sensors such as cameras, LiDAR, RADAR, etc. Regarding Digit's sensor suite: the torso RGB camera has a low frame rate relative to the mobility of Digit which can lead to odometry drift due to low frequency updates. Additionally, the downward orientation of the torso RGB-Depth camera results in a low number of possible feature points that can be extracted from the camera's point cloud. Therefore, the LiDAR sensor at the top of Digit's torso is leveraged for SLAM. To accomplish this, LiDAR Odometry and Mapping (LOAM), a method introduced in [41], is used to estimate the odometry of the robot. LOAM introduces an odometry estimation algorithm and a mapping algorithm that execute in parallel.

The LiDAR odometry estimation algorithm runs at 10 Hz in which the correspondence of extracted features from consecutive LiDAR sensor sweeps are used to estimate the motion of the sensor. The estimated motion of the LiDAR is then used to correct distortions in the accumulated point cloud for a given sweep. The mapping algorithm runs at 1 Hz and uses the undistorted point cloud to generate a 3D map of the environment. Finally, LOAM combines transforms from both the odometry estimation algorithm and the mapping algorithm to generate a final transform of the LiDAR sensor with respect to the generated map, which is generated at 10 Hz. In this section an overview of the LOAM method is described, while a more detailed description can be found in [41].

4.4.1 Feature Point Extraction

First, feature points must be extracted from the point cloud to be used for correlation between consecutive sweeps of the LiDAR sensor. To begin, each point in the point cloud is assigned a c value, which evaluates the smoothness of the local surface,

$$c_i = \frac{1}{|\mathcal{S}| \cdot \left\| \boldsymbol{X}^L_{(k,i)} \right\|} \cdot \sum_{j \in \mathcal{S}, j \neq i} \left(\boldsymbol{X}^L_{(k,i)} - \boldsymbol{X}^L_{(k,j)} \right), \tag{4.5}$$

where $\boldsymbol{X}^L_{(k,i)}$ and $\boldsymbol{X}^L_{(k,j)}$ are the points i and j in the LiDAR coordinate system, L, received during the sensor sweep k, and $\mathcal{S}$ is the set of consecutive points in a scan. The points with the highest smoothness values are considered as edge points, and points with the lowest values are considered as planar points. To ensure an even distribution of features, a maximum of 2 edge points and 4 planar points are extracted from each of 4 identical subregions. For a point to be selected as an edge point or planar point, it must meet several criteria:

- The number of edge points or planar points does not exceed the subregion maximum,
- A surrounding point is not already selected, and
- It does not belong to a surface that is close to parallel to the LiDAR scan, or on the boundary of an occluded region.

4.4.2 Feature Point Correspondence

At the end of each sweep, the accumulated point cloud, $\mathcal{P}_k$, during the sweep is projected to time stamp t_{k+1} to generate the projected point cloud, $\overline{\mathcal{P}}_k$. The projected point cloud is used together with the point cloud accumulated during the next sweep $\mathcal{P}_{k+1}$ for sensor motion estimation. First, correspondences must be found between $\overline{\mathcal{P}}_k$ and $\mathcal{P}_{k+1}$. Features are extracted from $\mathcal{P}_{k+1}$ using the previously described method, resulting in a set of edge points $\mathcal{E}_{k+1}$ and a set of planar points $\mathcal{H}_{k+1}$. These edge points and planar points are then projected back to the beginning of the sweep to obtain a set of projected edge points $\tilde{\mathcal{E}}_{k+1}$ and a set of projected planar points $\tilde{\mathcal{H}}_{k+1}$ using the current transform estimate.

We now can use these point sets to find correspondences between the projected cloud $\overline{\mathcal{P}}_k$ and consecutive point cloud $\mathcal{P}_{k+1}$ in the form of edge lines as the correspondence for edge points and planar patches as the correspondence for planar points. The edge line for point $i \in \tilde{\mathcal{E}}_{k+1}$ is represented as two points (j, l) where j is the closest neighbor of i in point cloud $\overline{\mathcal{P}}_k$ and l is the closest neighbor of i in the two consecutive scans. The planar patch for point $i \in \tilde{\mathcal{H}}_{k+1}$ is represented as the three points (j, l, m) where j is the closest neighbor of i in point cloud $\overline{\mathcal{P}}_k$ and l, m are the two closest neighbors of i, one in the same scan of j and the other in the two consecutive scans to that of j.

With correspondences found for the feature points, the distances between feature points and their correspondence can be found. For a point $i \in \tilde{\mathcal{E}}_{k+1}$ who has a correspondence edge line (j, l) with $j, l \in \overline{\mathcal{P}}_k$, the point to line distance is:

$$d_{\mathcal{E}} = \frac{\left|\left(\tilde{\boldsymbol{X}}^L_{(k+1,i)} - \overline{\boldsymbol{X}}^L_{(k,j)}\right) \times \left(\tilde{\boldsymbol{X}}^L_{(k+1,i)} - \overline{\boldsymbol{X}}^L_{(k,l)}\right)\right|}{\left|\overline{\boldsymbol{X}}^L_{(k,j)} - \overline{\boldsymbol{X}}^L_{(k,l)}\right|}, \tag{4.6}$$

and for a point $i \in \tilde{\mathcal{H}}_{k+1}$ who has a correspondence planar patch (j, l, m) with $j, l, m \in \overline{\mathcal{P}}_k$, the point to patch distance is:

$$d_{\mathcal{H}} = \frac{\left|\begin{matrix}\left(\tilde{\boldsymbol{X}}^L_{(k+1,i)} - \overline{\boldsymbol{X}}^L_{(k,j)}\right) \\ \left(\left(\overline{\boldsymbol{X}}^L_{(k,j)} - \overline{\boldsymbol{X}}^L_{(k,l)}\right) \times \left(\overline{\boldsymbol{X}}^L_{(k,j)} - \overline{\boldsymbol{X}}^L_{(k,m)}\right)\right)\end{matrix}\right|}{\left|\left(\overline{\boldsymbol{X}}^L_{(k,j)} - \overline{\boldsymbol{X}}^L_{(k,l)}\right) \times \left(\overline{\boldsymbol{X}}^L_{(k,j)} - \overline{\boldsymbol{X}}^L_{(k,m)}\right)\right|}, \tag{4.7}$$

where $\tilde{\boldsymbol{X}}^L_{(k+1,i)}, \overline{\boldsymbol{X}}^L_{(k,j)}, \overline{\boldsymbol{X}}^L_{(k,l)}$ are the coordinates of points i, j, and l in the LiDAR coordinate system, respectively.

4.4.3 Motion Estimation

It is assumed that there is a constant angular and linear velocity during each LiDAR sweep. Using this assumption, the pose transformation can be linearly interpolated during each sweep. First, let $\boldsymbol{T}^L_{k+1}$ be the LiDAR transform between time stamps $[t_{k+1}, t]$, where $\boldsymbol{T}^L_{k+1} = [t_x, t_y, t_z, \theta_x, \theta_y, \theta_z]$ is comprised of a 3D translation, $\boldsymbol{t}$, and rotation, $\boldsymbol{\theta}$. Now, for a given

Algorithm 1 LOAM motion estimation algorithm [41]

```
1: procedure LidarOdometry($\overline{\mathcal{P}}_k, \mathcal{P}_{k+1}, \boldsymbol{T}^L_{k+1}$)
2:     if at beginning of LiDAR sweep then
3:         $\boldsymbol{T}^L_{k+1} \leftarrow \mathbf{0}$;
4:     Detect edge and planar points from $\mathcal{P}_{k+1}$, insert into $\mathcal{E}_{k+1}$ and $\mathcal{H}_{k+1}$;
5:     for a number of iterations do
6:         for all edge point $\in \mathcal{E}_{k+1}$ do
7:             Find edge line correspondence and distance using (4.6), apply to (4.13);
8:         for all planar point $\in \mathcal{H}_{k+1}$ do
9:             Find planar patch correspondence and distance using (4.7), apply to (4.13);
10:        Compute a bisquare weight for each row of (4.13);
11:        Update $\boldsymbol{T}^L_{k+1}$ for a nonlinear iteration based on (4.14);
12:        if nonlinear optimization converges then
13:            Break;
14:    if at the end of a LiDAR sweep then
15:        Reproject each point in $\mathcal{P}_{k+1}$ to $t_{k+2}$ and form $\overline{\mathcal{P}}_{k+1}$;
16:    else
17:        Return $\boldsymbol{T}^L_{k+1}$;
```

point $i \in \mathcal{P}_{k+1}$, the pose transform between $[t_{k+1}, t_i]$, $\boldsymbol{T}^L_{(k+1,i)}$, can be linearly interpolated using $\boldsymbol{T}^L_{k+1}$,

$$\boldsymbol{T}^L_{(k+1,i)} = \frac{t_i - t_{k+1}}{t - t_{k+1}} \boldsymbol{T}^L_{k+1}. \tag{4.8}$$

Additionally, a rotation matrix $\boldsymbol{R}$ can be constructed using the Rodrigues formula [29]:

$$\boldsymbol{R} = \boldsymbol{I} + \hat{\omega} \sin\theta + \hat{\omega}^2 (1 - \cos\theta) = e^{\hat{\omega}\theta}, \tag{4.9}$$

where θ is the magnitude of the rotation in $\boldsymbol{T}^L_{(k+1,i)}$, $\theta = \|\boldsymbol{T}^L_{(k+1,i)}[4:6]\|$, and $\hat{\omega}$ is the skew symmetric matrix of ω, with $\omega = \boldsymbol{T}^L_{(k+1,i)}[4:6] / \|\boldsymbol{T}^L_{(k+1,i)}[4:6]\|$.

Using the estimated pose transform and the rotation matrix, a correspondence can be formed between a point in $\mathcal{E}_{k+1}$ or $\mathcal{H}_{k+1}$ and its corresponding point in $\tilde{\mathcal{E}}_{k+1}$ or $\tilde{\mathcal{H}}_{k+1}$:

$$\boldsymbol{X}^L_{(k+1,i)} = \boldsymbol{R} \tilde{\boldsymbol{X}}^L_{(k+1,i)} + \boldsymbol{T}^L_{(k+1,i)}[1:3], \tag{4.10}$$

where $\boldsymbol{X}^L_{(k+1,i)}$ is the coordinates of point i in $\mathcal{E}_{k+1}$ or $\mathcal{H}_{k+1}$, $\tilde{\boldsymbol{X}}^L_{(k+1,i)}$ is the corresponding point in $\tilde{\mathcal{E}}_{k+1}$ or $\tilde{\mathcal{H}}_{k+1}$, and $\boldsymbol{T}^L_{(k+1,i)}[1:3]$ is the 1^{st} to 3^{rd} elements of $\boldsymbol{T}^L_{(k+1,i)}$.

Using the correspondences, geometric relationships can be formed between an edge point and its corresponding edge line and a planar point and its corresponding planar patch:

$$f_{\mathcal{E}}\left(\boldsymbol{X}^L_{(k+1,i)}, \boldsymbol{T}^L_{k+1}\right) = d_{\mathcal{E}}, i \in \mathcal{E}_{k+1}, \tag{4.11}$$

$$f_{\mathcal{H}}\left(\boldsymbol{X}^L_{(k+1,i)}, \boldsymbol{T}^L_{k+1}\right) = d_{\mathcal{H}}, i \in \mathcal{H}_{k+1}. \tag{4.12}$$

These relationships are stacked for each edge feature point and each planar feature point to obtain a nonlinear function,

$$\boldsymbol{f}\left(\boldsymbol{T}^L_{k+1}\right) = \boldsymbol{d}, \tag{4.13}$$

where each row of $\boldsymbol{f}$ is a feature point and each row of $\boldsymbol{d}$ is the correspondence distance. The Jacobian matrix, $\mathbf{J}$, of $\boldsymbol{f}$ with respect to $\boldsymbol{T}^L_{k+1}$ can be computed and used to solve (4.13) by iteratively minimizing $\boldsymbol{d}$ to zero:

$$\boldsymbol{T}^L_{k+1} \leftarrow \boldsymbol{T}^L_{k+1} - (\mathbf{J}^T\mathbf{J} + \lambda \text{diag}(\mathbf{J}^T\mathbf{J}))^{-1}\mathbf{J}^T\boldsymbol{d}, \tag{4.14}$$

where λ is determined by the Levenberg-Marquardt method [12]. At the end of each LiDAR sweep, the LOAM odometry estimation algorithm uses the solved pose transform to undistort the point cloud $\overline{\mathcal{P}}_k$, where both the estimated transform and the undistorted point cloud are then used in the mapping algorithm. This motion estimation algorithm can be seen in Algorithm 1.

4.4.4 LiDAR Mapping

As previously mentioned, the mapping algorithm runs at a lower frequency of 1 Hz, called only at the end of each LiDAR sweep. The mapping algorithm maintains an accumulated mapped point cloud $\mathcal{Q}_k$, which contains the mapped points up to sweep k, and a transform, $\boldsymbol{T}^W_k$, of the pose of the LiDAR with respect to the map at the end of sweep k. From the odometry estimation, the mapping algorithm receives the solved pose transform which is used to extend $\boldsymbol{T}_k^W$ for one sweep from t_{k+1} to t_{k+2} into $\boldsymbol{T}_{k+1}^W$, and the undistorted point cloud which is projected into the world coordinates to obtain $\bar{\mathcal{Q}}_{k+1}$.

Feature points are then extracted from $\bar{\mathcal{Q}}_{k+1}$ in the same way as described in Section 4.4.1, except 10 times the maximum feature points are extracted. To compute the correspondence between feature points in $\bar{\mathcal{Q}}_{k+1}$ and mapped points in $\mathcal{Q}_k$, two points on an edge line and three points on a planar patch are selected to allow the same distance formulation as (4.6) and (4.7). The geometric relationships in (4.11) and (4.12) are reformulated for these new sets of feature points, and then the nonlinear optimization is again solved using the Levenberg-Marquardt method to finally register $\bar{\mathcal{Q}}_{k+1}$ onto the map. The final pose of the LiDAR sensor is the combination of the pose transform $\boldsymbol{T}_{k+1}^L$ from the odometry algorithm and the pose transform $\boldsymbol{T}_k^W$ from the mapping algorithm.

4.5 Mapping

Although LOAM generates both an odometry estimation and a mapped point cloud, the mapped point cloud was not used directly for environment mapping. Although LOAM provides a high fidelity depth map of the navigated environment, the computational cost of maintaining a 3D map and executing 3D planning within this map is high. Additionally, for many environments, such as during indoor navigation, the level of environment information provided in a 3D map is not always necessary for successful navigation and may unnecessarily increase computational load. Instead, a 2D representation of the environment was used for mapping and planning is executed in 2D instead of 3D. To enable this, we create a 2D layered cost maps presented in [26] to map both the global environment and the robots local region.

4.5.1 Global Map

As the robot navigates through the environment, long term mapping of environment obstructions and general environment structure is required. The purpose of the global map is not to track local, dynamic obstacles but rather capture and store macro-scale information of the environment. To enable this, the global map uses a lower resolution of 0.1 m and only updates at a rate of 2 Hz. To populate the global map, wall and large obstructions

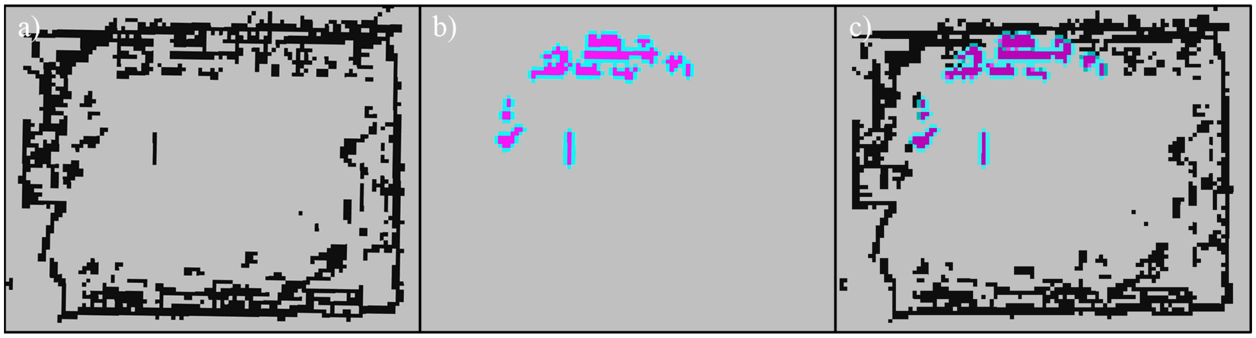

Figure 4.3: Mapping results from the same environment as shown in Fig. 4.1. a) Global map with black cells representing Occupied space and grey cells representing Free space. b) Local map with purple cells representing Occupied cells, blue cells representing the Inflated regions around obstacles, and grey cells representing Free space. c) The fused global and local maps.

are detected using the LiDAR sensor, and the local region of the global map is updated with the latest local map information. The odometry estimation from LOAM is used for two purposes during map creation: 1) At each update step, the pose estimate allows the global map to be accurately updated using LiDAR readings relative to the already mapped global environment; and 2) At each update step, the pose estimate is used to accurately position the local map within the global map to update information regarding close dynamic obstacles.

The global map is represented as a cost map, where the map is comprised of a grid of cells with each cell given a state representing environment information. Cells in the global map have two possible states:

Occupied: Any 3D point detections from the obstacle point cloud extracted earlier are projected onto the 2D plane of the global map. Any cells that contain projected points are considered to be occupied by obstacles. All occupied cells are considered untraversable.

Free: Any cell that is not occupied is considered free and traversable for the robot. Unexplored regions of the environment are by default considered to be free.

4.5.2 Local Map

While the global map is used to store long-term information, capturing all known environment information for explored areas, the local map is intended to only capture a small region local to the robot. This local map captures a 8 m x 8 m region around the robot with the map's origin being the robot's center of mass. The map updates at a frequency of 10 Hz, which allows it to capture local dynamic obstacles and facilitate real-time obstacle avoidance. Additionally, to better enable obstacle avoidance, the local map uses a higher resolution of 0.05 m per cell to more precisely capture obstacle positions within the environment. The local map is populated using detections from both the segmented obstacle point cloud and the LiDAR sensor measurements. Similar to the global map, the local map cells can be given Occupied and Free states which are identical to that of the global map, but introduces a new cell state:

Inflated: Cells within a certain distance of obstacles are inflated with non-zero cell costs. These non-zero costs are used to penalize trajectory planning through regions close to obstacles while not completely prohibiting trajectories from entering these regions.

CHAPTER 5

REAL-TIME MOTION PLANNING

In this chapter we describe the motion-planning component of the proposed navigation system. This motion planning is comprised of a two-stage planner and the Digit default low-level walking gait controller. During navigation through unknown environments, generating in real-time collision-free paths through the global environment is required. As described in Section 4.5, we use a 2D grid-based representation for the environment. Considering the need for real-time collision-free path generation and planning in a grid-based representation, D* Lite is used as the global planner as presented in [20]. Capable of fast re-planning, D* Lite is designed to solve goal-directed navigation problems in graph-represented environments and can be directly applied to the global 2D cost map described in Section 4.5. However, global planning is not sufficient as the D* Lite planner does not consider the physical constraints of the robot during path generation. Introducing these physical constraint into the global planner is possible, but when navigating in large environments, optimizing a kinematically constrained trajectory in real-time over long distances can become infeasible. Therefore, a local planner, Timed-Elastic-Band (TEB) [37, 36], is leveraged to generate kinematically feasible trajectories only in the local map that follow the global path provided by D* Lite.

5.1 D* Lite Global Planner

D* Lite as introduced in [20] builds off a variant of the popular A* planning algorithm [11], Lifelong Planning A* (LPA*) [19]. The D* Lite algorithm is an incremental graph-search planning algorithm that can be used to solve goal-directed navigation problems. In a terrain modelled as an eight-connected graph, the D* Lite algorithm continuously calculates the shortest path between the current robot vertex and a goal vertex. The edge costs of this graph can change during navigation towards the goal vertex, with the only restriction on the graph being that the successors and predecessors of a given vertex must be determinable.

For every vertex, s, in the graph, D* Lite maintains two goal distance estimates: an estimate of the goal distance, $g(s) = h(s, s_{goal})$, where h is the Euclidean distance from the current vertex s to the goal vertex s_{goal}; and a one-step lookahead value based on the g-value, $rhs(s)$:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in \text{Pred}(s)} \left(g\left(s'\right) + c\left(s', s\right)\right) & \text{otherwise,} \end{cases} \tag{5.1}$$

where $c(s', s)$ is the cost to traverse from a neighboring vertex, s', to the current vertex, s. The heuristic h does not need to be the Euclidean distance, but must satisfy the following: $h(s, s')$ must be nonnegative, $h(s, s') \leq c^*(s, s')$, and $h(s, s'') \leq h(s, s') + h(s', s'')$ for all vertices $s, s', s'' \in \mathcal{S}$, with $c^*(s, s')$ denoting the cost of the shortest path from s to s'.

Furthermore, a vertex is classified as locally consistent if its g-value equals its rhs-value, otherwise it is classified as locally inconsistent. When all vertices are locally consistent, a shortest path can be generated between the goal vertex s_{goal} to a start vertex s_{start} by moving from the current vertex s, starting at the start vertex, to the successor vertex, s' that minimizes $c(s, s') + g(s')$ until the goal vertex is reached. However, when some edge costs of the graph change (when new obstacles are detected) the vertices will not all be locally consistent. Instead of making all vertices locally consistent, D* Lite will instead use heuristics to only update the vertices that are relevant for shortest path generation. To achieve this, a priority queue is maintained that contains the locally inconsistent vertices. The vertices in the priority queue are those vertices that need their g-values updated to become locally consistent. The priority of vertices in the priority queue are based on a key value given as $k(s) = [k_1(s), k_2(s)]$, with

$$k_1(s) = \min(g(s), rhs(s)) + h(s, s_{start}) + k_m, \tag{5.2}$$

$$k_2(s) = \min(g(s), rhs(s)), \tag{5.3}$$

where k_m is a scalar whose value is incremented each time the planner re-plans, and will be expanded on later in this section.

Algorithm 2 Optimized D* Lite initialization algorithm [20]

```
1: procedure Initialize()
2:     $U = \emptyset$;
3:     $k_m = 0$;
4:     $\forall s \in S, rhs(s) = g(s) = \infty$;
5:     $rhs(s_{goal}) = 0$;
6:     $U$.Insert($s_{goal}$, $[h(s_{start}, s_{goal}); 0]$);
```

First, the D* Lite algorithm begins by initializing the search problem, shown in Algorithm 2, setting the g-values and rhs-values to infinity. The D* Lite algorithm begins searching for a shortest path from the goal vertex to the start vertex, thus as part of the initialization the goal vertex's rhs-value is set to 0, and adds the goal vertex to the empty priority queue, as it is the only locally inconsistent vertex. After initialization, ComputeShortestPath(), shown in Algorithm 3, is called, which on the first run operates similar to A*. As new vertices are removed from the priority queue, the true values of a vertex are only calculated as they are expanded. ComputeShortestPath() removes the vertex u with the smallest priority, k_{old}, from the priority queue. This priority from the priority queue may no longer be accurate due to edge cost changes. CalculateKey() is used to calculate the most current priority for vertex u, if $k_{old} <$ CalculateKey(), it is reinserted into the priority queue with the new key value, if $k_{old} \geq$ CalculateKey(), then vertex u is expanded. A locally inconsistent

Algorithm 3 Optimized D* Lite shortest path generation algorithm [20]

```
procedure CalculateKey($s$)
    return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m); \min(g(s), rhs(s))]$;
procedure UpdateVertex($u$)
    if $(g(u) \neq rhs(u)$ AND $u \in U)$ then
        $U$.Update($u$, CalculateKey($u$));
    else if $(g(u) \neq rhs(u)$ AND $u \notin U)$ then
        $U$.Insert($u$, CalculateKey($u$));
    else if $(g(u) = rhs(u)$ AND $u \in U)$ then
        $U$.Remove($u$);
procedure ComputeShortestPath()
    while ($U$.TopKey() $<$ CalculateKey($s_{start}$) OR $rhs(s_{start}) > g(s_{start})$) do
        $u = U$.Top();
        $k_{old} = U$.TopKey();
        $k_{new} =$ CalculateKey($u$);
        if $k_{old} < k_{new}$ then
            $U$.Update$(i, k_{new})$;
        else if $g(u) > rhs(u)$ then
            $g(u) = rhs(u)$;
            $U$.Remove($u$);
            for all $s \in$ Pred($u$) do
                if $s \neq s_{goal}$ then
                    $rhs(s) = \min(rhs(s), c(s, u) + g(u))$;
                UpdateVertex($s$);
        else
            $g_{old} = g(u)$;
            $g(u) = \infty$;
            for all $s \in$ Pred$(u) \cup \{u\}$ do
                if $rhs(s) = c(s, u) + g_{old}$ then
                    if $s \neq s_{goal}$ then
                        $rhs(s) = \min_{s' \in \text{Succ}(s)} (c(s, s') + g(s'))$;
                UpdateVertex($s$);
```

vertex, s, may be locally overconsistent if $g(s) > rhs(s)$, in which case expanding the vertex sets the g-value equal to the rhs-value, or the vertex may be locally underconsistent if $g(s) < rhs(s)$, in which case expanding the vertex sets the g-value equal to infinity. Expanding a locally overconsistent vertex makes the vertex locally consistent, but expanding a locally underconsistent vertex may make the vertex either locally consistent or locally overconsistent. Furthermore, the expansion of locally inconsistent vertices may effect the local consistency of its successors. Thus, ComputeShortestPath() updates the rhs-values of the successors, checks for local consistency, and appropriately adds or removes them from the priority queue.

After executing ComputeShortestPath(), the shortest path can be followed by navigating from the start vertex, s_{start}, to the goal vertex, s_{goal}, by moving to the successor, s', of the current vertex, s, that minimizes $c(s, s') + g(s')$ until the goal position is reached or it is determined no finite-cost path exists from s_{start} to s_{goal}.

With an initial shortest path found, the robot can begin to move along the generated path. When navigating in an unknown environment, new obstacles may be encountered during navigation which results in edge cost changes. With edge costs changes detected in the graph, two processes occur: first, the heap ordering of the priority queue must be maintained and second, the values and keys of the vertices potentially affected by the edge changes must be updated.

When edge cost changes are detected in the graph after moving from a given vertex, s, to a new vertex, s', the priorities of the affected vertices in the priority queue must be recalculated as they were initially based on heuristics calculated at the old vertex position of the robot. Unless these priorities are recalculated, they do not follow the constraint that the priority of a vertex in the priority queue is the same as its key value. However, constantly recalculating priorities in the priority queue can quickly become computationally expensive as the number of vertices in the priority queue increases. To avoid this constant reordering, D* Lite adapts a method from the D* planning algorithm introduced in [40], which maintains lower bounds on priorities. When the robot moves from a given vertex, s, to a new vertex,

Algorithm 4 Optimized D* Lite shortest path execution algorithm [20]

```
1: procedure Main()
2:     $s_{last} = s_{start}$;
3:     Initialize();
4:     ComputeShortestPath();
5:     while $s_{start} \neq s_{goal}$ do
6:         if $rhs(s_{start}) = \infty$ then
7:             There exists no known path
8:         $s_{start} = \underset{s' \in \mathrm{Succ}(s_{start})}{\mathrm{argmin}} (c(s_{start}, s') + g(s'))$;
9:         Move to $s_{start}$;
10:        Scan for changed graph edge costs;
11:        if Edge costs changes detected then
12:            $k_m = k_m + h(s_{last}, s_{start})$;
13:            $s_{last} = s_{start}$;
14:            for all all directed edges $(u, v)$ with changed edge costs do
15:                $c_{old} = c(u, v)$;
16:                Update edge cost $c(u, v)$;
17:                if $c_{old} > c(u, v)$ then
18:                    if $u \neq s_{goal}$ then
19:                        $rhs(u) = \min(rhs(u), c(u, v) + g(v))$;
20:                else if $rhs(u) = c_{old} + g(v)$ then
21:                    if $u \neq s_{goal}$ then
22:                        $rhs(u) = \min_{s' \in \mathrm{Succ}(u)} (c(u, s') + g(s'))$;
23:                UpdateVertex($u$);
24:            ComputeShortestPath();
```

s', and edge cost changes are detected, the first element of the priorities in the priority queue can have decreased by at most $h(s, s')$ (the distance between vertices s and s'). Thus to maintain the lower bounds on the priorities, $h(s, s')$ should be subtracted from the first element of the priorities of all vertices in the priority queue. Thus, when new vertices are added to the priority queue, the first element of their priorities are $h(s, s')$ too small, and must have the value added to the first element of their priorities every time the edge costs change. When the robot moves and the edge costs change again, this constant offset of the priorities increases. To account for this, a constant value, k_m, is added to the first element of newly calculated priorities. Every time the robot moves from s to s' and the edge costs change, the value of k_m is increased by $h(s, s')$.

Additionally, when edge cost changes are detected in the graph after moving from a given vertex, s, to a new vertex, s', UpdateVertex() is called to update the rhs-values and keys of the potentially affected vertices. Once the affected vertices are updated, ComputeShortestPath() is executed again to obtain the new shortest path, and the robot can continue navigation. This global planner executes at a rate of 5 Hz and the most up-to-date global path is represented as a sequence of waypoint vertices within the global cost map. The main global path generation process can be seen in Algorithm 4.

5.2 Timed-Elastic-Band Local Planner

The purpose of the local planner is to generate a smooth, kinematically feasible trajectory that follows the global path generated by the D* Lite global planner. The Timed-Elastic-Band (TEB) method introduced in [37, 36] originally implements a trajectory optimization method for differential drive models. We adapt this method for use in bipedal robots by applying simplifying assumptions on the kinematic constraints of the low-level controller to mimic those of a differential drive robot. The bipedal locomotion is modeled as a differential drive model to avoid several problems introduced from the omnidirectional movement of bipedal robots. For example, lateral velocity in bipedal locomotion is difficult to control due to oscillating behaviors in the lateral direction.

The TEB problem is defined as an open-loop optimization task to find the sequence of controls needed to move the robot from an initial pose, $\mathbf{s}_s$, to a goal pose, $\mathbf{s}_f$. A trajectory is defined as a sequence of robot poses $\mathcal{S} = \{\mathbf{s}_k | k = 1, 2, \dots, n\}$ where $\mathbf{s}_k = [x_k, y_k, \beta_k]^\top$ denotes the robot pose at time k, with x_k and y_k denoting the 2D position of the robot and β_k denoting the orientation of the robot. The TEB planner then augments the trajectory with positive time intervals $\Delta T_k, k = 1, 2, \dots, n$. The sequence of robot poses and time intervals are then joined to form the parameter vector $\mathbf{b}^* = [\mathbf{s}_1, \Delta T_1, \mathbf{s}_2, \Delta T_2, \mathbf{s}_3, \dots, \Delta T_{n-1}, \mathbf{s}_n]^\top$. TEB solves the non-linear optimization task defined as:

$$V^*(\mathbf{b}) = \min_{\mathbf{b}} \sum_{k=1}^{n-1} \Delta T_k^2, \tag{5.4}$$

which is subject to several constraints defined as equality and inequality constraints.

Non-holonomic constraint: An equality constraint $\mathbf{v}_k$ enforces the kinematic constraints of the robot. According to [37] this non-holonomic constraint can be defined with a geometric interpretation to assume the robot can only execute smooth paths with consecutive poses $\mathbf{s}_k$ and $\mathbf{s}_{k+1}$ sharing a common arc of constant curvature. To enforce this constraint the angle ϑ_k between pose $\mathbf{s}_k$ and direction $\mathbf{d}_{k,k+1} = [x_{k+1} - x_k, y_{k+1} - y_k, 0]^\top$ has to be equal to the similar angle ϑ_{k+1} at pose $\mathbf{s}_{k+1}$,

$$\vartheta_k = \vartheta_{k+1}, \tag{5.5}$$

$$\begin{bmatrix} \cos(\beta_k) \\ \sin(\beta_k) \\ 0 \end{bmatrix} \times \mathbf{d}_{k,k+1} = \mathbf{d}_{k,k+1} \times \begin{bmatrix} \cos(\beta_{k+1}) \\ \sin(\beta_{k+1}) \\ 0 \end{bmatrix}, \tag{5.6}$$

and thus the resulting equality constraint applied to the optimization task in (5.4) is given by:

$$\mathbf{h}_k(\mathbf{s}_{k+1}, \mathbf{s}_k) = \left(\begin{bmatrix} \cos(\beta_k) \\ \sin(\beta_k) \\ 0 \end{bmatrix} + \begin{bmatrix} \cos(\beta_{k+1}) \\ \sin(\beta_{k+1}) \\ 0 \end{bmatrix} \right) \times \mathbf{d}_{k,k+1}. \tag{5.7}$$

Velocity and acceleration constraints: Additionally, limitations of the linear and angular velocities and accelerations are applied. First, the linear and angular velocities are approximated between two consecutive poses s_k and s_{k+1}:

$$v_k = \Delta T_k^{-1} \| [x_{k+1} - x_k, y_{k+1} - y_k]^\top \| \gamma(\mathbf{s}_k, \mathbf{s}_{k+1}), \tag{5.8}$$

$$\omega_k = \Delta T_k^{-1} (\beta_{k+1} - \beta_k), \tag{5.9}$$

where $\gamma(\mathbf{s}_k, \mathbf{s}_{k+1})$ is a sign extraction function which is approximated by a smooth sigmoidal approximation that maps to the interval $[-1, 1]$:

$$\gamma(\mathbf{s}_k, \mathbf{s}_{k+1}) \approx \frac{\kappa \langle \mathbf{q}_k, \mathbf{d}_{k,k+1} \rangle}{1 + |\kappa \langle \mathbf{q}_k, \mathbf{d}_{k,k+1} \rangle|}. \tag{5.10}$$

Next, the accelerations are approximated in a similar manner to the velocities between consecutive poses:

$$a_k = \frac{2(v_{k+1} - v_k)}{\Delta T_k + \Delta T_{k+1}}, \tag{5.11}$$

$$\dot{\omega}_k = \frac{2(\omega_{k+1} - \omega_k)}{\Delta T_k + \Delta T_{k+1}}. \tag{5.12}$$

Finally, the velocity and acceleration constraints are applied to (5.4) as:

$$\mathbf{v}_k(\mathbf{s}_{k+1}, \mathbf{s}_k, \Delta T_k) = [v_{max} - |v_k|, \omega_{max} - |\omega_k|]^\top, \tag{5.13}$$

$$\boldsymbol{\alpha}_k(\mathbf{s}_{k+2}, \mathbf{s}_{k+1}, \mathbf{s}_k, \Delta T_{k+1}, \Delta T_k)) = [a_{max} - |a_k|, \dot{\omega}_{max} - |\dot{\omega}_k|]^\top. \tag{5.14}$$

Obstacle avoidance: The local planner considers obstacles, $\mathcal{O}_l, l = 1, 2, ..., R$, as simply-connected regions in the form of points, circles, polygons, and lines in $\mathbb{R}^2$. To ensure a minimum separation ρ_{min} between pose $\mathbf{s}_k$ and all obstacles, the inequality constraint applied to (5.4) is:

$$\mathbf{o}_k(\mathbf{s}_k) = [\rho(\mathbf{s}_k, \mathcal{O}_1), \rho(\mathbf{s}_k, \mathcal{O}_2), ..., \rho(\mathbf{s}_k, \mathcal{O}_R)]^\top - [\rho_{min}, \rho_{min}, ..., \rho_{min}]^\top, \tag{5.15}$$

where ρ is the minimal Euclidean distance between the obstacle and the robot pose. This equality restricts the possible robot positions (x_k, y_k) to $\{\mathbb{R}^2 \setminus \bigcup_{l=1}^{R} \tilde{\mathcal{O}}_l\}$ where $\tilde{\mathcal{O}}_l$ is the obstacle region inflated by ρ_{min}. This leads to the nonlinear problem (5.4) containing many local minima. Due to the computational burden, local optimization methods are used for the online optimization, causing the optimality of the solution to depend on the global path, which acts as an initial solution for the local optimization.

The nonlinear task shown in (5.4) is computationally expensive to solve directly considering the hard constraints. The TEB method instead applies an unconstrained optimization technique to the problem. Thus, the problem is transformed into an approximate nonlinear least-squares optimization problem that can be more efficiently solved. The constraints are applied to the approximated objective function as additional quadratic penalty functions and applied according to [33]. The kinematic equality constraint is expressed as:

$$\phi(\mathbf{h}_k, \sigma_h) = \sigma_h \|\mathbf{h}_k\|_2^2, \tag{5.16}$$

and the inequality constraints are approximated as:

$$\chi(\mathbf{v}_k, \sigma_v) = \sigma_v \|\min\{\mathbf{0}, \mathbf{v}_k\}\|_2^2, \tag{5.17}$$

$$\chi(\boldsymbol{\alpha}_k, \sigma_\alpha) = \sigma_\alpha \|\min\{\mathbf{0}, \boldsymbol{\alpha}_k\}\|_2^2, \tag{5.18}$$

$$\chi(\mathbf{o}_k, \sigma_o) = \sigma_o \|\min\{\mathbf{0}, \mathbf{o}_k\}\|_2^2, \tag{5.19}$$

where $\sigma_i, i \in \{\mathbf{h}, \mathbf{v}, \boldsymbol{\alpha}, \mathbf{o}\}$ are scalar weights and the min operator is applied row-wise. The overall optimization problem is approximated with the objective function:

$$\mathbf{b}^* = \underset{\mathcal{B}\setminus\{\mathbf{s}_1, \mathbf{s}_n\}}{\operatorname{argmin}} \tilde{V}(\mathbf{b}), \tag{5.20}$$

$$\tilde{V}(\mathbf{b}) = \sum_{k=1}^{n-1} [\Delta T_k^2 + \phi(\mathbf{h}_k, \sigma_h) + \chi(r_k, o_r) + ... + \chi(\mathbf{v}_k, \sigma_v) \\ + \chi(\mathbf{o}_k, \sigma_o) + \chi(\boldsymbol{\alpha}_k, \sigma_\alpha)] + \chi(\boldsymbol{\alpha}_n, \sigma_\alpha), \quad (5.21)$$

where $\mathbf{b}^*$ is the optimal parameter vector. This optimization problem is solved using a variant of the Levenberg–Marquardt algorithm implemented in the open-source graph optimization framework g2o [21]. This optimization problem is repeatedly solved during runtime where at each sampling interval only the first control input is commanded to the robot to account for disturbances and changes in the environment. In the proposed navigation system, the robot is commanded by a translational and rotational velocity calculated according to (5.8) and (5.10). At the end of each interval, the start and goal poses are updated and the previous trajectory is sampled with respect to its length to maintain a desired temporal step size ΔT_{ref}. A new sample is added between poses $\mathbf{s}_k$ and $\mathbf{s}_{k+1}$ if ΔT_k is larger than ΔT_{ref}, otherwise pose $\mathbf{s}_{k+1}$ is removed. To prevent oscillations in the number of samples, a hysteresis ΔT_{hyst} is applied.

As previously mentioned, the application of local online optimization results in many local minima in the nonlinear problem, where achieving the global minimum relies on the initial global path. The TEB method is further improved to search for the global minimum by optimizing several trajectories from distinctive topologies in parallel. The detailed descriptions of these optimizations can be found in [36].

The final output from the TEB local planner is sequences of translational and rotational velocity commands that can be used to command the robot. To execute the local trajectory generated by TEB, the velocity commands are executed using Digit's default low-level controller provided by Agility Robotics.

CHAPTER 6

RESULTS

This chapter focuses on the results of the proposed navigation system. Initial experiments were conducted in simulation to validate the perception, localization, mapping, and planning systems for real-time navigation. Simulated experiments were conducted to explore navigation in unknown environments, navigation in environments with dynamic obstacles, and long distance navigation. After the navigation system was validated in simulation, it was adapted to be used on the Digit bipedal robot and tested in static and dynamic environments.

6.1 Simulation Experiments

As previously mentioned, for simulation experiments a Turtlebot3 differential drive robot is configured with a sensor suite to mimic that of the Digit bipedal robot. The Turtlebot3 robot is placed in one of two simulated Gazebo worlds used for validation. The first Gazebo world, as seen in Section 6.1.1 and Section 6.1.2, is a 6.0 m x 6.0 m hexagon shaped environment. Inside this hexagon are nine static, 0.15 m radius and 2.0 m tall cylinders organized in a grid pattern 0.5 m away from each other. The second Gazebo environment used for simulation validation can be seen in Section 6.1.3. This environment is 6.0 m x 30.0 m and contains three distinct rooms for navigation. The first two rooms contain static, rectangular prism shaped obstacles, in the first room they are placed along the walls of the environment and in the second room they are interspersed throughout the room. The third room contains multiple cube shaped dynamic obstacles.

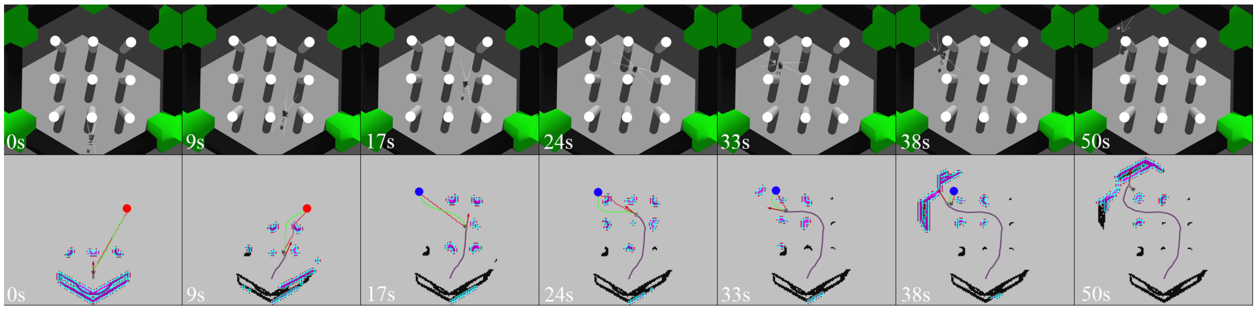

Figure 6.1: Results from experiment 6.1.1 with the top row showing the simulated world view and the bottom row showing the environment representation available to the robot. The red and blue circles represent the first and second goals. Black grid cells are obstacles in the global map, purple and blue grid cells are obstacles and inflated cells in the local map, respectively. The red arrow is the current estimated odometry. The red path and green paths are the global path and local trajectory, respectively. The purple path shows the robot's traveled path. As unknown obstacles are discovered, re-planning occurs. Midway through execution, a new goal is provided and is reached.

6.1.1 Static Obstacles, Unknown Environment

Experiment results are shown in Fig. 6.1. The Turtlebot3 is placed in the first Gazebo simulated environment previously described, with the task of navigating in an environment that is unknown with static obstacles. At $t = 0$, an initial goal position is selected. From $t = 0$ to $t = 9$ the robot navigates along the path until an obstacle is discovered obstructing the current trajectory, at which point the global path and local trajectory are quickly re-planned to account for the new obstacle. While continuing to navigate to the first goal, a new goal position is given. The system quickly re-plans and begins navigating to the new goal. At $t = 24$, a blocking obstacle is again detected and re-planning occurs. From $t = 24$ to $t = 50$ the robot successfully executes the trajectory and reaches the goal position. From this experiment, the navigation system's ability to successfully command a robot in static, unknown environments is validated. This ability is necessary for navigation systems to operate in society as navigation through a previously unexplored environment is often needed.

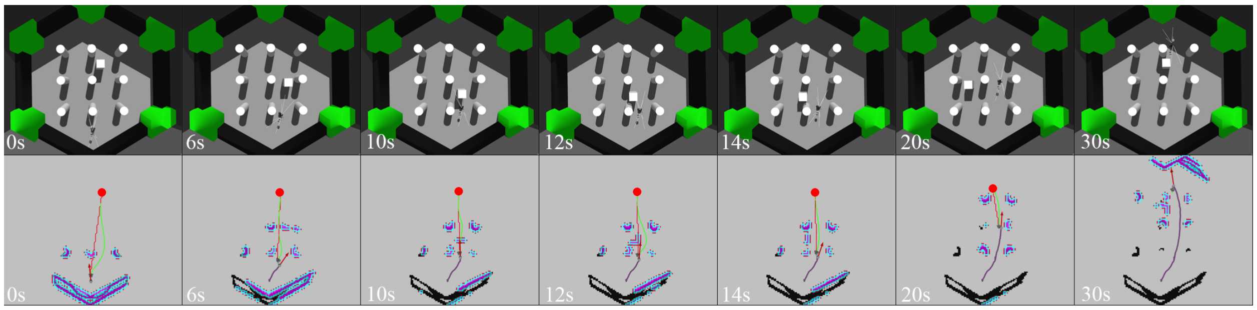

Figure 6.2: Results from experiment 6.1.2 with the top row showing the simulated world view and the bottom row showing the environment representation available to the robot. The red circle represents the goal position. Black grid cells are obstacles in the global map, purple and blue grid cells are obstacles and inflated cells in the local map, respectively. The red arrow is the current estimated odometry. The red path and green paths are the global path and local trajectory, respectively. The purple path shows the robot's traveled path. As the unknown dynamic obstacles is discovered, the robot stops, begins to reverse, and then plans around the obstacle. Once the dynamic obstacle has passed, the robot begins to navigate forward and successfully reaches the goal.

6.1.2 Dynamic Obstacles, Unknown Environment

Experiment results are shown in Fig. 6.2. The Turtlebot3 is placed in the same Gazebo environment as the experiment in Section 6.1.1, however a dynamic obstacle has been placed in the environment in the form of a 0.15 m x 0.15 m x 2.0 m rectangular prism moving in a square pattern in the center of the simulated environment. This task explores navigating in an unknown environment that contains dynamic obstacles. At $t = 0$, an initial goal position is selected. The robot begins to navigate until $t = 6$ when the dynamic obstacle comes into view and blocks the path. The planners attempt to re-plan around the obstacle but determine the current path is untraversable. The robot stops and begins to reverse until $t = 12$ when a viable trajectory is discovered. At $t = 14$, the robot stops reversing and begins to execute the trajectory as the dynamic obstacle has passed. From $t = 14$ to $t = 30$, the robot executes the trajectory and reaches the goal. The successful completion of this experiment validates the navigation systems ability to react to dynamic obstacles, necessary in the complex environments common in society.

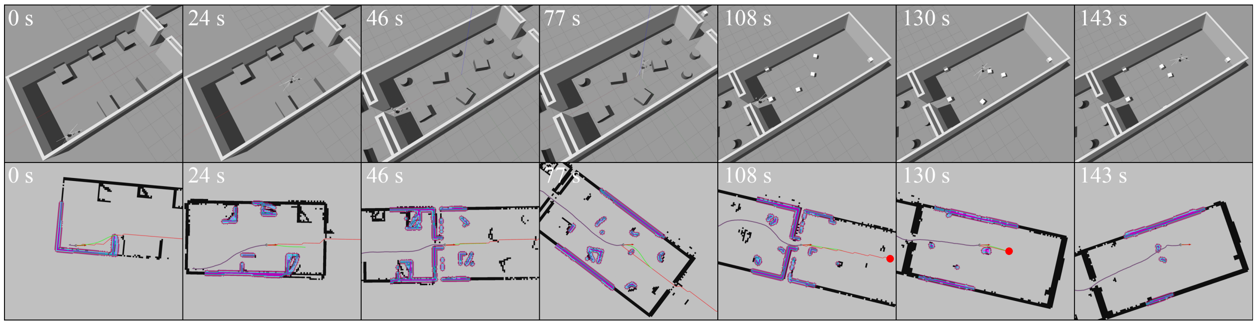

Figure 6.3: Results from experiment 6.1.3 with the top row showing the simulated world view and the bottom row showing the environment representation available to the robot. The red circle represents the goal position. Black grid cells are obstacles in the global map, purple and blue grid cells are obstacles and inflated cells in the local map, respectively. The red arrow is the current estimated odometry. The red path and green paths are the global path and local trajectory, respectively. The purple path shows the robot's traveled path. As the robot discovers new regions and obstacles in the unknown environment, the navigation system quickly reacts to obstacles obstructing the current path and trajectory. The navigation system is able to safely plan around the static obstacles in the first two rooms of the Gazebo world. As the robot encounters dynamic obstacles in the final room, the navigation system is able to safely navigate around the moving obstacles and successfully reaches the goal position.

6.1.3 Dynamic Obstacles, Large Unknown Environment

Experiment results are shown in Fig. 6.3. The Turtlebot3 is placed in the second Gazebo environment previously described, consisting of three rooms with static and dynamic obstacles. This task explores navigating over long distances in an unknown environment that contains dynamic obstacles. At $t = 0$, an initial goal position is selected and the robot begins to navigate towards this goal. At $t = 24$, the robot has successfully navigated around the first obstacle in front of the start position and has detected the connecting doorway to the next room ahead and begins to navigate towards this opening. By $t = 46$, the robot has successfully navigated through the doorway connecting the first two rooms and begins to plan around the visible obstacles interspersed in room two. At $t = 77$, the robot is safely navigating around the static obstacles in room two and is heading towards the detected doorway between rooms two and three. Once the robot passes through the doorway into room three at $t = 108$, the dynamic obstacles in the final room are detected. While navigating

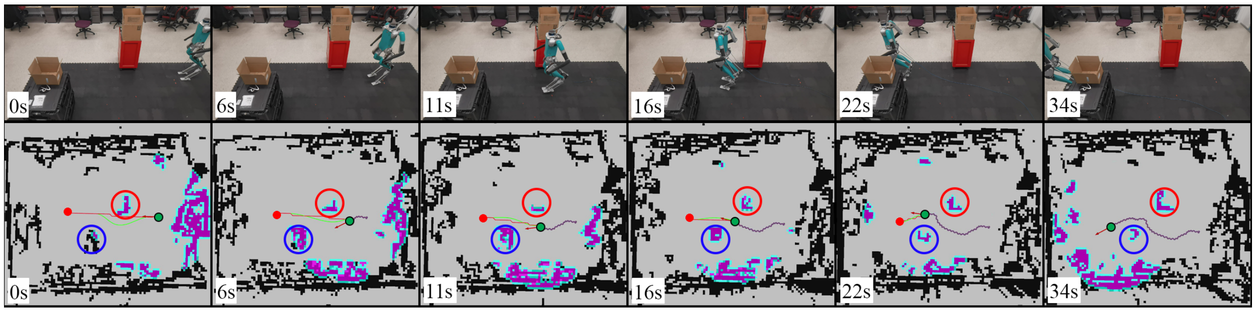

Figure 6.4: Results from experiment Section 6.2.1 with the top row showing the camera view of the environment and the bottom row showing the environment representation available to the robot. The solid red circle represents the goal position. The red and blue hollow circles represent the first and second obstacles, the second being a dynamic obstacle in Fig. 6.5. Black grid cells are obstacles in the global map, purple and blue grid cells are obstacles and inflated cells in the local map, respectively. The red arrow is the current estimated odometry, and the green circle shows the robots current position. The red path and green paths are the global path and local trajectory, respectively. The purple path shows the robot's traveled path. Digit is able to successfully navigate around static obstacles blocking the initial trajectory and safely reaches the goal position.

to the goal position, the robot is able to successfully navigate around the dynamic obstacles at $t = 130$ and reaches the goal position at $t = 143$. The successful completion of this final simulated experiment validates the navigation system's ability to successfully navigate over long distances, which is essential for autonomous operation in society.

6.2 Hardware Experiments

The navigation system computation is executed on an external desktop using an AMD Ryzen 5800X CPU and connected through a TCP connection to Digit. In this work, two hardware experiments were conducted indoors: 1) Planning in a static environment and 2) Planning in a dynamic environment. Both experiments are conducted in a 10.0 m x 8.0 m indoor room with different obstacles in each experiment. Each test is deemed successful once the Digit robot is able to navigate collision-free from the start position to the goal position within a 0.2 m final goal position tolerance and 0.2 rad final goal orientation tolerance.

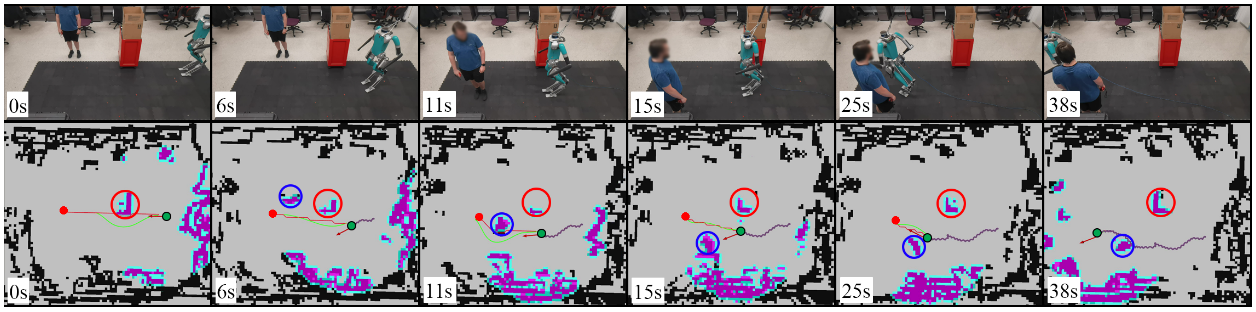

Figure 6.5: Results from experiment Section 6.2.1 with the top row showing the camera view of the environment and the bottom row showing the environment representation available to the robot. The solid red circle represents the goal position. The red and blue hollow circles represent the first and second obstacles, the second being a dynamic obstacle. Black grid cells are obstacles in the global map, purple and blue grid cells are obstacles and inflated cells in the local map, respectively. The red arrow is the current estimated odometry, and the green circle shows the robots current position. The red path and green paths are the global path and local trajectory, respectively. The purple path shows the robot's traveled path. During navigation, a dynamic obstacle is discovered and as the obstacle begins to move, the trajectory deforms to maintain a collision free path. The dynamic obstacle finally comes to a rest and Digit is able to successfully navigate around the obstacles and safely reach the goal position.

6.2.1 Static environment

As shown in Fig. 6.4, the 10.0 m $\times$ 8.0 m environment contains two static 0.5 m $\times$ 1.0 m and 1.0 m $\times$ 0.5 m obstacles. At $t = 0$ a goal position is given to Digit and the robot begins to rotate to safely avoid the first obstacle. Digit continues to travel forward between $t = 0$ and $t = 11$ at which point Digit begins to rotate to avoid the second obstacle. Digit continues along the generated trajectory until the robot safely reaches the goal position at $t = 34$. This experiment successfully validates the ability to adapt the navigation system in hardware and execute on a bipedal robot.

6.2.2 Dynamic environment

The second experiment is conducted in the same environment as the previous experiment, except the second static obstacle has been replaced with a dynamic obstacle. At $t = 0$ a goal position is given to Digit. Initially, the dynamic obstacle is not visible to Digit due to being obscured by the first obstacle. Digit begins to execute the generated trajectory and at $t = 6$ the dynamic obstacle is detected. At this moment, the dynamic obstacle begins to navigate across the currently planned trajectory. As the obstacle moves in the environment, the planned trajectory begins to deform to maintain a collision-free path, which can be seen at $t = 11$. Between $t = 11$ and $t = 15$, Digit begins to slow to a stop as the dynamic obstacle continues to move across the robots path. At $t = 15$, the obstacle comes to a rest and Digit is able to begin executing a collision-free path to the goal. Digit is able to successfully execute the trajectory between $t = 15$ and $t = 38$ and safely reaches the goal position at $t = 38$. Many environments that target mobile robot integration requires robust navigation in dynamic environments. The success of this experiment validates the robustness of the navigation system when executing on a bipedal robot and in these common dynamic environments.

CHAPTER 7

CONCLUSION

In this thesis a real-time navigation system for bipedal robots that is robust in complex, dynamic environments was developed. First, a depth-based sensor suite of three depth cameras and a LiDAR sensor was used to capture 3D point clouds of the environment. These point clouds are first processed to reduce the raw point cloud size, even out the density, and remove possible erroneous detections farther than 2.9 m from the robot and below the ground plane. After processing, Random Sample Consensus was used to segment obstacle points from ground points into separate point clouds. Using the sensor measurements from the LiDAR sensor, LiDAR Odometry and Mapping was used to estimate the odometry of the robot and generate a 3D point cloud map of the explored environment. This 3D point cloud map was not directly used for environment mapping, but instead the sensor detections were used to populate a global and local 2D cost map of the global and local environments.

Once robot odometry is estimated and the environment is mapped, a two-stage planner was used to generate collision-free, kinematically feasible trajectories. A D* Lite global planner was used to generate high level paths directly in the 2D cost map representation. This global planner is capable of re-planning in real-time and robust to a changing environment. The global path generated from this planner is then used by the Timed-Elastic-Band local planner as an initial estimate of the optimal trajectory through the environment. The local planner follows the global path and applies several constraints to generate an optimal trajectory that respects non-holonomic, velocity, acceleration and obstacle avoidance constraints. From this trajectory, translational and rotational velocities were calculated and executed using the default Digit bipedal robot walking gait controller provided by Agility Robotics.

While the proposed navigation system was shown to successfully execute in complex environments, there exist several limitations. The current sensor suite configuration results in blind spots on the left and right sides of the robot. This prevents the navigation system from quickly reacting to dynamic obstacles that move in these blind spots. In future work, a more robust sensor suite providing 360^o sensor coverage would allow for better dynamic obstacle reactions. Additionally, the use of 2D layered costmaps limits the navigable environments to only flat 2D terrains. While this is sufficient for many complex environments, it is often desirable to navigate through inclines, stairs, and uneven terrain. Using a mapping and planning system that can utilize 3D information would allow for more robust navigation in complex 3D terrains. Finally, although the local planner is able to generate robust trajectories, the assumption of differential drive kinematic constraints limits the capabilities of bipedal robots. An extension to omnidirectional locomotion would allow for better utilization of the unique capabilities of bipedal robots.

BIBLIOGRAPHY

[1] Léo Baudouin et al. “Real-time replanning using 3D environment for humanoid robot”. In: *2011 11th IEEE-RAS International Conference on Humanoid Robots*. IEEE. 2011, pp. 584–589.

[2] Khaled Belghith, Froduald Kabanza, and Leo Hartman. “Randomized path planning with preferences in highly complex dynamic environments”. In: *Robotica* 31.8 (2013), pp. 1195–1208. DOI: `10.1017/S0263574713000428`.

[3] J. Bruce and M. Veloso. “Real-time randomized path planning for robot navigation”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 3. 2002, 2383–2388 vol.3. DOI: `10.1109/IRDS.2002.1041624`.

[4] Joel Chestnutt et al. “An adaptive action model for legged navigation planning”. In: *2007 7th IEEE-RAS International Conference on Humanoid Robots*. IEEE. 2007, pp. 196–202.

[5] Sébastien Dalibard et al. “Dynamic walking and whole-body motion planning for humanoid robots: an integrated approach”. In: *The International Journal of Robotics Research* 32.9-10 (2013), pp. 1089–1103.

[6] D. Ferguson, N. Kalra, and A. Stentz. “Replanning with RRTs”. In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* 2006, pp. 1243–1248. DOI: `10.1109/ROBOT.2006.1641879`.

[7] Dave Ferguson, Maxim Likhachev, and Anthony Stentz. “A Guide to Heuristic-based Path Planning”. In: (Jan. 2005).

[8] Dave Ferguson and Anthony Stentz. “The Delayed D* Algorithm for Efficient Path Replanning”. In: vol. 2005. May 2005, pp. 2045–2050. DOI: 10.1109/ROBOT.2005.1570414.

[9] Martin A. Fischler and Robert C. Bolles. “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography”. en. In: *Communications of the ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/358669.358692.

[10] Johannes Garimort, Armin Hornung, and Maren Bennewitz. “Humanoid navigation with dynamic footstep plans”. In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 3982–3987.

[11] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[12] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed. New York, NY, USA: Cambridge University Press, 2003. ISBN: 0521540518.

[13] Arne-Christoph Hildebrandt et al. “Real-Time Path Planning in Unknown Environments for Bipedal Robots”. In: *IEEE Robotics and Automation Letters* 2.4 (2017), pp. 1856–1863. DOI: 10.1109/LRA.2017.2712650.

[14] Armin Hornung et al. “Anytime search-based footstep planning with suboptimality bounds”. In: *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*. IEEE. 2012, pp. 674–679.

[15] Weiwei Huang, Junggon Kim, and Christopher G Atkeson. “Energy-based optimal step planning for humanoids”. In: *2013 IEEE International Conference on Robotics and Automation*. IEEE. 2013, pp. 3124–3129.

[16] Dimitrios Kanoulas et al. “Footstep Planning in Rough Terrain for Bipedal Robots Using Curved Contact Patches”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 4662–4669. DOI: 10.1109/ICRA.2018.8460561.

[17] Sertac Karaman and Emilio Frazzoli. *Sampling-based Algorithms for Optimal Motion Planning*. 2011. DOI: `10.48550/ARXIV.1105.1186`. URL: `https://arxiv.org/abs/1105.1186`.

[18] L.E. Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. DOI: `10.1109/70.508439`.

[19] S. Koenig and M. Likhachev. "Incremental A∗". In: *Advances in Neural Information Processing Systems*. Ed. by T. Dietterich, S. Becker, and Z. Ghahramani. Vol. 14. MIT Press, 2001. URL: `https://proceedings.neurips.cc/paper/2001/file/a591024321c5e2bdbd23ed35f0574dde-Paper.pdf`.

[20] Sven Koenig and Maxim Likhachev. "D*Lite." In: Jan. 2002, pp. 476–483.

[21] Rainer Kümmerle et al. "G2o: A general framework for graph optimization". In: *2011 IEEE International Conference on Robotics and Automation* (2011), pp. 3607–3613.

[22] Steven M. LaValle. "Rapidly-exploring random trees : a new tool for path planning". In: *The annual research report* (1998).

[23] Zhongyu Li et al. *Vision-Aided Autonomous Navigation of Underactuated Bipedal Robots in Height-Constrained Environments*. 2021. eprint: `arXiv:2109.05714`.

[24] Maxim Likhachev, Geoffrey Gordon, and Sebastian Thrun. "ARA*: Anytime A* with Provable Bounds on Sub-Optimality". In: vol. 16. Jan. 2003.

[25] Maxim Likhachev et al. "Anytime Dynamic A*: An Anytime, Replanning Algorithm". In: *Proceedings of 15th International Conference on Automated Planning and Scheduling (ICAPS '05)*. June 2005, pp. 262–271.

[26] David V. Lu, Dave Hershberger, and William D. Smart. "Layered costmaps for context-sensitive navigation". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 709–715. DOI: `10.1109/IROS.2014.6942636`.

[27] Eitan Marder-Eppstein. *Wiki*. URL: `http://wiki.ros.org/move_base`.

[28] Philipp Michel et al. “Vision-guided humanoid footstep planning for dynamic environments”. In: *5th IEEE-RAS International Conference on Humanoid Robots, 2005.* IEEE. 2005, pp. 13–18.

[29] Richard M Murray, Zexiang Li, and S Shankar Sastry. *A mathematical introduction to robotic manipulation*. CRC press, 2017.

[30] Toshiya Nishi and Tomomichi Sugihara. “Motion planning of a humanoid robot in a complex environment using RRT and spatiotemporal post-processing techniques”. In: *International journal of humanoid robotics* 11.02 (2014), p. 1441003.

[31] Koichi Nishiwaki, Joel Chestnutt, and Satoshi Kagami. “Autonomous navigation of a humanoid robot over unknown rough terrain using a laser range sensor”. In: *The International Journal of Robotics Research* 31.11 (2012), pp. 1251–1262.

[32] C. Nissoux, T. Simeon, and J.-P. Laumond. “Visibility based probabilistic roadmaps”. In: *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No.99CH36289).* Vol. 3. 1999, 1316–1321 vol.3. DOI: 10.1109/IROS.1999.811662.

[33] Jorge Nocedal and Stephen J. Wright. *Numerical optimization: with 85 illustrations.* eng. Cham: Springer International Publishing. ISBN: 9780387227429.

[34] Nicolas Perrin et al. “Weakly collision-free paths for continuous humanoid footstep planning”. In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2011, pp. 4408–4413.

[35] Sebastian Pütz, Jorge Santos Simón, and Joachim Hertzberg. “Move Base Flex A Highly Flexible Navigation Framework for Mobile Robots”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 3416–3421. DOI: 10.1109/IROS.2018.8593829.

[36] Christoph Rösmann, Frank Hoffmann, and Torsten Bertram. “Integrated online trajectory planning and optimization in distinctive topologies”. In: *Robotics and Autonomous Systems* 88 (2017), pp. 142–153. ISSN: 0921-8890. DOI: `https://doi.org/10.1016/j.robot.2016.11.007`.

[37] Christoph Rösmann et al. “Trajectory modification considering dynamic constraints of autonomous robots”. In: *ROBOTIK*. 2012.

[38] Radu Bogdan Rusu and Steve Cousins. “3D is here: Point Cloud Library (PCL)”. In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 1–4. DOI: `10.1109/ICRA.2011.5980567`.

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

[40] Anthony Stentz. “The Focussed D* Algorithm for Real-Time Replanning”. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI’95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, pp. 1652–1659. ISBN: 1558603638.

[41] Ji Zhang and Sanjiv Singh. “Low-drift and Real-time Lidar Odometry and Mapping”. In: *Autonomous Robots* 41.2 (Feb. 2017), pp. 401–416.