## ABSTRACT

| | |
|---|---|
| Title of Thesis: | Planning Footsteps of Humanoid Robots |
| Degree Candidate: | Timothy Charles Giles Jr. |
| Degree and Year: | Master of Science, 2018 |
| Thesis Directed By: | David C. Conner, Ph.D., Assistant Professor, Department of Physics, Computer Science and Engineering |

Problems that are solved intuitively by humans become difficult to replicate in a machine. For example, humans are able to walk around and navigate environments without having to consciously think about which footsteps to take. This thesis focuses on a subset of motion planning problems: footstep planning and execution on a humanoid robot. If a robot is not able to accurately plan and execute a step plan, then further functionality of the robot will be hindered. Therefore, we implemented accurate planning and execution using state of the art tools for a small laboratory humanoid robot.

**PLANNING FOOTSTEPS OF HUMANOID ROBOTS**

by

Timothy Charles Giles Jr.

Thesis submitted to the Graduate Faculty of
Christopher Newport University in partial
fulfillment of the requirements
for the degree of
Master of Science
2018

Approved:

David C. Conner, Chair _____

Roberto Flores _____

Anton Riedl _____

## DEDICATION

To my father, for his love and support, and for being an example of hard work and perserverance. To my grandparents, for their love and support. Thank you for everything.

## ACKNOWLEDGEMENTS

I would like to thank my committee for taking time out of their busy schedules to work with me and help to refine my thesis. None of this would have been possible without my adviser, Dr. David C. Conner, and his knowledge of robotics and artificial intelligence. Thank you for directing me through this process and making sure I could see the forest from the trees. I also thank Dr. Roberto Flores for sparking my interest in robotics and artificial intelligence. Thank you for the countless discussions and the advice you have given me over these years. Furthermore, I thank Dr. Anton Riedl for helping provide the right language needed for this thesis. I also thank Alexander Stumpf because without his hard work and feedback, I would not have a working framework for my thesis. Finally, I thank my professors at CNU that have positively impacted my life.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# CHAPTER I:
# INTRODUCTION

As humans advance technology, they tend to adhere to the policy of work smarter, not harder. In the past, the mindset was that robots would eventually perform mundane tasks. This would free up the owner to perform more relevant tasks as well as other tasks that robots were unable to solve. However, solving most problems with robots is not a simple task.

When humans try to model themselves via robots, problems that can be solved intuitively become difficult to replicate in a machine. Humans are able to pick up objects, manipulate doors and other tools, as well as walk around without having to consciously think about what steps to take. To describe these problems and solve them mathematically takes a great deal of effort. Because of this difficulty, we will focus on one particular problem, planning a successful sequence of footsteps. We will then take this plan and have a small humanoid robot execute this plan.

Currently there is research that focuses on using robots in disaster situations to provide humanitarian relief [1–4]. This would provide a solution that is less costly than inserting human workers into a dire situation. Although relevant in a real robotic system, we will focus on accurate planning and execution of locomotion.

While wheeled motion is easier to model and solve, it is not as versatile as bipedal motion. Given some room where the only path is to step over and onto obstacles, a wheeled robot would likely not be able to solve the problem. This type of scenario can occur in disaster situations where man-made environments degrade and become cluttered [1].

Our humanoid robot of choice will be the NAO from Aldebaran, which is available in our lab. Although our humanoid, seen in Figure 1, is too small to be effective in one of the

Figure 1: NAO robot

situations presented at the DARPA Robotics Challenge [1], we can still model and solve a scale version of a similar environment. Using NAO as a scaled model will allow us to demonstrate our footstep planner.

While there are 2D motion planning applications that have a successful implementation for the NAO, these applications are obsolete[1]. Additionally, there are 3D motion planning frameworks that have been implemented for larger humanoids but, to our knowledge, none of these have been implemented on the NAO. Without accurate footstep planning and execution, further abilities of the robot can be hindered. In our disaster relief situation, this type of error is unacceptable and helps provide motivation for a working and accurate solution. Applying these state of the art tools allows us to solve one part of the complex robotic system.

[1]These applications were developed in 2014 and are not maintained

The goal of this thesis is to implement a state of the art footstep planner using the NAO's configuration, while the objectives of this thesis are:

1. Create accurate configuration files for the state of the art footstep planner
2. Investigate apparent state dependency issue with the footstep planner
3. Solve issues that prevent accurate 2D planning in the system
4. Create custom cost function for planner that generates desired paths

In Chapter II, we discuss background and related work. After this we declare the scope of this thesis in Chapter III. We discuss the work needed to improve the original planner in Chapter IV. After implementing our changes, we compare our results against baseline results in Chapter V. Finally, we summarize our work and present future work opportunities in Chapter VI.

# CHAPTER II:
# BACKGROUND

## NAO

NAO is a humanoid open source robot from Aldebaran. Our NAO is an H25 V5 model which is the most current model. It is 574mm tall, 275mm wide, has a depth of 311mm, and a weight of 5.4kg [5]. Our NAO offers 25 degrees of freedom which allows it to mimic human physiology [6]. It has four directional microphones, two cameras, one speaker, and touch sensors. These input and output devices allow accurate feedback with its target audience.

NAO features "autonomous life" as Aldebaran describes it [7]. The robot is able to determine new and old faces, as well as follow a moving face. This is possible due to the two onboard cameras and native image segmentation software. NAO is also able to listen to English, French, and Japanese questions and answer in the respective language. To the untrained eye, these behaviors make NAO appear alive, but are only programmed performances. Manipulating and creating new behaviors is achieved through a few different tools: *Choregraphe*[2], Python, or C++.

*Choregraphe*, shown in Figure 2, is a GUI tool that Aldebaran provides to chain stock behaviors to define new behaviors. A developer is also able to write new behaviors in Python and add them into *Choregraphe*. This allows rapid prototyping of new behaviors without being bogged down in other ecosystems, such as NAOqi or Robot Operating System (ROS). The main disadvantages with using *Choregraphe* is that a developer is limited in use of the NAOqi API, and behaviors will execute slower compared to their pure C++ or Python equivalents [8].

[2] `http://doc.aldebaran.com/1-14/software/choregraphe/choregraphe_overview.html`

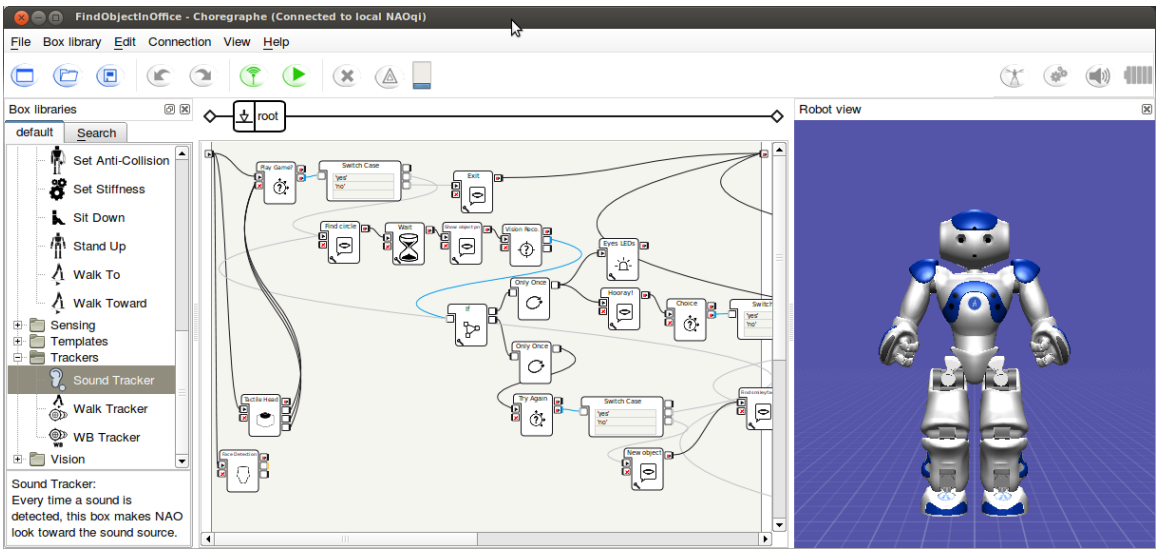


Figure 2: Aldebaran's *Choregraphe* interface

NAOqi is the framework Aldebaran implemented for the NAO that deals with common robotics needs such as parallelism, resources, synchronization and events [9]. It was also developed to be cross-platform and cross-language, allowing trivial use of the API in either language. According to Aldebaran, most behaviors will be in Python while services will be in C++. A behavior is a high-level capability of the robot, while a service provides efficient computation for services and behaviors.

Although not a focus of this thesis, it is relevant to understand the model of locomotion that the NAO uses. According to Aldebaran's documentation, the NAO uses a simple dynamic model inspired by the work of Kajita *et al.* [10] which is then solved using quadratic programming [11]. Essentially, every footstep is broken down into a double leg and single leg support round. During the double support round, the time taken is one third of the actual step time. Six degrees of freedom (6DOF) interpolation using the SE(3) transformation equations framework is used for the foot swing path. Then the walk is initialized, the NAO executes its plan, and ends with a round of double support [12].

## Applications of NAO

Most of the current applications of NAO are research based. The NAO provides a complete and affordable solution for researchers in artificial intelligence and robotics. Some of the focus areas are human-robot interactions [13], humanoid navigation [14–20], and collaborative robot systems [13]. Another research area revolves around programming NAOs to play a game of soccer. Robocup, the organization that hosts the competitions eventually wants to field a team of fully autonomous human sized humanoids, not NAOs, that will win a soccer game against the most recent World Cup champions [21]. In the medical field, NAOs have been used to help young patients in their physical rehabilitation process [22] as well as increase quality of life for elderly patients [23]. There are some commericial based applications, but these mainly fall under education tools [24, 25]. To make these applications of NAO easier to use and target a larger audience, most developers create the functionality using the Robot Operating System (ROS) ecosystem.

## Robot Operating System (ROS)

In order to advance robotics as a whole, Quigly *et al.* devised and created a framework called the Robot Operating System (ROS) [26]. This framework makes writing software easier for multiple robots with varying hardware. ROS is set up with extreme modularity in mind, "since the required breadth of expertise is well beyond the capabilities of any single researcher" [26]. The design goals of ROS can be summarized as thus: Peer-to-peer, tools-based, multi-lingual, thin, and free and open-source.

According to the authors, "...a central data server is problematic if the computers are connected in a heterogenous network" [26], therefore a peer-to-peer approach was chosen. There is an additional benefit with having a peer-to-peer model, robots on the network only communicate with each other as needed. Figure 3 shows this peer-to-peer model in more detail. This example illustrates multiple nodes communicating with each other in a real

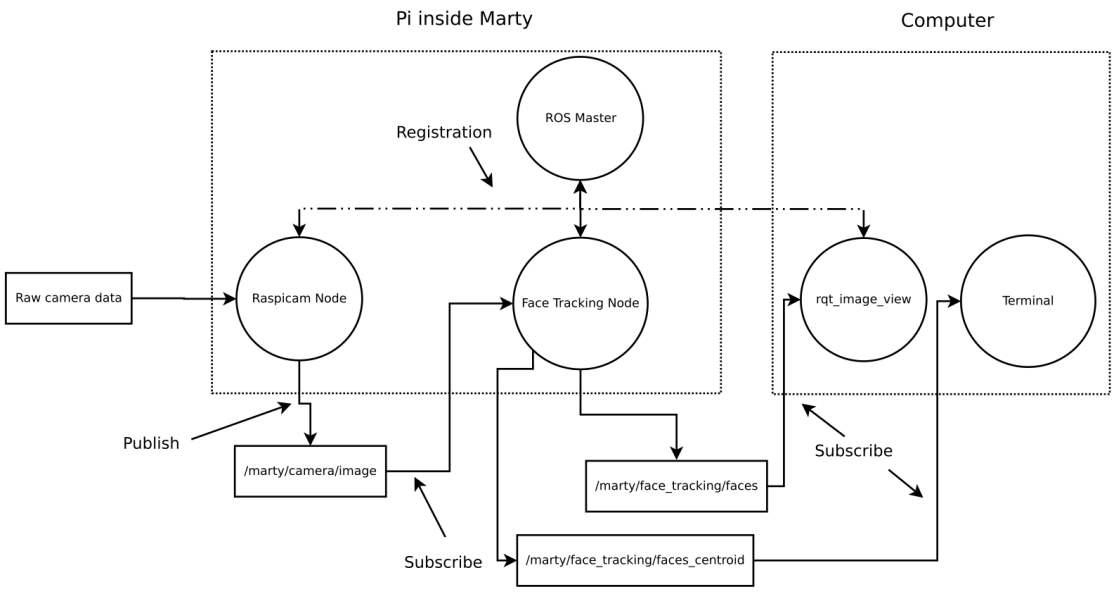



Figure 3: *Marty* is a robot that has a Raspberry Pi running ROS on it, while *Computer* is a different machine on the same network [27]

life situation using the publisher/subscriber model of communication in ROS. The "ROS Master", also know as "roscore", is a central master node that keeps track of all registered nodes so that communications can be efficient. Therefore, this saves precious bandwidth on the wireless bridge between machines.

By a tools-based approach, the authors mean using a microkernel and modular tools to build the runtime environment. Each tool in the stack has a directed purpose, such as: "navigate the source code tree...visualize the peer-to-peer connection topology..." [26]. The authors wanted to push everything into separate packages to reduce complexity of the system as well as stablize the system. This goal also falls in line with the thin methodology.

Another reason that ROS was developed was due to potentially reusable code being welded to middleware[3]. To move away from this high coupling, the authors "encourage all drivers and algorithm development to occur in standalone libraries that have no dependencies on ROS" [26]. This allows developers to create and unit test their software without being bogged down in whichever middleware they are using. The authors have also reused code

[3]Middleware is software that acts as a bridge between an operating system and applications

from other open-source projects to help develop the full ROS stack, which ties into the final goal of free and open-source.

While there are other robot operating systems, such as Webots and Microsoft Robotics Studio, ROS is the first that is free and open source. ROS features the BSD[4] license which allows developers to create commercial and non-commercial products. Since ROS is thin and does not need to link functional modules together, any projects "can use fine-grain licensing of their various components" [26].

ROS is comprised of nodes, messages, topics, and services. Nodes are the functionality that is developed, and are synonomous with software modules. We can think of these modules as literal nodes in our peer-to-peer network. These nodes communicate with each other through messages that are defined by ROS or a developer. These messages can be primitives, arrays of primitives, arrays of user defined types, other messages and more. ROS uses a publisher/subscriber approach for communications. A node can publish a message to a given topic, but nothing can be done with this message unless another node subscribes to the same topic. ROS also defines services, which have a specific request and response type. Services do not subscribe to topics, but are invoked with a specific passed message. For example, one could have a node that processes images with a service called, "process_rgb_image" that would need an rgb image passed to the service and return some modification of it.

### Integrating NAO with ROS

As previously stated, most researchers try to develop new robotics software in the ROS ecosystem to improve portability and code reuse. This is no different with the NAO robot. The ROS packages for NAO are essentially wrappers for the NAOqi API that exposes the

[4] `https://opensource.org/licenses/BSD-3-Clause`

needed parts to ROS [28]. There are three separate clusters of packages needed for the robot: basic configuration, hardware drivers and simulation, and high level capabilities.

In the basic configuration is a package called "naoqi_bridge_msgs" which describes robot specific messages and services. NAOqi is used by several Aldebaran robots, but the specific robot driver translates these messages into specific commands. The "nao_description" package publishes the robot's model to ROS in the Unified Robot Description Format (URDF), which is simply XML for describing a robot's model. Lastly, there is a package for furnishing 3D surface meshes to simulation software, aptly named "nao_meshes".

Next we move into the hardware layer which also provides functionality to simulation software. In Figure 4, the ROS packages would wrap around the outer box and send commands down to NAOqi. Since both ROS and NAOqi feature multiple languages, there are two launch points for the actuator and basic sensor drivers: a C++ or Python version. The missing piece in the hardware section is a package called "nao_dcm_bringup". This node starts up the Device Communication Manager (DCM). The DCM is one of the most important parts of the NAO since it is "in charge of the communication with almost every electronic device in robots...excepting sound...and cameras" [29]. Once the DCM is running, a user is able to send commands to the robot via ROS messages, nodes, and services. There is no way around the DCM, as it also runs some security needed for the robot to function [29].

Lastly, there are high level capabilities that have been created by various researchers such as tele-operation, pose management, path following, human-robot interaction, general planning, and footstep planning [28]. The tele-operation package provides the ability to use a gamepad or joystick to remotely control NAO. The pose management packages allows a user to send a predefined pose name message to NAO. This stack includes controller for manipulating joints to achieve the pose goal. NAO is able to walk to a target location or follow a planned 2D path closely via the "nao_path_follower" package. The interaction stack is actually a metapackage, or a package that describes other packages; these packages are

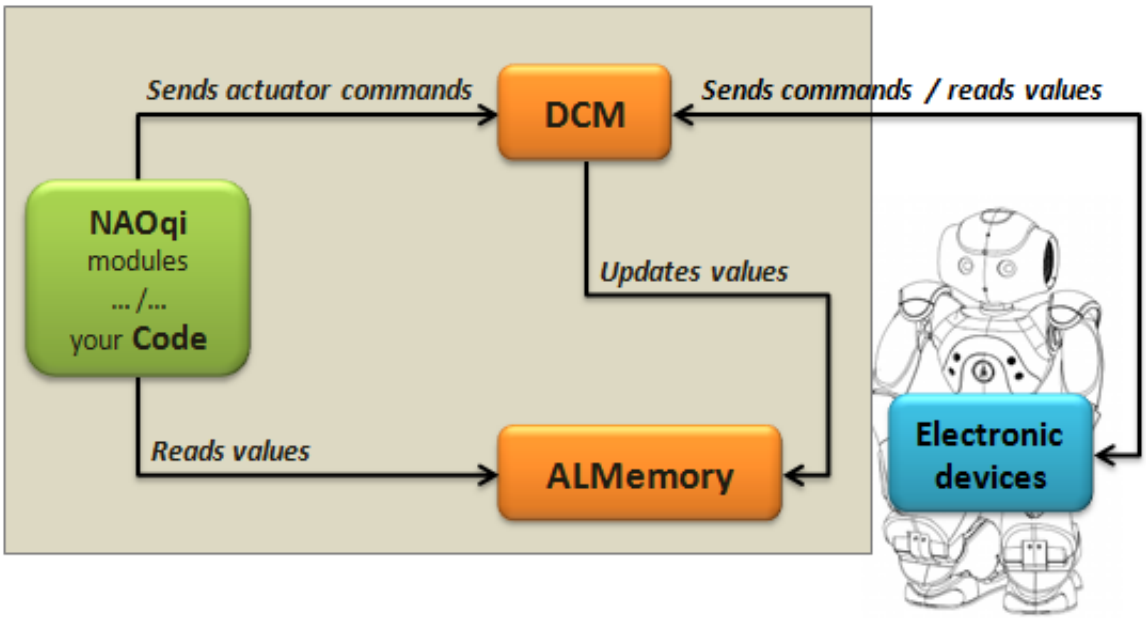



Figure 4: Command flow in NAO [29]

"nao_audio","nao_interaction_launchers", "nao_interaction_msgs", and "nao_vision". These modules are simple wrappers of the NAOqi API. In the general planning sense, joint control and manipulation, we can use the integrated MoveIt![30] framework. MoveIt! is an open-source motion planning library that allows real-time manipulation of robot joints through a graphical user interface. In our case, we use RViz which is a ROS application for visualization. Figure 5 shows NAO's left arm being pulled towards the sky. A user then could execute this command and NAO's arm will move to the planned position.

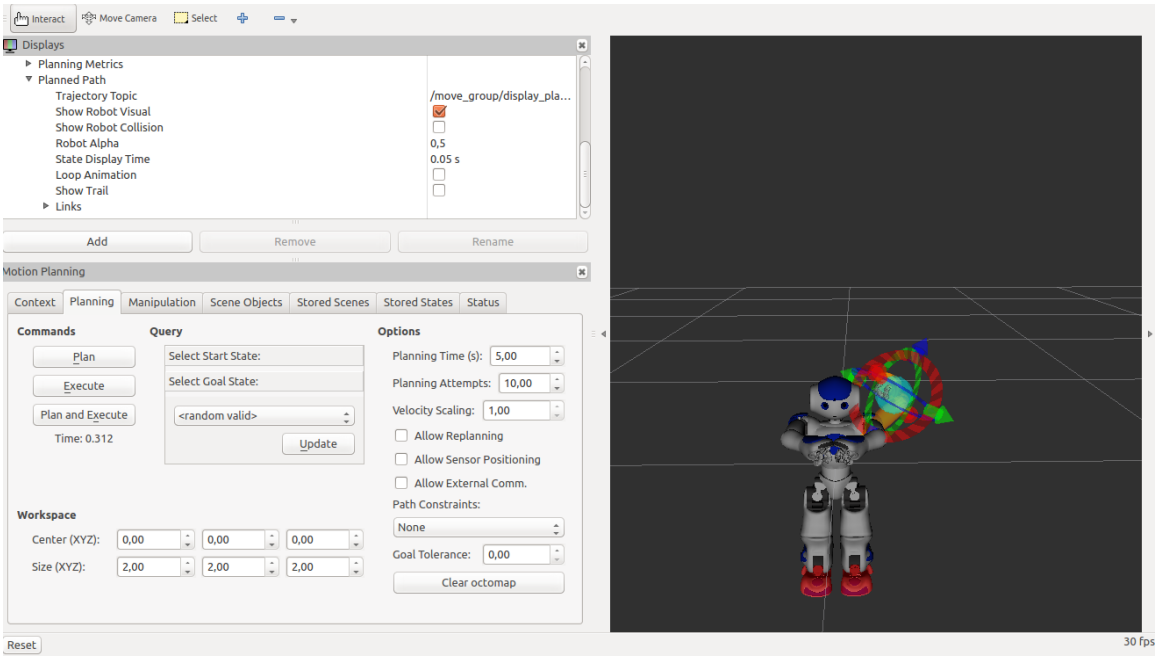


Figure 5: MoveIt! inside of RViz [31] interface

## Graph Planning

Path planning problems can be modeled as abstract graphs, therefore we may use graph search algorithms to effectively solve these problems. One of the most important of these algorithms is A-Star (A*)[5]. A* is an extenstion of Dijkstra's that uses a search heuristic to direct which nodes are expanded first [32]. The main benefit of A* is that it is optimal, or it will find a solution if one exists. However, it is wholly dependent on the heuristic to avoid expanding irrelevant states. This is one of the major benefits of Weighted A* (wA*) [33].

Weighted A* (wA*) behaves the same as A* except that wA* will inflate the heuristic function by some factor $\epsilon \geq 1$. By doing so, wA* will potentially find a path faster by performing a greedy search based on the heuristic. Using wA* guarantees that our path will cost no more than $\epsilon$ times the cost of an optimal path. However, wA* does not reuse

[5]Details of the described algorithms can be found in Appendix A

information, so solving a dynamic graph will complicate this algorithm. To solve this issue, other planners were developed, such as D* Lite and Anytime Repairing A* (ARA*).

As with wA*, D* Lite is an extension of A* but also an improvement of D* [20]. D* Lite excels in efficient incremental searches, i.e. dynamic environments. However, for the algorithm to perform efficiently, it searches in reverse order from goal state to current state. The first search is simply an A* search since there is no information available to be reused. For every subsequent search, D* Lite will maintain the costs for all visited states as a heuristic cost when determining the current optimal path from the current state to the goal state. D* Lite is one approach to solving dynamic path planning problems, however it requires precomputation in order to be successful. In contrast, Anytime Repairing A* (ARA*) does not require precomputation to provide a solution.

As with previous algorithms, Anytime Repairing A* (ARA*) is an extension of A*. ARA* search executes multiple wA* searches while efficiently reusing previous information, similiar to D* Lite. On the first ARA* search, our heuristic factor $\epsilon$ is set to be large, usually $\epsilon > 5$. In constrast to standard A*, we will use whatever computational time we have left to create a more optimal solution. The next iteration of ARA* will reduce $\epsilon$ by some increment to try and find a better path. If ARA* is unable to find the best path in time, we know that our path will cost no more than $\epsilon$ times the true cost. This approach allows ARA* to find a path faster than regular A* but also approach optimality if given enough time. Again, ARA* is dependent on the search heuristic; however, the authors propose that randomized A* (R*) will solve this problem [16].

Randomized A* (R*) search is designed to have less dependency on the quality of the heuristic function. One of the main issues with graph search algorithms is the procedure entering a local minimum and wasting computation when exiting the minima. R* avoids

this by executing a "series of short range, fast wA* searches towards randomly chosen subgoals." [16]. As with ARA*, R* will iteratively lower the inflated heuristic value $\epsilon$ and rerun the search if time permits. Unlike ARA*, R* ensures exploration of the search space, i.e. it should avoid local minima.

### Humanoid Robot Footstep Planning

From the previously described high-level capabilities[6], we have a useful package called "footstep_planner". This planner is not specific to NAO, but is a generic humanoid planner. However, creating an accurate plan for humanoids is not a simple task.

Garimort *et al.* developed an approach for creating dynamic footstep plans and an implementation of D* Lite, an incremental heuristic search algorithm, to solve for the most optimal set of footsteps between start and goal [20]. Unlike with wheeled motion, which has up to two degrees of freedom, humanoid locomotion has up to six degrees of freedom per foot. Garimort's D* Lite separates each foot which brings the degrees of freedom down to three per foot, since the elevation, roll, and pitch of the feet is fixed for each step. Even with three degrees of freedom, creating and expanding successor states can become computationally expensive. The authors try to minimize this cost using the D* Lite algorithm. They state that D* Lite's "replanning capabilities can be exploited when the humanoid seriously deviates from the computed path during execution, when non-static obstacles change their locations, or when bad terrain locations are observed along the way" as a major incentive for implemenation [20].

Hornung's results, reproduced in Table 1, show the experimental results of using R* and ARA* for footstep planning in a cluttered environment. Both R* and ARA* found a solution quickly compared to A*, but these paths were suboptimal by a factor of $\epsilon$. If a quick reaction time is needed, R* and ARA* significantly outperform A*, while also giving a good enough

[6] `http://wiki.ros.org/nao`

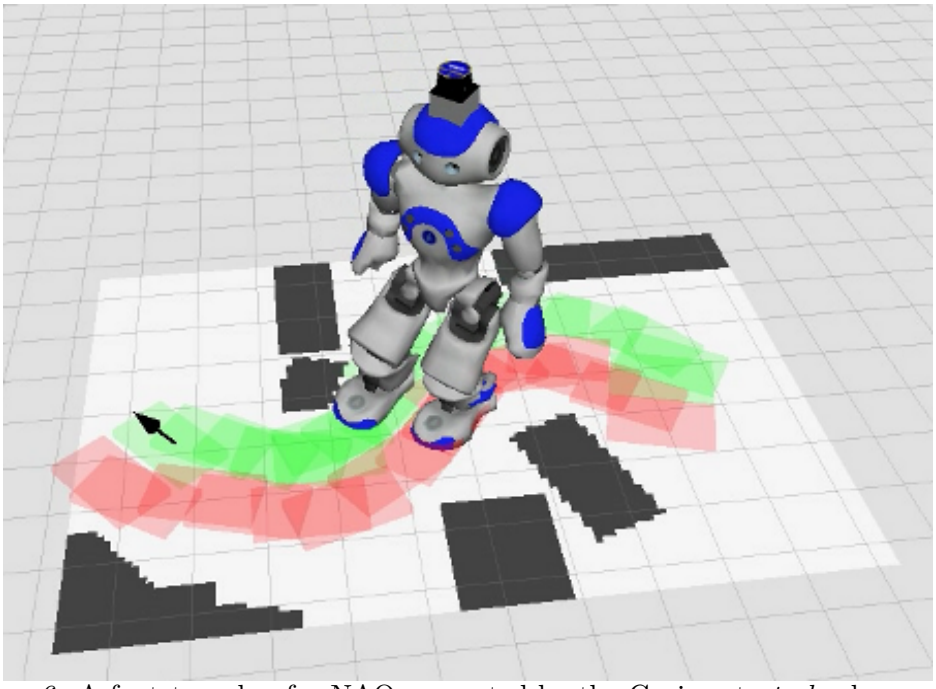


Figure 6: A footstep plan for NAO generated by the Garimort *et al.* planner [20]

path. The authors also proclaimed that these graph search algorithms could be extended into a 3D space. The implemenation of 3D motion planning would not be developed by Hornung and his colleagues due to them finishing their time at their university.

Table 1: Performance Comparison For First Solutions In A Densely Cluttered Environment. Table from [16] © 2013 IEEE

| Planner | Heuristic | Planning time [s] | Path costs |
|---|---|---|---|
| R* ($\epsilon$=5) | Euclidean | 0.32 ± 0.23 | 16.45 ± 3.16 |
| ARA* ($\epsilon$=5) | Euclidean | 2.15 ± 2.21 | 13.57 ± 1.15 |
| ARA* ($\epsilon$=5) | 2D Dijkstra | 0.56 ± 1.13 | 20.41 ± 5.08 |
| Optimal: A* ($\epsilon$=1) | Euclidean | 33.31 ± 15.00 | 11.06 ± 1.20 |

### Interactive 3D Footstep Planning

There are a few motivators for creating a 3D planning framework such as: the DARPA Robotics Challenge (DRC)[34], disaster scenarios, and real-world planning. In 2012, a prize competition called the DARPA Robotics Challenge was funded by the US Department of Defense to innovate human-robot interaction and coordination [2–4, 34]. The primary goal

of this challenge was to develop robots that could perform complex tasks in hazardous, man-made environments. Given that man-made enviroments exist in a 3D space, this presented a problem to the teams in the DRC. While a 2D footstep planner existed in Hornung's work [35], no one had implemented the 3D extension to his planner. Because of missing features and lack of a standardized open source framework, two separate research groups created their own open source motion planning frameworks, HPP and Team ViGIR's Footstep Planner [36, 37].

Mirabel *et al.* describe HPP as "software designed for complex classes of motion planning problems..." [36], such as: navigation among dynamic objects, multiped[7] locomotion, among others. Their motivation is based on constraint-based motion planning, such as multi-contact planning and Navigation Among Movable Obastacles (NAMO). The given example for multi-contact planning is when "an under-actuated multiped robot can only move through the contact forces exerted by its effectors on the environment" [36]. For NAMO, this involves manipulating an object out of the way so a clear path can be executed.

To further the point of constraint-based motion planning, HPP supports constraint graphs. According to the authors, using constraint graphs allows "the integration of the discrete, higher-level task scheduling problem into the motion planning problem" [36]. This is advantageous when dealing with NAMO environments since parallel motion problems may need to be solved for a working solution.

HPP uses abstract classes so a developer can focus on the details for a specific robot. The framework is designed that it can be used out of the box or used to test and implement new planning algorithms. HPP does not use any remote operator feedback or use perception data from the robot to improve calculations, where Team ViGIR's framework does use this data [37].

[7] Multi-legged but described as "multiped" in [36]

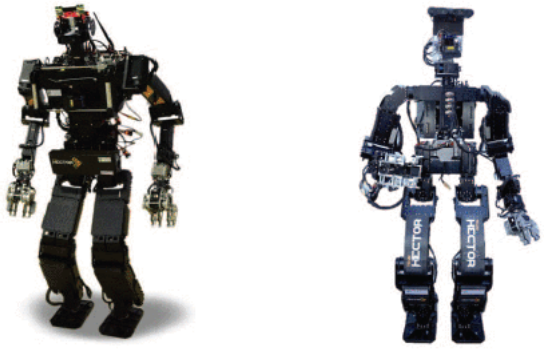
In contrast, the planner developed by Stumpf for Team ViGIR uses a decoupled step planner and walking controller, and allows a remote operater to change single steps then check to see if the new plan is still feasible. If the new plan cannot be done, the planner gives that feedback and allows the operator to move more single steps or allow the planner to create a new plan from scratch. Therefore, for the focus of this project, we will use Stumpf's framework over HPP.

Since there are few planning frameworks for humanoid navigation, resources are wasted in reimplementing these foundations for a particular robot. This motivated Stumpf to create a ROS package that would simplify this process. Additionally, the need to include ROS drove some of the functionality for the planner, as it "has to provide a complete set of basic functionalities, but also has to be extendable for new application-specific features"[37]. Because his design uses a plugin and parameter system[8], new users can add modules to the framework for their specific robot without having to worry about reimplementing complex algorithms. Furthermore, this design decreases chances of errors and reduces development time for new features. Figure 7 shows four different humanoid robots that helped provide motivation for this planner.

One of the design goals for this plugin management system is keeping the overhead cost low to maintain efficient planning. To execute this plan, the planner requests all plugins needed for a single iteration of planning. The planner will then initialize these plugins with passed parameters for the particular execution. So now we have this fully-fledged framework extensible to particular use cases.

[8]More details of Stumpf's footstep planner plugin model can be found in Appendix B

(a) Atlas (DRC Finals) (b) ESCHER



(c) Johnny #5 (DRC Finals) (d) Johnny #5 (2016)

Figure 7: Robots that have a successful implementation of Stumpf's footstep planner. Image from [37] © 2016 IEEE

The framework also includes a common interface to step controllers on the robots. This tool allows easy access to low-level walking controllers as well as seamless integration of step planning updates. The implemenation of this tool follows the rest of the framework and thus can be adapted by implementing a robot specific *StepControllerPlugin*[9]. For remote supervision purposes, this controller will frequently report the internal state and show step execution feedback in the attached graphical user interface.

In order to deal with 3D environments, the footstep planner also comes with a terrain modeler. While there are other approaches for dealing with mapping or surface reconstruction [38], Stumpf decided to create a light-weight approach to model 3D environments. This model consists of a grid-based elevation map and a 3D octree-based structure [38] that will hold point cloud data and the estimated normals. This approach simplifies the 3D

[9]Further details about this plugin can be seen in Appendix B

planning problem, taking 6D state space and reducing it to a 3D space. This is calculated by looking at the x, y, yaw of each footstep and projecting foot sole onto the surface to determine elevation, roll, and pitch of the foot. Figure 8 shows the terrain modeler in action in Stumpf's planner. The middle column shows the point clouds that the robot is generating via sensors as well as the proposed footstep plan. The right column shows a remote operator view of the current environment after the point clouds have been used to model the terrain. This approach provides a method for online 3D footstep planning on any robot system that can generate point clouds in a global reference frame.

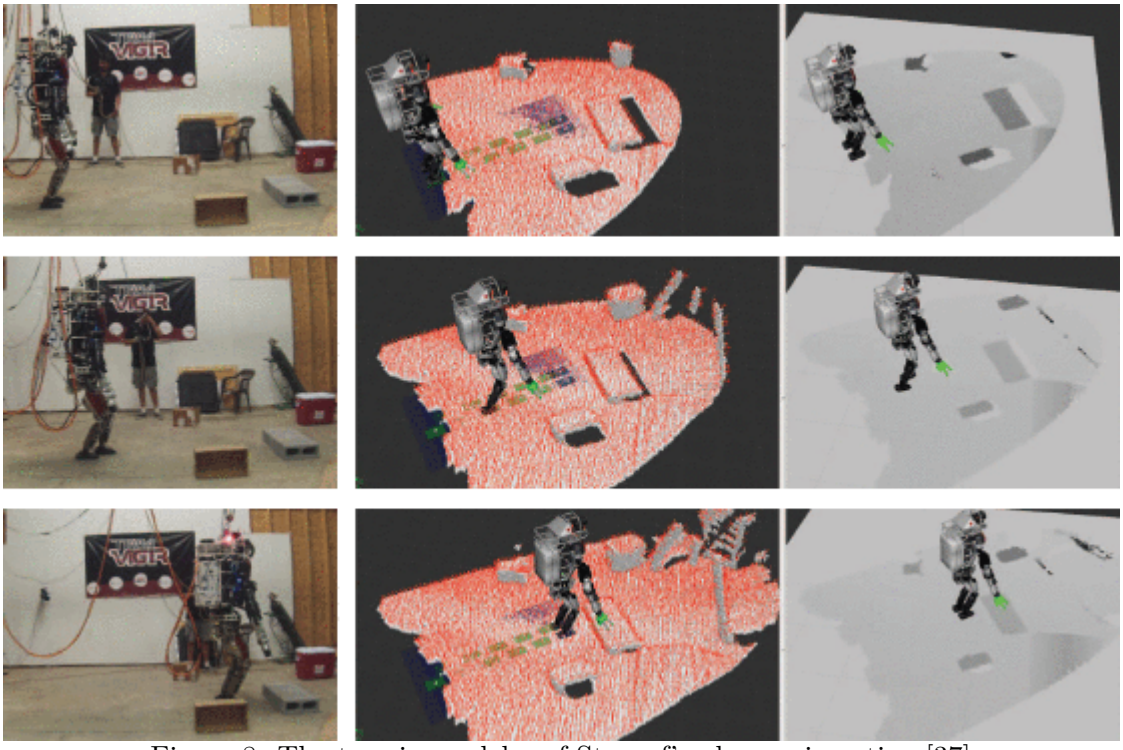


Figure 8: The terrain modeler of Stumpf's planner in action [37]

Another change, from Hornung's approach, that Stumpf introduced is operator interaction. With previous implemenations of footstep planning, a plan would be generated and executed without input from a remote operator. In most cases, this would be acceptable; however the robot will sometimes plan a risky footstep that endangers the rest of the plan's execution. To avoid this issue, Stumpf designed the framework in such a way that allows immediate

and constant feedback of the footstep plan. This feedback is described as "collaborative autonomy" and uses state machines to allow operator validation prior to state execution [39]. These services allow the remote operator to aid the planner by adjusting single steps, afterwards the planner will readjust the change into 3D space. Then the planner will give immediate feedback whether this adjusted footstep is safe or not. According to the authors, this interactive feedback approach "significantly improves mission performance during locomotion tasks" [37].

This feedback is vital when working in challenging environments, i.e. complex man-made locations, disaster situations, and so on. The remote operator must be aware of the conditions surrounding the robot, as well as the state of the robot. Given complex, cluttered environments, footstep planning becomes more difficult. These conditions can cause immense slowdown of finding a path, and without some type of feedback, an operator can only guess as to what the issue may be. With immmediate feedback, operators can preempt certain steps to help the planner solve the path more quickly.

### Flexible Behavior Engine

The Flexible Behavior Engine (FlexBE) [40] uses hierarchical finite state machines to control robot behaviors and execution. Using this hierarchical approach allows the development of complex behaviors, i.e. multiple state machines, in an intuitive container. FlexBE also allows robotic behaviors to have human input in determining state changes. Essentially, a robot can ask its operator for help when transitioning from state to state or vice-versa. FlexBE was designed to be used by an operator that is not a developer, sporting an intuitive graphical user interface to manage the different states. Figure 9 shows the different states and state machines used, such as *Locomotion Config*, a hierarchical finite state machine, and *Decide_Step_Back*, a simple finite state machine, and various transitions based on the outcome of a particular state. This figure also shows different colored transitions, green, red, and gray, to show different levels of autonomy when a transition is triggered.

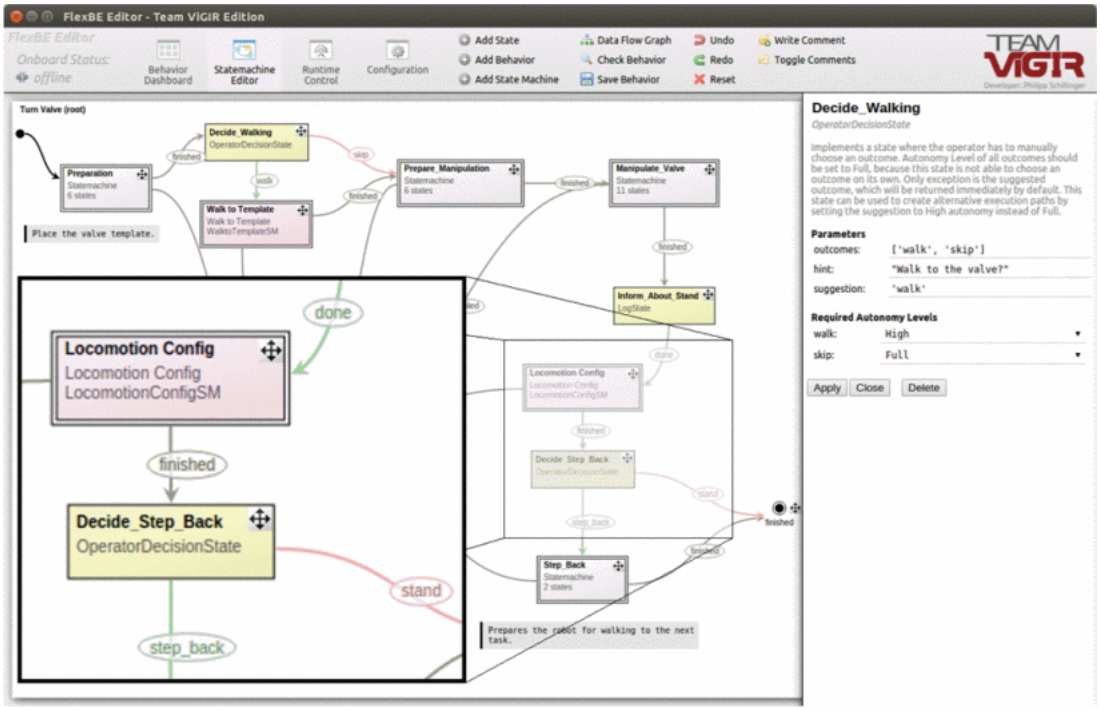



Figure 9: FlexBE's graphical user interface. Image from [40]

An operator is able to manage the states as well as the autonomy level need to trigger a state transition. For example, there are four levels of autonomy: High, Medium, Low, and None. Any state transition can have its own independent level of autonomy, so that low-risk tasks will not interrupt the remote operator while high-risk tasks will interrupt. At the same time, these levels can be changed on the fly allowing responsive behaviors due to dynamic changes in environment.

Additionally, behaviors may be modified on the fly as well. Because of the design of the framework, an operator, who is not skilled in software developement, is able to add and change states to create new behaviors. This system is akin to Aldebaran's *Choregraphe*, but is free and open source, and designed specifically for use in ROS.

Another benefit of FlexBE is the native integration of the planner and FlexBE. A sample behavior involving the planner and FlexBE would be:

1. Ask the operator for a goal pose in RViz

2. Plan a path between start and goal poses
3. Ask operator to approve or deny the plan
4. If approved:
    (a) Execute each step
    (b) If any step fails, abort operation and ask operator for new start and goal poses or quit behavior
5. If denied:
    (a) Ask for new start and goal poses and replan or quit behavior

After the initial states are created, any future operator can reuse states to create new behaviors that fit their needs. While the ViGIR footstep planner has been used on larger humanoid robots such as Atlas, ESCHER and THORMANG, previously seen in Figure 7, to our knowledge no other group has implemented it on the smaller NAO humanoid. Additionally, to our knowledge, there has been no integration of NAO and the Flexible Behavior Engine [40].

# CHAPTER III:
# PROJECT SCOPE

## Problem Statement

In order to help advance the field of robotics, we implemented Stumpf's footstep planner using NAO's configuration. This allows other researchers to study footstep planning algorithms for humanoids on a platform that costs significantly less than ATLAS or THORMANG, previously seen in Figure 7. However, to get Stumpf's planner working, we needed to create the proper robot configuration in his framework, as well as create custom plugins to achieve desired walking behavior. Solving localization of robot in motion is outside of scope of this thesis, but will influence how the robot executes footsteps in the real world. Although Stumpf's planner has 3D path generation functionality, 3D planning and footstep execution is outside of scope for this project. For 3D planning, we need a whole-body motion controller which is something that is not available in the current ROS NAO stack. Additionally, for proper 2D or 3D footstep plan execution, a custom NAO hardware driver is needed.

## Motivation

Although we are not using the full functionality of the Team ViGIR footstep planner, the system still provides more benefits to the end user than other comparable systems. For instance, ViGIR's planner is a standard ROS node using up-to-date interfaces for communication. This allows easier dialogue between other ROS nodes that an end user might use, such as RViz or the Flexible Behavior Engine (FlexBE)[37, 40]. Furthermore,

the planner is designed so that only a minimum amount of files need to be configured for a new robot. For our experiments, only two configuration files needed to be generated. The first file contains the robot's physical dimensions while the second file contains planner specific information. These files will be discussed in detail in Chapter IV.

## Scope

The scope of this thesis is defined as such:

- Creating accurate NAO configuration files for ViGIR's planner
- Investigate issues with apparent state dependent planning issues in current open source implementation
- Fixing bugs that prevent accurate 2D planning
- Creating a custom cost function that creates desired walking path generation

Items that are relevant to accurate footstep execution and the ViGIR planner, but will not be covered:

- A whole body robot controller. This functionality is not included in the standard NAO ROS stack and would require a sigificant amount of work to implement.
- Accurate localization. Without proper localization, the robot will drift during the execution of footstep plans and finish with some deviation of the goal pose.
- Enabling 3D planning component of ViGIR planner. Without a whole body controller, accurate 3D motion and balancing will be impossible.

# CHAPTER IV:
# SYSTEM DEVELOPMENT

## Footstep Planner Tutorial

For Team ViGIR's footstep planner, two files are launched: one containing the robot's configuration and the other to start up RViz, which allows the user to graphically plan start and goal poses. The launch file that deals with the robot's configuration has two corresponding YAML[10] files, one dealing with physical dimensions of the robot, the other contains parameters for the planner[11]. For the physical dimensions file, typically called "robotName_params.yaml", a size of the upper body, in meters, is needed. This allows the planner to check for obstacles that are at body height, an essential part of robot path planning, as well as visually see the body positions of the resulting path in 3D space in RViz. In the same file, the size of the feet, in meters, is needed as well as, nominal separation distance between the feet, and the transform (tf)[12] frame id and coordinates for each foot[13]. For the planner parameters file, typically called "robotName_planning_params.yaml", this is where a user would specify a particular plugin set and which search algorithm to use, how much time the planner has to search, and the discrete cell size for the robot. This file also allows the user to define the max toe in, toe out angle for the robot and, most importantly, the range, in meters, of possible steps.

## Original Planner

During our experiments with the footstep planner, we discovered a major bug that prevented the system from accurately executing the ARA* algorithm. Instead of searching around

[10] YAML Ain't Markup Language `http://www.yaml.org`
[11] These details can be seen in Appendix C
[12] `http://wiki.ros.org/tf` Package used to keep track of multiple reference frames
[13] Sample and NAO configuration can be seen in Appendix C

obstacles, the planner appeared to search states that were previously expanded. Because of this behavior, seen in Figure 10, the planner could not solve trivial environments.

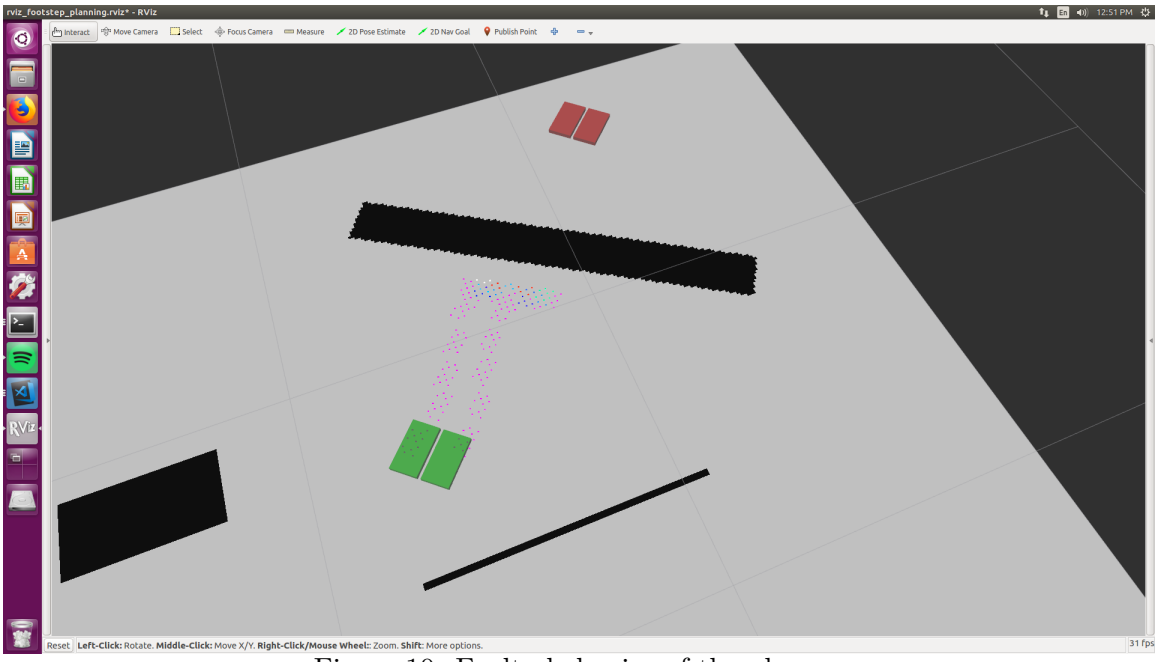


Figure 10: Faulty behavior of the planner

We determined that this bug was because of a calculation for comparing footstep states, found deep in the footstep planner codebase. This equality calculation, seen in Figure 11, was iterating over the entire potential path and comparing each field of each previous and successor state in the path. Each state is compared by a hashtag, which is generated by a state's x, y, yaw, and leg name. If these are equivalent, then the operation wanted to compare the current state's predecessor to the compared state's predecessor. This would recursively call the previously defined equality calculation and start the process over until the current leg and prior leg were the same. After these calls were finished, then the calculation would check if the current state's successor is the same as the compared state's successor. Again, this would start another recursive stack of equality calculations until all successor states were compared.

```
1  bool PlanningState::operator==(const PlanningState& s2) const
2  {
3      if (ivHashTag != s2.getHashTag())
4          return false;
5  // Performance loss with the following statement
6      if (ivpPredState != s2.ivpPredState)
7          return false;
8  // Performance loss with the following statement
9      if (ivpSuccState != s2.ivpSuccState)
10         return false;
11     return (ivX == s2.getX() && ivY == s2.getY()
12          && ivYaw == s2.getYaw()
13          && ivState.getLeg() == s2.ivState.getLeg());
14 }
15
16 bool PlanningState::operator!=(const PlanningState& s2) const
17 {
18     return !operator==(s2);
19 }
```

Figure 11: The faulty planning state equality calculation

After fixing this, seen in Figure 12, the planner behaved as expected; it would expand states around obstacles and calculate a path around instead of expanding the same states over and over. However, given start and goal poses and obstacles such as Figure 13, a path that involves strafing[14], or side to side, motion is generated. This path is produced by optimizing the euclidean distance function. Instead of this strafing path, we would prefer smooth curving motions which take advantage of the robot forward stepping distance.

[14]Term popularized by first person shooter video games, `https://en.wikipedia.org/wiki/Strafing_(gaming)`

```
1  bool PlanningState::operator==(const PlanningState& s2) const
2  {
3      if (ivHashTag != s2.getHashTag())
4          return false;
5      // if (ivpPredState != s2.ivpPredState)
6          // return false;
7      // if (ivpSuccState != s2.ivpSuccState)
8          // return false;
9      return (ivX == s2.getX() && ivY == s2.getY()
10             && ivYaw == s2.getYaw()
11             && ivState.getLeg() == s2.ivState.getLeg());
12 }
```

Figure 12: The corrected planning state equality calculation

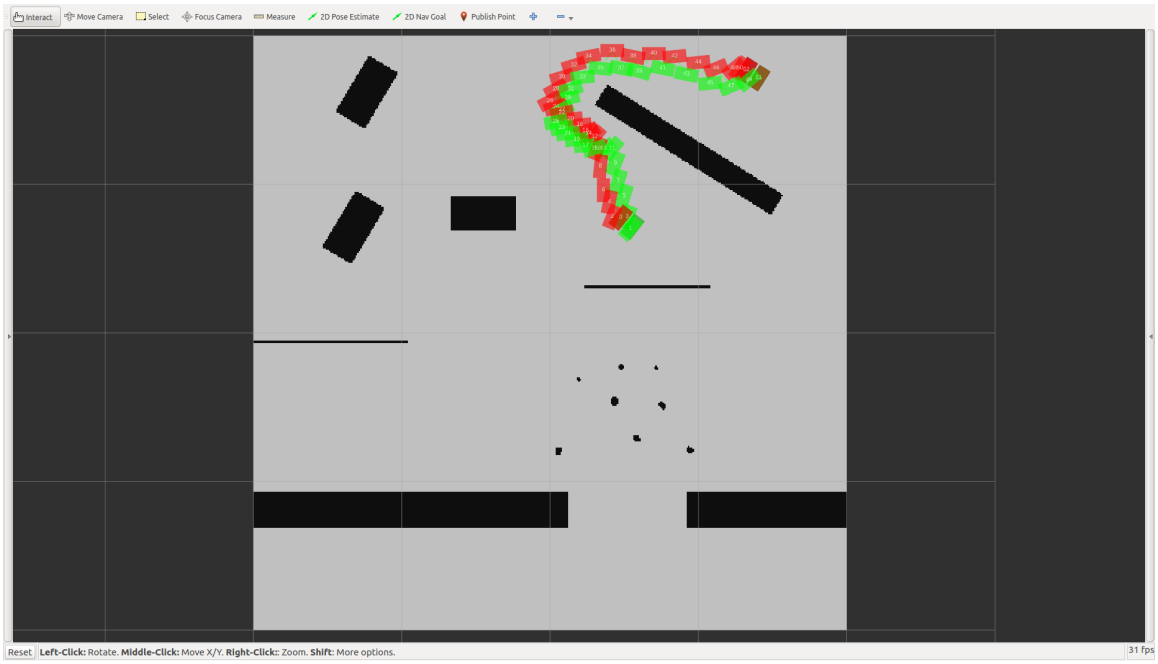



Figure 13: An optimal, but strafing, path from the planner

In the default setup, the cost function is comprised of three *StepCostEstimator* plugins: A constant cost, the euclidean cost, and a ground support cost which can be ignored in 2D planning. In other words, the default planner will try to minimize the total distance

traveled, which can lead to strafing and other behaviors that we do not necessarily want. In order to fix this strafing conduct, we needed to create a custom plugin that would use a cost function that avoids strafing.

### Discussion of Most Relevant Plugins

Stumpf's nomenclature, for our relevant plugins, can be seen in Table 2.

Table 2: Description of Relevant Plugins used in ViGIR planner

| Plugin Type | Description |
|---|---|
| Step Cost Estimators | References the estimate of the actual "real world" cost; in reality this is an arbitrary calculation of the assigned planning cost, and does not necessarily represent an actual cost such as energy or time. |
| Heuristics | References the estimate cost of the cheapest path from an arbitrary state to the goal. This cost does not necessarily represent an actual cost. |

While exploring this task, we discovered the needed plugin was a custom *StepCostEstimator*. The *StepCostEstimator* is used in the planner to determine the risk and cost of step transitions. In order to calculate a certain cost for various transitions, we implemented a custom cost function. The cost function we use is comprised of two parts, a scaling vector of constants and the actual cost calculation. The scaling vector, $K$, contains of five floating point numbers that help to incentivize certain behaviors. For example, seen in Equation

3, $K_0$ is the robot forward movement cost, $K_1$ is the robot backwards movement cost, $K_2$ is the robot lateral cost, $K_3$ is the robot strafing cost, and $K_4$ is the robot different angle cost:

$$\begin{aligned}
&\text{Let } \Delta x = \text{x-coordinate stance foot} - \text{x-coordinate swing foot} \\
&\text{Let } \Delta y = \text{y-coordinate stance foot} - \text{y-coordinate swing foot} \\
&\text{Let } \Delta\theta = \text{yaw stance foot} - \text{yaw swing foot} \\
&\text{Let } \Delta\tilde{x} = cos(\theta_{stance}) \cdot \Delta x + sin(\theta_{stance}) \cdot \Delta y \qquad (1) \\
&\text{Let } \Delta\tilde{y} = -sin(\theta_{stance}) \cdot \Delta x + cos(\theta_{stance}) \cdot \Delta y \qquad (2) \\
&\text{Let } f = 1.333,\ g = -33.33 \\
&\text{Let } A = f - g \cdot \Delta\tilde{x} \text{ where } 0 \leq A \leq 1 \\
&\text{Let } C = K_0 \cdot \max(0, \Delta\tilde{x}) \qquad (3) \\
&\qquad + K_1 \cdot \max(0, -\Delta\tilde{x}) \\
&\qquad + K_2 \cdot \text{abs}(\Delta\tilde{y}) \\
&\qquad + K_3 \cdot \min(1, \max(0, A)) \cdot \text{abs}(\Delta\tilde{y}) \\
&\qquad + K_4 \cdot \text{abs}(\Delta\theta)
\end{aligned}$$

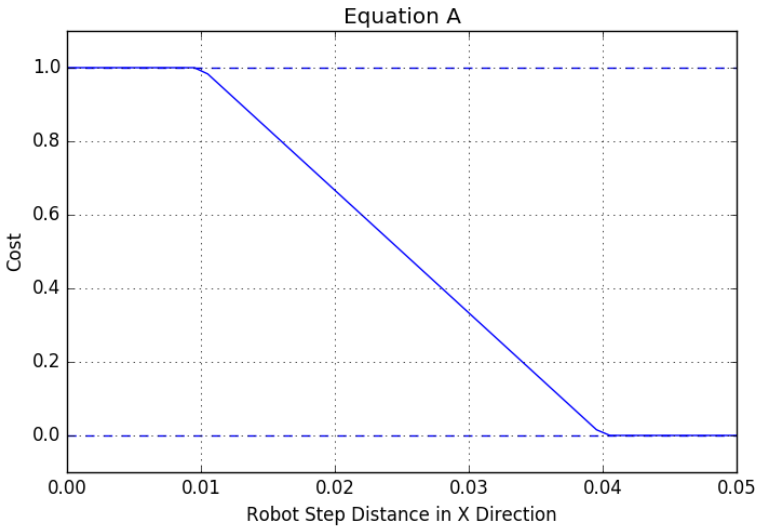



Figure 14: Equation A that helps reduce the amount of strafing in footstep plans, used in Equation 3

When performing these calculations, it is critical to remember which reference frame [41], or coordinate system, one is working in. In most situations, we have a world, or fixed, frame and a robot, or moving, frame. The only way to make these frames compatible with each other is by using a homogenous transformation matrix [41]. In our case of footstep planning, we use the 2D version, seen in Equation 4.

Figure 15: Comparison of references frames in the same environment. The red axis represents the $x$ direction, the green axis represents the $y$ direction, and the blue axis represents the $z$ direction.

All step cost plugins work in robot stance foot frame and so we need to convert the footstep placements from world to robot stance foot frame. In order to perform this conversion, we use part of Equation 4 known as the rotation matrix, seen in Equation 5:

$$\begin{bmatrix} cos(\theta_1) & -sin(\theta_1) & x_t \\ sin(\theta_1) & cos(\theta_1) & y_t \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} cos(\theta_1) & -sin(\theta_1) \\ sin(\theta_1) & cos(\theta_1) \end{bmatrix} \tag{5}$$

Referring back to Equations 1 and 2, this is why we use $\Delta\tilde{x} = cos(\theta_{stance}) \cdot \Delta x + sin(\theta_{stance}) \cdot \Delta y$ and $\Delta\tilde{y} = -sin(\theta_{stance}) \cdot \Delta x + cos(\theta_{stance}) \cdot \Delta y$ for these equations. These equations allow us to transform our world frame coordinates to local robot frame coordinates which in turn gives us numbers that work with our cost function. For example, $\Delta x$ should be a positive number if the robot is moving forward. However, if the robot is moving forward

in the negative $x$ direction with respect to the world frame, then $\Delta x$ can become negative when it should be a positive number. This will wreak havoc on our cost function because we assign a much higher cost when the robot travels backwards.

### Planner After Custom Plugin

After implementing these formulas and calculations, we are able to enforce a cost on step transitions that disincentivizes strafing motions and incentivizes smooth curving motions as $\epsilon$ approaches one. Using default settings, a path planned around a complicated obstacle, previously seen in Figure 13, has an optimal cost and fifty-two steps in the plan. However, many of these steps are strafing steps. Using our custom plugin, a path planned around the same obstacle, seen in Figure 16, has sixty-eight steps and a suboptimal cost according to the planner. There are also strafing steps in this path due to the suboptimal plan. The main reason the custom plugin results in a suboptimal cost is because of the default heuristics used.

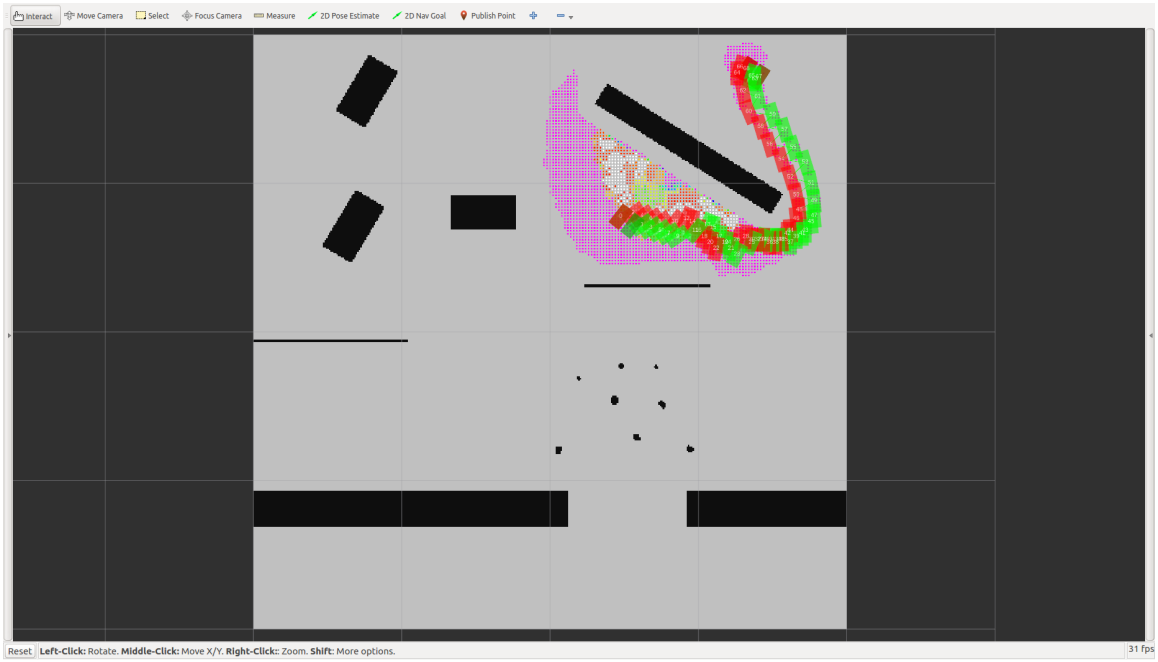


Figure 16: A suboptimal plan with an uninformed heuristic that expanded around 99,000 states for this path with final $\epsilon = 3.8$

The primary heuristic used is euclidean distance. If our environment is an open world, then this heuristic works well. However, as Figure 13 shows, our current environment is cluttered and complex. Because of this clutter, the euclidean heuristic will thrash against obstacles since it believes it can find a way through the obstacle. This thrashing wastes computational time and can result in suboptimal paths, depending on complexity of barriers. To reduce this thrashing behavior, we used a more informed euclidean heuristic that is aware of impediments. This heuristic is calculated by precomputing a 2D Dijkstra search from goal to start, using euclidean distance for the cost between edges. Front loading this work helps save time later on, as the 2D Dijkstra search only occurs when the environment or the goal pose changes. Without this informed heuristic, we would expand around 99,000 states and have suboptimal paths seen in Figure 17. Afterwards, this dropped to around 97,000 states expanded and resulted in optimal paths depicted in Figure 18.

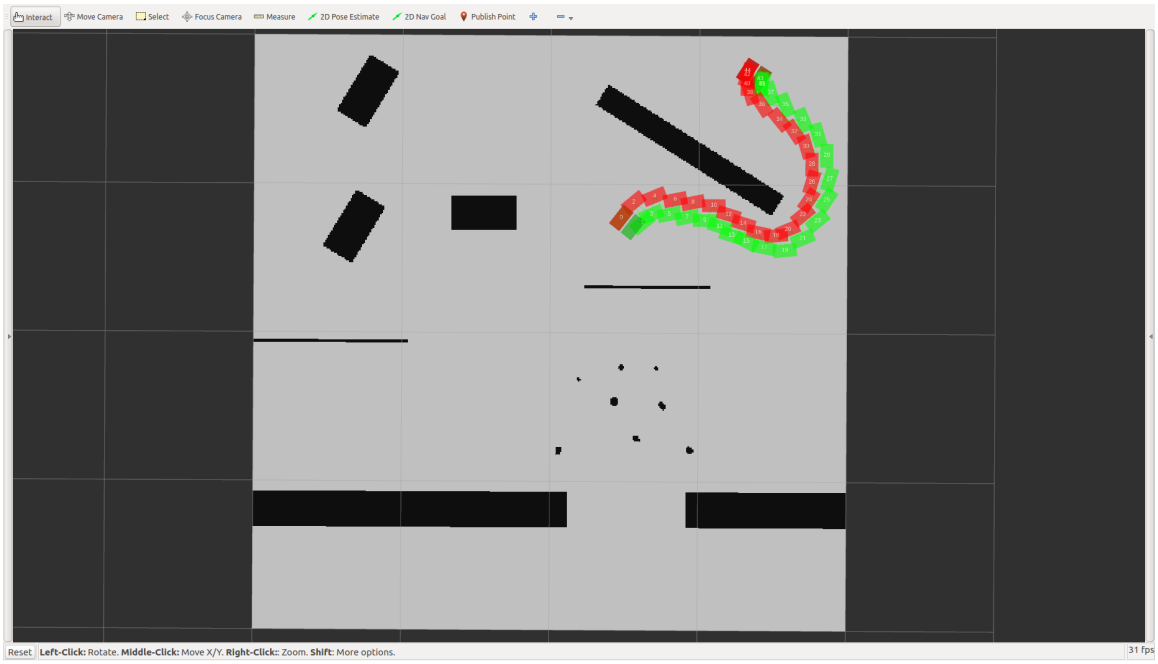


Figure 17: A suboptimal plan, using custom estimator, with no strafing

Figure 18: An optimal path, using the custom estimator and informed heuristic, with no strafing that expanded around 97,000 states for this path

# CHAPTER V:
# EXPERIMENTAL RESULTS

## Comparison Metrics

We use two primary metrics for comparing plans, body centroid distance, defined as the sum of the euclidean distance the body centroid travels, and number of steps in the plan. Since different configurations use different cost functions, we cannot directly compare path cost to each other. Additionally in the case of our custom step estimator, Equation 3, we are able to manipulate each scaling factor. Therefore we are unable to compare plan cost with the custom step estimator unless the scaling factors are equal.

## Planning in Simulation Results

The default planning configuration is defined as:

1. Planner algorithm: ARA*
2. Planning Time: 60.0 seconds
3. Initial epsilon($\epsilon$): 6.0
4. Decrease epsilon($\epsilon$): 0.2
5. Constant step cost: 0.06
6. Cell size: 0.02m
7. Number of angle bins: 64
8. Different angle cost: 0.0
9. CPU threads: 4

We used four different scenarios for our experiments: No obstacle, trivial obstacle, intermediate obstacle, and difficult obstacle navigation. Start and goal poses for each scenario can be seen in Figure 19 through Figure 22 respectively. Green feet placement represents start and red feet placement represents goal. Through experimentation with the planner, we believe these scenarios accurately represent four different classes of problems. However, due to an apparent state dependency issue that will be explained later, the data does not explicitly show these levels of difficulty.

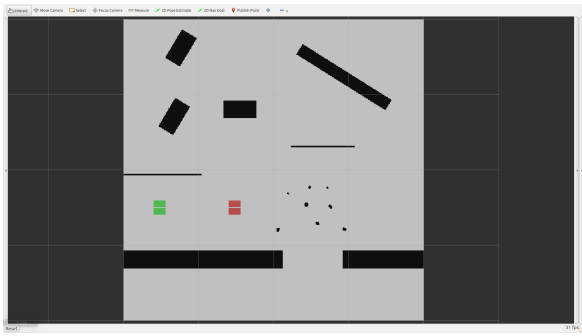


Figure 19: No obstacle start and goal poses in RViz

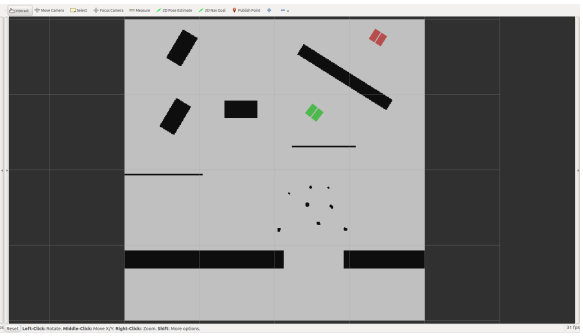


Figure 20: Trivial obstacle start and goal poses in RViz

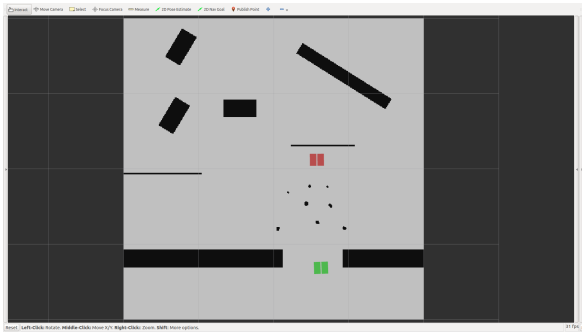


Figure 21: Intermediate obstacle start and goal poses in RViz

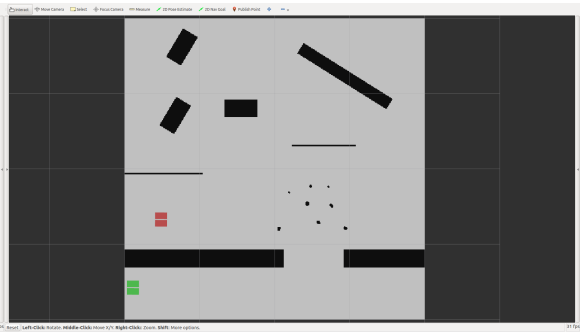


Figure 22: Difficult obstacle start and goal poses in RViz

## Methodology

For each scenario, a particular configuration is set. For baselines, we run each scenario given Hornung's configurations [16, 20]. These configurations are shown in Table 3. For reference, the step cost heuristic helps the planner determine the expected number of steps to take from start to goal.

Table 3: Baseline Configurations for Planner

| Configuration ID | Step Cost Function | Heuristic(s) |
|---|---|---|
| Hornung A | Euclidean distance | Euclidean distance |
| Hornung B | Euclidean distance | 2D Dijkstra distance, Step Cost |

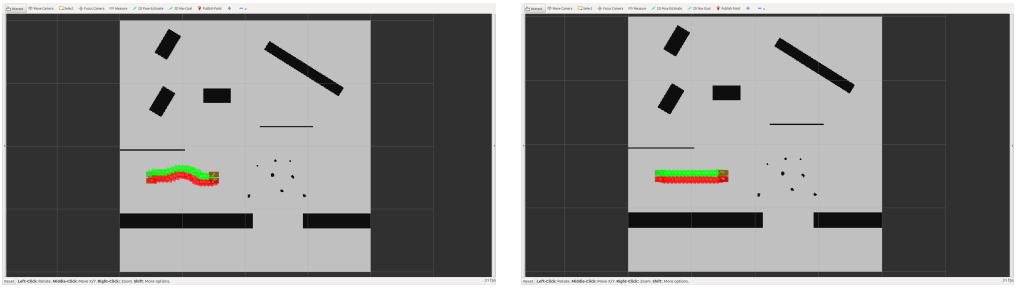
When we first ran our automated testing, we discovered a bug that causes major performance issues when executing iterations of each scenario. On the first run a configuration would expand $n$ amount of states. On the last iteration, in our case the fifth iteration, the number of expanded states is around $\frac{n}{2}$. This last iteration uses the full amount of planning time but generates worse paths. This pattern appeared regardless of planning scenario or configuration of the planner.

To circumvent this problem and acquire fair data, we used an alternate approach of publishing start and goal poses to the planner. Using this alternate approach is much slower than our previous testing but there are no negative performance issues, so we decided on manually publishing start and goal poses. We ran the Hornung A configuration five times for each scenario and discovered there was no deviation in body centroid distance, number of steps, final epsilon, planning time, and path cost. Additionally, the deviation in the number

of expanded states was minimal; given a configuration that expanded 20000+ states each run, we calculated a deviation of less than one-percent. Because of this trend with the Hornung A data, we only executed one run through of each scenario for all other planner configurations. These baseline results can be seen in Tables 4 through 7. Finally, if final $\epsilon$ is 100000, this means no path was found.

Table 4: Baseline Results for Planner given Figure 19 configuration

| Config ID | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] | Path Cost |
|---|---|---|---|---|---|---|
| Hornung A | 2.664515 | 38 | 2.6 | 19273 | 60 | 3.361104 |
| Hornung B | 2.583373 | 36 | 1.4 | 18554 | 60 | 3.16814 |

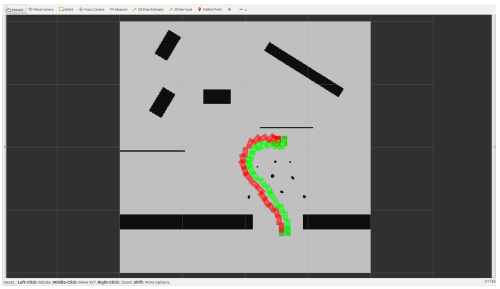


Hornung A path Hornung B path

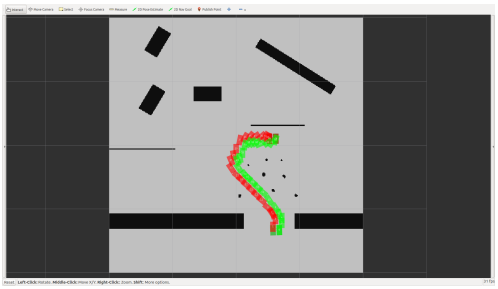
Figure 23: Planner generated paths corresponding to Table 4

Table 5: Baseline Results for Planner given Figure 20 configuration

| Config ID | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] | Path Cost |
|---|---|---|---|---|---|---|
| Hornung A | 4.770912 | 40 | 3.4 | 21749 | 60 | 4.478373 |
| Hornung B | 4.866316 | 46 | 2 | 20288 | 60 | 4.944012 |



Hornung A path

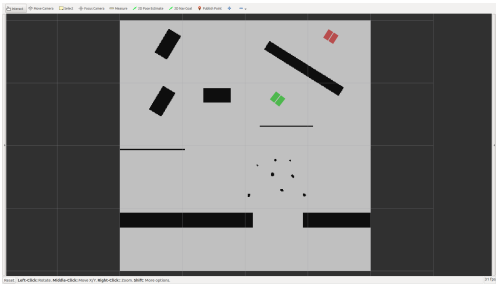


Hornung B path

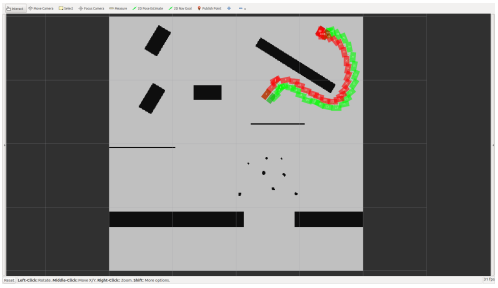
Figure 24: Planner generated paths corresponding to Table 5

Table 6: Baseline Results for Planner given Figure 21 configuration

| Config ID | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] | Path Cost |
|---|---|---|---|---|---|---|
| Hornung A | 0 | 0 | 100000 | 20431 | 60 | 0 |
| Hornung B | 6.454742 | 52 | 1.8 | 18901 | 60 | 5.8732 |



Hornung A path

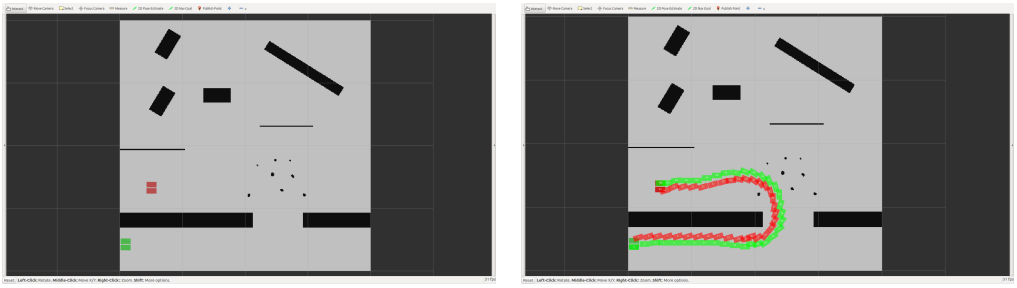


Hornung B path

Figure 25: Planner generated paths corresponding to Table 6

Table 7: Baseline Results for Planner given Figure 22 configuration

| Config ID | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] | Path Cost |
|---|---|---|---|---|---|---|
| Hornung A | 0 | 0 | 100000 | 20105 | 60 | 0 |
| Hornung B | 5.235499 | 70 | 1 | 3440 | 10.724 | 9.032137 |



Hornung A path Hornung B path

Figure 26: Planner generated paths corresponding to Table 7

In order to show improvements over the original open-sourced system, we will stage our experiments into three steps: first, how the ViGIR planner behaves pre-fix; second, how the system acts after our fix but with default planner settings; and finally, how the system acts with our custom plugins. For reference, the step cost heuristic helps the planner determine how many expected steps it should take to go from a state to the goal. These configurations are summarized in Table 8.

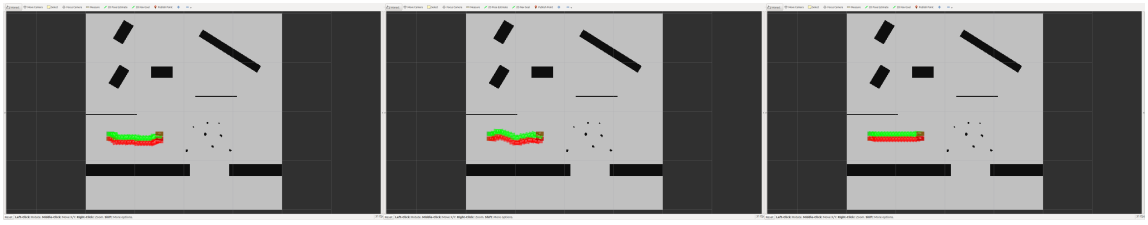
Table 8: Experimental Configurations for Planner

| Configuration ID | State Generator | Step Cost Function | Heuristics |
|---|---|---|---|
| ViGIR A | Faulty | Euclidean distance | Euclidean distance, Step Cost |
| ViGIR B | Standard | Euclidean distance | Euclidean distance, Step Cost |
| ViGIR C | Standard | Euclidean distance | 2D Dijkstra distance, Step Cost |
| Custom A | Standard | Equation 3 | Euclidean distance, Step Cost |
| Custom B | Standard | Equation 3 | 2D Dijkstra distance, Step Cost |

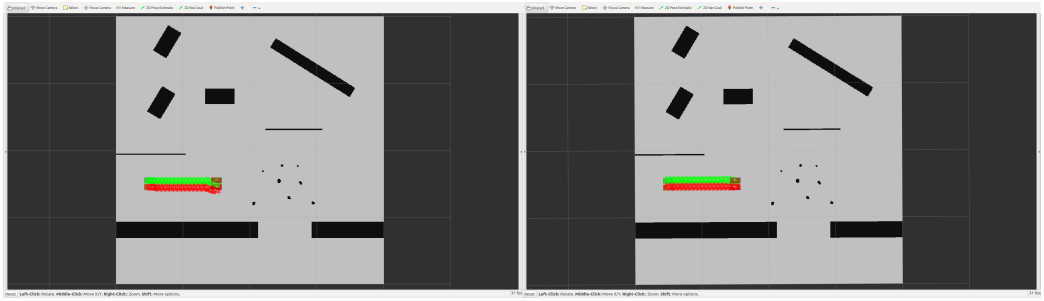
Finally, our results from running each configuration once given a planning scenario are seen in Table 9 through Table 12. We do not claim this is the optimal configuration for this system. Additionally, we do not claim to have exhausted all possible combinations of configuration for these experiments. However, we believe this information is useful for future work in this system. For reference, our vector $K$ for our custom cost function is defined as such: $K_0 = 1.0, K_1 = 2.0, K_2 = 1.0, K_3 = 2.0, K_4 = 0.0003$.

Table 9: No Obstacle Results given Figure 19 configuration

| Config ID | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] | Path Cost |
|---|---|---|---|---|---|---|
| ViGIR A | 2.660472 | 44 | 2.2 | 8749 | 60 | 3.733941 |
| ViGIR B | 2.676065 | 38 | 1.6 | 18029 | 60 | 3.378522 |
| ViGIR C | 2.583373 | 36 | 1.4 | 18062 | 60 | 3.168140 |
| Custom A | 2.625728 | 38 | 3 | 17285 | 60 | 6.095494 |
| Custom B | 2.584899 | 36 | 3.4 | 17990 | 60 | 5.764372 |



ViGIR A path ViGIR B path ViGIR C path

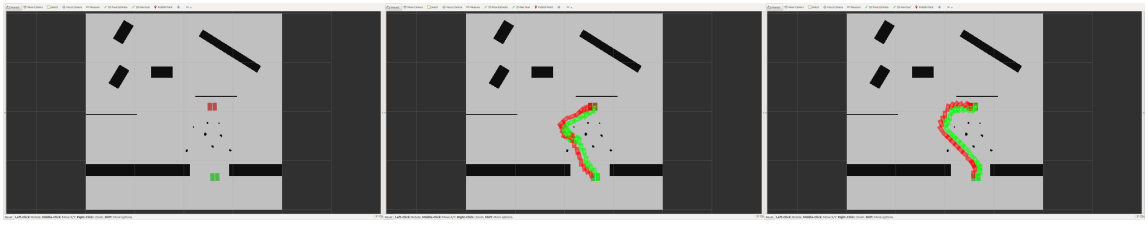


Custom A path Custom B path

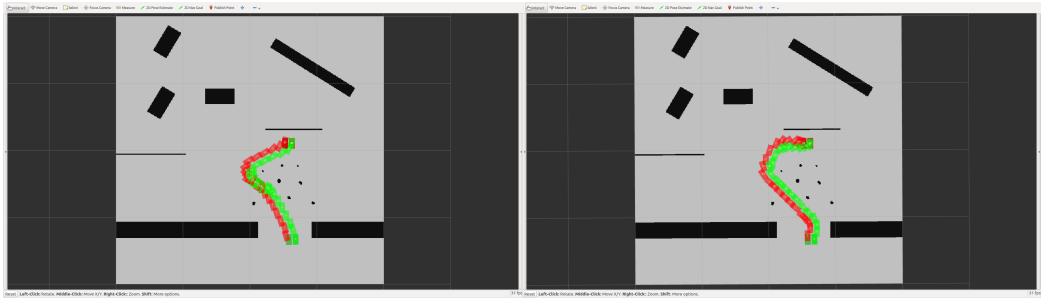
Figure 27: Planner generated paths corresponding to Table 9

Table 10: Trivial Navigation Results given Figure 20 configuration

| Config ID | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] | Path Cost |
|---|---|---|---|---|---|---|
| ViGIR A | 0 | 0 | 100000 | 13575 | 60 | 0 |
| ViGIR B | 4.738458 | 46 | 2.2 | 21029 | 60 | 4.806523 |
| ViGIR C | 4.866316 | 46 | 2 | 20497 | 60 | 4.944012 |
| Custom A | 4.755381 | 48 | 3 | 21036 | 60 | 6.681904 |
| Custom B | 4.862185 | 44 | 2.8 | 20731 | 60 | 5.784568 |



ViGIR A path ViGIR B path ViGIR C path

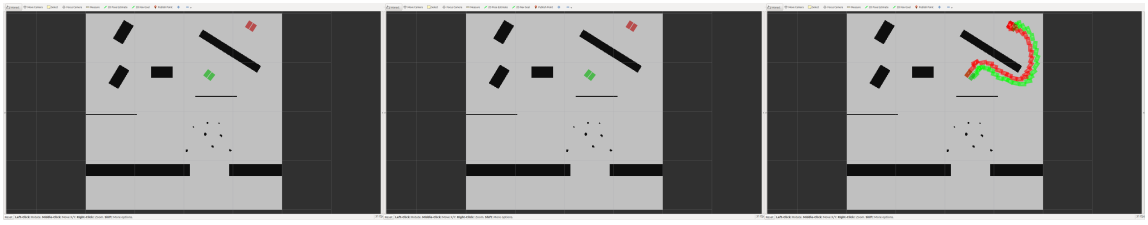


Custom A path Custom B path

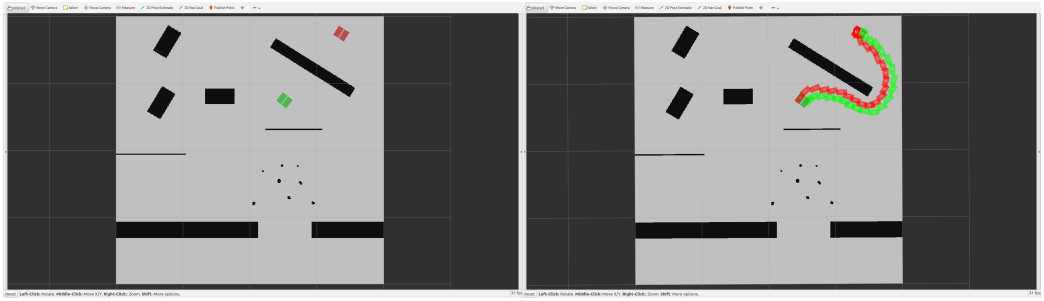
Figure 28: Planner generated paths corresponding to Table 10

Table 11: Intermediate Navigation Results given Figure 21 configuration

| Config ID | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] | Path Cost |
|---|---|---|---|---|---|---|
| ViGIR A | 0 | 0 | 100000 | 17156 | 60 | 0 |
| ViGIR B | 0 | 0 | 100000 | 19941 | 60 | 0 |
| ViGIR C | 6.454742 | 52 | 1.8 | 19289 | 60 | 5.873284 |
| Custom A | 0 | 0 | 100000 | 19733 | 60 | 0 |
| Custom B | 6.261546 | 44 | 2 | 19082 | 60 | 5.811205 |



ViGIR A path ViGIR B path ViGIR C path

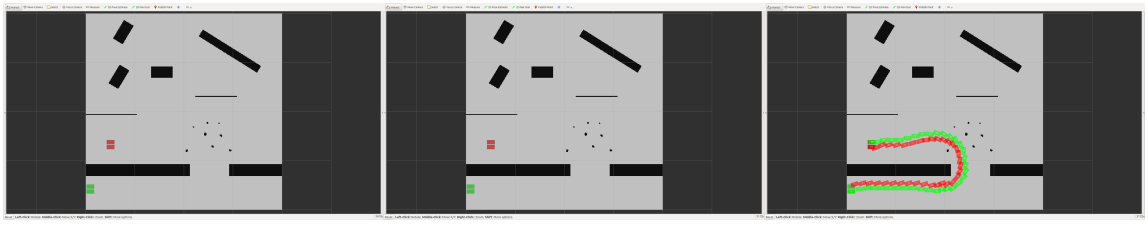


Custom A path Custom B path

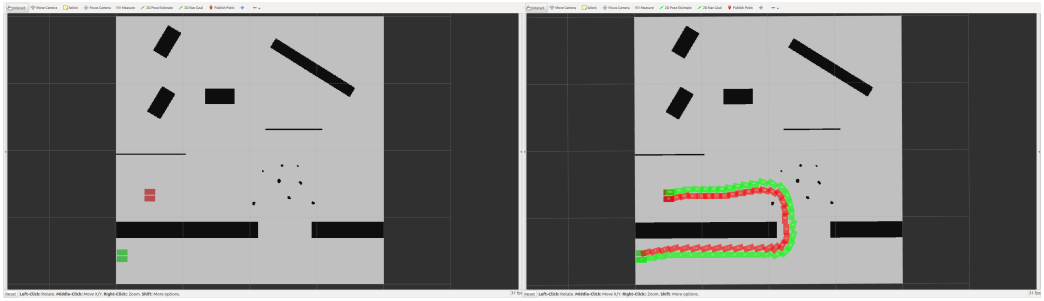
Figure 29: Planner generated paths corresponding to Table 11

Table 12: Difficult Navigation Results given Figure 22 configuration

| Config ID | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] | Path Cost |
|---|---|---|---|---|---|---|
| ViGIR A | 0 | 0 | 100000 | 10715 | 60 | 0 |
| ViGIR B | 0 | 0 | 100000 | 19377 | 60 | 0 |
| ViGIR C | 5.235499 | 70 | 1 | 3440 | 10.684 | 9.032137 |
| Custom A | 0 | 0 | 100000 | 18800 | 60 | 0 |
| Custom B | 5.153382 | 70 | 1.2 | 19599 | 60 | 8.894609 |



ViGIR A path ViGIR B path ViGIR C path



Custom A path Custom B path

Figure 30: Planner generated paths corresponding to Table 22

During our tests, we discovered that planning performance depends on the start and goal states in an interesting way. For example, going north one meter with no obstacles results in around 40 expanded states, but going south one meter results in around 4000 states. The same results appeared with east and south paths respectively. We hypothesize that

this issue is related to the discrete bins used to discretize the x, y, and angle coordinates, as well as the interplay with the footstep generation step of the planner. For Table 13, we sent paths of one meter magnitude from $-\pi$ radians to 1.565 radians with an increment of $\frac{\pi}{1024}$ radians. For simplification, we only use a subset of the previous data to demonstrate our point. Tracking the root cause of this issue is beyond the scope of this thesis, and is left for future work.

Table 13: Subset of data showing apparent state dependency issue

| Orientation | Body Centroid Distance [m] | Number of Steps | Final $\epsilon$ | Number of Expanded States | Planning Time [s] |
|---|---|---|---|---|---|
| -170.86° | 1.0199 | 17 | 1 | 48 | 1.43 |
| -170.51° | 1.0315 | 17 | 1 | 147 | 4.42 |
| -170.17° | 1.0673 | 19 | 1.2 | 4081 | 120.51 |
| -169.82° | 1.0629 | 19 | 1.4 | 4026 | 120.52 |
| -169.48° | 1.0339 | 17 | 1 | 114 | 3.52 |

As previously mentioned in the Methodology section of Chapter V, we discovered a potential memory leak or bug that negatively impacts planner performance. When running our automated tests we discovered that the planner slows down linearly with each new start and goal pair sent. These results are seen in Figure 31. This graph uses the full set of data discussed in Table 13. We believe this issue is related to re-initializing the planner via action goals, since this performance decrease does not appear when manually setting start and goal poses in RViz or manually publishing the start and goal via rqt[15]. We hypothesize that this is unrelated to the previously mentioned state dependency issue.

[15]A ROS application that allows manually publishing message, as well as other features. `http://wiki.ros.org/rqt`

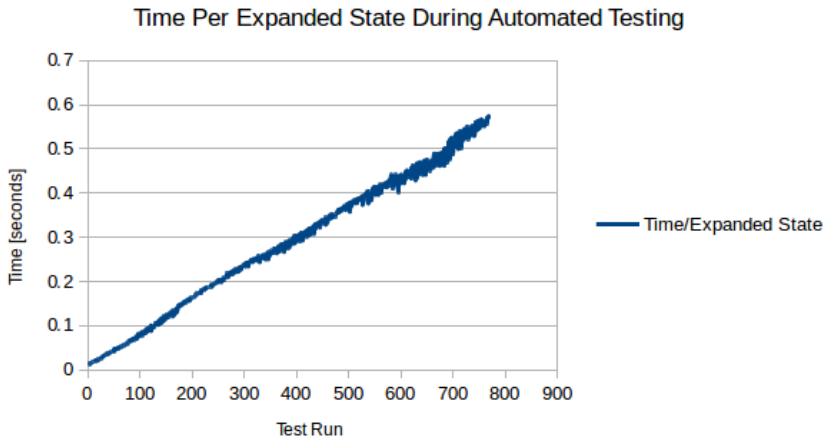



Figure 31: Graph of Decaying Performance of Planner Over Time

# CHAPTER VI:
# SUMMARY

## Conclusion

In this thesis, we presented a working implementation of Team ViGIR's footstep planner with a configuration setup for the NAO humanoid robot. Additionally, we corrected several issues, seen in Chapter IV, in the ViGIR planner that prevented accurate behavior of the search algorithms. Using the ViGIR framework, we created our own step cost estimator, seen in Chapter IV, that incentivizes smooth motions and disincentivizes strafing motions in the planner. Futhermore, we implemented a more informed heuristic using a precomputed 2D Dijkstra's search which can be seen in Chapter IV.

## Future Work

We identified two major bugs which will need further investigation in the ViGIR planner. Although the planner works in many situations, we discovered through automated testing that there is a state dependency issue in addition to a timing or memory leak. Given certain paths, such as one meter north on an empty map, the planner's performance is negatively effected. This negative effect is seen as the number of states expanded grows by an order of magnitude, as well as planning time, compared to similar paths. In addition, if running the planner in an automated way, the planner's performance slows down linearly as each planning task is completed. This slowdown does not appear when using the planner in the standard way, i.e. sending start and goal poses in RViz or publishing start and goal poses via rqt. This evidence further shows that one of the issues appears to be with the re-initialization of the planner.

In our experiments, we did not consider 3D planning or robot localization. Although the

ViGIR planner has 3D capabilities, we were unable to use them because of a lack of a whole body controller for the NAO robot. For this thesis, the whole body controller is outside of scope but is currently in development for future work. The NAO would also need equipment for gathering point cloud data in order to take advantage of the built in terrain classifier in ViGIR's planner. Having this equipment would also help alleviate robot localization issues. Without proper localization, executed paths will drift from planned paths. This drift can result in the robot failure when navigating a cluttered environment.

## Appendix A: Graph Planning

### Graph Planning Detailed Overview

Path planning problems can be modeled as abstract graphs, thus we need to use graph search algorithms to effectively solve these problems. One of the most important of these algorithms is A-Star (A*). A* is an extention of Dijkstra's that uses a search heuristic to direct which nodes are expanded first [32].

A* search expands states according to an evaluation function

$$f(s) = g(s) + h(s) \tag{6}$$

where $g(s)$ are the actual costs of the optimal path from the start state to current state $s$ and some heuristic function $h(s)$ dispenses the estimated costs to the goal from some state $s$. However for A* to keep its optimality, our heuristic $h(s)$ must be admissible, i.e.

$$\forall s : h(s) \leq c(s, \text{goal}) \tag{7}$$

where $c(s, \text{goal})$ means the true cost of navigating from state $s$ to the goal. The efficiency of A* is wholly dependent on the heuristic function, a well-informed heuristic can reduce the number of irrelevant states expanded. A* will, if there is a solution, always give an optimal path from some starting node to a goal node. If A* is not given a time limit, then the previous property is not an issue. However, in real situations, there is some restriction for available computational time. To manuever around this issue, Weighted A* (wA*) was created [33].

Weighted A* (wA*) behaves the same as A* except that wA* will inflate the heuristic function by some factor $\epsilon \geq 1$. By doing so, wA* will potentially find a path faster by performing a greedy search based on the heuristic. In addition, there is a chance for the

algorithm to reduce irrelevant states in order to guarantee sub-optimality. Using wA* guarantees that our path will cost no more than $\epsilon$ times the cost of an optimal path. Therefore, the quality of the path can be traded off for a quicker search. wA* does not reuse information, so solving a dynamic graph will complicate this algorithm. Given a real life environment, where dynamic objects may change our cost, it is beneficial to reuse calculations where possible. To solve this issue, other planners were developed, such as: D* Lite and Anytime Repairing A* (ARA*).

As with wA*, D* Lite is an extension of A* but also an improvement of D* [20]. D* Lite excels in efficient incremental searches, i.e. dynamic environments. However, for the algorithm to perform efficiently, it searches in reverse order from goal state to current state. The first search is simply an A* search since there is no information available to be reused. For every subsequent search, D* Lite will maintain the costs for all visited states as a heuristic cost when determining the current optimal path from the current state to the goal state. Additionally, D* Lite will maintain the lower boundary of the optimal path costs regardless of iteration. D* Lite is one approach to solving dynamic path planning problems, however it requires precomputation in order to be successful. In contrast, Anytime Repairing A* (ARA*) does not require precomputation to provide a solution.

As with previous algorithms, Anytime Repairing A* (ARA*) is an extension of A*. ARA* search executes multiple wA* searches while efficiently reusing previous information, similiar to D* Lite. On the first ARA* search, our heuristic factor $\epsilon$ is set to be large. In the case of Table 1, $\epsilon = 5$, however values for $\epsilon$ can range from five to one-hundred. This will create a suboptimal path quickly as previously described. In constrast to vanilla A*, we will use whatever computational time we have left to create a more optimal solution. The next iteration of ARA* will reduce $\epsilon$ by some increment to try and find a better path. Given

enough time, ARA* will reduce $\epsilon$ to 1 and thereby find the most optimal path. If ARA* is unable to find the best path in time, we know that our path will cost no more than $\epsilon$ times the true cost. This approach allows ARA* to find a path faster than regular A* but also approach optimality if given enough time.

Since ARA* is based off A*, it is also dependent on the search heuristic. A more informed heuristic will increase efficency of ARA* search and vice versa. Therefore, designing the heuristic becomes difficult given complex environments. However, the authors propose that randomized A* (R*) will solve this problem [16]. Before diving into R*, it is relevant to provide context to this randomized approach by the use of Rapidly-exploring Random Trees (RRT) [42].

LaValle introduced the concept of a Rapidly-exploring Random Tree (RRT) as a method to solve many different path planning problems [42]. In contrast to A* and its previously mentioned derivatives, RRTs take a quasi-randomized approach for determining which state to expand next. By using some control inputs, the RRT will drive itself toward some randomly selected points. This design heavily biases the RRT to explore unvisited states. Unlike with A*, RRTs are not dependent on some informed heuristic function. However, RRTs are not guaranteed to provide an optimal solution nor have a boundary on how suboptimal the solution will be [16]. Another A* derivative, Randomized A* (R*), claims to minimize solution cost and provide a bound on the suboptimality of the solution [43].

Randomized A* search is designed to have less dependency on the quality of the heuristic function. One of the main issues with graph search algorithms is the procedure entering a local minima and wasting computation when exiting the minima. R* avoids this by executing a "series of short range, fast wA* searches towards randomly chosen subgoals." [16].

R* creates a graph $\Gamma$ with scantly placed states and then at each iteration will expand some state in $\Gamma$. A state is expanded by generating some $k$ random successor states at a distance $\Delta$. Any goal state that is within distance $\Delta$ is added to the successor list for that current state.

Then R* will try to connect the separated states in $\Gamma$ in the easiest way possible. However if this "easy" local path requires expanding too many states, R* will put a label telling itself to avoid the local path unless absolutely necessary for the suboptimal boundary. As with ARA*, R* will iteratively lower the inflated heuristic value $\epsilon$ and rerun the search if time permits. Unlike ARA* however, R* ensures exploration of the search space, i.e. it should avoid local minima. In summary, R* will start and sample infrequent states towards the goal and around any obstacles. Then as the heuristic approaches 1, R* will expand more states and create a more dense graph in order to find the most optimal solution.

## Appendix B: Plugin Details

### Stumpf's Plugin Details

Stumpf's solution to plugin management is based on *pluginlib*[16] which has the ability to load classes (plugins) from shared libraries and build them using the ROS build framework [37]. The main idea is a user will create a new plugin with specific functions and this plugin will be injected into the framework as is. This way all existing code is untouched during execution which reinforces less errors and quicker development time. Most plugins should be modeled after the divide and conquer approach, i.e. create highly cohesive and highly modular packages that solve some focused problem.

The plugin manager will determine the provided features of a plugin by checking its inheritance tree. Figure 32 shows the concept of the plugin inheritance with simple plugin classes. This allows developers to acquire dependencies easily from a plugin database without having to know implemenation details of dependent plugins. The manager will also set up ROS services and action servers automatically to allow access to management functionality with the included graphical user interface.

In addition to the plugin manager, Stumpf's implemenation uses a parameter manager to help define different scenarios. Parameters are a way to define different scenarios at run time using the YAML[17] syntax for configuration files that is common in ROS development. The current implemenation allows quick software adjustments by providing parameter sets to a plugin to dictate its behavior. Additionally, the parameter manager comes with its own graphical user interface to modify parameter sets. These tools allow us to deal with different humanoid configurations.

To materalize this plugin system, an execution pipeline must be created. In this pipeline

[16] http://wiki.ros.org/pluginlib
[17] http://yaml.org/

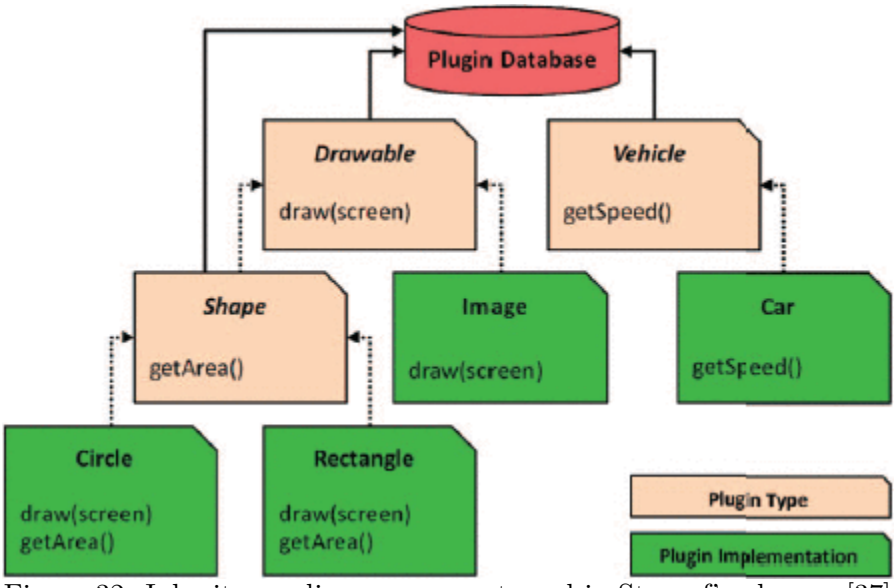


Figure 32: Inheritance diagram concept used in Stumpf's planner [37]

there will be injection points that are exposed to developers to create new functionality. For each of these points, a plugin type has been defined:

- *CollisionCheckPlugin*: Collison checks for a given state and/or transition.
- *HeuristicPlugin*: Computes heuristic value (estimated remaining cost) to the goal state.
- *PostProcessPlugin*: Post-processing of a generated step or step plan.
- *ReachabilityPlugin*: Defines the set of valid step transitions.
- *StateGeneratorPlugin*: Determines the next state(s) to visit.
- *StepCostEstimatorPlugin*: Estimates cost and risk for given step transition.
- *StepPlanMsgPlugin* (singleton): Marshalling interface for robot specific data carried in each single step and step plan.
- *TerrainModelPlugin* (singleton): Provides 3D terrain model of environment.

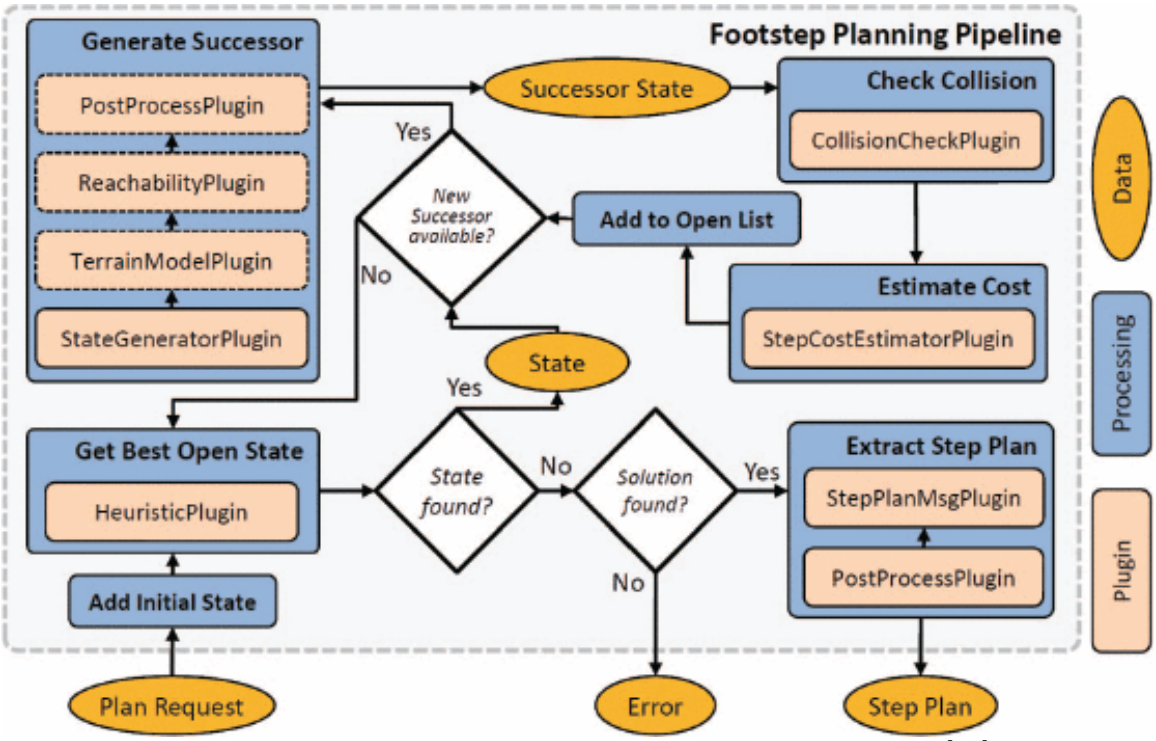



Figure 33: Plugins embedded into footstep planning pipeline [37]

Every plugin, except the singletons, can exist multiple times or as different implemenations in our plugin database. Every plugin that is the same type contributes individual computational results, so that all active *StepCostEstimatorPlugins* are summed up to a total step cost.

To extend this framework, we create a new plugin according to this instruction set[18] and inject it into the planner. We modify the current plugin set to use our new plugin and adjust any parameters as needed. This operation is shown in figure 34. A key feature of this plugin setup is that changing the active set does not require recompilation of the framework. Therefore, we are able to have highly dynamic changes if needed without much hassle.

[18]http://wiki.ros.org/vigir_pluginlib

```
plugin_sets:
  default:
    [...]

    # StateGeneratorPlugin
    reachability_state_generator: none

    # StepPlanMsgPlugin
    step_plan_msg_plugin: none

    # ReachabilityPlugins
    reachability_polygon: none

    # StepCostEstimatorPlugins
    const_step_cost_estimator: none
    euclidean_step_cost_estimator: none
    my_step_cost_estimator:
      params:
        my_param: 42

    # HeuristicPlugins
    step_cost_heuristic: none
    euclidean_heuristic: none

    [...]
```

Figure 34: A sample configuration file with a user-created plugin and parameter [37]

## Appendix C: ViGIR Footstep Planner Bringup Details

### ViGIR Footstep Planner Tutorial

To replicate our experiments, follow these instructions:

### Computer Setup

- Follow the ROS Installation[19] for setting up your ROS system on Ubuntu.
  - This system has been tested on ROS Kinetic Release on Ubuntu 16.04.
  - We recommend *ros-kinetic-desktop-full* as the base for this system; it includes most of the required packages for our system.
- Install the catkin_tools[20] package.
  - `$ sudo apt-get install python-catkin-tools`
  - This is required by many of CHRISLab scripts, and is generally preferred over the standard catkin_build system.
- Install ROS stand alone tools
  - `$ sudo apt-get install python-rosinstall`
  - CHRISLab main install script uses rosinstall tools
- Finally, install your favorite editor or IDE. We mainly use CLion, QtCreator, Atom, and VSCode.

[19] `http://wiki.ros.org/ROS/Installation/`
[20] `https://catkin-tools.readthedocs.io/en/latest/installing.html`

**ViGIR Footstep Planner Software Setup**

**If installing a new workspace, remove existing workspace setup from /.bashrc, and reopen terminal sourcing only the /opt/ros/kinetic/setup.bash prior to running this script**.

1. Create workspace root folder (e.g. ~/vigir_footstep_install) and change to that directory
2. Clone the install setup repository
   - `$ git clone https://gitlab.pcs.cnu.edu/vigir_footstep_devel_one/vigir_footstep_install.git`
3. Checkout the correct branch `$ git checkout kinetic_devel`
4. Run the install script `$ ./install.sh`
5. Follow on-screen instructions to add the new setup to .bashrc and re-source the terminal
6. Test the setup by running:
   - `roscd`
   - This command should place your terminal in the workspace root /src folder (e.g. ~/vigir_footstep_install/src).
7. Navigate back to the workspace root `cd $WORKSPACE_ROOT`.
8. Install the ViGIR footstep planner
   - `$ ./rosinstall/install_scripts/install_vigir_footstep.sh`
9. Build and install any external libraries installed in the `$WORKSPACE_ROOT/external` folder
   - `$ cd $WORKSPACE_ROOT/external/sbpl`
   - `$ mkdir build`

- `$ cd build`
- `$ cmake ..`
- `$ make`
- `$ sudo make install`

10. Navigate back to `$WORKSPACE_ROOT` and build the system
    - `$ catkin build`
    - `$ . setup.bash`

### Running ViGIR Planner Demonstrations

- The following commands run the default demo for the THORMANG humanoid robot.
- `$ roslaunch vigir_footstep_planner footstep_planner_test.launch`
- `$ roslaunch vigir_footstep_planning rviz_footstep_planning.launch`
- While these commands run the demo for the NAO humanoid
- `$ roslaunch vigir_footstep_planner footstep_planner_test_nao.launch`
- `$ roslaunch vigir_footstep_planning rviz_footstep_planning.launch`

## LITERATURE CITED


[1] *DARPA Robotics Challenge (DRC) (Archived)*, https://www.darpa.mil/program/darpa-robotics-challenge.

[2] S. Kohlbrecher, A. Stumpf, A. Romay, P. Schillinger, O. von Stryk, and D. C. Conner, "A comprehensive software framework for complex locomotion and manipulation tasks applicable to different types of humanoid robots," Frontiers in Robotics and AI **3**, 31 (2016).

[3] A. Romay, S. Kohlbrecher, A. Stumpf, O. von Stryk, S. Maniatopoulos, H. Kress-Gazit, P. Schillinger, and D. C. Conner, "Collaborative autonomy between high-level behaviors and human operators for remote manipulation tasks using different humanoid robots," Journal of Field Robotics **34**, 333–358 (2017).

[4] S. Kohlbrecher, A. Romay, A. Stumpf, A. Gupta, O. von Stryk, F. Bacim, D. A. Bowman, A. Goins, R. Balasubramanian, and D. C. Conner, "Human-robot teaming for rescue missions: Team ViGIR's approach to the 2013 DARPA Robotics Challenge Trials," Journal of Field Robotics **32**, 352–377 (2015).

[5] *Aldebaran documentation*, http://doc.aldebaran.com/2-1/family/robots/dimensions_robot.html.

[6] *Find out more about nao*, https://www.ald.softbankrobotics.com/en/robots/nao/find-out-more-about-nao.

[7] *Aldebaran documentation*, http://doc.aldebaran.com/2-1/nao/nao_life.html.

[8] *Nao software 1.14.5 documentation*, http://doc.aldebaran.com/1-14/software/choregraphe/choregraphe_overview.html.

[9] *Nao software 1.14.5 documentation*, http://doc.aldebaran.com/1-14/dev/naoqi/index.html#naoqi-overview.

[10] S. Kajita and K. Tani, "Experimental study of biped dynamic walking in the linear inverted pendulum mode," in Proceedings of 1995 IEEE International Conference on Robotics and Automation, Vol. 3 (May 1995), 2885–2891 vol.3.

[11] P. B. Wieber, "Trajectory free linear model predictive control for stable walking in the presence of strong perturbations," in 2006 6th IEEE-RAS International Conference on Humanoid Robots (Dec. 2006), pp. 137–142.

[12] *Nao software 1.14.5 documentation*, http://doc.aldebaran.com/1-14/naoqi/motion/control-walk.html#id3.

[13] M. Johnson, J. Bradshaw, P. Feltovich, C. Jonker, B. Van Riemsdijk, and M. Sierhuis, "The fundamental principle of coactive design: interdependence must shape autonomy," Coordination, organizations, institutions, and norms in agent systems VI, 172–191 (2011).

[14]S. Oßwald, A. Hornung, and M. Bennewitz, "Learning reliable and efficient navigation with a humanoid," in 2010 IEEE International Conference on Robotics and Automation (May 2010), pp. 2375–2380.

[15]A. Hornung, S. Böttcher, J. Schlagenhauf, C. Dornhege, A. Hertle, and M. Bennewitz, "Mobile manipulation in cluttered environments with humanoids: integrated perception, task planning, and action execution," in 2014 IEEE-RAS International Conference on Humanoid Robots (Nov. 2014), pp. 773–778.

[16]A. Hornung, A. Dornbush, M. Likhachev, and M. Bennewitz, "Anytime search-based footstep planning with suboptimality bounds," in 2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012) (Nov. 2012), pp. 674–679.

[17]C. Lutz, F. Atmanspacher, A. Hornung, and M. Bennewitz, "Nao walking down a ramp autonomously," in 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (Oct. 2012), pp. 5169–5170.

[18]A. Hornung and M. Bennewitz, "Adaptive level-of-detail planning for efficient humanoid navigation," in 2012 IEEE International Conference on Robotics and Automation (May 2012), pp. 997–1002.

[19]S. Oßwald, A. Görög, A. Hornung, and M. Bennewitz, "Autonomous climbing of spiral staircases with humanoids," in 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (Sept. 2011), pp. 4844–4849.

[20]J. Garimort, A. Hornung, and M. Bennewitz, "Humanoid navigation with dynamic footstep plans," in 2011 IEEE International Conference on Robotics and Automation (May 2011), pp. 3982–3987.

[21]*Robocup federation official website*, `http://www.robocup.org/`.

[22]*Robots can help young patients engage in rehab*, (Sept. 2017) `http://theconversation.com/robots-can-help-young-patients-engage-in-rehab-54741`.

[23]B. N. De Carolis, S. Ferilli, G. Palestra, and V. Carofiglio, "Towards an empathic social robot for ambient assisted living.," in Essem aamas (2015), pp. 19–34.

[24]*Ask nao*, `https://asknao.aldebaran.com/`.

[25]*For education & research*, `https://www.ald.softbankrobotics.com/en/solutions/education-research`.

[26]M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in ICRA Workshop on Open Source Software (2009).

[27]robotical.io, *Robotical :: Deep-Dive with ROS*, `https://robotical.io/learn/article/Deep-Dive%20with%20ROS/Learn/`.

[28]*Nao ros wiki*, `http://wiki.ros.org/nao`.

[29] *Dcm documentation*, `http://doc.aldebaran.com/2-1/naoqi/sensors/dcm.html`.

[30] S. Chitta, I. Sucan, and S. Cousins, "Moveit![ros topics]," **19**, 18–19 (2012).

[31] *Rviz documentation*.

[32] M. Likhachev, G. J. Gordon, and S. Thrun, "ARA*: Anytime A* with provable bounds on sub-optimality," in Advances in neural information processing systems (2004), pp. 767–774.

[33] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving* (Addison-Wesley, 1984).

[34] E. Krotkov, D. Hackett, L. Jackel, M. Perschbacher, J. Pippine, J. Strauss, G. Pratt, and C. Orlowski, "The darpa robotics challenge finals: results and perspectives," Journal of Field Robotics **34**, 229–240.

[35] A. Hornung, D. Maier, and M. Bennewitz, "Search-based footstep planning," in Proc. of the icra workshop on progress and open problems in motion planning and navigation for humanoids, karlsruhe, germany (2013).

[36] J. Mirabel, S. Tonneau, P. Fernbach, A. K. Seppälä, M. Campana, N. Mansard, and F. Lamiraux, "Hpp: a new software for constrained motion planning," in 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (Oct. 2016), pp. 383–389.

[37] A. Stumpf, S. Kohlbrecher, D. C. Conner, and O. von Stryk, "Open Source Integrated 3D Footstep Planning Framework for Humanoid Robots," in 2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids) (Nov. 2016), pp. 938–945.

[38] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: an efficient probabilistic 3d mapping framework based on octrees," Autonomous Robots **34**, 189–206 (2013).

[39] S. Kohlbrecher, A. Romay, A. Stumpf, A. Gupta, O. von Stryk, F. Bacim, D. A. Bowman, A. Goins, R. Balasubramanian, and D. C. Conner, "Human-robot Teaming for Rescue Missions: Team ViGIR's Approach to the 2013 DARPA Robotics Challenge Trials," Journal of Field Robotics **32**, 352–377 (2015).

[40] P. Schillinger, S. Kohlbrecher, and O. von Stryk, "Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics," in IEEE International Conference on Robotics and Automation (May 2016).

[41] K. M. Lynch and F. C. Park, *Modern Robotics: Mechanics, Planning, and Control* (University Press, 2017).

[42] S. M. LaValle, *Rapidly-exploring random trees: a new tool for path planning*, tech. rep. (Iowa State University, 1998).

[43] M. Likhachev and A. Stentz, "R* search," Lab Papers (GRASP), 23 (2008).