IF GOINGTOCRASH() DONT();

A senior design project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science at Harvard University

Anthony J.W. Kenny
S.B. Candidate in Electrical Engineering

Faculty Advisor: Vijay Janapa Reddi

Harvard University School of Engineering and Applied Sciences
Cambridge, MA

March 1st, 2020
Version: 4.1

Abstract

This thesis demonstrates the design of RISC-V computer architecture that supports faster execution of motion planning algorithms for drone applications. First, it shows the analysis of computational performance of Rapidly-exploring Random Tree (RRT), a sampling-based motion planning algorithm commonly used in autonomous drones. Having identified collision detection as the biggest area of opportunity for improved performance, it describes the process of designing specialized hardware, taking advantage of parallelization, that quickly detects collisions. Finally, it presents how this specialized functional unit can be implemented in a processor, and the RISC-V Instruction Set Architecture (ISA) extended to invoke execution to massively reduce the execution time of collision detection.

Contents

List of Acronyms

API Application Programming Interface

CISC Complex Instruction Set Computer

CPI Cycles Per Instruction

FPGA Field Programmable Gate Array

CPU Central Processing Unit

GPU Graphics Processing Unit

GUI Graphical User Interface

ISA Instruction Set Architecture

PRM Probabalistic Road Map

RISC Reduced Instruction Set Computer

RRT Rapidly-exploring Random Tree

RTOS Real-Time Operating Systems

UAV Unmanned Aerial Vehicle

2D 2-Dimensional

3D 3-Dimensional

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Summary

1.1.1 Background

The Unmanned Aerial Vehicle (UAV) has been utilised in military applications extensively throughout the late 20th and early 21st century. However, over the last decade, their use in non-military uses, such as commerical, scientific, agricultural, and recreational, such that the number of civilian drones vastly outnumber military UAVs.Particularly in the commercial sector, such rapid growth in the number and range of applications means that autonomy is key for the profitable adoption of UAVs. Such autonomy relies on efficient computation of motion planning algorithms. However, the implementation of these algorithms can be quite computationally expensive, and thus slow and/or detrimentally power consuming. As such, this thesis aims to design specialized hardware to more efficiently compute motion plans for autonomous drones.

Robotics

For well over 2000 years, the concept of robotics, albeit not always with such a term, has fascinated humans. As early as the first century A.D., the Greek mathematician and engineer, Heron of Alexandria, described more than 100 different machines and automata in *Pneumatica* and *Automata* [1]. In 1898, Nikola Tesla demonstrated the first radio-controlled vessel. Since then, the world has seen widespread application of robotics in manufacturing, mining, transport, exploration, and weaponry. For the last few decades, robots have operated in controlled, largely unchanging environments (e.g. an assembly line) where their environment and movements are largely known *a priori*.
However, in recent years a new generation of autonomous robots has been developed for a wide range of real-world, complex applications. The increasing trend the use of autonomous robots is shown in Figure 1.1. These new robots, unlike those traditional ones described

Figure 1.1: The use of Autonomous Robots over time

above, are required to adapt to the changing environment in which they operate. As such, they must perform motion planning in real time.

Motion Planning

Motion Planning refers to the problem of determining how a robot moves through a space to acheive a goal. Chapter 2 provides a detailed explanation of motion planning and of RRT, a commonly used motion planning algorithm.
On the algorithmic level, motion planning has been extensively studied and many solutions exist. However, current algorithms running on regular Central Processing Unit (CPU)s are too slow to execute in real time for robots operating in complex environments. Simply solving this problem with more raw computing power, using energy hungry Graphics Processing Unit (GPU)s may have merit in tethered robots. On the other hand, untethered applications, such as autonomous drones, where limiting power consumption is a primary concern, this strategy is infeasible.

Something

1.1.2 Problem Definition

Problem Statement

Motion planning algorithms implemented in software that runs on general purpose CPUs cannot execute quickly enough for fully autonomous UAVs to operate in high-complexity environments. The state-of-the-art strategy of using power-hungry GPUs to accelerate the execution of these algorithms requires too much power to be cost-effective or feasible for UAVs to sustain flight for useful periods of time.

End User

This thesis aims to provide developers of autonomous drones with specialized hardware for motion planning. Such developers have a need for computing hardware that executes motion planning algorithms faster and more power efficiently than existing methods. This thesis will provide a processor design that is synthesizable on an Field Programmable Gate Array (FPGA), giving developers a processer for which a Real-Time Operating Systems (RTOS), or bare metal code, can be deployed.

1.2 Prior Work

1.2.1 Hardware Acceleration

Hardware acceleration refers to the strategy of using computer hardware specifically designed to execute a function more efficiently than can be achieved by software running on a general purpose CPU. Specialized hardware designed to perform specific functions can yield significantly higher performance than software running on general purpose processors, and lower power consumption than GPUs.

Computer Implementation Hierarchy

To briefly frame the space in which this thesis operates, consider the typical computer implementation hierarchy, demonstrated in Figure 1.2. **User level applications**, such as Google Chrome, Microsoft Word, and Apple's iTunes, sit at the top of the abstraction hierarchy. These applications are implemented in **High-Mid Level Languages**, such as C/C++, Python, Java, etc. These programming languages have their own hierarchy, but for the purpose of this thesis, it is sufficient to understand that these programming languages are then compiled into **Assembly Language**. Assembly language closely follows the execution of instructions on the **processor**, and is defined by an **ISA**. An ISA can be thought of as the contract between software programmers and processor engineers, agreeing what instructions the processor is able to implement. This assembly code is finally loaded into the processor's instruction memory and executed.

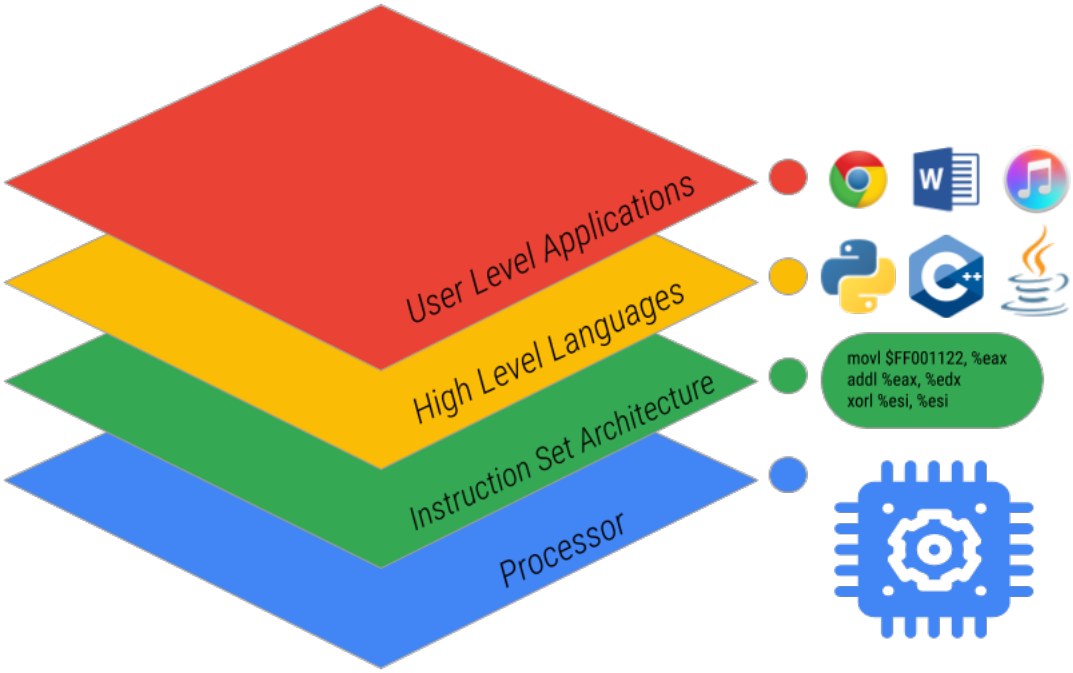


Figure 1.2: Simple Visualization of Computer Implementation Hierarchy

As will be outlined in Section 1.3, this thesis operates extensively on the lower two levels of this hierarchy, extending an existing ISA and building hardware at the processor level that supports these extensions.

Acceleration of Motion Planning

Accelerating motion planning with hardware is a fairly well studied problem.
A Motion Planning Processor on Reconfigurable Hardware [2] studied the performance benefits of using FPGA-based motion planning hardware as either a motion planning processor, co-processor, or collision detection chip. It targeted the feasibility checks of motion planning (largely collision detection) and found their solution could build a roadmap using the Probabalistic Road Map (PRM) algorithm up to 25 times faster than a Pentium-4 3Ghz CPU could.
In *A Programmable Architecture for Robot Motion Planning Acceleration* [3], Murray et al. built on the work of the aformentioned paper, to accelerate several aspects of motion planning in an efficent manner.
FPGA based Combinatorial Architecture for Parallelizing RRT [4] studies the possibility of building architecture to allow multiple RRTs to work simultaneously to uniformly explore

a map. Taking advantage of hardware parallelism allows systems such as this to compute more information per clock cycle.
Finally, in the paper *Robot Motion Planning on a Chip* [5], Murray et al. describe a method for contructing robot-specific hardware for motion planning, based on the method of constructing collision detection circuits for PRM that are completely parallelised, such that edge collision computation performance is independent of the number of edges in the graph. With this method, they could compute motion plans for a 6-degree-of-freedom robot more than 3 orders of magnitude faster than previous methods.

1.2.2 RISC-V

RISC-V (pronounced "risk-five") is an ISA developed by the University of California, Berkeley. It is established on the principles of a Reduced Instruction Set Computer (RISC), a class of instruction sets that allow a processor to have fewer Cycles Per Instruction (CPI) than a Complex Instruction Set Computer (CISC) (x86, the ISA on which macOS and linux operating systems run, is an example of a CISC instruction set). What makes RISC-V unique is its open-source nature. What makes CPU design so expensive is that is requires expertise across many disciplines (compilers, digital logic, operating systems, etc). RISC-V was started with the philosophy of creating a practical, open-source ISA that was usable in any hardware or software without royalites. The first report describing the RISC-V Instruction Set was published in 2011 by Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović [6].

Extending RISC-V

RISC-V is designed cleverly in a modular way, with a set of base instruction sets and a set of standard extensions. As a result, processors can be designed to only implement the instruction groups it requires, saving time, space and power on instructions that won't be used. In addition, another goal of RISC-V is to provide a basis for more specialized instruction-set extensions or more customized accelerators. This is described in the most recent *RISC-V Instruction Set Manual* [7]. This is a powerful feature, as it does not break any software compatability, but allows for designers to easily follow the steps outlined in Figure 1.3. From a hardware acceleration point of view, this is particularly useful as it allows the designer to directly invoke whatever functional unit or accelerator they implement from assembly code.

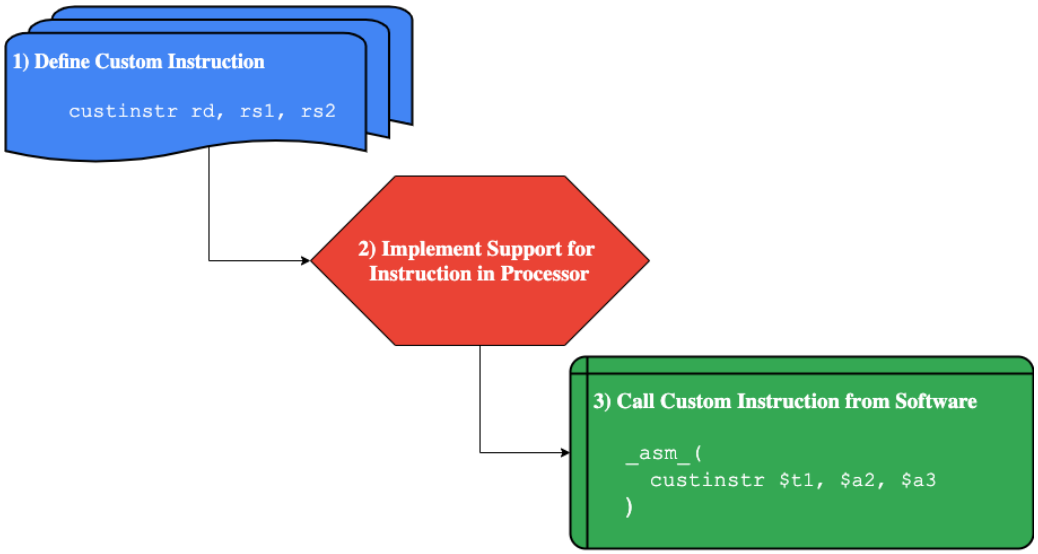


Figure 1.3: Typical Process of Adding Non-Standard Extension to RISC-V ISA

Accelerating RISC-V Processors

Having only been released in 2011, RISC-V is still a relatively unexplored opportunity for non-education applications. However, it shows promise in the commercial space, with Alibaba recently developing the Xuantie, a 16-core, 2.5GHz processor, currently the fastest RISC-V processor. Recently there has been promising research into accelerating computationally complex applications, particularly in edge-computing, with RISC-V architecture. *Towards Deep Learning using TensorFlow Lite on RISC-V*, a paper co-written by the faculty advisor of this thesis, V.J. Reddi, presented the software infrastructure for optimizing the execution of neural network calculations by extending the RISC-V ISA and adding processor support for such extensions. A small number of instruction extensions achieved coverage over a wide variety of speech and vision application deep neural networks. Reddi et al. were able to achieve an 8 times speedup over a baseline implementation when using the extended instruction set. *GAP-8: A RISC-V SoC for AI at the Edge of the IoT* proposed a programmable RISC-V computing engine with 8-core and convolutional neural network accelerator for power efficient, battery operated, IoT edge-device computing with order-of-magnitude performance improvements with greater energy efficiency.

1.3 Project Overview

1.3.1 Proposed Solution

This thesis proposes a non-standard RISC-V Instruction Set Extension, supported by a functional unit embedded in a FPGA synthesizable processor design that more rapidly computes motion planning for autonomous UAVs

System Overview

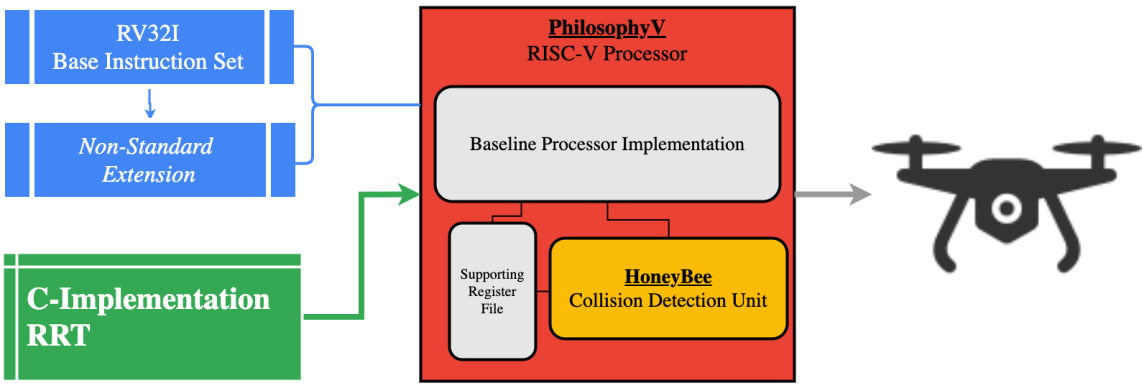


Figure 1.4: System Diagram of Overall Project

1.3.2 Project Specifications

1.3.3 Project Structure

Chapter 2

Motion Planning in Software

This thesis aims to design hardware that executes motion planning algorithms faster than those same algorithms can execute on generic hardware. Chatper 2 introduces motion planning and details the process of implementing and analysing RRT to identify computational hot-spots in the algorithm and thus identify the biggest opportunities for hardware optimization.
Section 2.1 provides an introduction to Motion Planning Algorithms in general. Section 2.2 outlines the RRT algorithm, and describes the implementation of RRT for this project. Finally, Section 2.3 outlines a method for performance analysis of RRT and the results of such analysis.

2.1 Background

Motion Planning Algorithms refer to the set of algorithms that find possible sequences of valid configurations for a robot in a space.

2.2 Rapidly-Exploring Random Tree

RRT is an algorithm designed to efficiently search, and thus plan a path through, a high-complexity environment by randomly sampling points and building a tree. The algorithm randomly samples points, draws an edge from the nearest currently existing node in the tree, to grow the tree in the space. It is inherently biased to grow towards large unsearched areas of the problem. RRT was developed by S. LaVelle[8] and J. Kuffner[9]. It is used in autonomous robotic motion planning problems such as autonomous drones, the focus of this thesis.

2.2.1 Algorithm

Building the Tree

Put simply, RRT builds a tree (referred to as a graph) of possible configurations, connected by edges, for a robot of some physical description. It does so by randomly sampling the configuration space and adding configurations to the graph. From this graph, a path from the initial configuration to some goal configuration can be found, given a high enough number of iterations. As such, RRT can be considered probabalistically complete. The pseudo-code for RRT can be seen in Algorithm 2.1

Algorithm 2.1: Rapidly-Exploring Random Tree in Free Configuration Space

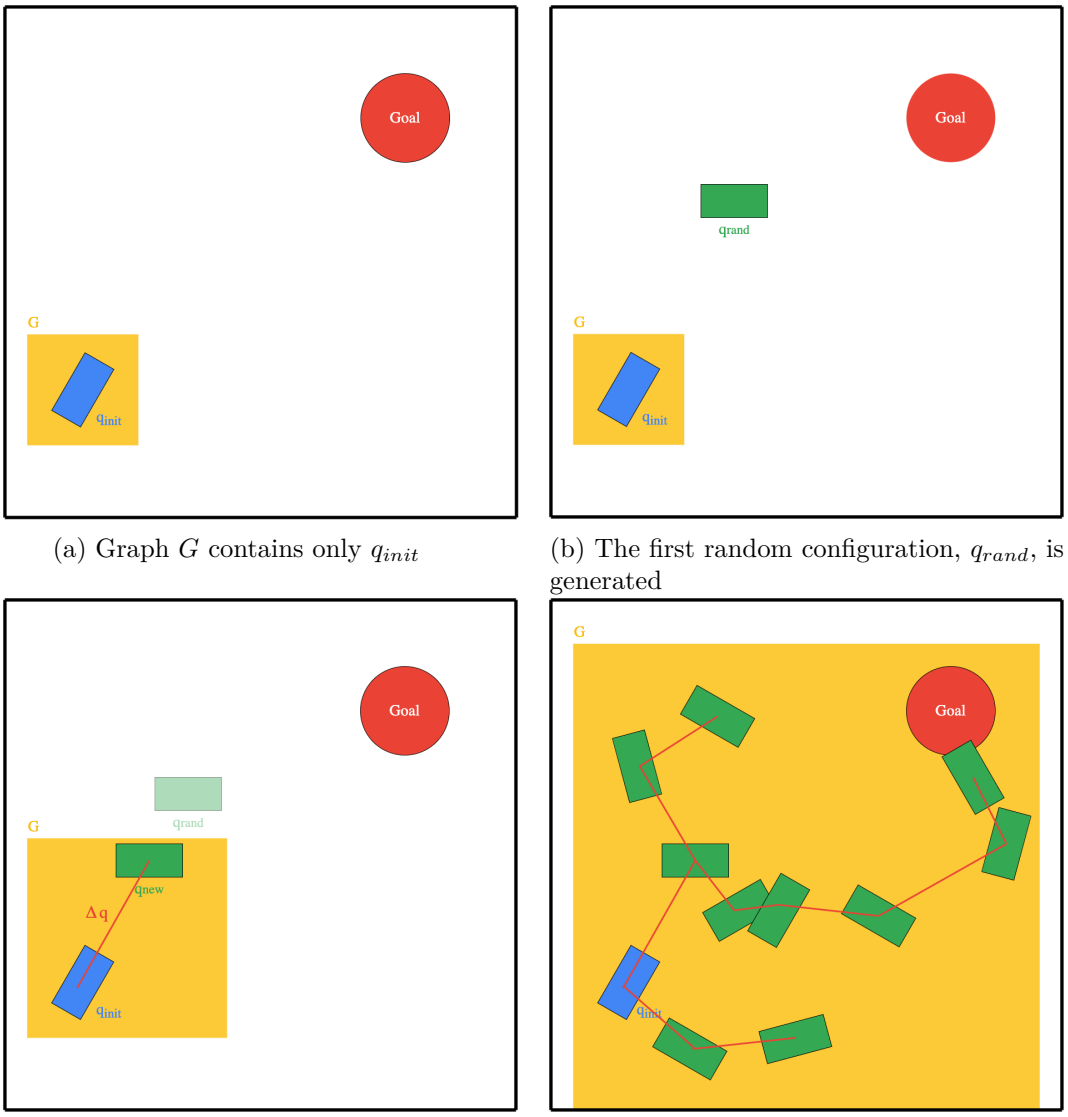
Inputs: Initial configuration q_{init},
Number of nodes in graph K,
Incremental Distance Δq

Output: RRT Graph G with K configurations $[q]$ & edges $[e]$

```
G.init();
for k = 1 to K do
    q_rand ← randomConfiguration();
    q_near ← nearestVertex(q_rand, G);
    q_new ← newVertex(q_near, q_rand, Δq);
    G.addVertex(q_new);
    G.addEdge(q_near, q_new);
end
```

Algorithm 2.1 can be visually represented in Figure 2.1. Consider a 2-Dimensional (2D) robot operating in a 2D workspace. A Graph G is initialized containing an initial configuration, q_{init}, with constraints on the number of nodes that the graph can hold, K, and the maximum distance between two nodes, Δq. This is shown in Sub-figure 2.1a. A random configuration for the robot, q_{rand} is generated (2.1b). The nearest existing configuration in G, q_{near}, is found. (In the first iteration, $q_{near} = q_{init}$, shown in Sub-figure

2.1c). The distance between q_{near} and q_{rand} is calculated. If this distance is less than Δq, $q_{new} = q_{rand}$. If not, q_{new} is selected, typically by moving by Δq from q_{near} towards q_{rand} (2.1c). q_{new} is then added to G. This is repeated for K configurations.



(a) Graph G contains only q_{init}

(b) The first random configuration, q_{rand}, is generated

(c) In first iteration, $q_{near} = q_{init}$. Distance between q_{init} and q_{rand} is greater than Δq, so q_{new} is generated and added to G

(d) This is repeated K times. For G, $K = 10$, and the red line represents the edges between configurations

Figure 2.1: Step by step demonstration of RRT Algorithm for 2D robot in 2D space

Collision Detection

Algorithm 2.1 shows how RRT builds a graph of possible configurations connected by edges in a free configuration space. However, in real-world applications, a robot's configuration space often contains obstacles. As such, collision detection must be included in the algorithm. The two types of collisions the algorithm must check for are *configuration collisions* (those where the robot would collide with an obstacle in a given configuration) and *edge collisions* (where the robot would collide when moving between two collision free configurations).

The RRT with configuration and edge collision detection can be seen in Algorithm 2.2. The method of implementing RRT with collision detection to model a drone in 3D space is detailed in Section 2.2.2.

Algorithm 2.2: Rapidly-Exploring Random Tree with Collision Detection

Inputs: Initial configuration q_{init},
Number of nodes in graph K,
Incremental Distance Δq,
Space S containing obstacles

Output: RRT Graph G with K configurations $[q]$ & edges $[e]$

```
G.init();
for k = 1 to K do
    while !pointCollision(q_new) do
        q_rand ← randomConfiguration();
        q_near ← nearestVertex(q_rand, G);
        q_new ← newVertex(q_near, q_rand, Δq);
    end
    e_new ← newEdge(q_near, q_new)
    if !edgeCollision(e_new) then
        G.addVertex(q_new);
        G.addEdge(q_near, q_new);
    else
        k = k − 1;
    end
end
```

2.2.2 Implementation

With RRT selected as the benchmark algorithm against which to test specialised hardware, this project required an implementation of the algorithm that satisfied the following criteria.

Requirement	Description and Justification
C/C++ Implementation	As outlined in Section 1.3.3, the critical step in determining the design of specialized hardware to accelerate RRT is CPU performance analysis of the algorithm to determine computational hot-spots. Implementations in C allow for the use of certain CPU profiling tools, described in Section 2.3.1, unlike higher-level languages such as Python.
Models Drone as Point	In reality, implementing RRT for a drone would model the robot as a 3-Dimensional (3D) object defined by coordinates $\{x, y, z\}$ and Euler angles $\{\alpha, \beta, \gamma\}$. However, for simplicity's sake, modelling the drone as a point defined by coordinates $\{x, y, z\}$ will suffice. Time permitting, this could be revisited.
Mirrors Algorithm	In order for the results of CPU performance analysis to be easy to understand, software implementation of RRT should call functions that mirror the functions described in Algorithms 2.1 and 2.2.

Table 2.1: Technical Specifications for RRT Implementation

The original intention was to find an existing implementation of RRT that could fulfill these requirements. Most open source implementations found online were in Python, and all those implemented in C were unsuitable[10][11][12][13], as they had extraneous GUIs, reliance on external Application Programming Interface (API)s, and other features that would distort analysis of algorithmic hot-spots.

As a result, it was necessary to build a C implementation of RRT from the ground up that satisfied the requirements in Table 2.1. It can be found in this project's GitHub repository. It follows Algorithm 2.1 closely. For monitoring correctness, I build in an optional GUI that shows the tree, starting node, and obstacles.

Implementation in 2D

The first step was to implement RRT with a 2-Dimensional workspace.

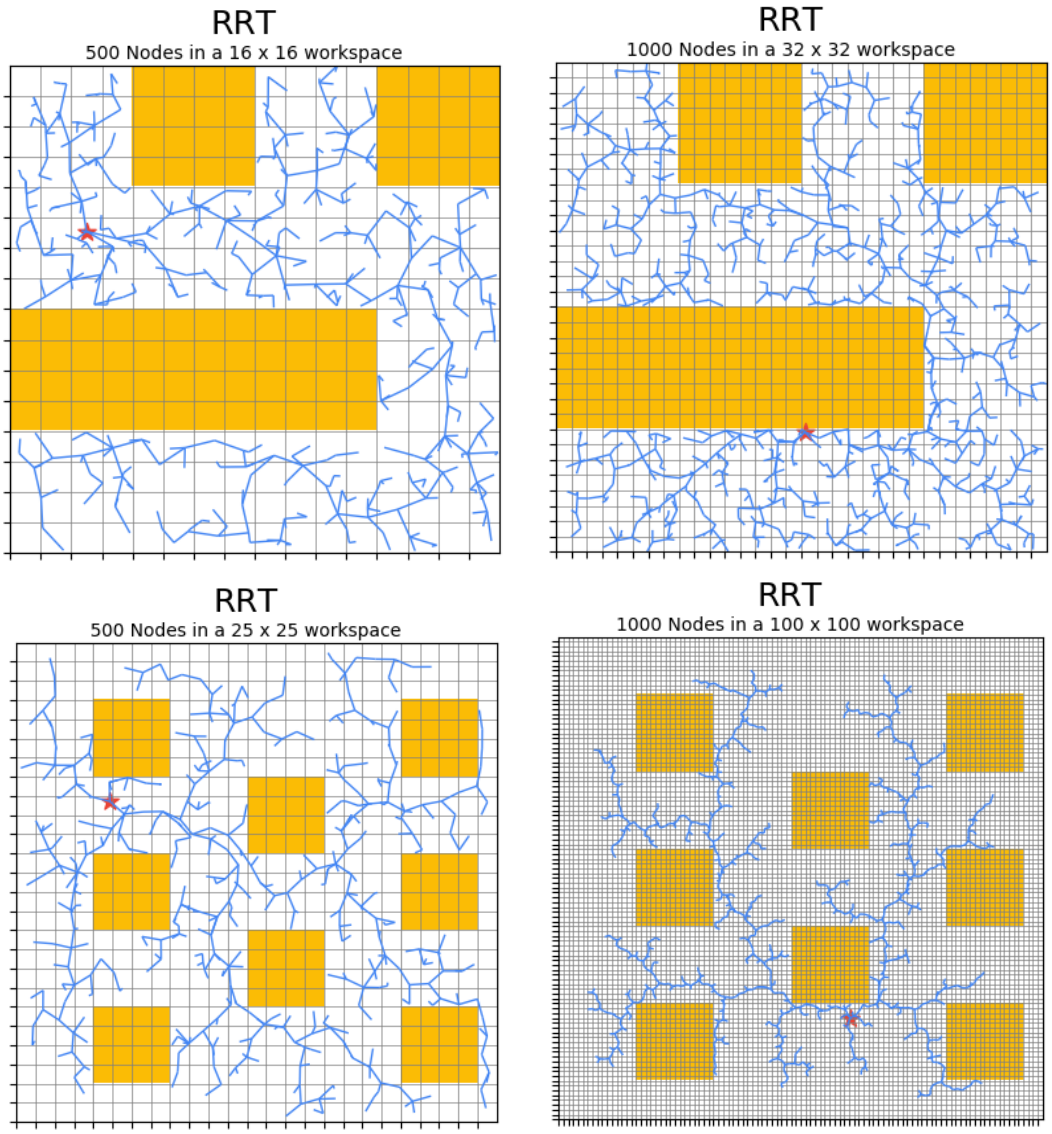


Figure 2.2: 2D RRT Implementation shown by GUI

Implementation in 3D

Figure 2.3: 3D RRT Implementation shown by GUI

2.3 Performance Analysis

2.3.1 Methodology

To restate, the aim of this thesis is to design a computer processor with reduced execution time of motion planning algorithms, such as RRT. As such, it is important to understand the elements of the algorithm that have the highest percentage of CPU execution time. To determine this, it was necessary to implement my own, naive but typical, RRT in C. This program could then be compiled and analysed using a software performance profiling tool. With this, I could design experiments to determine the critical RRT functions (those occupying a majority of CPU time) and see how this varies given different parameters.

VTune Profiler

VTune Profiler performance profiler is an application for software performance analysis. It provides functionality to examine hot-spots for CPU execution time through a top down analysis, shown below in Figure 2.4. As can be seen from the figure, the top down analysis tool shows the percentage of CPU time taken up by each function. I used this tool to profile the algorithm's performance as I changed certain parameters.

Figure 2.4: VTune Amplifier TopDown Analysis Example

Internal Timing

The limitation of VTune Profiler is that it can only profile software running on Intel processors, which implement the x86-64 ISA. As such, when the time comes to analyse performance of the software running on a RISC-V processor, another method will be required. A simple and effective way of measuring execution performance is to insert timing functionality into the software itself.

Comparison

Before proceeding to use either of these methods to profile the software implementation of RRT, it was important to verify that the two methods yielded similar results for the same program. Table 2.2 summarizes the results of analysis of a simple C executable. The program calls 5 functions, $\{A, B, C, D, E\}$, each a simple iteration in which a integer is incremented. Since the Internal Timing method returned similar results to the (trusted) VTune Profiler, it was considered to be a reliable method. While it was encouraging to see both methods returned similar results for absolute execution time, the more important metric was the similarity in percentage of total execution time.

function	Vtune Profiler		Internal Timing	
	time (s)	time (% total)	time (s)	time (% total)
A	0.488	57.4%	0.497	57.6%
B	0.2	23.5%	0.198	23.1%
C	0.102	12.0%	0.099	11.5%
D	0.048	5.7%	0.049	5.6%
E	0.012	1.4%	0.019	2.2%

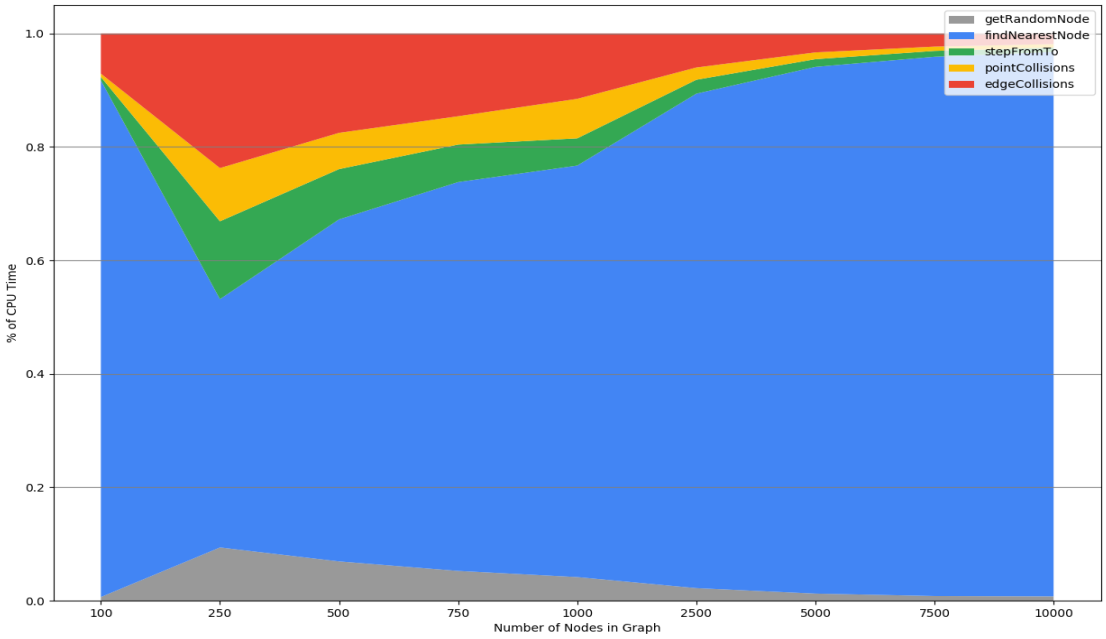
Table 2.2: Comparison of Timing Methods

Experimental Design

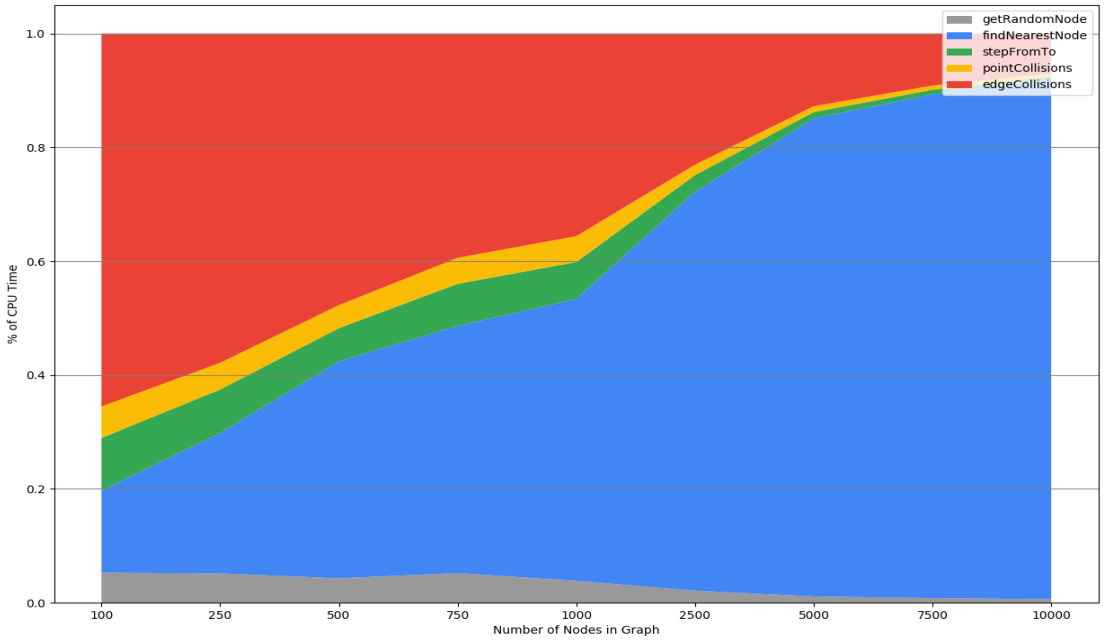
In profiling RRT in software, the goal was to find the critical task across different values of K and sizes of configuration space. Multiple tests were run, varying these two constraints, to find this critical function. The results of this analysis can be found in Section 2.3.2.

2.3.2 Results

Figure 2.5 shows the profile of functions within RRT, for $100 \leq K \leq 10000$, and cubic configuration spaces with dimensions $\{4, 8, 16, 32\}$. Each subfigure shows a similar profile, with the % of CPU Execution Time taken by findNearestNode increasing with K. This is to be expected. . However, it is also seen that edgeCollisions increases with larger configuration spaces, taking up the overwhelming majority of execution time for a 32x32x32 configuration space.

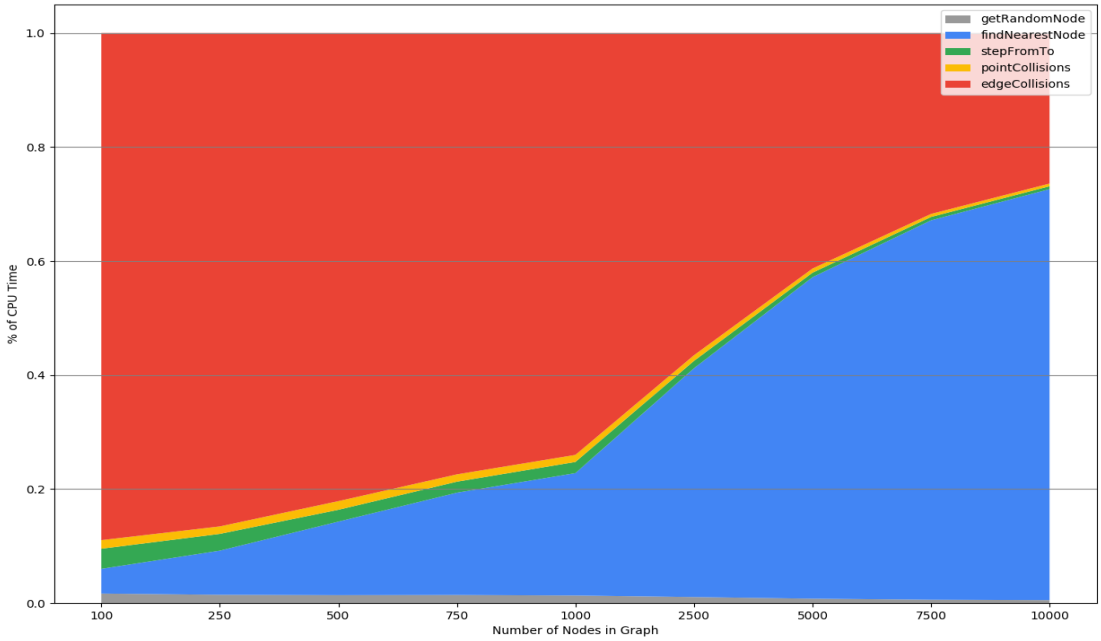


(a) 4x4x4 Configuration Space

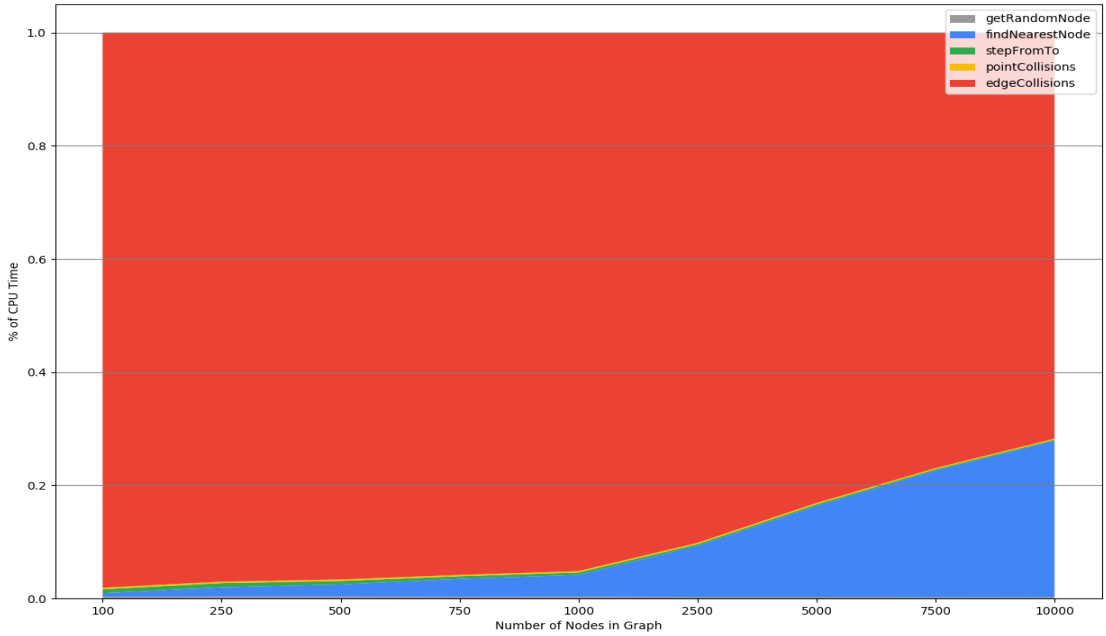


(b) 8x8x8 Configuration Space

Figure 2.5: RRT Functions as a % of Total CPU Exectution Time



(c) 16x16x16 Configuration Space



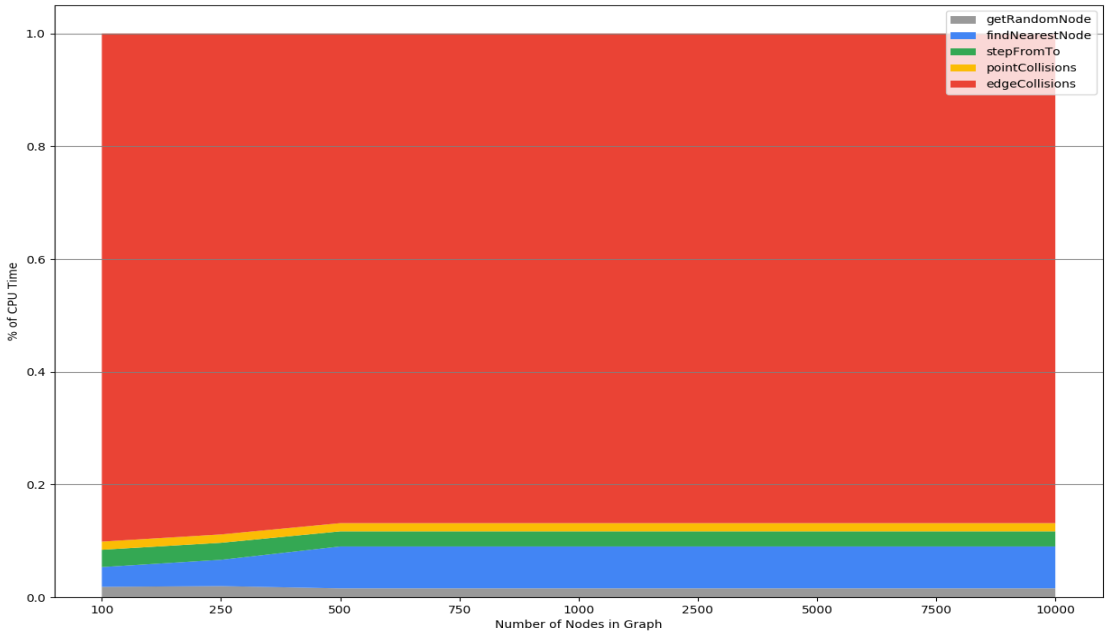
(d) 32x32x32 Configuration Space

Figure 2.5: RRT Functions as a % of Total CPU Exectution Time (cont.)

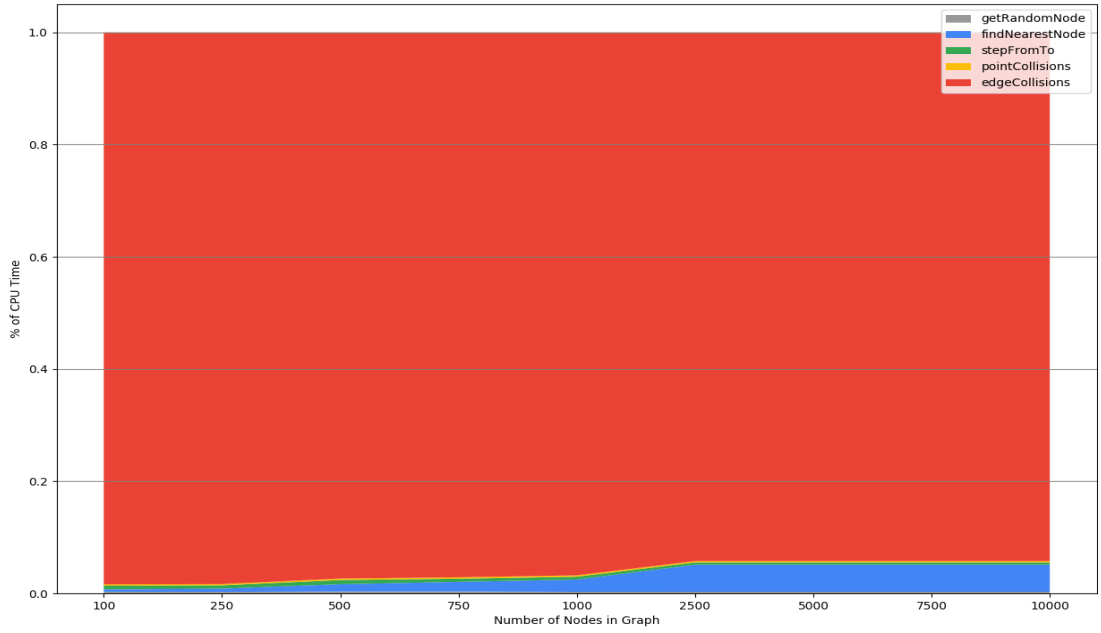
Furthermore, the computational load of findNearestNode can be reduced through a variety of software optimizations. A simple one used here to demonstrate that fact is storing nodes in seperate "buckets," sorted by their x value. By using only two buckets, the execution time of findNearestNode fell drastically. Figure 2.6b shows edge collision detection accounting for over 95% of execution time for $100 \leq K \leq 10000$. This is consistent with the profiling results of RRT in prior work[14].

Conclusion

From the above data, it was identified that, as prior work suggested, edge collision detection shows the greatest promise for potential speedup through specialized hardware. The next chapter details the process of designing and building this hardware.



(a) 16x16x16 Configuration Space



(b) 32x32x32 Configuration Space

Figure 2.6: RRT Functions Exectution Time, with Bucket Optimization

Chapter 3

Motion Planning in Hardware

3.1 Background to Hardware Optimization

3.2 HoneyBee

The Honey Bee has long been renowned for its tireless work ethic. But people rarely give the Honey Bee credit for its remarkable navigation and collision avoidance strategies during flight. As such, it is quite appropriate that this functional unit, designed to work tirelessly, rapidly and efficiently to execute collision detection computations, is named HoneyBee.

3.2.1 Design

It was demonstrated in Section 2.3.2 that the critical function of RRT was edge collision detection.

3.2.2 Build

High Level Synthesis

3.2.3 Measurement and Analysis

3.2.4 Iterations

Dimensions	Mac OS	Ubuntu	1	2	3	4
4x4x4	2	2	21.6	1.5	0.44	0.47
8x8x8	23	19	151	5.53	2.2	1.79
16x16x16	166	180	1133	41.37	13.08	12.11
32x32x32	1317	1424	8783	328	103	104

Table 3.1: Simulated performance of HB-A in microseconds

Chapter 4

RISC-V Processor

4.1 RISC-V ISA

4.2 Extending RISC-V

4.3 PhilosophyV

Chapter 5

Conclusion

5.1 Discussion of Results

Bibliography

[1] H. (Alexandrinus), *De gli automati, overo machine se moventi, Volume 2*.

[2] N. Atay and B. Bayazit, "A motion planning processor on reconfigurable hardware," in *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2006, pp. 125–132, 2006.

[3] S. Murray, W. Floyd-Jones, G. Konidaris, and D. J. Sorin, "A Programmable Architecture for Robot Motion Planning Acceleration," tech. rep.

[4] G. S. Malik, K. Gupta, K. M. Krishna, and S. R. Chowdhury, "FPGA based combinatorial architecture for parallelizing RRT," in *2015 European Conference on Mobile Robots, ECMR 2015 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., nov 2015.

[5] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, G. Konidaris, and D. Robotics, "Robot Motion Planning on a Chip," tech. rep.

[6] V. I. B. U.-l. Isa, A. Waterman, Y. Lee, D. Patterson, K. Asanovi, and B. U.-l. Isa, "The RISC-V Instruction Set Manual v2.1," *2012 IEEE International Conference on Industrial Technology, ICIT 2012, Proceedings*, vol. I, pp. 1–32, 2012.

[7] A. Waterman, K. Asanovic, and SiFive Inc, "The RISC-V Instruction Set Manual," vol. Volume I:, 2019.

[8] S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," *In*, vol. 129, pp. 98–11, 1998.

[9] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics Research*, vol. 20, pp. 378–400, may 2001.

[10] RoboJackets, "RRT," 2019. https://github.com/RoboJackets/rrt.

[11] M. Planning, "rrt-algorithms," 2019. https://github.com/motion-planning/rrt-algorithms.

[12] Sourishg, "rrt-simulator," 2017.
https://github.com/sourishg/rrt-simulator.

[13] Vss2sn, "Path Planning," 2019.
https://github.com/vss2sn/path{_}planning.

[14] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT," in *IEEE International Conference on Intelligent Robots and Systems*, pp. 3513–3518, 2011.

Appendices

Appendix A

Project Repository