Safe Navigation for Bipedal Robots in Static Environments

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By
Archit S. Rede, B.S.
Graduate Program in Mechanical Engineering

The Ohio State University
2021

Master's Examination Committee:
Prof. Ayonga Hereid, Advisor
Prof. Mrinal Kumar

ABSTRACT

Motion planning is a field with a large body of research, ranging from applications to industrial robotics to mobile robotics. A*, RRT, and RRT* are used as a basis for the preliminary results due to their versatility. In particular, A* provides a simple implementation that can be combined with other methods of local planning or upgraded with improved versions of A*. RRTs can be improved by modifying the local planner, sampling, and other parameters that may allow for asymptotic optimality as well as safety. Many methods have been proposed in recent and past literature that attempt to solve safe motion planning problems such as learning-based, grid-based, etc. The problem of safe bipedal navigation is based on the need to find a path or trajectory that allows for the center-of-mass (CoM) and the footsteps to be planned safely within the configuration space (C-space). The motion planning framework proposed will first use A*, RRT, and RRT* as a basis for solving the 2D path planning problem by implementing the algorithms on both toy datasets as well as occupancy maps. The map processing will allow the discrete occupancy map to be converted to a usable C-space for the chosen RRT* algorithm. Then the framework will simulate the robot, Digit, in the Agility Robotics (AR) simulator using a PID controller. The results showed that the map processing algorithm successfully creates a C-space. Additionally, initial safety results are provided for A* and RRTs, which show the strict safety and lenient safety, which is successfully implemented by modifying the input grid data. Finally, the PID results showed that the planned path can be followed relatively closely to the desired RRT* waypoint path. The next steps to improve the framework using chance-constrained path planning, hybrid A*, and possibly learning-based approaches.

Dedicated to my family, friends, fellow peers from the lab, and Dr. Hereid for supporting me through my journey in robotics and artificial intelligence.

ACKNOWLEDGEMENTS

I wanted acknowledge Dr. Ayonga Hereid, my advisor, for his support and help with my motion planning research. After coming back to Ohio State, Dr. Hereid was very welcoming to my diverse experiences and allowed me to find my passion for robotics and artificial intelligence. I also wanted to acknowledge Jack Beokhaimook, Guillermo Castillo, Victor Cauna Paredes as well as several undergraduate students for helping me throughout my research, both with their expertise as well as providing me the foundational tools required for me to complete my master's thesis. Finally, I wanted to acknowledge Dr. Rephael Wenger for providing me with some input on my research and helping me find a solution that helped my path planning project. I would also like to thank Dr. Mrinal Kumar for joining the thesis defense and allowing me to defend the work I have compiled together in this document. To my friends and family, thank you for all of the support throughout my graduate education.

VITA

May 2018 ..... B.S., Biomedical Engineering, The Ohio State University

Jun 2018 to Sep 2018 ..... Supply Chain Intern, SBP Consulting

Feb 2019 to Dec 2019 ..... Software Engineering Intern, Lincoln Electric Automation - Wolf Robotics

Aug 2020 to Nov 2020 ..... Biomedical Engineering Intern, Battelle Memorial Institute

Jan 2021 to May 2021 ..... Graduate Teaching Associate, Department of Mechanical Engineering, The Ohio State University

Fields of Study

Major Field: Mechanical Engineering

Table of Contents

List of Tables

List of Figures

List of Algorithms

CHAPTER 1

INTRODUCTION

Motion planning is a research topic that has significance in robotics perception, navigation, and autonomy. The problem of motion planning depends on a variety of components within the planning hierarchy. Many solutions to the motion planning problem require a good problem definition as well as a strong definition of what "motion" and "planning" are referring to in terms of the problem and its corresponding solutions. The main components of a planning problem are: 1) state definition, 2) action space, 3) configuration space, 4) task space, 5) environment definition, and 6) heuristics. State definition often refers to position, orientation, velocity, etc while environment definition refers to the type of environment, which may be static or dynamic, simulation or real-world, 2D or 3D, etc. The configuration space is the largest space that describes the start, goal, obstacle space, and free space. Task space is often a subspace of the configuration space that describes the region where a specific task might take place, and action space is the set of actions that can be taken within a given space. Heuristics are measurements of cost or rewards that might be useful in finding optimal plans. Some subtopics within motion planning specifically focus on generating feasible and optimal paths using algorithms that search across grid space and continuous space. Such planning problems are designated by "path" or "trajectory" planning. The focus of this paper is the application of an algorithm used in the grid space called A* as well as a continuous space algorithm using randomly-exploring tree methods such as RRT and RRT*. This section will provide some motivation and background for the topic of motion planning and current research in bipedal navigation as well as outline the rest of the thesis.

1.1 Motivation and Background

1.1.1 Brief History of Motion Planning

Before moving on to the recent works in bipedal navigation, a brief history of motion planning from the perspective of grid-based and sampling-based algorithms will be discussed. More specifically, A*, RRT, and RRT* will be discussed with some historical background alongside improvements made to these algorithms from surveys of grid-based and sampling-based algorithms. This section serves as an introduction to the algorithms used as a basis in the bipedal navigation framework discussed in a later chapter. First, A* and its successor algorithms will be discussed, and then, the RRT algorithms and its successor algorithms will be discussed in more detail.

A* is a graph search algorithm that can find the optimal path, which corresponds to the shortest path. The original authors in [13] discuss the A* algorithm approach as combining two approaches: (1) mathematical approaches that use graphs and their properties within algorithms used to find the minimum cost path and (2) heuristic approaches that use information about the problem domain to improve the computational efficiency of graph searching algorithms. A* is considered an admissible algorithm because it generally finds the optimal path for any graph. The proofs are shown in more detail in [13]. While the original authors focus on mathematical proof of optimality in this paper, the proposal of A* has made it a common algorithm to use as a basis for solving discrete motion planning problems. Many algorithms seek to improve on the limitations of A*. The authors in [31] discuss limitations and classes of algorithms that are inspired from A*. The main limitation of A* is that the nodes of the graph can be expanded exponentially in the length of the optimal path, which means that there is a storage limitation. The choice of the heuristic function determines the complexity of A* and its descendants. Based on the book, there are five classes of algorithms inspired by A* that seek to improve those limitations: incremental, memory-concerned, parallel, anytime, and real-time [31]. Incremental methods tend to improve the plan over time using previous search information. Some examples include

Fringe-Saving A*, Lifelong Planning A* (LPA*), D*, and D* Lite, which are described in more detail in [35],[20],[34], and [18], respectively. Out of these algorithms, D* Lite is commonly used for robot path planning in dynamic environments. Memory-concerned methods try to improve the use of the storage methods in A* with the closed and open lists. Examples include iterative deepening A* and Frontier A* as described in [22] and [21], respectively. Parallel class of A* inspired algorithms attempts improves A* to make it useful for parallel execution. An example is the Parallel Retracting A* (PRA*) as described in more detail in [10]. Anytime class algorithms tend to improve A* in that they can improve computation time, trading shortest path optimality for improved computation time. Anytime Dynamic (AD*) is an example of an anytime algorithm as described in [26]. Real-time A*-based algorithms seek to search with time-based constraints. An example is Real-Time Adaptive A* [19]. Another extension of A* is called Hybrid A* as described in [8] which uses a modified A* applied to 3D kinematic state space with a state-update rule that finds the continuous state within each of the discrete nodes of A*. Additionally, hybrid A* improves the path using numerical nonlinear optimization, allowing for locally and sometimes globally optimal paths.

Rapidly-exploring random trees (RRT) are a class of sampling algorithms that solve the navigation problem by finding feasible and safe paths. The original authors describe the properties of RRT in [30]. The main advantages of RRT described in the paper are the following:

1. Expansion of an RRT is biased towards unexplored regions of the state or configuration space.

2. The distribution of nodes in the tree approaches the sample distribution, which allows for more consistent behavior.

3. RRT is probabilistically complete.

4. RRT is simple, which makes it easier for analyzing performance.

5. RRT is always connected, even if there are a minimal number of edges.

6. RRT can solve path planning but can also be modified and incorporated in larger planning frameworks.

7. Path planning algorithms can be built with RRT as a basis, which reduces the need to include some sort of steering between two states or configurations. This allows RRT to be more applicable to solving motion planning problems.

These properties of RRTs allow them to be a common basis for or integration with other planning algorithms [30]. RRT* attempts to improve this algorithm by developing shorter paths, which is described in the following by the authors in [16]. RRT* tends to overcome the limitations of RRT's lack of asymptotic optimality. Other methods involving RRTs have been developed and are discussed closely in a survey of sampling-based planning techniques [9]. A couple of other examples are kinodynamic RRT and RRT-Connect. RRT-Connect uses bidirectional tree search while kinodynamic integrates kinematic and dynamic constraints with RRT. Generally, RRT can be improved from the perspective of sampling, guided exploration, metrics, collision detection, heuristics, post-processing, and local planning, which means that many algorithms can improve the algorithm in certain conditions [9]. All three algorithms output non-smooth, continuous, and feasible paths with the key differences being in approaches, and RRT* and A* are considered proven to have optimality in the shortest path sense. In the case of RRT, RRT does not provide optimal paths as mentioned earlier. Hybrid A* allows for smoother paths, which is not the current goal of the thesis project. However, for future works, it may be useful to consider hybrid A* as an alternative. Note that because there are many ways to improve both A* and RRTs, for solving navigation problems it makes more sense to start with A*, RRT, or RRT* as a basis before improving the framework. Because of the advantages of RRTs and A* algorithms, many algorithms have been developed to improve their limitations. More specific discussion on the algorithms from a pseudocode and implementation perspective will be done in Chapter 3.

1.1.2 State-of-the Art in Robot Motion Planning and Legged Robot Navigation

Within the field of study of robot motion planning, many methods attempt to solve the problem of safe navigation. Before discussing methods used in bipedal navigation, some recent works in motion planning will be discussed. Some classical algorithms for path planning were discussed in the last section with the focus on A* and RRTs as a basis. The authors in [41] survey methods for optimal motion planning, discussing the importance of appropriate problem definition for finding optimal motion plans. Optimality depends on the optimization problem defined. In other words, a well-defined optimal motion planning problem can be solved to find anything from the shortest path to time-optimal planning. Algorithm classes that tend to be able to solve for optimal plans include (1): grid-based methods that are optimal, (2) sampling-based methods that are asymptotically optimal, and (3) trajectory optimization methods. In [41], the authors describe obstacle representations as important to optimal motion planning methodologies in addition to considering how different methods approach obstacle representation. Additionally, many planning problems require additional constraints such as dynamics, kinematics, and other more general constraints outside of obstacle avoidance. A comparison of optimal planning methods is important to understand why one algorithm class may be more useful for solving a given motion planning problem. Similarly, newer studies are surveyed such as the one conducted by the authors in [39]. The focus of this survey is on learning-based robot motion planning. Since learning is the solution considered to the motion planning problem, there are three classes of learning-based motion planners: (1) supervised learning, (2) unsupervised learning, and (3) reinforcement learning. Under supervised learning planners, the authors mention data-driven end-to-end motion plannings for autonomous ground robots, which use expert information from simulated environments. The method requires Convolutional Neural Networks (CNN) that retrieve information from input data from lasers. Another study was mentioned that describes using Deep Neural Networks (DNN) and lidar scans to predict speed and direction for robots [11]. Recurrent Neural Networks (RNN) have also been proposed to solve motion planning problems for static and near-optimal planned paths, which was discussed in the survey [1].

Unsupervised learning methods do exist but are less common. One example discussed in the survey was called PROM-Net, which makes visual predictions for robot motions from raw video frames. Because it is unsupervised, the methodology is lightweight and easy to implement [32]. Reinforcement learning (RL) methodologies such as Dyna-PI described in [36] and Deep Q Network combined with artificial potential fields described in [42], and policy gradient was also discussed as solutions to solving motion planning problems. Two types of RL methods exist: (1) value-based and (2) policy-based [39]. The importance of integrating learning into existing motion planning algorithms or as part of a larger framework of motion planning algorithms is evident from the number of methodologies studied in the survey. In addition to optimality, learnability appears to be an important direction for improving motion planning solutions.

Another paper described by the authors in [29] focuses on methodologies for robot motion planning that involve dynamic environments. Within the scope of the paper, RRT, PRM, and anytime algorithms were discussed as potential solutions. The paper argues that the shift in motion planning in dynamic environments has shifted from classical search to heuristic approaches [29]. Another paper discusses navigation for legged robots, describing the problem as solving two different problems: (1) motion before footsteps and (2) footsteps before motion [14]. The work describes designing a motion planner that chooses foot contacts and stances before finding the motions due to the constraints involved in legged motion on irregular terrains. The authors in [40] discuss using RRT* as part of a hierarchical framework that allows the robot to replan the path as new terrain is perceived by the robot's sensors. This study was done on a quadruped robot. Another paper discusses a series of studies done in human-aware robot navigation [23]. In the context of human-robot interaction, natural motions under social constraints become an important problem in motion planning. While social constraints are not important in the context of designing an initial planning framework, it is important to understand how the robot will navigate in a socially acceptable way in real environments where humans may interact with it. The authors in [38] reviews methods for global path planning using occupancy grids, mentioning Probabilistic

Roadmaps (PRMs), Visibility Graphs (VGs), and RRTs as common methods for solving motion planning problems on an occupancy map. The paper also mentions using algorithms such as Space Skeletonization as part of the map processing for use in motion planning [38].

An approach to obstacle avoidance is chance-constrained path planning. The authors in [2] discuss a method of chance-constrained optimal path planning that uses a probabilistic representation of uncertainty on an aircraft. In another paper, the work described in [5], a hybrid of a polyhedron and nonlinear differentiable surface definitions of safe bounds is used for real-time, chance-constrained motion planning on an aerial vehicle. An approach described in [27] uses a chance-constrained RRT for robust path planning in uncertain environments. Authors of the paper described in [3] discuss another approach to aircraft and ground vehicle control involving chance-constrained stochastic control. This body of work implies that chance-based constraints appear to be important representations for obstacle avoidance. However, for implementation and integration of planning algorithms used with the bipedal robot, Digit, the obstacle avoidance uses a brute-force method that is easier to implement initially. However, future works may require leveraging chance-constrained methods for obstacle avoidance.

Some recent studies related to bipedal navigation and motion planning are from [43]. The study uses a robot called Cassie, which is a robot with just a pelvis and two legs. The approach taken in this paper is an inspiration for the work described in the thesis project. The motion planning framework is based on a hierarchy that involves RRT combined with Discrete-Time Control Barrier Functions (DCBFs) and Model Predictive Control (MPC) as well as another method which is an information-theoretic approach called SAFE-IIG. The approach decouples the pelvis trajectory from the discrete footsteps by modeling the robot as a linear inverted pendulum [37]. For more information about DCBFs and MPC combination, another study is conducted by authors in [43], which inspired the paper on safe navigation for Cassie discussed earlier. Another paper discusses vision-aided autonomous navigation with height constraints using Cassie, modeling the leg dynamics as vertically-actuated Spring-Loaded Inverted Pendulum Model (vSLIP) [25].

With this large body of work, the motivation for the thesis project comes from designing a new framework for a robot called Digit that is described in Chapter 2. There is a large body of work done in safe motion planning, and safe bipedal robot navigation requires more results and implementations on hardware. Many of the approaches discussed for solving motion planning and bipedal navigation use algorithms that tend to be complex and address specific improvements over other methods. While many discuss implementations on hardware, one drawback of complex algorithms is that the approaches may be for a specific robot model from the hardware perspective. Another drawback of these algorithms is that the complexity usually requires a very well-defined problem and a well-defined solution. In the case of optimal paths, a well-defined optimization problem is required whereas learning requires a learning problem to be solved in the context of motion planning. General planning algorithms like A*, RRT, and RRT* are a good basis for starting a navigation framework for a specific robot model. In particular, the problem definition and approaches discussed in Chapter 2 and Chapter 3 seek to define the robot framework from scratch for the robot model used in the simulations. This chapter first introduced A* and RRTs as a basis for future navigation frameworks, which is important due to the choice of using simple algorithms for designing a working framework for implementation in simulation and on real hardware. While hardware experiments were not conducted in time for the preliminary results of the navigation framework available for Digit, the software framework is ready for implementation on hardware experiments as part of future works. Additionally, current works have varied approaches that may be time-consuming from learning, information-theoretic, and optimization perspective for testing initially with Digit, so the framework will likely work with more complex navigation algorithms once tested well with the simple algorithms used in the body of work described in this thesis.

1.2 Main Contributions

The main contribution of the body of work on this thesis is the integration of a simple motion planning framework, consisting of A*, RRT, and RRT* path planners with occupancy map data and simulation for a bipedal robot called Digit. Other smaller contributions include

adding small modifications to the traditional A*, RRT, and RRT* as a safety measure for path planning as well as using simple image analysis techniques to convert a discrete grid space to a continuous 2D space. The final contribution for simulation was to integrate a PID controller for in-simulation robot movement using velocity commands. Because of these main contributions, the algorithms provided in the motion planning framework are now ready to test on hardware. At The Ohio State University, this motion planning framework is the first available for Digit with the possibility to continuously improve and push for stronger and more original motion planning frameworks for safe bipedal navigation.

1.3 Outline of Thesis

This section will focus on outlining the rest of the thesis chapter by chapter.

Chapter 2 will focus on developing a mathematical formulation of bipedal safe navigation. The emphasis on the problem definition will be on 2D path planning and extensions into higher dimensional planning. The robot utilized in the bipedal navigation problem solving is also described. Finally, the current state of the motion planning framework for bipedal navigation will be discussed in conjunction with the mathematical formulations and the tools involved.

Chapter 3 will focus on the algorithms used to solve the bipedal safe navigation problem. The emphasis will be on 2D navigation using A* and RRT approaches to plan paths in discrete and continuous spaces, respectively. The map processing technique will also be discussed as an algorithm for converting the discrete grid space into a continuous planning space. Finally, this section will discuss an implementation of a PID controller that allows the bipedal robot to walk along a safe path in the Agility Robotics simulator.

Chapter 4 will focus on the results of the approaches discussed in Chapter 3. Preliminary results from A* and RRT will be discussed. RRT* will be the focus of algorithmic results as the basis of the motion planning framework for safe bipedal navigation. The results from the mapping and PID controller will also be discussed in this section. There will be a mix of figures that were outputs of ROS, RVIZ, Agility Robotics, MATLAB, and Python that will be used for the analysis of the results.

Chapter 5 will discuss the conclusions from the approaches used for safe navigation as well as some insights from future work and ideas that might help improve the current state of the motion planning framework for safe bipedal navigation. Additionally, some unanswered questions will be discussed as preparation for new research insights or questions that came from pursuing this project.

CHAPTER 2

Problem Formulation

2.1 Motion Planning Problem

Motion planning is a field of study that has diverse applications in academia and industry. However, problem definition and mathematical formulation are required prior to discussing the approaches that are used to solve the problem. Current research in bipedal robotics, as discussed in the Chapter 1, focuses on hierarchical planning frameworks to separate footstep planning and center of mass (CoM) trajectory or path planning. While this section will discuss the necessity for hierarchical planning, many of the results discussed later will focus on 2D path planning for the CoM and implementation on Digit. The results shown in Chapter 4 will emphasize building the initial framework that will eventually lead to integration with safe footstep planning. In this study, the focus of the motion planning problem will focus on safe navigation using 2D path planning and map processing techniques. This section will first discuss the necessary mathematical definitions from topology, geometry, and metric spaces for describing the safe navigation problem. After providing the mathematical foundation for the safe navigation problem definition in the context of bipedal robotics, the robot utilized in simulation, Digit from Agility Robotics, will be discussed in detail. Finally, the software framework for the current state of the motion planning framework will also be discussed.

2.1.1 Topology and Metric Spaces

Topology and geometry are important topics within the field of robotics. Mathematical definitions utilized in robotics depend on the definitions used in these branches of higher

mathematics. We will first define topological spaces, manifolds, and metric spaces that are required for understanding the mathematical formulation of safe navigation.

Definition 2.1 ([24])**.** A set X is called a topological space if there is a collection of subsets of X called open sets for which the following axioms hold:

1. The union of any number of open sets is an open set.
2. The intersection of a finite number of open sets is an open set.
3. Both X and $\emptyset$ are open sets.

Now suppose that X is a topological space. A subset $C \subset X$ is defined to be a *closed set* if and only if $X \setminus C$ is an open set. An example of a closed set is a closed interval such as $[0, 2]$. An open set is the complement of the closed set, which for the example given is $(-\infty, 0) \cup (2, \infty)$. However, it should be noted that not all intervals can be considered closed or open since its complement consists of both open and closed intervals. In the context of topology, any topological space X and $\emptyset$ are both open and closed [24],[6].

Definition 2.2 ([24])**.** For a given topological space $M \subseteq \mathbb{R}^n$, that space is called a manifold if for every $x \in M$ an open set $O \subset M$ exists such that:

1. $x \in O$.
2. O is homeomorphic to $\mathbb{R}^n$.
3. n is fixed for all $x \in M$,

where n is the dimension of M.

When a topological space is said to be homeomorphic to another topological space, within the field of topology, those two spaces are considered to be topologically equivalent. An example would be a torus that is topologically homeomorphic to a coffee mug. Two topological spaces that are homeomorphic to each other cannot be cut or pasted into the other. Examples of a 1D manifold include the unit circle defined by $\mathbb{S}^1 = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$

and $\mathbb{R}$. Examples of 2D manifolds include $\mathbb{R}^2$, which represents a plane, and $\mathbb{S}^1 \times \mathbb{S}^1$, which represents a torus [24].

When considering a path in motion planning, topological definitions should be used to maintain consistency with related areas in robotics and motion planning. Let us formally define a path as a continuous function mapping as follows:

Definition 2.3 ([24])**.** Let X be a topological space that is also a manifold. A path can be defined as a mapping $\tau : [0, 1] \mapsto X$.

Connectivity is also an important concept to understand within the scope of paths and topology. A topological space X is considered to be *connected* if and only if it cannot be represented as the union of two disjoint, non-empty, open sets. A topological space is considered to be *path connected* if for all $x_1, x_2 \in X$, there exists a path τ such that $\tau(0) = x_1$ and $\tau(1) = x_2$ [24].

Two other things to consider mathematically within motion planning include metric spaces. To maintain mathematical consistency, a metric space is defined as:

Definition 2.4 ([24])**.** (X, ρ) is a topological space X equipped with a function $\rho : X \times X \mapsto \mathbb{R}$ such that for any $a, b, c \in X$:

1. Nonnegativity: $\rho(a, b) \geq 0$.
2. Reflexivity: $\rho(a, b) = 0$ if and only if $a = b$.
3. Symmetry: $\rho(a, b) = \rho(b, a)$.
4. Triangle inequality: $\rho(a, b) + \rho(b, c) \geq \rho(a, c)$.

Common metric spaces used in motion planning include L_1, L_2, and L_∞. The most common are the first two, representing Manhattan and Euclidean metric spaces, respectively. For cases where orientation is involved as part of the planned path, there are more specific metrics, likely used for higher-dimensional planning [24].

With these formal definitions, we can now formalize the mathematical definitions associated with the configuration space (C-space) that is used commonly in robotics. After specifically defining the configuration space and the geometry utilized in robotics, the bipedal safe navigation problem can be well defined and solved [24],[28],[6].

2.1.2 Configuration Space and Geometry

The configuration space or C-space C is a topological space and a manifold as described in the previous section. The topological importance of the C-space is how it is used to describe the environment in which an agent may act to solve a navigation problem. Within the C-space, there are more subsets that describe free space and obstacle spaces. Within the configuration space, each possible configuration $q \in C$ is defined by the minimum required number of coordinates to define the state space.

Formally, free space C_{free} is a topological space such that $C_{free} \subseteq C$. The obstacles space C_{obs} is defined as a topological space such that $C_{obs} \subseteq C$ and $C \setminus C_{free}$. Examples of the C-space are manifolds such as $\mathbb{R}^2$ for a point mass, describing (x, y), and $\mathbb{R}^2 \times \mathbb{S}^1$ for a 2D rigid body, describing (x, y, θ). Both of these C-spaces are important in robotics from the perspective of planning. The first one based on $\mathbb{R}^2$ describes a point mass robot whose orientation does not matter for creating safely planned paths. The C-space of the 2D rigid body describes a robot with a definite geometry that depends on position and orientation when planning safe paths [24], [28].

Additionally, geometry is an important concept to consider from the perspective of Lie algebra groups from which rigid transformations and rotations can be defined in the context of robotics. Note that the C-space is separate from the environment in which the robot is planning. To make the distinction, the world will be defined as W that consists of obstacles $o \in O$ and robots $a \in A$, where $O, A \subseteq W$ [24]. Free space F is the part of W that excludes O and A but using the world space notations is not useful in the context of most planning solutions.

Formally, let h_a, h_o be mappings such that: $h_a : A \mapsto W$ and $h_o : O \mapsto W$. Such mappings allow for defining obstacles and robots as either point masses or as rigid bodies

with a shape, usually polygonal [24]. A common world may be defined as an occupancy or grid map that can be processed into a configuration space, which will be discussed further in the motion planning framework section. Now, for a quick introduction to common Lie algebra groups used in robotics for rigid body transformations consist of special orthogonal and special euclidean groups. Rotation matrices are usually defined by special orthogonal groups while transformation matrices are defined by special euclidean groups.

Definition 2.5 ([28])**.** A rotation matrix R is part of Lie algebra group called $SO(n)$, which is the set of all $n \times n$ rotation matrices that satisfy (1) $R^T R = I$ and (2) $\det R = 1$.

Definition 2.6 ([28])**.** A homogeneous transformation matrix is defined by $SE(n)$ in the form of $(n+1) \times (n+1)$ matrix T:

$$T = \begin{bmatrix} \mathrm{R} & \mathrm{p} \\ 0 & 1 \end{bmatrix} \tag{2.1}$$

Transformation matrix is defined by: $R \in SO(n)$ and n-dimensional point $p \in \mathbb{R}^n$. Most commonly, n is usually 2 or 3 for rigid body rotations and transformations in $\mathbb{R}^2$ or $\mathbb{R}^3$. Some common rotation matrices are based on Euler angles, roll-pitch-yaw, and quaternions. The discussion of the properties of the transformation and rotation matrices as well as their role in transforming other matrices is beyond the scope of the problem definition. Now we have enough mathematical background to define the safe bipedal navigation problem [28].

2.1.3 Safe Bipedal Navigation Problem Definition

With the mathematics formally defined in the previous sections, the problem of safe bipedal navigation can be defined more precisely. The problem formulation for the general motion planning problem is stated as follows:

Definition 2.7 ([24])**.** The formal definition for a general feasible motion planning problem.

1. Let X be a nonempty state space, which can be finite, countably infinite, or uncountably infinite.

2. Then for each state $x \in X$, an finite or infinite action space $U(x)$ is defined.

3. A state transition function f that produces a state $f(x, u) \in X$ for every $x \in X$ and $u \in U(x)$. The state transition equation is derived from f as $\dot{x} = f(x, u)$.

4. An initial state $x_I \in X$.

5. A goal set $X_G \subset X$.

Given a C-space C, the problem is defined as finding some path τ in C_{free} such that $\tau(0) = x_I$ and $\tau(1) \in X_G$.

The general problem definition describes the motion planning problem for both discrete and continuous spaces. If a path is time-varying, then it is called a trajectory. For safe navigation, another set called $C_{safe} \subseteq C_{free}$ may be useful if safety is desired over optimality. C_{safe} represents the set of all safe configurations or states that are a subset of the free space. If the safety set is a proper subset, then the safety set is considered strictly safe. Otherwise, the safety set is leniently safe. For real robotic systems, C_{safe} may be more useful, especially in unknown environments or in real-time applications as it can plan far enough away from obstacle-based constraints. If the plan requires optimality, then the user of the planning algorithm must define a well-defined optimization problem for their use case and then define a solution that completely solves that well-defined problem. Within the context of the safe bipedal navigation, there are two subsets of the safety set $C_{safe} = C_{footstep} \cup C_{CoM}$, meaning that the planning is hierarchical and seeks to find both safe footsteps in the safety set as well as safe center-of-mass (CoM) trajectory. If the C-space is not readily available, then a functional mapping is required to map points from the world, including the robot and obstacles, to the C-space. A motion planning problem becomes a navigation problem when the robot needs to process input perception data to plan paths or trajectories within the world. Normally that involves finding the C-space and then mapping the path back to the world after finding the solution from the implementation of a planner.

2.2 Digit Robot Description

2.2.1 Digit Specifications

Digit is a bipedal robot that was designed and built by Agility Robotics. The robot comes with 1 lidar, 1 RGB depth camera, 3 monochrome depth cameras, 1 GPS, 1 inertial mass unit (IMU), 20 actuated joints, and 30 degrees of freedom. The robot's design also includes an industrial computer for processing program inputs for use in different environments. The leg model is based on Cassie, which was the robot model before Digit. There are some springs that allow for passive dynamics alongside the actuated joints. Because bipedal locomotion requires a continuous leg swing and discrete foot contact dynamics, bipedal locomotion is inherently a hybrid dynamical system. The figure below shows the joints of the robot as shown in Figure 2.1 below. The robot also comes with a gamepad controller, which in conjunction with the default controller for the robot can be used to move the robot like a remote control [4].

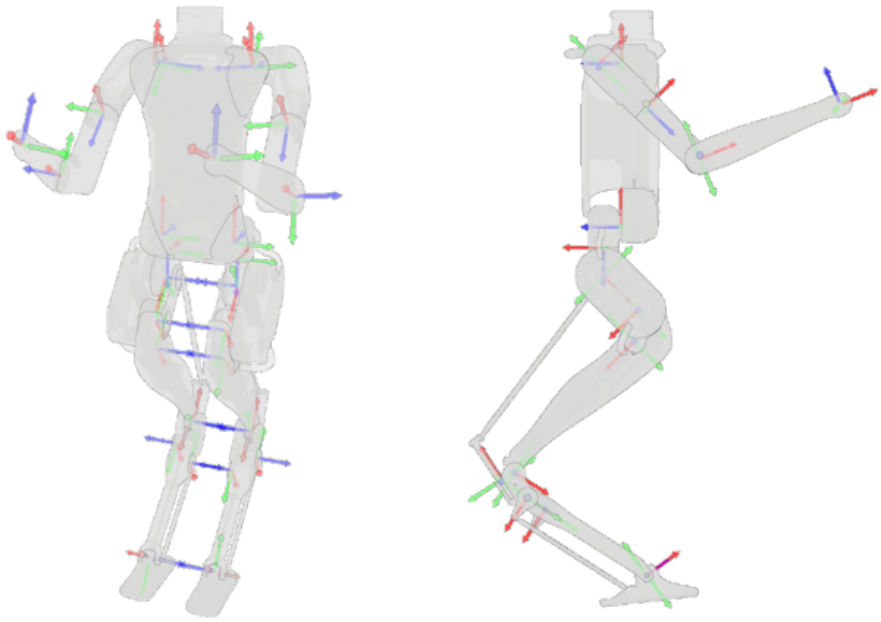


Figure 2.1: Model of Digit from Agility Robotics.

2.2.2 Simplifying Assumptions

Some assumptions are needed to consider the approaches described in Chapter 3. Firstly, the robot model for bipedal navigation has its pelvis kinematics decoupled from its leg dynamics for the purposes of planning. Another assumption is that the planning algorithms used to solve the path planning problem, A*, RRT, and RRT*, do not use a state transition function. Another assumption specific to the environment is obstacles are static for the purposes of initial results with path planning. The robot is always modeled as a point mass for path planning and as a point mass robot for path planning for the CoM. An additional assumption is that a cluster of obstacles, as well as single points, will be represented as circles for continuous planning and as discrete square grids for discrete planning. Safety is assumed to mean avoiding obstacles with CoM only since the footstep planning is outside the scope of this thesis project. An assumption is also required for the task space and the configuration space. Since the task is global path planning, the task space is assumed to be the same as the larger configuration space. Chosen algorithms for preliminary results will be considered for their simplicity so that a larger navigation framework can be developed that integrates or adds to these simple algorithms. Motion planning in the context of this thesis project is mainly path planning. Another assumption is that the C-space will always be converted from input from the robot's vision sensors. An assumption on safety is that the robot avoids narrow spaces for planning, at least in the context of bipedal navigation and control. Also, control of the legs through dynamics is considered a separate step to achieve full-bipedal navigation and requires the CoM to find safe footsteps, hence being a control problem that is outside the scope of the thesis project.

2.2.3 Simplified Models

The simplified models that are considered for the robot are purely based on geometrical models as stated in the mathematical definitions. Two simplified models are considered for this robot. The model used for the 2D navigation is the point mass robot, model. This model assumes that the robot only has (x, y) positions. From the perspective of path planning and

search, this model is one of the simplest models and is often used as a precursor to kinematic and dynamic models. When considering the mathematical definition of the robot as $a \in A$, the mapping is a trivial mapping where $h_a : \mathbb{R}^2 \mapsto \mathbb{R}^2$. Similarly, the mapping from the world W to the C-space $h_c : W \mapsto C$ will contain the same position information of the robot. The second model is another simplified model that depends on omnidirectional robot kinematics. However, for the context of the available controllers, omnidirectional kinematics is not considered in the current state of the path planning framework. The set of nonlinear equations are described by Equation 2.2:

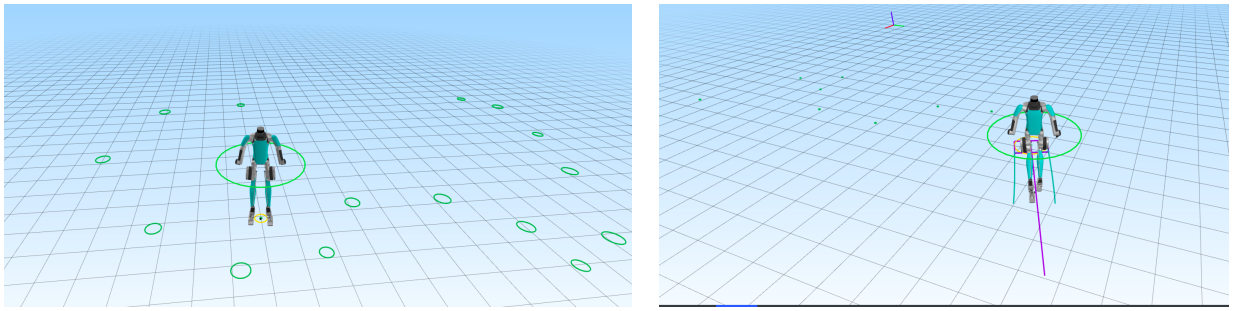
$$\begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ \omega \end{bmatrix} \tag{2.2}$$

θ is a heading angle computed as shown in the PID controller in Section 3.3. v_0 is the initial velocity of the robot that is user-defined. v_x, v_y, ω are the particle velocities computed from the PID controller. These velocities are based on particle kinematics, which is the simplified model used currently for the path planning framework. Simplified models are used so that the path planning framework has some framework to start with respect the safe path planning prior to planning safe trajectories. Although the robot is treated like a particle, it should be noted that the planned paths are for the CoM. The position information is updated using the default controller from Agility Robotics using the "moveto" JSON command as described in Section 2.2.4.

2.2.4 Agility Robotics Simulator

The Agility Robotics (AR) simulator is built using the MuJoCo physics simulator. The AR simulator can be started from the command line using a standard Linux command "./ar-control world.toml". The first argument starts the simulator while the second one creates a world for the robot to move in the AR simulator. The simulator can be interacted with through a website at "localhost:8080", which comes with AR simulator documentation, JSON command information, and some interfaces for sending sample JSON commands. The

simulator at the localhost site is the same one that can be accessed through the gamepad controller and has the same functionalities. The most relevant command and interactions that are available are setting waypoints, goto commands, and moveto commands. The "goto" and "moveto" commands work differently in that "goto" uses the desired position that allows the default robot controller, which is Agility Robotics' robust model-based controller, to go to the desired position. The "moveto" command sends velocity commands to allow the default controller to walk with a user-defined desired linear and angular velocity. Any commands sent to the simulator are in JSON format specified by the Agility Robotics documentation. In addition, through the default controller and the simulator, the robot has some default gestures it can perform as part of a "user-friendly" design. Figure 2.2 shows the robot in the AR simulator below. The circles in Figure 2.2a are selected waypoints, and the green dots are the waypoints as the robot is moving to them as shown in Figure 2.2b. The JSON command composer is shown in Figure 2.3.



(a) Digit in AR Simulator. (b) Digit moving from waypoint to waypoint.

Figure 2.2: AR Simulator with Digit in the environment.

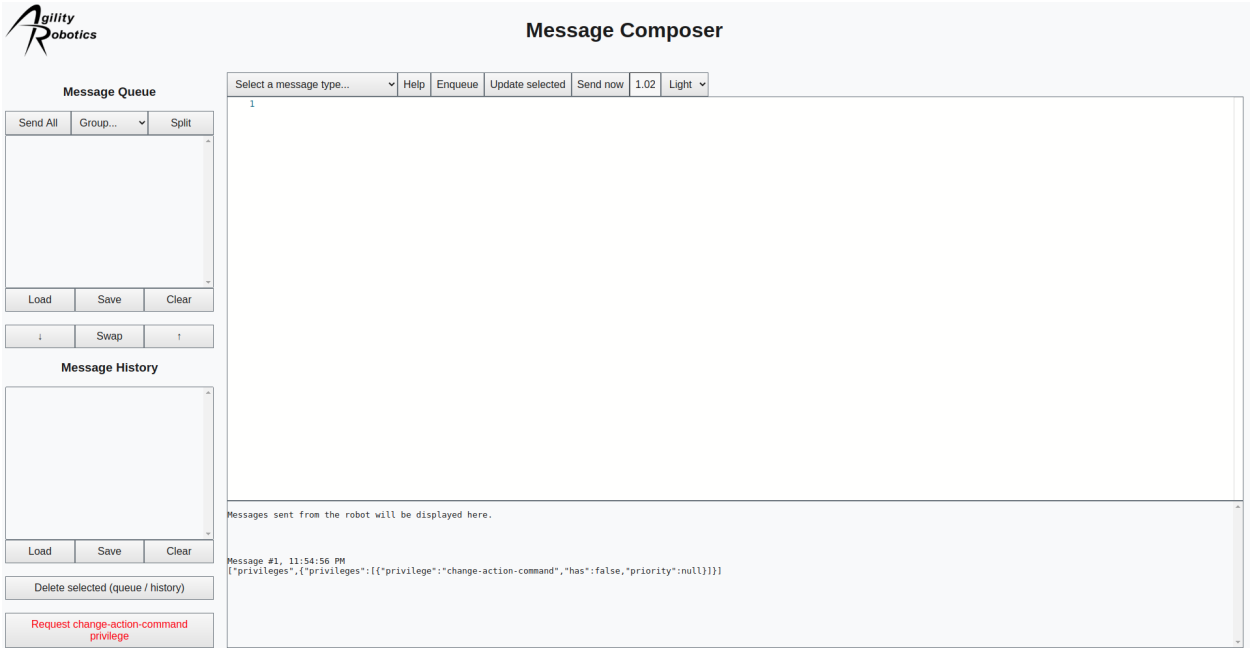


Figure 2.3: Message composer for JSON commands.

2.3 Motion Planning Framework

2.3.1 Robot Operating System (ROS)

Robot Operating System (ROS) is a software framework that is designed for communication between hardware or simulation and software implementations for solving robotics problems. The general framework includes subscribing and publishing topics for receiving and sending data from the robot through sensor processing. Data is sent through ROS messages for different modules such as Navigation, Mapping, etc. The use of ROS for this project was mainly accessing data for receiving an occupancy map that was built through OctoMap mapping techniques. After the vision data is retrieved, the map is processed in MATLAB for the path planning algorithm. After saving the planned path as a CSV file, the data is imported into Python for use in simulation and visualization in the AR simulator and RVIZ, respectively. A PID controller is implemented to send the robot to its desired waypoints from the planned paths in real-time. Once the simulation is over, the simulation data is saved locally for analysis. A WebSocket node was used to access ROS and publish desired path

and the simulated PID path that was sent through the controller described in Chapter 3 as well as send JSON commands to the AR simulator while simultaneously requesting the base kinematics from the robot. In Figure 2.4, the available motion planning framework can be seen.

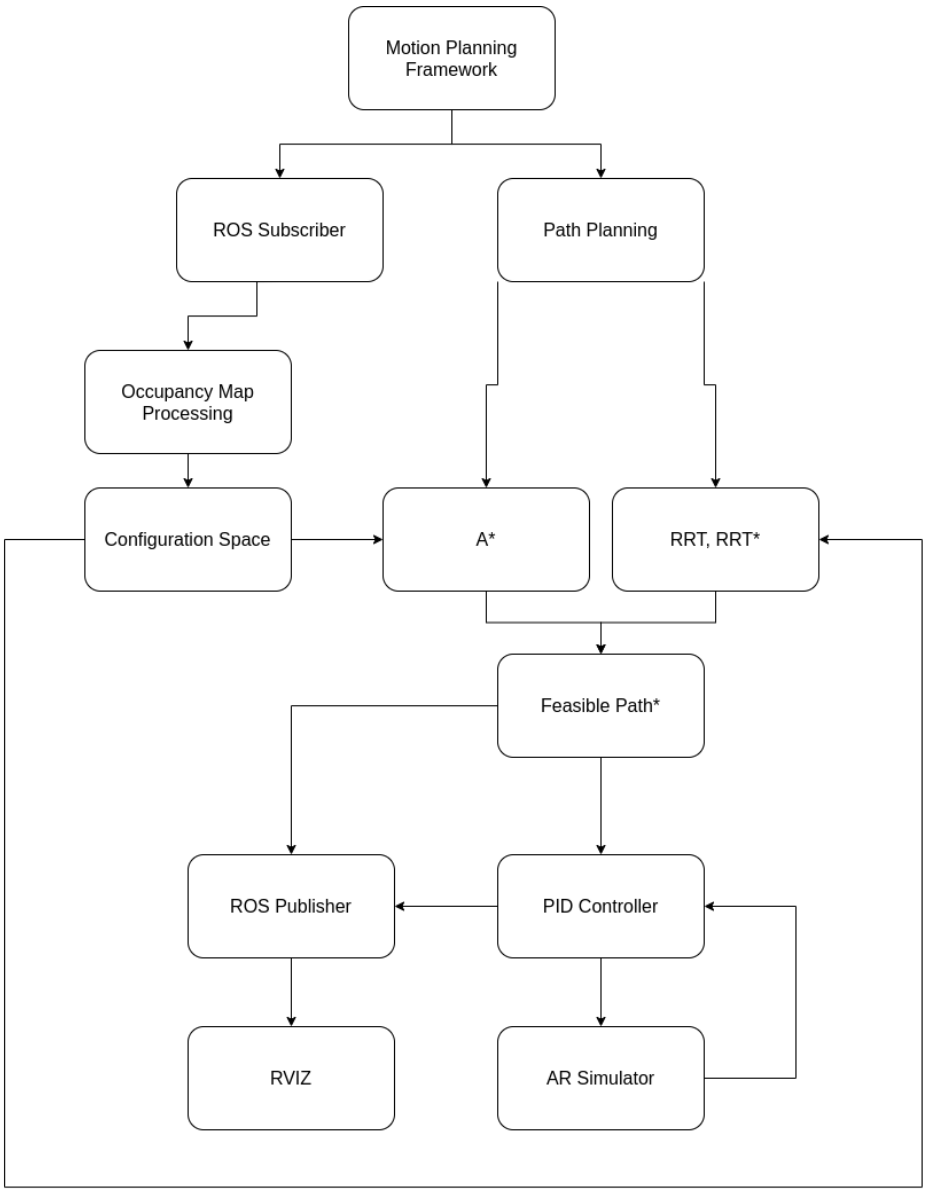


Figure 2.4: Motion planning framework for Digit.

2.3.2 Occupancy Map Processing

The occupancy map is taken in through OctoMap, detailed in [15], which is a mapping technique that allows the processing of point cloud data into occupancy maps. After using ROS to get the map, which is a discrete world in RVIZ, which is described by W_{RVIZ}. Figure 2.5 and Figure 2.6 show the OctoMap framework and visualization of the occupancy map in RVIZ, respectively. Figure 2.5 shows how the tf topic maps all of the point cloud data processing that comes in from the rosbag data. Typically, the rosbag is run after starting RVIZ. The output of running rosbag data is Figure 2.6. Each time the rosbag data is run, the map is slightly altered, which makes it create a slightly different map every time the rosbag is run. The map processing algorithms are able to handle the randomness of the data transfer that occurs when running the rosbag.

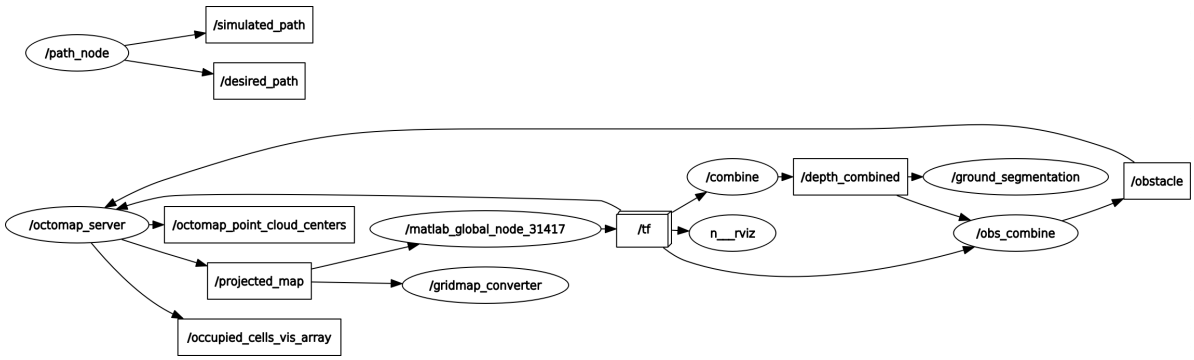


Figure 2.5: Motion planning framework for creating the occupancy map in ROS.

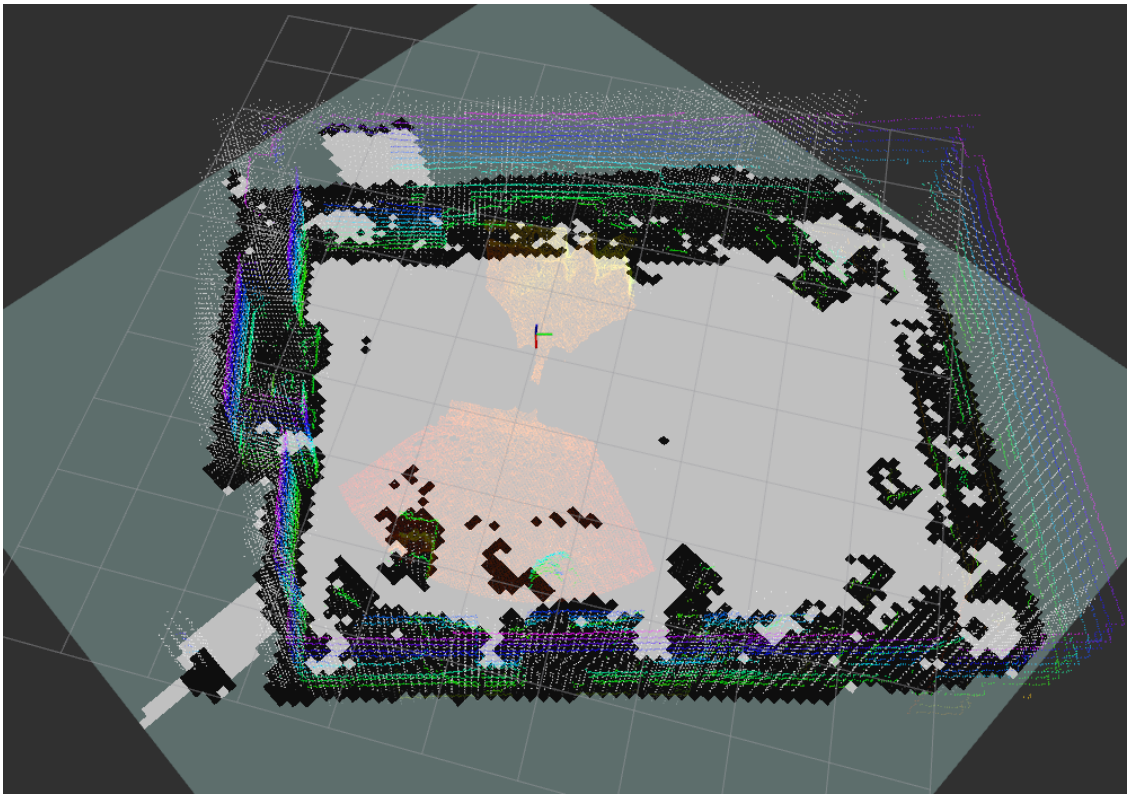


Figure 2.6: RVIZ visualization of occupancy map data.

This world is mapped to a continuous space using a function from the mathematical definition to convert this discrete world into a suitable C-space for planning, which is defined as h_c. In the case of discrete grid data, the function is an identity function that maps this world to a C-space that is equivalent to the original world. For continuous C-space, the discrete grid data is mapped using the function mentioned earlier. The specific mapping occurs by adjusting the obstacles directly defined in the C-space to build a new C-space. The details of the functions used for the mapping are discussed in Chapter 3. Additionally, a safety margin is added directly to project a larger size for the collision detector so that a strictly safe path is enforced from the 2D navigation framework. In the case of discrete grids, the strict safety is in the form of a high maximum cost that refers to the occupancy cost once a movement is made to a location where an obstacle exists within the occupancy

grid. Lenient safety is enforced on the free space that directly borders an obstacle region on the grid. Once those boundary points are found within the grid, a cost greater than 1 is used to represent the occupancy cost in those regions. For the continuous C-space, the leniency is in-built into the default collision detector in that the plan allows paths to border on the edge of an obstacle region.

2.3.3 2D Navigation

Prior to testing on the occupancy map data, the algorithms for map processing and navigation are implemented on toy datasets that simulate sample grid data. The navigation framework currently supports 2D navigation, more specifically, generating feasible paths using discrete and sampling-based approaches. In addition, for 2D navigation, the robot model that is used for generating feasible paths is the point mass robot. The algorithms chosen for the safe navigation were A*, RRT, and RRT*. In the next chapter, the focus will be on describing these algorithms. Some preliminary results for toy datasets are shown in **??** for all three path planning algorithms. RRT* was chosen for implementation on the input occupancy map created from OctoMap due to advantages over A* for higher-dimensional planning as well as a shorter path than RRT. A* was implemented in Python and integrated with the AR simulator with objects added to the simulator from the toy dataset. RRT and RRT* were both implemented in Python and MATLAB. RRT* was chosen over RRT for integration with the simulator due to RRT having a non-optimal path. A* was integrated with the simulator using the "goto" command, which sends the robot from waypoint to waypoint from a path or a series of interactive clicked waypoints in the web-based simulator. RRT* was integrated with the simulator using the "moveto" command, which sends linear and angular velocities to the low-level API. The input velocities were computed using a PID controller, which will be discussed in more detail in Chapter 3. In Chapter 5, a future iteration of the 2D navigation framework will be discussed. Note that the A* and RRT* were modified to include stricter safety, which will be discussed more specifically in the next chapter.

CHAPTER 3

Approaches to Safe Navigation

This chapter focuses on the algorithms used to solve the 2D safe bipedal navigation problem described in the previous chapter. The chapter first discusses the map processing algorithms used to map the discrete world to a continuous C-space as well as some brief descriptions of the image processing techniques used to help estimate obstacles into circles. A*, RRT, and RRT* algorithms are discussed as well as some modifications that were made.

3.1 Map Processing from Cameras and Lidar

The map processing, as mentioned in the framework described in Section 2.3, depends on whether the solution is chosen to solve discrete or continuous motion planning problems. Lidar and four depth cameras are used to create this map. OctoMap is the basis for creating the occupancy map, which is then processed using Algorithm 1. The details of how OctoMap works are in the source paper [15]. For the discrete case, to find the border of the obstacle points in a 2D grid, the border is the mapping of the free space to a set of closed curves surrounding the obstacles. The lenient safety finds the borders of the immediate free space boundary while enforcing lower occupancy costs. The resulting algorithm allows A* to find its shortest path, which will be discussed in the next section. More discussion on the specific functions used in discrete planning will be discussed in the section on A*. For continuous 2D space, the brute-force map processing algorithm is shown in Algorithm 1.

Algorithm 1: Occupancy Map to Configuration Space

Input: occupancy map W, map width w, map height h, threshold radius r_{th}, default radius r_0

Output: C-space C

Let $w : W \mapsto \{0, 1\}, \forall (x, y) \in W$

Find obstacle space $O = \{(x, y) \in W \mid w(x, y) = 1\}$

Find free space $F = \{(x, y) \in W \mid w(x, y) = 0\}$

Let $f_{cc} : (W, w, O) \mapsto (W_{new}, w_{new})$

Let $w_{new} : W_{new} \mapsto \{0, 1, ..., n\}$

$O_{sep} = \{O_i \subset O \mid \forall (x, y) \in O_i, w_{new}(x, y) \in \{1, 2, ..., n\}\}$

$B = \{(x, y) \in O \mid (x, y) \in f_{bt}(W_{new}, w_{new}, F)\}$

Let $C^*_{free} = \{(x, y) \mid x \in [0, w], y \in [0, h]\}$

Add each $(x, y) \in B$ as a new O_i to O_{sep}

for $O_i \in O_{sep}$ **do**

- $i \leftarrow$ current label $\in \{1, 2, ..., n\}$
- $n_{pts} \leftarrow Length(O_i)$
- $(x^i_c, y^i_c) \leftarrow \frac{1}{n_{pts}} \sum_{i=1}^{n_{pts}} (x_i, y_i) \in O_i$
- $r^i_{max} \leftarrow \max\{\{\sqrt{(x_i - x^i_c)^2 + (y_i - y^i_c)^2} \mid (x_i, y_i) \in O_i, (x^i_c, y^i_c)$ is the CoM$\}\}$
- $(\delta x, \delta y) \leftarrow \max_{(x,y)} O_i - \min_{(x,y)} O_i$
- $\delta x \leftarrow \delta x + 1$
- $\delta y \leftarrow \delta y + 1$
- $r^i_{max} \leftarrow r^i_{max} + \frac{\max\{\delta x, \delta y\}}{\delta x + \delta y}$
- **if** $r^i_{max} < r_{th}$ **then**
  - $r^i_{max} \leftarrow r_0$
- Estimate circle c from r^i_{max} and CoM (x^i_c, y^i_c)
- Add c to C_{obs}

$C_{free} = C^*_{free} \setminus C_{obs}$

$C = C_{free} \cup C_{obs}$

Looking closer at Algorithm 1, the inputs to the algorithm are the occupancy map or an input discrete grid, the height of the map, the width of the map, a threshold radius, and a default estimated radius. The output of the algorithm is the C-space C as defined in Chapter 2. From lines 1-10 of Algorithm 1, the objective is to set up the necessary intermediate variables to convert the grid space into a continuous space. First, the obstacle space O is found by finding grid locations that equal 1. Then the free space F is found by finding the grid locations that equal 0. The mapping to the value corresponding to 0 or 1 is w. To ensure that the input world data is still available, a new world W_{new} is initialized, which is an output of connected components. Connected components are shown mathematically in line 4 with the function call f_{cc}. f_{cc} is used to label each location in W with a separate obstacle number. The algorithm starts by scanning through the input dataset. If a node is unlabeled, then a preliminary label is given to it. The neighbors of that node are checked based on the 4-connected or 8-connected grid rule to see if any of the neighbors share the same value as that of the current node. For the purposes of map processing, 8-connected is assumed. The algorithm then rescans the dataset to finish labeling the dataset and returns the labeled dataset with components labeled separately. The input of $f_{cc}s$ includes W and O, and the algorithm outputs an updated W_{new} that has each obstacle location with a corresponding label i. The connected components are considered to be the groups of obstacle points that are associated with a given label. The specific descriptions of the implementation of connected components within MATLAB as well as variations are described in [17],[12]. After the obstacle labels are found, the boundaries of W_{new} are found using a boundary tracing algorithm called Moore-Neighbor tracing algorithm [33],[7], as shown in line 5 with the f_{bt} function call. The Moore-Neighbor algorithm uses the 8-connectivity rule that is used in connected components and attempts to find boundary points by scanning through the dataset. If a pixel is associated with any of the connected components obstacle labels, then that pixel is considered a boundary point. It is not a boundary point, then that the next pixel is investigated. The boundary points are the outermost and innermost pixels associated with a given connected components label. f_{bt} takes the W_{new} and F to find all the possible

boundaries to the free space and returns a set of boundary points. Each boundary point in B is treated as a separate obstacle if its obstacle label is associated with the largest connected component that is also an obstacle. Those points are then added to the O_{sep} in line 8 of Algorithm 1, where each O_i represents a set of points corresponding to label i. C_{free} is initialized as a rectangular continuous space with intervals between 0 and the width of the map for the x-axis and intervals between 0 and the height of the map for the y-axis while C_{obs} is an empty set. Both algorithms, connected components, and Moore-neighborhood boundary tracing, were not implemented from scratch were instead a default function from MATLAB was used to process images; furthermore, more information about the MATLAB implementation of the algorithms can be found in the following books [7],[12].

The for loop in line 11 of Algorithm 1 takes each labeled group of obstacle points, O_i, and treats it as a single obstacle. For each O_i, the center of mass (CoM), denoted by (x_c^i, y_c^i) is computed, and the maximal radius r_{max}^i is computed by finding the distance between each point in the labeled obstacle and its CoM. Using the min and max x and y pixel locations in each labeled obstacle O_i, $\delta x, \delta y$ represent the absolute difference of the min and max x and y, respectively. A factor is added to the maximum radius that represents the ratio of the max possible difference between the x and y min and max and the sum of those to absolute differences, which allows the factor to be in the interval $[0, 1]$. This factor is added to overestimate the circular obstacles after computing the maximum radius and allow for the circular estimation to enforce stricter safety. If the maximum radius is below a threshold, then a default radius is used to estimate the circle for that obstacle; furthermore, this default radius usually will be applied to the single boundary points that are treated as separate obstacles since the estimation is often too small. Once all of these computations are completed, the C-space can be defined by updating the configuration free space and taking the union of the configuration free space and the configuration obstacle space. The results on the toy dataset and the occupancy map are shown in Chapter 4 as well as on the vision-based occupancy map data from Digit.

3.2 2D Path Planning

3.2.1 A* Algorithm

As an example of a discrete path planning technique, A* is a classical algorithm in search that is used as a baseline for testing path planning on real robots or in simulation. A* is a part of a class of forward search algorithms that solve the discrete motion planning problem defined in the last chapter. A* is considered resolution-complete because it will find a solution if one is feasible at a given level of discretization of the C-space, represented by the shortest path. The heuristic is based on a metric, which in the context of a metric space is either the Manhattan distance or the Euclidean distance. The A* pseudocode is shown below in Algorithm 3 [28]. As defined by the pseudocode, A* takes inputs of a start, goal, and heuristic, all of which are defined by 1, N, and h in the pseudocode. An empty priority queue or open list called X_o is initialized with the start node as its first element. The past cost, defined by G, of the first element is then set to 0 as there are no parent nodes to the start node. Then the past cost of all the other nodes is set to infinity. The while loop checks to see if the priority queue still has unexplored nodes and does not stop the loop until the current node x in the loop is the goal node. Before checking if the current node is the goal node, the current node is removed from X_o and added to an empty map or closed list called X_c. If the current node is equal to the goal node, then A* returns SUCCESS and the path to the current point, terminating the while loop. However, if the current node x is not the goal node, then the algorithm loops through all the neighbors, from set X_p of the current node that is not in X_c. For each neighbor x_p of the current node x, the tentative past cost G_t is set equal to the sum of the past cost of the current node and the cost F_c of the current node to the neighbor. If the tentative past cost is less than the past cost of the neighbor, the algorithm set the past cost of the neighbor is set to the tentative past cost. The current node is set to the parent x_{par} of the neighbor node. Then the neighbor is added to X_o, which is sorted based on the estimated total cost F. The estimated total cost of the neighbor is set equal to the sum of the past cost of the neighbor and heuristic cost to go of the neighbor.

Algorithm 2: ReconstructPath

Input: X_c, x
$\tau = \{x\}$
while $x \in X_c$ **do**
    $x = X_c[x]$;
    τ.append(x);
end
return τ

If the algorithm fails to reach a goal node, then FAILURE is returned. In Python, the A* code returns an empty list alongside FAILURE [28].

Path reconstruction is not described by A*but is relatively simple to implement in code, as shown in Algorithm 2. The path reconstruction algorithm takes the current node x and closed list X_c and backtracks through X_c to find the final path. Traditionally, A* does not solve the collision detection or obstacle avoidance problem from the context of safety. However, by adding varying costs to nodes when creating a graph from the grid space nodes, where each edge is made from an assumption of 8 neighbors for each node in the grid. Nodes that correspond to an obstacle are given occupancy costs of MAX_OCC_COST, which is set to an arbitrarily large number to allow for obstacle avoidance. During the step where the tentative past cost is computed, nodes with no obstacles from the occupancy grid have a tentative cost of 1 added, representing the movement cost to go to that neighbor node. Nodes with obstacle locations have a tentative cost of 1 plus MAX_OCC_COST, representing movement cost plus occupancy cost, added to the tentative past cost for a given neighbor node. Similarly, for safety considerations, the immediate boundary points of the obstacles have the same occupancy cost as MAX_OCC_COST, which allows the algorithm to directly avoid the immediate region surrounding the obstacles. Additional layers of safety can be found by finding the boundaries of the previous layers of safety.

For example, suppose the first layer of safety is found, then the boundary of the obstacles is the first layer of safety. The boundary of that first layer of safety adds another layer of safety, which can continue until the user is satisfied with the safety considerations for the

Algorithm 3: A* Search [28]

Input: x_I start position, X_g goal set, heuristic cost-to-go h
$x_I = 1$, where $1 \in \{1, 2, .., n\}$
$X_o \mapsto \{x_I\}$
$G[x_I] \mapsto 0$, $G[x_n] \mapsto \infty$ for node $x_n \in \{2, ..., N\}$
while $X_o \neq \emptyset$ **do**
    node $x \mapsto$ first node in X_o, remove from X_o
    add x to X_c
    if $x \in X_g$ **then**
        return SUCCESS and ReconstructPath(X_c, x)
    end
    Let X_p be the neighbors of x
    for $x_p \in X_p \mid x_p \notin X_c$ **do**
        $G_t \mapsto G[x] + F_c(x, x_p)$
        if $G_t < G[x_p]$ **then**
            $G[x_p] = G_t$
            $\mathrm{x_{par}}[x_p] = x$
            $F[x_p] \mapsto G[x_p] + h(x_p)$
            Add $\min_{x_p}\{F\}$ to X_o
        end
    end
end
return $FAILURE$

grid. The occupancy cost added to each layer of safety decreases after the first layer of safety is added (i.e., the second layer of safety and onward), which allows for some leniency for the planned path should the goal be in the higher layers of safety. However, that depends solely on the occupancy cost associated with each layer. If the obstacle nodes and the nodes corresponding to the first layer of safety have sufficiently high occupancy costs, the algorithm will find the shortest path that avoids the obstacles. If the additional layers of safety have low enough occupancy costs, then the algorithm will be sure to converge to the shortest path that avoids the obstacles and the first layer of safety. Another thing to note is that the heuristic cost to go is dependent on the metric chosen, which as stated before is usually Euclidean or Manhattan distance. The preliminary results of A* are shown in Chapter 4.

3.2.2 RRT Algorithms

Sampling-based algorithms are a class of algorithms that allow an agent or robot to sample within the C-space or state-space while attempting to find a feasible path from start to goal. The algorithms implemented as part of the 2D navigation framework are called rapidly-exploring random trees (RRTs). RRTs rely on tree representations from data structures for forward searching from start to goal. The two algorithms covered in this section will be RRT and a variant of RRT called RRT*. RRT is a standard sampling-based planner that is used as a basis, but it often returns longer non-smooth paths. It should be noted though that longer implies based on a metric such as Manhattan or Euclidean distance and not on the number of waypoints. The number of waypoints may be fewer in RRT-based algorithms than for A*. Sampling-based algorithms are not resolution-complete and do not tend to find resolution-optimal solutions that grid-based algorithms often find. However, the sampling algorithms tend to find feasible solutions quicker in higher-dimensional spaces. The trade-off between higher dimensionality and optimality is often considered before using an algorithm as part of a navigation framework. Sampling-based algorithms are called probabilistically complete, which means that the probability of finding a feasible solution approaches 100% when the number of samples approaches infinity. The algorithm pseudocode is shown below in Algorithm 4 [28].

RRT is a relatively simple algorithm from an implementation and pseudocode perspective. The algorithm first initializes an empty tree, which can get expanded as more samples from the C-space or state space are taken. Given a start and a goal, just as with A*, RRT builds the tree until it hits the goal or returns FAILURE. While the size of the tree is less than a maximum tree size, the algorithm samples from X, which is either a sample from the state space or the C-space. The nearest point to the sampled state or configuration is found by sorting the node cost for each sample in the tree. Then x_{new} is computed from a maximum distance from x_{near} to x_{samp}. Then the local planner is used to find the motion that gets the new state or configuration. As long as the motion is collision-free the new configuration is added to the tree with an edge from the nearest point to the new point. Once the new state

Algorithm 4: Rapidly-Exploring Random Tree (RRT) [28]

```
Initialize search tree T_n with x_start;
while T_n.size()<MAX_TREE_SIZE do
    x_samp ← X ~ U(x_min, x_max)
    x_near ← nearest node in T_n to x_samp
    compute x_new from a maximum distance of D from x_near to x_samp;
    employ a local planner to find a motion from x_near to x_new
    in the direction of x_samp
    if the motion is collision-free then
        add x_new to T_n with an edge from x_near to x_new
        if x_new ∈ X_g then
            return SUCCESS and feasible τ from x_start to x_new
        end
    end
end
return FAILURE
```

or configuration is equal to the goal state or configuration, RRT finishes and returns either SUCCESS. Otherwise, the algorithm returns FAILURE. Due to RRT* producing shorter paths, RRT* was chosen for further simulation experiments over RRT [28].

The pseudocode for RRT* is shown below in Algorithm 5. RRT* is changed from RRT in that there is an additional for loop where all the x in the set of nearest points are tested to see if there are two conditions: (1) collision-free and (2) the total cost from x_{start} to x_{new} is minimized. RRT and RRT* are implemented in Python and MATLAB. Obstacles are part of the input to these algorithms in the available navigation framework, which means that they can be modified for the purposes of making the plan safe. The 2D path planned by RRT* is much shorter and smoother than RRT, which means that the method for finding the shortest path within the sampling-based method requires checking other nearest nodes. However, RRT* is considered asymptotically optimal, as described in Chapter 1. In Chapter 4, the results will be discussed in more detail. In the specific implementation of RRT and RRT* for safety, strict safety is enforcing a safety margin on the estimated obstacles from the mapping for use in the collision detection, which forces the planner to plan a safety margin away from

Algorithm 5: RRT* [28]

Initialize search tree T_n with x_{start}
while $T_n.size()$<MAX_TREE_SIZE **do**
  $x_{samp} \leftarrow X \sim U(x_{min}, x_{max})$
  $x_{near} \leftarrow$ nearest node in T_n to x_{samp}
  compute x_{new} from a maximum distance of D from x_{near} to x_{samp};
  employ a local planner to find a motion from x_{near} to x_{new}
  in the direction of x_{samp}
  if *the motion is collision-free* **then**
    add x_{new} to T_n with an edge from x to x_{new}
    after testing all $x \in X_{near}$ such that
    (1) the motion is collision-free and
    (2) total cost is minimized from x_{start} to x_{new}
    if $x_{new} \in X_g$ **then**
      return SUCCESS and feasible τ from x_{start} to x_{new}
    end
  end
end
return $FAILURE$

the estimated obstacle boundary. The preliminary results for RRT and RRT* are shown in Chapter 4 in addition to the results with the occupancy map are shown using RRT* [28].

3.3 PID Controller

In order to send the robot in the AR simulator to the desired locations, a PID controller was designed to enable the robot to walk from waypoint to waypoint. The controller computes the angular velocity ω_d from the controller. The gains were tuned until the controller was reduced to a PD controller that approximately sent velocity commands to the AR simulator and displayed the results in RVIZ. The following equation was used to design the controller. While the distance was tracked to make sure that the controller was moving from waypoint to waypoint, the controller is based purely on the error in the heading angle. The heading angle is computed from the desired positions and the current positions, using the atan2 function. The PID controller was implemented in Python so that the controller could send

the appropriate JSON commands from the WebSocket node. First, the WebSocket node is initialized, then the controller requests the base kinematics to get the pose and twist information through ROS. After receiving the base kinematics, the controller uses the yaw angle error as the heading error for the PID controller. The mathematical equation for the PID controller update step is shown:

The desired heading angle is computed from the desired position and the actual position. The heading angle is computed using the atan2 function, which will be referred simply arctan. Equation 3.1 is shown below:

$$\theta_d = \arctan 2(y_d - y, x_d - x) \tag{3.1}$$

x_d, y_d are desired x-value and y-value, and x, y are actual x-value and y-value from base kinematics from the robot. Using the desired heading θ_d and a constant initial velocity v_0, the desired velocities v_x, v_y are computed as shown in Equations 3.2 and 3.3.

$$v_x^d = v_0 \cos\theta_d \tag{3.2}$$

$$v_y^d = v_0 \sin\theta_d \tag{3.3}$$

From the desired heading angle and the measured heading angle, the error in the yaw angle is found, as shown in Equation 3.4. Using the previous yaw error and the current yaw error divided by a time step dt, as shown in Equation 3.5. Finally, the desired angular velocity is computed in Equation 3.6. All of the variables without a subscript described for the PID controller are considered measured variables that are taken from a JSON call to get the robot's current base kinematics. The desired values are all computed based on the base kinematics.

$$e_{yaw} = \theta_d - \theta \tag{3.4}$$

$$\dot{e}_{yaw} = (e_{yaw} - e_{yaw}^{prev})/dt \tag{3.5}$$

$$\omega_d = K_p e_{yaw} + K_d \dot{e}_{yaw} \tag{3.6}$$

The desired angular velocity and the desired linear velocities are then sent as inputs to the "moveto" JSON command for updating the robot's base kinematic information. The results of the PID controller are shown in Chapter 4. This PID controller is designed to be simple for the purposes of sending velocity commands to the AR simulator. By doing so, fewer variables and gains need to be tuned for the purposes of making a more robust controller. In other words, controlling only the heading angle allows for simple adjustments to the K_d and K_p terms in the PID controller output, which is the desired angular velocity ω_d. Desired linear velocities are also only computed from just the heading angle, meaning only one variable is needed to send velocity commands to the AR simulator.

CHAPTER 4

Results

This chapter focuses on the implementation results in simulation with Digit. The results first discuss the output of the map processing algorithms for both the preliminary results for the toy and real vision dataset. Then the output of A*, RRT, and RRT* are discussed alongside the preliminary output from the mapping algorithm. Finally, the preliminary results of the PID controller are discussed with the plots of position, orientation, and velocities. In addition, the path traversed by the robot using the PID controller will be shown to show how closely the robot follows the desired position and orientation.

4.1 Map Processing Results

4.1.1 Toy Data Processing

For the grid space algorithm with A*, the C-space doesn't change, so the results of the processing do not really matter with the discrete case. With the toy data processing for the continuous case, the map processing algorithm was able to label obstacles into separate regions for circular obstacle estimation. In Figure 4.1, the labels are associated with different colors that were found through the connected components algorithm. The results from this algorithm work well with the toy dataset. Note that the different colored obstacles are based on the safety region found from A* that was part of the map. The input for this data processing was the grid found from adding safety regions, which is why the results returned these specific results. For the purposes of testing the boundary tracing and connected components, this toy dataset was sufficient to continue with the real vision data represented as

the occupancy map.

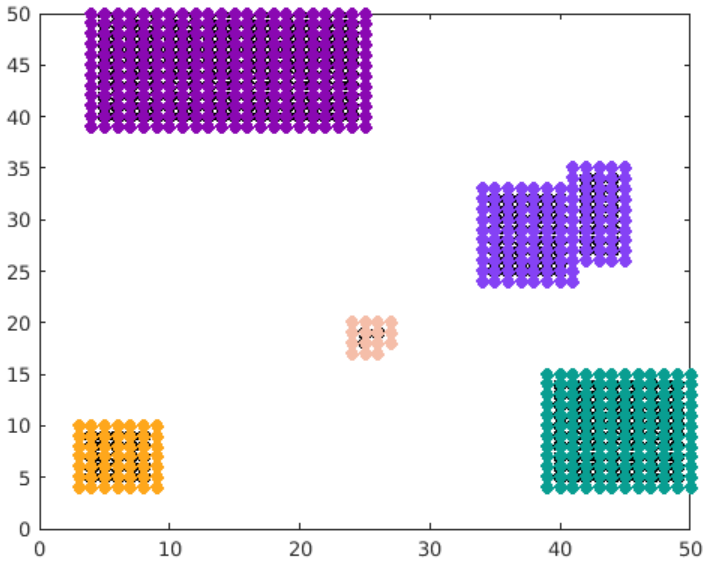
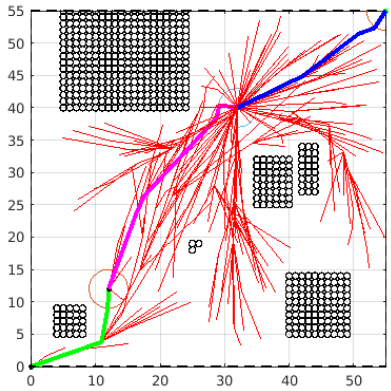


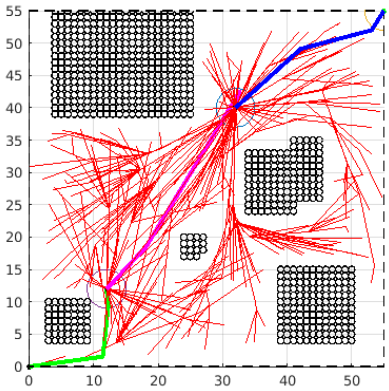
Figure 4.1: Results of obstacle separation technique using connected components. The different colors are based on labels from the connected components algorithm.

After processing the toy dataset to separately label each obstacle, the obstacles were approximated as circles. For Figure 4.2 below, (a) and (b) show the obstacle grid positions treated as their own center of mass with the resolution of the grid used to define the radius. While this method works as a way of representing obstacles, there are a few issues that are found numerically. For example, in the collision detection step of RRT and RRT*, the algorithms for the detection of a collision can fail due to numerical limitations. These numerical issues will be discussed in the next chapter. Also, Figure 4.2a shows the circles with the original data while Figure 4.2b shows the data with the safety region around the obstacles that make the obstacles larger in size within the C-space. Figure **??** shows the circle estimation technique described in Chapter 2 and how it overlaps with the original data from

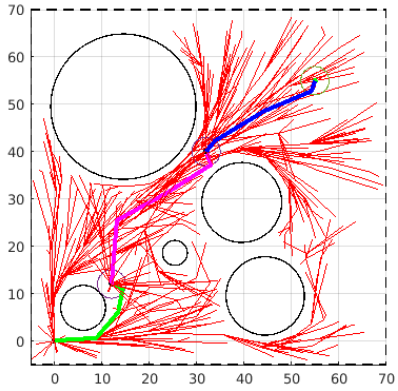
Figure 4.2a in Figure 4.2c. As can be seen, the obstacles are larger in size and circumscribe the original obstacles completely. The main result of interest here is the reduction of the number of obstacle grid points from hundreds of points to five larger obstacles. It should be noted though that in the toy datasets free space was not accounted for in the algorithm for planning since the boundaries of the sampling space were well defined. The paths shown in the figures are results of RRT* navigating around the obstacles. The different colored feasible paths are the path of a sequential plan that represents having multiple plans in the same free space. Note that to get from start to goal using most planners, a single planner can be employed or multiple planners can be employed to solve the navigation problem. The choice of whether or not to use multiple planners depends on the robot and whether the robot needs or requires multiple plans for performing other tasks other than navigating around their environment. Another detail that is different from the actual vision data is that this data assumes that the obstacles are generally always rectangular, meaning that the data processing and circle estimation is much easier. In the next section, the results of the obstacle labeling and boundary tracing algorithms will be discussed in further detail.



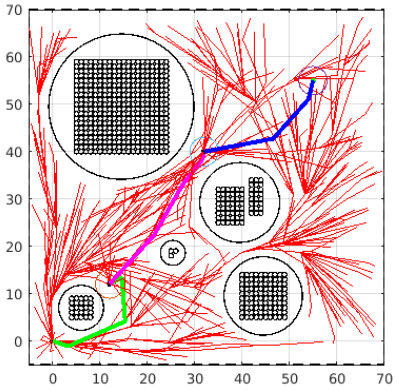
(a) Small obstacles.



(b) Small obstacles with predefined obstacles.



(c) Obstacles estimated as larger circle.



(d) Smaller obstacles vs larger obstacles.

Figure 4.2: RRT* with obstacle estimation. (a) Discrete obstacle positions converted to small circles from original data. (b) Discrete obstacle positions converted to small circles directly from safety region from toy grid. (c) Discrete obstacle positions grouped into larger circles. (d) Discrete obstacle positions grouped into larger circles with original small circles overlapped.

4.1.2 Occupancy Map Data Processing

Figure 4.3 shows the results with RRT*, the results of obstacle estimation, obstacle labeling, and boundary tracing all overlapped. As mentioned before, free space is needed to be used to confine the planning since proper bounds were not easy to define with the noisiness in the construction of the points on the walls. The algorithm implemented to find the C-space

from the discrete grid computes all the boundaries for each obstacle and each free-space region. The boundary of the free space is the edge of the obstacle, which means that the walls points can be used to confine planning inside the walls or outside the walls. Since the points on the wall are non-smooth and piecewise, the chosen method was to estimate the walls as circles, which is shown in the figure. Note obstacles are estimated as circles based on the number of points that are labeled to that obstacle or whether they are free-space boundary points corresponding to the wall edges. Additionally, the results show that narrow regions are cluttered with obstacles, which makes planning in the nearby free space to be unnecessary and infeasible. The results of the planning will be discussed in more detail in the RRT algorithms section, but the planned path is able to find a feasible path despite having over 1000 obstacles. In navigation, each obstacle is a constraint, so the planner found a feasible path despite having many constraints on the planning.

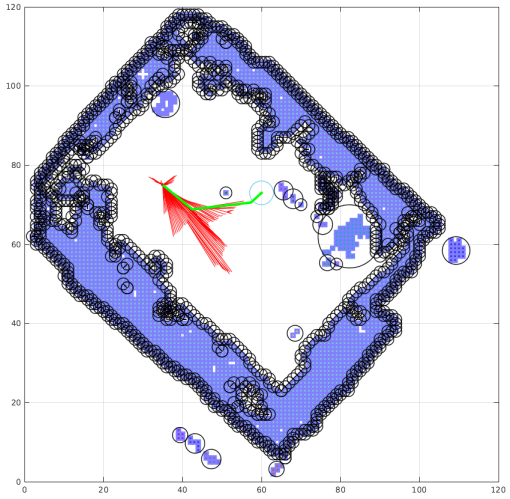


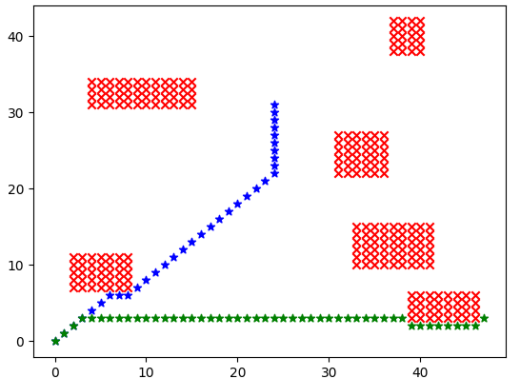
Figure 4.3: Occupancy map implementation with RRT*. Circles are obstacles estimated from boundary points and connected components.

4.2 Path Planning Results

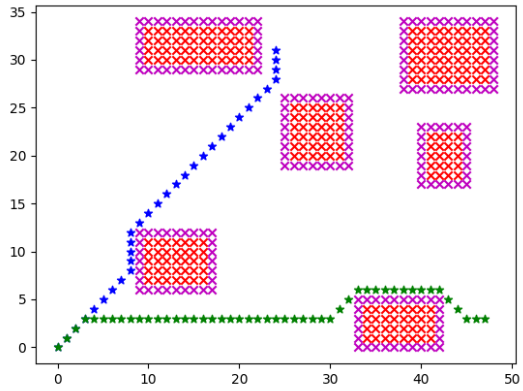
4.2.1 A* Algorithm

A* generates the shortest path in any grid-based planning. The results shown in Figure 4.4 show that A* is able to find the shortest path from start to goal. The paths are denoted by "*" characters while the obstacles are denoted by red "x" characters. In Figure 4.4a and Figure 4.4b A* finds a path for the blue and green paths successfully. However, the black path shown in the Figure 4.4c shows a third path. In the implementation of the code that led to these results, A* was only able to find the paths for blue and green paths in both (a) and (b). The main reason for these results was that the grid space was randomly generating obstacles, which caused no paths to be returned for the black path. Additionally, the first layer of safety is denoted by purple "x" characters. The user of these algorithms can pre-define the occupancy cost associated with any safety layers added. The second layer is defined by cyan "x" characters, and the third layer is defined by yellow "x" characters. Each of these layers has an associate occupancy cost. One other remark is that the user of these algorithms can extend the boundary of any of these layers for planning should they want to implement the first layer as two layers of boundaries on the obstacles. If the user of these algorithms for finding the boundaries of obstacles and safety layers, eventually the entire C-space will look similar to a contour plot. However, the recommendation is to refrain from the usage of the safety layers to create contour maps since it may reduce the shortest path convergence of A*. As mentioned earlier, A* generates more waypoints than RRT or RRT*. The number of waypoints for these is each discrete grid space that is free and satisfies the safety considerations.

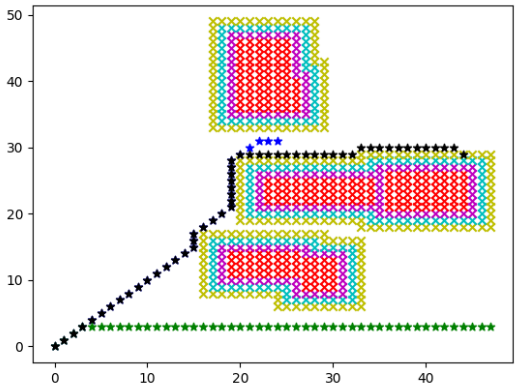
A* does solve the 2D navigation problem, but there are still some limitations with A*. A* is both a forward search and a grid-based approach to solving the discrete path planning problem defined in Chapter 2. A* is not as useful in higher dimensional planning. There is a class of algorithms that attempt to improve A* and are based on improving A* in the context of grid space planning as well as dynamically planning around obstacles. Due to problems



(a) A* with obstacle avoidance.



(b) A* with stricter obstacle avoidance.



(c) A* with both strict and lenient obstacle avoidance.

Figure 4.4: A* preliminary results with safety regions.

with higher-dimensional planning and algorithmic complexity, these algorithms were avoided in favor of RRT algorithms. The results of these algorithms are shown in the next section as the main method for planning.

4.2.2 RRT Algorithms

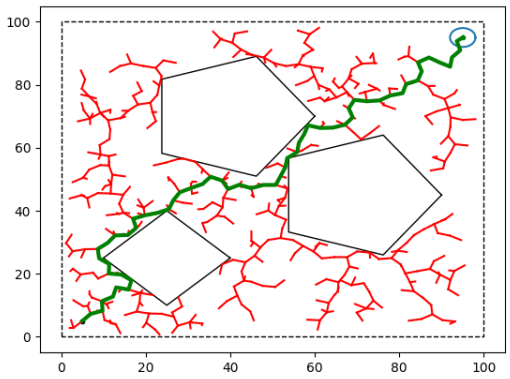
RRT algorithms sample the configuration or state-space to solve the feasible path problem. In the implementations in the current navigation framework, some sample solutions are shown on toy continuous C-space datasets in Figure 4.5a and Figure 4.5b. (a) shows the

Table 4.1: Average values for run time, cost, number of iterations, and number of waypoints are available over 11 runs. The total number of obstacles and the safety margin information is also available.

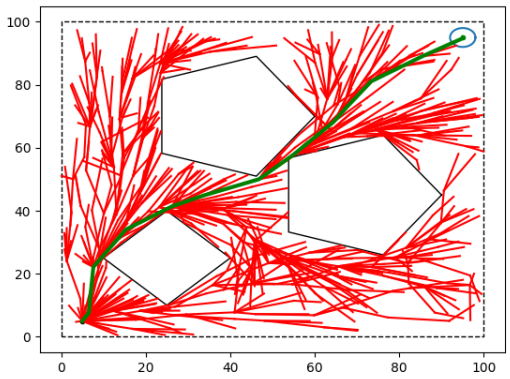
Run Time	# of Obs.	# of Waypoints	Cost	# of Iterations	Safety Margin
197.6335 seconds	1055	4.3636	25.4052 m	348.1818	0.35 m

results of RRT while (b) shows the results of RRT*. RRT has a long and non-smooth path and generally does not solve the shortest path problem. RRT* attempts to solve this problem by checking all of the nearest points, which results in a much shorter and smoother path. RRT* does not find the optimal global path from start to goal either. RRT tends to explore more than RRT*, which leads RRT to be non-smooth and longer. However, RRT* focuses on shortening the path, which means that the exploration is reduced. Additionally, Figure 4.5a and Figure 4.5b are leniently safe in that the planned path is allowed to plan near to the edge of the obstacle. Figure 4.5c is the same plot from the mapping section, but notice that the planned paths are planned far away from the obstacles. These planned paths are generally a safety margin away from the obstacle edge at all times, which means that it will pick the safer plan. Table 4.1 shows some results for 11 runs of RRT* on the the dataset from Figure 4.5c. The table consists of average run time, number of obstacles, number of waypoints, cost, and number of iterations. When receiving the occupancy map data from ROS, the data slightly changes, depending on the frequency at which the rosbag file is running. These changes are enough to simulate different iterations of the C-space mapping algorithm and the path planning results.

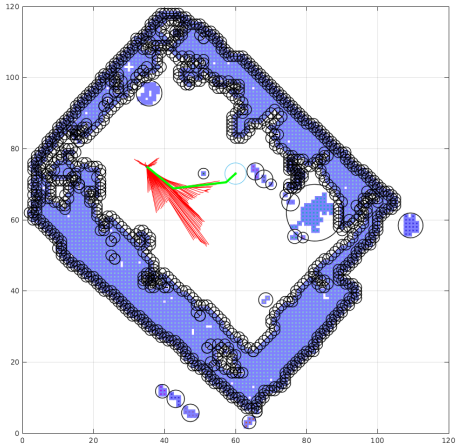
Based on the tabular information provided, the average run time of RRT* over 11 runs in 197.6335 seconds, which is approximately 3 minutes. The total number of obstacles was 1055, and the average number of waypoints is 4.3636, which implies that it finds 4 waypoints slightly more often than 5 or more waypoints. The average cost is 25.4052 m, which is based on the distance metric. The average number of iterations is 348.1818. Finally, the safety margin is set at 0.35 m. Given these results, 3 minutes average planning time is fairly fast in the context of finding feasible paths with 1055 obstacles at a safety margin of 0.35, which



(a) RRT results.



(b) RRT* results.



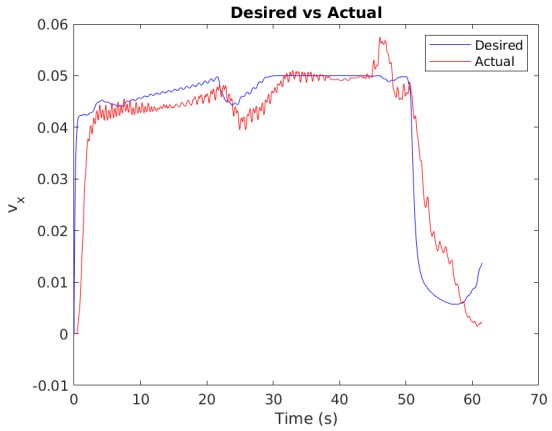
(c) RRT* results in occupancy grid.

Figure 4.5: Results of RRT algorithms.

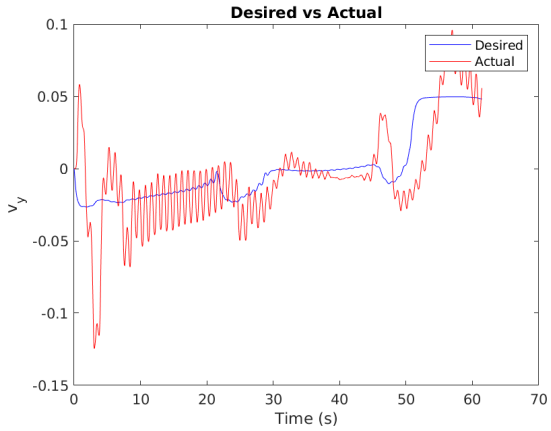
means there is a lot of exploration. Note that the safety versus exploration trade-off occurs in terms of computational time for the RRT* algorithm. A larger safety margin reduces the number of available states or configurations that are selected randomly.

4.2.3 PID Controller

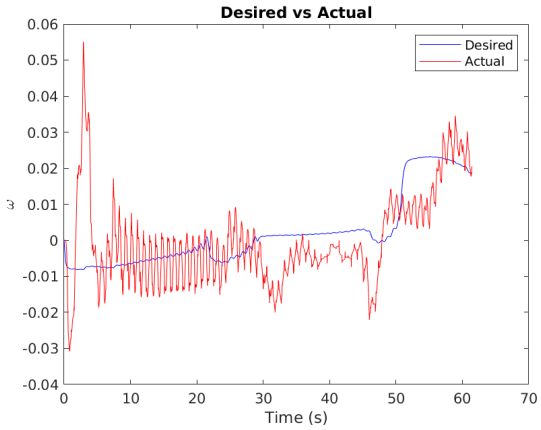
The results of the PID controller are shown in Figure 4.6. Looking at Figure 5.1a, v_x actual linear velocity in the x-direction appears to approach the desired values. Similarly,



(a) v_x vs. t.



(b) v_y vs. t.



(c) ω vs. t.

Figure 4.6: Linear and angular velocities.

in Figure 5.1b, the actual y velocity is a bit noisier but appears to be generally following the desired y velocity. The actual ω appears to approach the desired values but is noisier in some areas, which tend to correspond with changes in waypoints. The angular velocity-based controller seems to provide good results in terms of how the robot behaves in simulation. There are still improvements and tuning that can be done with the PID controller, but the preliminary results are good enough to start implementations with the real robot.

For now, the results are satisfactory, but some potential sources of error may be causing the robot to have unsteady velocities at times when it reaches the waypoints. Other sources of error could also come from the bipedal locomotion, which tends to be periodic and creates

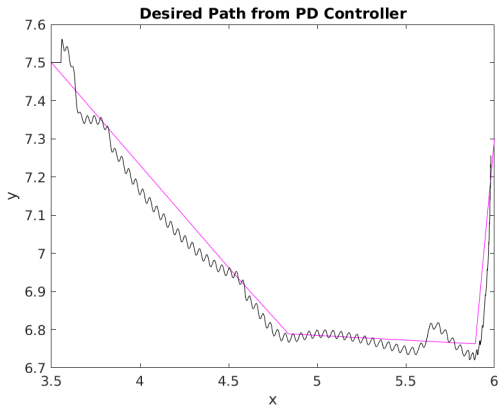


Figure 4.7: PID controller results with desired path vs PID path.

a periodic position curve when plotting x-y paths that is the output of the PID controller. However, some smaller overshoots still seem to suggest some requirements for properly tuning the controller. It is also possible that a better control design overall will improve the robustness of the PID controller. These results are considered preliminary since more tests need to be done to ensure that the real robot can achieve similar results as in simulation.

Figure 4.7 shows the x-y path. The black path is the output from the PID controller whereas the pink path is defined by the desired waypoint path from RRT*. As can be seen, the y position tends to overshoot more often, which could be an issue with the y velocity. Interestingly, an omnidirectional behavior is shown when the robot gets to the second-to-last waypoint. It solely uses the y velocity to sidestep to the final waypoint. It is possible that the controller is exhibiting some omnidirectional behavior unintentionally or it could also be part of the error that still needs to be reduced by better tuning of the PID gains.

The next chapter will focus on drawing conclusions from the results provided. The future improvements and deliverables will also be discussed in more detail as well.

CHAPTER 5

Conclusion and Future Work

5.1 Conclusions

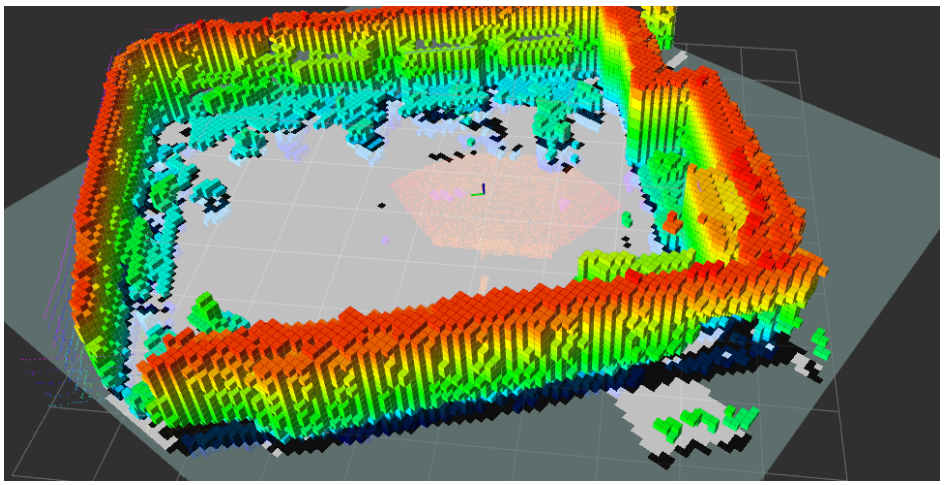
This chapter discusses the conclusions and future research directions based on the results of implementing the planning algorithms on Digit within the simulator. The conclusions are separated based on the conclusions based on the map processing results and those drawn from the path planning results.

5.1.1 Map Processing

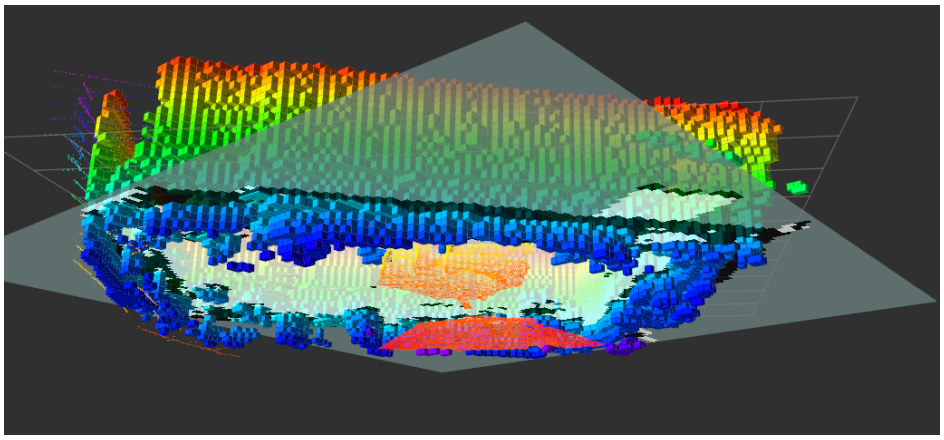
From the results from the last chapter, the map processing techniques tend to work for converting the discrete grid to a continuous space. However, there are some limitations to the method. Firstly, circle estimation is relatively inaccurate and can take up a large portion of the free space if the circle estimation is not properly set. Additionally, the wall boundaries would ideally not be approximated by intersecting circles and rather a closed-form function that can be used for obstacle avoidance to confine path planning within the walls. In some cases, ellipses may be better approximations, as they will allow for a maximum and minimum radius. However, for the dataset, it is limited since the data is at a particular orientation that is unknown. For the time being, the algorithm provided is sufficient for 2D path planning. It remains to be seen whether feasible trajectories can be found from this map processing method with 1000+ obstacles being estimated as circles.

As far as the map itself there are some limitations in that the map itself has some inaccuracies to it as well as seen below in Figure 5.1 figure shows a height map, looking at

the from the top view, there are some obstacles that are found that go through the ground, which is also seen in the bottom view. The mapping process can be improved to remove some of this excess noise, which will reduce a significant amount of the obstacles that are shown in the figures. In particular, the obstacles that are perceived through windows and underground are more concerning since there is no accurate information about that part of the world yet. The next section will detail some of the limitations of the 2D navigation framework.



(a) Top view.



(b) Bottom view.

Figure 5.1: Height map.

5.1.2 Path Planning

From the results of the planned paths, 2D navigation is now possible using both discrete and sampling-based methods. The chosen algorithm for analysis was RRT* due to stronger results with higher-dimensional planning. The current 2D navigation is still able to find feasible paths despite having 1000+ obstacles in the occupancy map. Some improvements need to be made on the average run time so that the navigation will be able to reduce overall run time. Additionally, 2D navigation should also seek to integrate omnidirectional kinematics and some form of dynamics to get feasible trajectories. One important lesson learned from the research project was that safety may be more important in unknown environments than optimality. More research and experiments need to be done to find metrics to measure safety vs optimality trade-offs. Safety in A* is limited in that continuously adding safety layers may reduce the ability to achieve globally optimal shortest paths while RRT* requires fewer estimated obstacles to run faster. Finally, trajectory optimization and planning are the next logical step as well as hardware experiments with path planning.

5.2 Future Work

5.2.1 Improvements to Navigation Framework

Some improvements to the navigation framework include generating feasible trajectories and reducing the number of obstacles for planning by finding a closed-form function for confining planned paths within the walls of the data. Perhaps considering planning methods in unknown environments would also enable Digit to be more robust in its planning and replanning if accounting for dynamic obstacles. Since the planners do not work with dynamic obstacles, the next logical steps would be to test hardware on the current framework, consider more mapping environments that limit underground points, and consider dynamic obstacles. Other improvements would be to design a proper optimization problem that can be solved with some sort of optimization technique for developing feasible trajectories. The PID controller needs to be more robust and the gains need to be tuned to reduce the errors in y position and velocities. Finally, integrating dynamics and kinematics is an important

step for improving navigation frameworks.

5.2.2 New Research Directions

Some new research directions that the lab can take include studying control barrier functions more carefully for the purpose of finding a closed-form expression for the boundary points of the wall. A potential method for finding such a solution could be function approximation techniques from machine learning as well as other methods that may be useful. Another useful direction would be considering more carefully the trade-off between optimality and safety, which also creates a relationship between safety and optimality. Reinforcement learning-based planners are also an interesting solution to continuous action space path or trajectory planning. Perhaps to improve map processing techniques, machine learning methods could be employed for learning how to label, if the use of the labeling is still useful for solving a particular problem. Other areas of safety may be to consider more dynamics and kinematics models to integrate a robust dynamics model made up of a hierarchy. Better alternatives for obstacle avoidance can come from chance-constrained path planning methods. For smoother paths, the use of hybrid A* would be helpful, particularly on the control implementations.

BIBLIOGRAPHY

[1] Mayur J. Bency, Ahmed H. Qureshi, and Michael C. Yip. "Neural Path Planning: Fixed Time, Near-Optimal Path Generation via Oracle Imitation". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 3965–3972. DOI: 10.1109/IROS40897.2019.8968089.

[2] Lars Blackmore, Masahiro Ono, and Brian C. Williams. "Chance-Constrained Optimal Path Planning With Obstacles". In: *IEEE Transactions on Robotics* 27.6 (2011), pp. 1080–1094. DOI: 10.1109/TRO.2011.2161160.

[3] Lars Blackmore et al. "A Probabilistic Particle-Control Approximation of Chance-Constrained Stochastic Predictive Control". In: *IEEE Transactions on Robotics* 26.3 (2010), pp. 502–517. DOI: 10.1109/TRO.2010.2044948.

[4] Guillermo Castillo et al. *Robust Feedback Motion Policy Design Using Reinforcement Learning on a 3D Digit Bipedal Robot*. Mar. 2021.

[5] Manuel Castillo-Lopez et al. *A Real-Time Approach for Chance-Constrained Motion Planning with Dynamic Obstacles*. Jan. 2020.

[6] Howie Choset et al. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Cambridge, MA: MIT Press, May 2005.

[7] *DIGITAL IMAGE PROCESSING USING MATLAB 2E*. Gatesmark, 2009. ISBN: 9781259084072. URL: https://books.google.com/books?id=dx0lM-XgabEC.

[8] Dmitri Dolgov et al. "Practical Search Techniques in Path Planning for Autonomous Driving". In: *AAAI Workshop - Technical Report* (Jan. 2008).

[9] Mohamed Elbanhawi and Milan Simic. "Sampling-Based Robot Motion Planning: A Review". In: *IEEE Access* 2 (2014), pp. 56–77. DOI: 10.1109/ACCESS.2014.2302442.

[10] M. Evett et al. "PRA*: Massively Parallel Heuristic Search". In: *Journal of Parallel and Distributed Computing* 25.2 (1995), pp. 133–143. ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1995.1036. URL: https://www.sciencedirect.com/science/article/pii/S0743731585710362.

[11] Mahmoud Hamandi, Mike D'Arcy, and Pooyan Fazli. "DeepMoTIon: Learning to Navigate Like Humans". In: (Mar. 2018).

[12] Robert M. Haralick and Linda G. Shapiro. *Computer and Robot Vision*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1992, pp. 28–48. ISBN: 0201569434.

[13] 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.

[14] Kris Hauser et al. "Motion Planning for Legged Robots on Varied Terrain". In: *The International Journal of Robotics Research* 27.11-12 (2008), pp. 1325–1349. DOI: 10.1177/0278364908098447. eprint: https://doi.org/10.1177/0278364908098447. URL: https://doi.org/10.1177/0278364908098447.

[15] Armin Hornung et al. "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees". In: *Autonomous Robots* (2013). Software available at https://octomap.github.io. DOI: 10.1007/s10514-012-9321-0. URL: https://octomap.github.io.

[16] Sertac Karaman and Emilio Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning". In: *CoRR* abs/1105.1186 (2011). arXiv: 1105.1186. URL: http://arxiv.org/abs/1105.1186.

[17] T. Kendall. *IMAGE PROCESSING with MATLAB*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016. ISBN: 1539784401.

[18] Sven Koenig and Maxim Likhachev. "Improved fast replanning for robot navigation in unknown terrain". In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)* 1 (2002), 968–975 vol.1.

[19] Sven Koenig and Maxim Likhachev. "Real-Time Adaptive A*". In: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '06. Hakodate, Japan: Association for Computing Machinery, 2006, pp. 281–288. ISBN: 1595933034. DOI: 10.1145/1160633.1160682. URL: https://doi.org/10.1145/1160633.1160682.

[20] Sven Koenig, Maxim Likhachev, and David Furcy. "Lifelong Planning A". In: *Artificial Intelligence* 155.1 (2004), pp. 93–146. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2003.12.001. URL: https://www.sciencedirect.com/science/article/pii/S000437020300225X.

[21] Richard E. Korf. "Breadth-first frontier search with delayed duplicate detection". In: *In Proceedings of the IJCAI03 Workshop on Model Checking and Artificial Intelligence*, pp. 87–92.

[22] Richard E. Korf. "Depth-first iterative-deepening: An optimal admissible tree search". In: *Artificial Intelligence* 27.1 (1985), pp. 97–109. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(85)90084-0. URL: https://www.sciencedirect.com/science/article/pii/0004370285900840.

[23] Thibault Kruse et al. "Human-aware robot navigation: A survey". In: *Robotics and Autonomous Systems* 61.12 (2013), pp. 1726–1743. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2013.05.007. URL: https://www.sciencedirect.com/science/article/pii/S0921889013001048.

[24] S. M. LaValle. "Planning Algorithms". In: Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006. Chap. 2,3,4,5.

[25] Zhongyu Li et al. “Vision-Aided Autonomous Navigation of Bipedal Robots in Height-Constrained Environments”. In: *CoRR* abs/2109.05714 (2021). arXiv: 2109.05714. URL: https://arxiv.org/abs/2109.05714.

[26] Maxim Likhachev et al. “Anytime Dynamic A*: An Anytime, Replanning Algorithm.” In: Jan. 2005, pp. 262–271.

[27] Brandon Luders, Mangal Kothari, and Jonathan How. “Chance Constrained RRT for Probabilistic Robustness to Environmental Uncertainty”. In: *AIAA Guidance, Navigation, and Control Conference*. DOI: 10.2514/6.2010-8160. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2010-8160. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2010-8160.

[28] Kevin M. Lynch and Frank C. Park. “Modern Robotics: Mechanics, Planning, and Control”. In: 1st. USA: Cambridge University Press, 2017. Chap. 2,3. ISBN: 1107156300.

[29] M.G. Mohanan and Ambuja Salgoankar. “A survey of robotic motion planning in dynamic environments”. In: *Robotics and Autonomous Systems* 100 (2018), pp. 171–185. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2017.10.011. URL: https://www.sciencedirect.com/science/article/pii/S0921889017300313.

[30] *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. Tech. rep. 1998.

[31] Luis Henrique Oliveira Rios and Luiz Chaimowicz. “A Survey and Classification of A* Based Best-First Heuristic Search Algorithms”. In: *Advances in Artificial Intelligence – SBIA 2010*. Ed. by Antônio Carlos da Rocha Costa, Rosa Maria Vicari, and Flavio Tonidandel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 253–262. ISBN: 978-3-642-16138-4.

[32] Meenakshi Sarkar, Prabhu Pradhan, and Debasish Ghose. *Planning Robot Motion using Deep Visual Prediction*. June 2019.

[33] Pratibha Sharma, Manoj Diwakar, and Niranjan Lal. “Article: Edge Detection using Moore Neighborhood”. In: *International Journal of Computer Applications* 61.3 (Jan. 2013). Full text available, pp. 26–30.

[34] Anthony Stentz. "Optimal and efficient path planning for partially-known environments". In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation* (1994), 3310–3317 vol.4.

[35] Xiaoxun Sun and Sven Koenig. "The Fringe-Saving A* Search Algorithm - A Feasibility Study". In: *IJCAI*. 2007.

[36] Richard S. Sutton. "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming". In: *Machine Learning Proceedings 1990*. Ed. by Bruce Porter and Raymond Mooney. San Francisco (CA): Morgan Kaufmann, 1990, pp. 216–224. ISBN: 978-1-55860-141-3. DOI: https://doi.org/10.1016/B978-1-55860-141-3.50030-4. URL: https://www.sciencedirect.com/science/article/pii/B9781558601413500304.

[37] Sangli Teng et al. "Toward Safety-Aware Informative Motion Planning for Legged Robots". In: *ArXiv* abs/2103.14252 (2021).

[38] Emmanouil Tsardoulias et al. "A Review of Global Path Planning Methods for Occupancy Grid Maps Regardless of Obstacle Density". In: *Journal of Intelligent & Robotic Systems* 84 (2016), pp. 829–858.

[39] Jiankun Wang et al. "A survey of learning-based robot motion planning". In: *IET Cyber-Systems and Robotics* n/a.n/a (). DOI: https://doi.org/10.1049/csy2.12020. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/csy2.12020. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/csy2.12020.

[40] Martin Wermelinger et al. "Navigation Planning for Legged Robots in Challenging Terrain". en. In: *Proceedings of the 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2016)*. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2016); Conference Location: Daejeon, South Korea; Conference Date: October 9-14, 2016. Zürich: ETH Zürich, 2016. DOI: 10.3929/ethz-a-010686519.

[41] Yajue Yang, Jia Pan, and Weiwei Wan. "Survey of optimal motion planning". In: *IET Cyber-Systems and Robotics* 1.1 (2019), pp. 13–19. DOI: https://doi.org/10.1049/iet-csr.2018.0003. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-csr.2018.0003. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-csr.2018.0003.

[42] Qingfeng Yao et al. "Path Planning Method With Improved Artificial Potential Field—A Reinforcement Learning Perspective". In: *IEEE Access* 8 (2020), pp. 135513–135523. DOI: 10.1109/ACCESS.2020.3011211.

[43] Jun Zeng, Bike Zhang, and Koushil Sreenath. "Safety-Critical Model Predictive Control with Discrete-Time Control Barrier Function". In: *2021 American Control Conference (ACC)*. 2021, pp. 3882–3889. DOI: 10.23919/ACC50511.2021.9483029.