THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF ELECTRICAL ENGINEERING

GAME-THEORETIC CONTROLLER SIMULATION FOR SAMPLING-BASED MULTI-ROBOT MOTION PLANNING ALGORITHMS

Cody M. Dillinger
Fall 2020

A thesis
submitted in partial fulfillment
of the requirements
for the baccalaureate degree
in Electrical Engineering
with honors in Electrical Engineering

Reviewed and approved* by the following:

Minghui Zhu
Associate Professor of Electrical Engineering
Thesis Supervisor

Julio Urbina
Associate Professor of Electrical Engineering
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

Motion planning is the process of a robot, vehicle, or other system calculating a collision-free trajectory and the corresponding control inputs to move from a starting state to a goal state set. Motion planning is increasingly important with developments in autonomous systems within factory environments, on the roads, in the air, and within our homes. Autonomous solutions in these applications can create increased safety, saved time, or saved financial costs. In recent years, feasible motion planning has evolved into optimal motion planning for single and multi-system scenarios. This thesis expands the simulation data on the i-Nash Trajectory Algorithm, a sampling-based game-theoretic algorithm for multi-robot motion planning. It analyzes data on the computational speeds and trajectory costs; it analyzes general performance and drawbacks of the algorithm upon increasing dimensions of the state space; it integrates the algorithm design with existing techniques such as the k-d tree for nearest neighbor searching and dual tree method of RRT-connect; it provides a Python code base for further expansion of similar simulations. Computational linearity in relation to the number of robots is verified in simulation for many simulation cases. Robots are non-cooperative, and they successfully reach a Nash Equilibrium. They find dynamically feasible paths that avoid both inter-robot collisions and static obstacle collisions, and each robot chooses their path in a decentralized, iterative manner such that no individual robot could unilaterally choose a lower cost feasible trajectory from its current options. The algorithm is also anytime, meaning that once the robots converge to a Nash Equilibrium they can continue to improve upon the solution by searching for more paths. Successful simulations are shown for various static obstacle arrangements, starting states, and goal sets. This thesis explores the algorithm for a larger number of double-integrator 4-dimensional state-space disc robots.

Table of Contents

List of Figures

Acknowledgements

I would like to thank Dr. Minghui Zhu for allowing me to be a part of the Networked Robotics Systems Laboratory and for introducing me to a very interesting and important realm of motion planning techniques. I would also like to thank Zhenyuan Yuan for your guidance in my research. It was very helpful to have your expertise and Dr. Zhu's expertise as I progressed through my thesis work. I would also like to thank Dr. Asok Ray for your early conversations with me regarding mathematical fundamentals; they positively impacted my view of academia as a whole. Thank you to Shicheng Liu for helping introduce me to types of robotic simulators that can be used for various motion planners. Thank you to Dr. Julio Urbina for your guidance in my Teaching Assistant roles, and for answering my questions as my honors advisor over the years. Thank you to Dr. Constantino Lagoa and Dr. Vishal Monga for providing me with some interesting Honors projects in Controls and DSP, for introducing me to these fields and inspiring a deeper interest.

I have spent five and a half years at The Pennsylvania State University, and have been extremely fortunate to work internships across various industries and locations every summer. With that, I thank every single mentor, manager, and friend built during my internships - as you have all had impacts on my life and my passions for engineering. I would also like to thank all of my colleagues and friends during my time at University Park for your time, energy, and support - and Schreyer for connecting me with these great people.

Thank you to my family for everything you have done for me. I would not be where I am today without you.

Chapter 1

Introduction

1.1 Motion Planning Techniques

1.1.1 Overview

There is an increasing focus on autonomous systems in factory environments, on the public streets, in the air, in space, and in our homes [1]. Motion planning plays a critical role in how these autonomous systems move. On the road, increasing automation will have drastic effects on safety. In urban environments, there is also great possibilities for improvements in traffic congestion. Industrial automation has already begun to decrease amounts of undesired or unsafe manual labor [2]. Investments in space technologies and space robotics are growing as well, sometimes serving as the only solution for unmanned tasks, or in other occasions decreasing risk of otherwise manual tasks [3]. There is also increasing usage of agricultural drones, such as for aerial imagery or seed planting [4].

Motion planning is the process of a robot, vehicle, or other system calculating a collision-free trajectory and the corresponding control inputs to move from its starting state to its goal set. For

mobile robots, the focus is moving through physical space without collision, but motion planning techniques can expand into higher dimensions for any robot state-space. This concept of motion planning is highly important regardless of application, environment, methods of robot sensing, methods of actuation, or size. As will be explained with sampling based planners, there are also examples of these algorithms applying well outside of the realm of robotics [5].

The idea is to incorporate both state dynamics and obstacles as constraints on the trajectory planning. In recent years, feasible path planning has evolved into optimal path planning for single robot and multi-robot systems at both the local and global perspective. Sampling based planners have been at the core of some of these developments.

There are varieties of motion planning techniques. Various reviews of these techniques have been given, such as in [5] and [6]. Some examples include graph search based planners or grid-based as in [7] and [8], sampling based planners as will be discussed most thoroughly, numerical optimization methods and interpolating curve planners as in [6], and artificial potential fields as in [9]. They each have their trade offs. These techniques should be compared in their computational complexity, time taken to find solutions, ability to guarantee those solutions (completeness), storage requirements, cost of trajectory solution, practicality, applicability in larger varieties of dynamics, scalability in multiple robot systems, and whether they are any-time. The term any-time refers to an algorithm that finds a solution, then continually improves upon the solution over time.

Graph search and grid-based algorithms implement a state space as a grid that represents where objects are; the most notable are cell-decomposition methods [8]. Some examples are A* and theta*. These may work well in small dimension state spaces for environments known a priori [6], but require a lot of memory and computational speed for higher dimension spaces since the entire space must be stored [5]. Interpolating Curve Planners can be used for creating very smooth paths through the state space; they generate paths through defined geometric curves such as polynomials, lines, circles, splines, and clothoids [6]. This is useful in considering comfort and feasibility of robot dynamics, but they can be computationally complex and even have non-convergence issues [10]. These are an older set of algorithms that are especially less useful with the onset of globally

optimal sampling based planners that require no interpolation. Artificial potential fields calculate some artificial potential that attracts the robot towards the goal and away from obstacles; they are conceptually simple and computationally simple, but they risk getting stuck in local minimums and thus are not complete [5].

1.1.2 Sampling Based Planners

Sampling based algorithms have proved to be computationally efficient in high dimension state spaces. They randomly sample from the state space to form a series of discrete state values connecting the starting point to the goal set. Where methods like the grid-search required storage of the entire state space, sampling methods scale much more easily with dimensions since they only require storage of a finite list of useful states. Traditionally, the state space is sampled randomly, leaving the mathematical proofs to probabilistic completeness; this means that the problem is guaranteed to return a solution in finite time with probability of one. There are also explorations of more deterministic sampling such as in [11] which applied neural networks to learn biased regions of the state space for sampling.

The Rapidly Exploring Random Tree (RRT) is one of the early useful examples of sampling based planners, which has proven to quickly find paths [12], and paths that are dynamically feasible [13]. There are even methods that have achieved asymptotically optimal conditions globally, as in RRT* and PRM* from [14]. This means the algorithm may find a feasible path at some point in time, but if a robot has more planning time, the cost of the path is improved, asymptotically approaching the most globally optimal path possible. This global optimality was an extremely important development in the field of motion planning.

Sampling based planners are generally seen as rather computationally feasible, so they can be compared in terms of completeness, optimality, their any-time nature, practicality, and scalability with number of robots. Scalability is discussed further in the next multiple-robot planner subsection. In terms of practicality, a difficulty in this realm of planners is properly integrating the discrete state lists with local solutions for varieties of robot dynamics [15].

Motion planning techniques are often discussed in the context of robotic systems moving through a state space. That being said, many of the related algorithms are abstracted into the general state space, allowing for a much greater variety of applications of planning techniques. Some of the sampling based algorithms were initially designed for continuous systems, but there have since been explorations in controlling hybrid dynamic systems. For example, in [16], a sampling based method was used to connect an initial state of a faulted power system to a desired operating point as a method of responding to cyber-physical security concerns. Cyber-physical faults refer to faults due to any system components, physical attacks, or cyber attacks. Decentralized planners become especially relevant in cyber-physical security, since decentralized planning involves the sharing of information among non-cooperative entities [17]. This thesis will focus largely on a decentralized sampling based planning method for mobile robots.

1.1.3 Multiple-Robot Planners

There is thorough literature for single-robot approaches in optimal sampling-based path planning. Multi-robot planning has also been explored, in prioritized [18], centralized [19], and distributed [20] methods. Prioritized methods simply give a rank to the robots, and the lower priority robots can only find paths that avoid collisions with the higher priority robot paths. This is computationally feasible but unrealistic in competitive situations; it is also easily scalable to more robots, but at the expense of optimality for lower priority robots [15]. Centralized methods treat all robots together as one system and are calculated on one processing system. This means that a social optimum can be achieved. A social optimum concludes that there is no possible smaller total cost of the sum of all robot path costs. This may be useful, but requires much more intense computations that scale exponentially with the number of robots; this requires a centralized processor and again is unrealistic in competitive scenarios [21]. Distributed systems have been analyzed in a few ways, such as encoding network goals as some collective function in which the partial derivatives for each robot generates the corresponding control law [15]. These methods utilize Lyapunov analysis to guarantee convergence of solutions - but they have not formed very optimal solutions in arrival

times or energy usage [22]. These algorithms are of course specifically relevant to scenarios with define-able collective objectives.

All of these issues described above were the motivation for the i-Nash Trajectory Algorithm designed in [21], which was the first distributed any-time algorithm for non-cooperative robots converging to a Nash Equilibrium. This algorithm is designed to be at-most linear with respect to the number of robots, making it very scalable compared to existing solutions. This is a trajectory-based game-theoretic algorithm. In a social optimum the total sum of costs of all robots is minimum, but in a Nash Equilibrium the individual robots cannot decrease their own cost given the current path plan of the other robots. This trajectory planning algorithm is then expanded to the more practical policy-planning algorithm in [23], which incorporates a feedback policy to keep a robot close to the trajectory as designed in the first algorithm. Additionally, in [15] an algorithm is proposed that integrates decentralized optimal feedback planning with decentralized conflict resolution; since i-Nash Trajectory and i-Nash Policy occur offline, this conflict resolution would be the online component that avoids collisions real-time by sensor feedback.

1.2 Contributions

The initial i-Nash Trajectory design paper [21] provided simulation results from first-order dynamics of holonomic disc robots. The simulation results of mean path length and times reaching goal were benchmarked with prioritized and any-time prioritized algorithms. Since this data had been collected, computational complexity and scalability are the important metrics that this thesis focuses on.

In this thesis, all simulations are created with original Python code. Mathematically proven path finding properties were verified visually. Robots are updated to contain second order dynamics, becoming four-dimensional state space double-integrator disc robots. Tests are performed for various robot numbers, static obstacle arrangements, starting states, and goal set states. A critical feature of the i-Nash algorithm is that the computational complexity is supposed to increase lin-

early with the robot number. Data is provided regarding computational complexity as it increases with the robot number. Linearity is verified to be practical for a large number of simulation cases with different inputs, where inputs are defined as the starting points and goal sets of robots. Linearity is easily proved mathematically, but has less clear practical implications for these different inputs - hence the value in increasing simulation results. Additionally, existing techniques in sampling algorithms (RRT-connect and k-d trees) are integrated with these simulations. Rather than just a path-displaying simulator, a full Pygame video simulator is also created here, displaying motion of disc robots across the screen. Various comments are made regarding the overall performance of the algorithm, including successes and drawbacks in the design. It is noted that there are limitations in the extend procedure and that additional difficulties arise in the global costs when increasing robot dynamics beyond first order; simple biased sampling is used to help combat these issues.

This thesis more thoroughly describes the details of the algorithm from [21], then describes the implementation method of simulation, the results data, conclusions, and potential next steps.

Chapter 2

Algorithm Descriptions

The i-Nash algorithm designed in [21] is more thoroughly described here. The Implementation section will then explore the programmatic methods.

The i-Nash Trajectory algorithm was the first distributed, anytime algorithm to compute open-loop Nash equilibrium for non-cooperative robots. It proved to return successful mean path lengths in comparison to prioritized and any-time prioritized methods. Simulations were performed for an eight robot scenario with small but frequent static obstacles, and a six robot scenario similar to a four way intersection. Both cases only used first-order dynamics and therefore a two dimensional state space of x and y positions.

2.1 i-Nash Trajectory

2.1.1 Primitive Procedures

The i-Nash Trajectory algorithm involves a number of important primitive procedures, which are defined mathematically here.

(a) *Sampling:* Sample(A) typically returns a uniformly random sample in A. A non-uniform

sampling procedure is also discussed in this thesis.

(b) *Local Steering:* Steer(x, y) returns a state z by steering state x towards state y at a maximum norm value η . Steer(x,y) = argmin$_z$|| $z - y$ || where z $\in B(x,\eta)$. Additionally, a trajectory σ(x,y) can only have one change in the label $\lambda(\sigma(x,y))$, where the λ associates each state a set of atomic propositions, such as x $\in$X$_F$, the feasible region, or x $\in$X$_G$, the goal region.

(c) *Nearest Vertex:* Returns state in S nearest x.

Nearest(x, S) = argmin$_y$|| $y - x$ || where y $\in$ S

(d) *Near Vertices:* Returns states in S within norm r of state x.

Near(x, S, r) = $\{y \in S \mid || x - y || \leq r\}$

(e) *Path Generation:* Given directed graph G with one root and no cycles, PathGeneration(G) generally returns the set of paths from root to leaf vertices; in this thesis it specifically returns a subset of those, only including the paths where the leaf vertices are at the goal region.

(f) *Collision Check of Paths:* Given path σ and set of paths Π , CollisionFreePath(σ, Π) returns 1 if σ collides no path in Π. $X_{F,i}$ is the feasible region for some robot i.

$X_{free} = \otimes X_{F,i} \cap \overline{X}_{collide}$

(g) *Feasible Paths:* Given a set of paths Σ_i for robot i, and a set of paths σ_{-i} which are the paths that all other robots currently have selected, Feasible(Σ_i, σ_{-i}) returns the set of paths $\sigma_i \in \Sigma_i$ in which CollisionFreePath(σ_i, σ_{-i}) returns 1.

In any of the rapidly exploring trees or graphs, on every iteration the graph G is expanded further throughout the state space, eventually finding connected vertices from the starting point to the goal set. The set of all vertices is defined as V. The set of all local edges connecting those vertices is defined as E. The graph G is the set of vertices V and edges E.

The first step in this process is to take a random sample from the state space. The simplest method is to take a uniformly random sample in all dimensions of the state space. It is possible to bias the random sampling to specific regions if necessary, affecting the spread of the tree or graph as analyzed in [24].

Following the random sampling, the new vertex to be added to the graph is calculated via the

steer procedure. The steer procedure returns this new vertex by steering the nearest vertex in the graph towards the randomly sampled vertex, at a maximum norm value. A Euclidean norm is used here. If the randomly sampled vertex was already within the maximum norm, then the randomly sampled vertex is the new vertex. The steer procedure is conceptually shown in Figure 2.1.

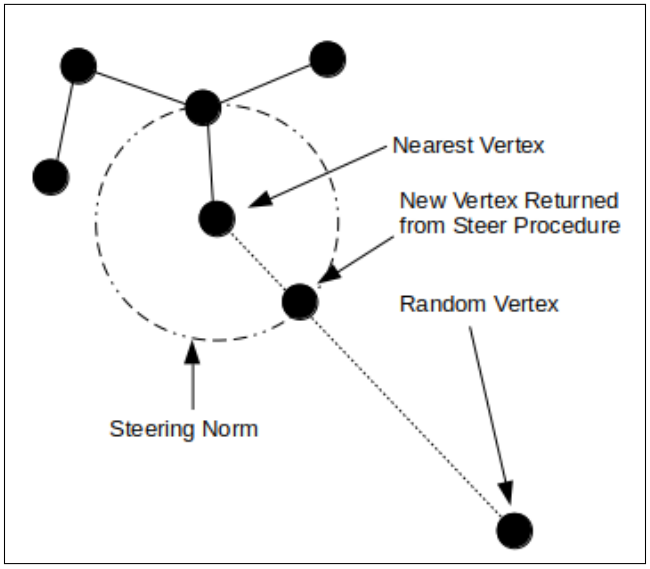

Figure 2.1: Steer procedure used in rapidly exploring random trees (RRT), rapidly exploring random graphs (RRG), and other sampling algorithms including i-Nash.

Now that the steer procedure has returned the new vertex, the next step is extending the graph to include that vertex. RRG, RRT, RRT*, PRM, and other algorithms all have modified versions of the extend procedure. The i-Nash algorithm utilizes something similar to the RRG method.

The i-Nash Trajectory algorithm utilizes an extend procedure similar to the Rapidly Exploring Random Graph (RRG). In RRG, connections are formed in both directions to all near vertices, which allows for eventual global optimality; the graph contains cycles, however, which was important to avoid in i-Nash to allow for a well-defined path generation procedure. RRT* forms a directed graph which is a subset of RRG, but requires re-wiring in order to achieve asymptotic global optimality, which was also important to avoid in i-Nash to guarantee convergence to Nash Equilibria. The extend procedure was then selected to add connections to the same near vertices as in RRG, except only in a single direction.

If there is no collision with the static obstacles, it adds the new vertex as a child from the near

vertices. The near vertices are defined as vertices within the near radius. The near radius varies with the number of existing vertices in the tree, as described in [14]. Note that with this first description, it seems that no vertex already in the graph can have a new parent after the connections have been formed. This is no longer true when the i-Nash-connect method is utilized. The reasoning for this is further explained in the implementation section. The i-Nash-connect method forms graphs from both the starting point as well as from the goal set, similarly to RRT-connect from [12]. Paths are considered found once vertices from the two graphs are within the defined near radius or when vertices from graph one are at the goal set. The extend procedure algorithm is shown in Figure 2.2

Algorithm 2: The Extend Procedure

$V \leftarrow V_{k-1}^{[i]}$;
$E \leftarrow E_{k-1}^{[i]}$;
$x_{\text{nearest}} \leftarrow \texttt{Nearest}(E, x_{\text{rand}}^{[i]})$;
$x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}}^{[i]})$;
if $\texttt{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $\mathcal{X}_{\text{near}} \leftarrow \texttt{NearVertices}(E, x_{\text{new}}, \min\{\gamma(\frac{\log k}{k})^{\frac{1}{n}}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    for $x_{\text{near}} \in \mathcal{X}_{\text{near}}$ **do**
        if $\texttt{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
            $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G = (V, E)$

Figure 2.2: Extend procedure used in the iNash algorithm from [21]. Inputs to NearVertices function were defined in [14].

2.1.2 I-Nash Trajectory Algorithm

The overall i-Nash algorithm works as follows. Initialize each robot's tree with the starting state as the root vertex. In each iteration k, follow the sample and extend procedures for every robot i. Check if any new robots have become active, meaning there is at least one set of edges connecting from the root to the goal set. This is how a path is defined, a set of discrete vertices connecting the root to the goal set. There are further details of how this is done in software, as described in the implementation section. If a robot has become active, add that robot to the active set, referred to as

A. Store all of the currently chosen paths for the active robots from the previous iteration k minus one. Call the BetterResponse procedure for every active robot. This BetterResponse procedure is an iterative method of updating robot i's chosen path. BetterResponse calls the PathGeneration procedure to generate a list of paths from the graph, the CollisionFreePath procedure to check for inter-robot collisions, then selects a low cost path. The formal i-Nash algorithm is shown in Figure 2.3, and the BetterResponse procedure is shown in Figure 2.4. This algorithm is guaranteed to converge to a Nash Equilibrium, but it is also an any-time algorithm; once it converges, robots continue to expand their graphs. If some robot finds and selects a new feasible and lower cost path, then the other robots will re-assess with the BetterResponse procedure and re-converge to the next Nash Equilibrium.

Algorithm 1: The iNash-trajectory Algorithm

for $i = 1 : N$ **do**
  $V^{[i]}(0) \leftarrow x^{[i]}_{\text{init}}$;
  $E^{[i]}(0) \leftarrow \emptyset$;
$A_k \leftarrow \emptyset$;
$k \leftarrow 1$;
while $k < K$ **do**
  for $i = 1 : N$ **do**
    $x^{[i]}_{\text{rand}} \leftarrow \texttt{Sample}(\mathcal{X}_i)$;
    $G^{[i]}_k \leftarrow \texttt{Extend}(G^{[i]}_{k-1}, x^{[i]}_{\text{rand}})$;
  for $i \in \mathcal{V}_R \setminus A_{k-1}$ **do**
    if $V^{[i]}_k \cap \mathcal{X}^G_i \neq \emptyset$ **then**
      $A_k \leftarrow A_{k-1} \cup \{i\}$;
  for $i \in A_k$ **do**
    $\tilde{\sigma}^{[i]}_k = \sigma^{[i]}_{k-1}$;
  for $i = 1 : N$ **do**
    if $i \in A_k$ **then**
      $\Pi^{[i]}_k \leftarrow \{\{\sigma^{[j]}_k\}_{j \in A_k, j<i}, \{\tilde{\sigma}^{[j]}_k\}_{j \in A_k, j>i}\}$;
      $\sigma^{[i]}_k \leftarrow \texttt{BetterResponse}(G^{[i]}_k, \Pi^{[i]}_k)$;
  $k \leftarrow k + 1$;

Figure 2.3: Overall mathematical description of the i-Nash-trajectory algorithm as provided by [21].

Algorithm 3: The `BetterResponse` Procedure

$\mathbb{P}_k^{[i]} \leftarrow \texttt{PathGeneration}(G_k^{[i]})$;
$\mathbb{P}_{\text{f}}^{[i]} \leftarrow \emptyset$;
for $\sigma^{[i]} \in \mathbb{P}_k^{[i]}$ **do**
    if $\texttt{CollisionFreePath}(\sigma^{[i]}, \Pi_k^{[i]}) == 1$ && $(\sigma^{[i]} \cap G_k^{[i]}) \in [\Phi_i]$ **then**
        $\mathbb{P}_{\text{f}}^{[i]} \leftarrow \mathbb{P}_{\text{f}}^{[i]} \cup \{\sigma^{[i]}\}$;
$\sigma_{\min}^{[i]} \leftarrow \sigma_{k-1}^{[i]}$;
for $\sigma^{[i]} \in \mathbb{P}_{\text{f}}^{[i]}$ **do**
    if $\texttt{Cost}(\sigma^{[i]}) \prec \texttt{Cost}(\sigma_{\min}^{[i]})$ **then**
        $\sigma_{\min}^{[i]} \leftarrow \sigma^{[i]}$;
        Break;
return $\sigma_{\min}^{[i]}$

Figure 2.4: BetterResponse Procedure used in the i-Nash algorithm as provided by [21]. Checks for inter-robot collisions and updates robot i's chosen path if a lower cost one is found.

Chapter 3

Implementation

All Python and Matlab code produced for this thesis can be found at the Github repository [25].

3.1 User Interface

The iNash trajectory algorithm is simulated using Python and Pygame. Pygame is a set of Python modules designed for creating simple video games, and is useful for any real-time drawing. Every line and every circle is drawn via a few lines of code providing the pixel locations, color, size, and window to be drawn on. Movements of robots are simulated by erasing the previous location of the robot and re-drawing it at it's location at the next time-step.

The code is set up so that when running, the user can input any number of robots. They can then click on any point in the window for the robots' starting points and click again for the center-points of the goal sets. The color of the dot for robot i's starting point is the same as the color of the dot for robot i's goal point. In the display, the dot with the black box around it is the goal set. In addition to displaying the two-dimensional position drawings, the window is also set up to display an x-position versus x-velocity plot as well as a y-position versus y-velocity plot. Every dot and line in

the position display therefore corresponds to a dot and line in the velocity plots, since every vertex in the algorithm is a fully defined state. There are buttons on the top-right that serve as commands to stop path-planning and start running a video simulation. The overall window size and static obstacle shapes can be adjusted easily in the classes.py file, and all collision checking will adjust accordingly. See Figure 3.7 for a clean image of the entire window for a simpler single-robot path display.

3.2 Near and Nearest Neighbor Searching Using k-d Tree for Spatial Sorting

The brute force method of finding the nearest vertex or near vertices would be to traverse through the entire robot's tree (or loop through an array of all of the vertices, for the simplest for-loop method) and calculate the distance from each vertex to the random vertex. At each calculation the distance would be compared either to the current nearest vertex or the near radius value. This process has O(n^2) computational complexity, where n is the cardinality of V, the number of vertices in graph G for some robot i. This can be reduced to at-worst O(n log(n)) computational complexity by using a spatial sorting algorithm. In this thesis, a k-d (k-dimensional) tree was used to sort the vertices. Each robot had its own k-d tree stored separately from its path planning tree. A k-d tree is a type of binary tree where each level of the tree corresponds to one dimension of the state space, meaning x position, y position, x velocity, or y velocity here. Each consecutive level iterates to the next state space dimension. At each of those levels, when inserting a new vertex, the value of the new vertex's state (one of the positions or velocities) is compared to the corresponding state value of the vertex already in the tree. If the new vertex has a greater state value than the state value of the vertex already in the tree, the new vertex is inserted as a right child (or traversed right to be inserted). This could be visualized in the state space by splitting every new section of the space in half - horizontally if comparing a y value and vertically if comparing an x value. The same thought process can be expanded to more dimensions. See Figure 3.1 for the structure of the k-d tree used

here.

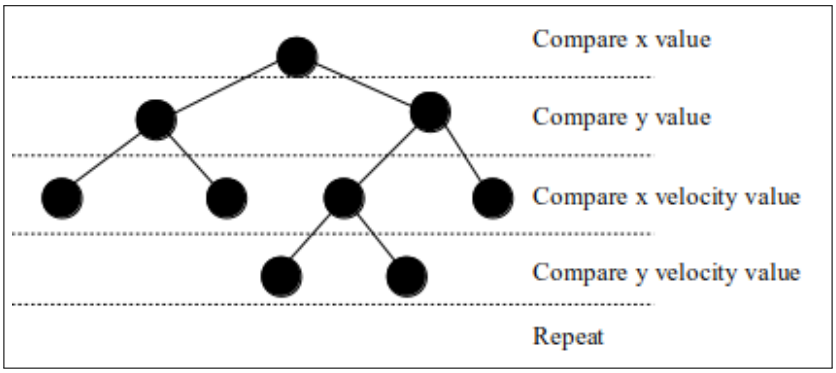

Figure 3.1: Structure of k-d tree used to sort all robot vertices.

The k-d tree can then be used to eliminate regions of the tree when traversing to find the nearest neighbor or near neighbors. For example, if the algorithm has found a vertex that has a Euclidean norm distance from the random one of D, then it is known that there is no reason to traverse in a direction that is guaranteed to only contain vertices with norm greater than D. In [14], there are mentions of the possibility of approximate-nearest neighbor searching. This method is not approximate. It is still finding exact neighbors. See Figure 3.2 for an example of the results.

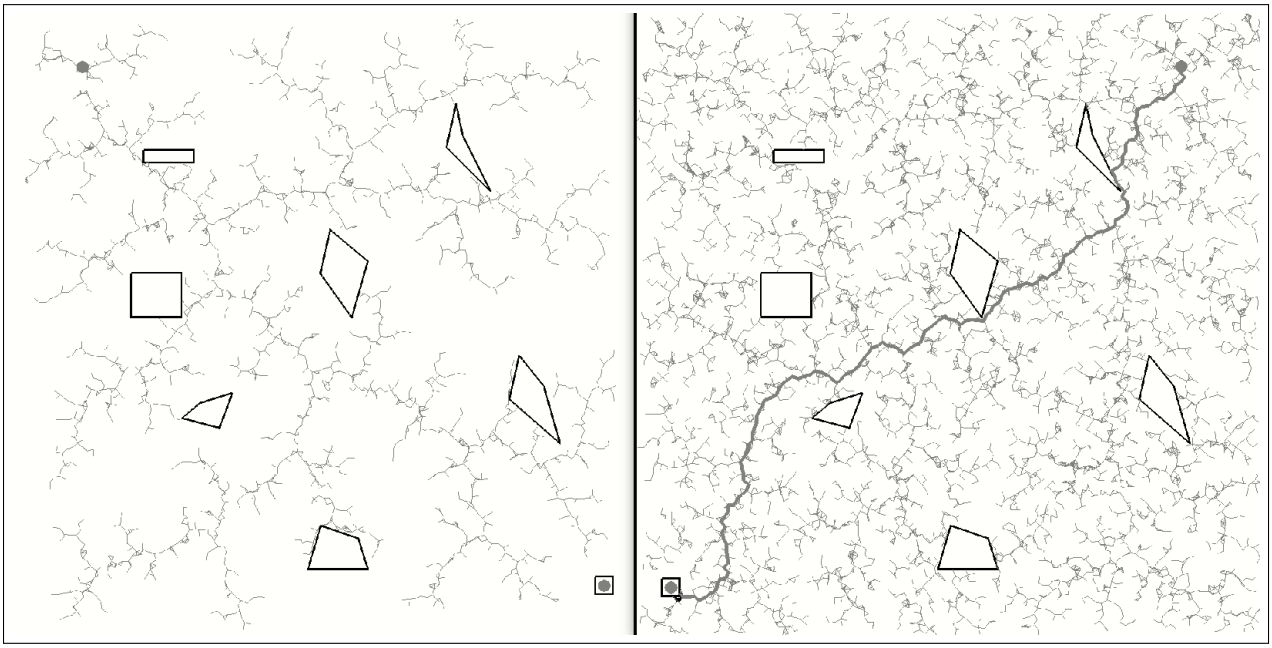

Figure 3.2: Simultaneous display of similar simulations. Only difference between the left and the right is the use of k-d tree sorting in the right algorithm. Notice the one using the sorting algorithm has thousands of more vertices in the same time-frame.

3.3 Static Obstacle and Goal Set Collision Checking

Every time an edge might be added between vertices, the algorithm checks for static obstacle collisions. All obstacles are stored as a series of vertices. The collision checking procedure checks for a collision between two lines, where the lines are defined by the x and y values regardless of the velocities. Even for continuous state space trajectory calculations, the trajectory is drawn via a series of smaller discrete edges; collisions are checked for these smaller edges. See the Robot Description section for further explanation of the continuous trajectory calculations. In collision checking, line 1 is an edge between two vertices in the trajectory, and line 2 is the edge between two vertices in the obstacle. This process is looped through each line of each obstacle; for a more elaborate environment, a sorting algorithm would be useful here to avoid checking for collisions with obstacles that are clearly too far away, but no sorting algorithm was used here. The same process is used to check if a line has collided with the goal set. In the case of the goal-set collision checking, if it collides then the intersection point is also calculated. That intersection point is then used as the new vertex, and this adjustment is completed in the steer procedure. This avoids robots passing through their goal set multiple times.

3.4 Path Procedures

3.4.1 Generation Procedure

A path was initially defined as an array of vertices, where the first element was the root vertex and the last element a vertex at the goal set:

$$\left[Vertex_0, Vertex_1, Vertex_2, ..., Vertex_m\right] \tag{3.1}$$

The vertex is mathematically defined as a fixed state value. In software, there was a class created, called Vertex. Each vertex in a graph is an object of the Vertex class, which contains additional

information other than just the fixed state value. An edge between vertices was defined as the pointer to a parent or child vertex; the Vertex class included the list of parent vertices and a list of child vertices. Each vertex has some location in the graph G, but it also has some separate location in the k-d tree with different parents and children. Therefore, each vertex object additionally stored a list of k-d tree child vertices. The continuous trajectory between a pair of vertices can be fully defined by a few parameters as returned by an optimal control problem. For the control solution to be described in another section, these parameters are the bang-bang parameters of switch time, arrival time, control magnitude, and sign of control input. These parameters were stored within a vertex object, with one set of these parameters for every child the vertex had (since there is one optimal control solution for every trajectory leading to some child vertex).

Defining a path as simply this list of vertices, however, created computational concerns due to an inefficient loop in the robot collision checking procedure. Checking for collisions did not just require checking between a discrete vertex in some path σ_i and another discrete vertex in some path σ_j. It involved checking for collisions for a few points along the continuous trajectories (as defined between two vertices in a path). This collision checking procedure will be defined even further in another section. Any time a robot was iterating through its trajectories to check for collisions, before searching along a continuous trajectory it would have to find which of its children vertices was equal to the next vertex in the path. Since this procedure was being called so frequently within the dominating computational procedure of inter-robot collision checking, it added an extremely large and unnecessary amount of computational time.

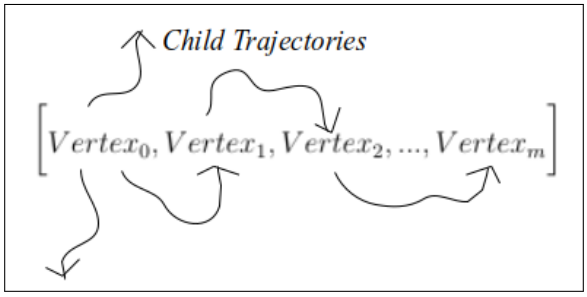

Figure 3.3: Every vertex i can have multiple child trajectories, resulting in later computational inefficiency without knowing which child trajectory ends at vertex i+1.

In order to fix this, the definition of a path was redefined; rather than it being a list of vertices from the Vertex class, a new class was defined as a PathVertex. The class PathVertex contains both the vertex as well as the label for which of its children is the next one in the path. Since the children are listed in an array, the label for which child is just an element number of the array. Thus, a path was redefined as:

$$\left[PathVertex_0, PathVertex_1, PathVertex_2, ..., PathVertex_m\right] \tag{3.2}$$

An alternative approach to this computational issue would be to keep the first definition of a path, only do collision checking for discrete vertex lists, and leave more detailed collision checking to online conflict resolution.

In the path generation procedure, the path is calculated in reverse and recursively, beginning from the goal-set and finishing at the robot root-vertex (which is its starting point). This avoids unnecessary recursion to vertices that are within the tree but have not yet reached the goal set. The algorithm works as follows. For each parent of the current vertex (starting with the goal set vertex), recursively call the path generation procedure and append the paths returned by that procedure to a parent-paths array. This results in a list of paths for each parent, so multiple lists of paths. Manipulate these lists into just one list of paths. Append the current vertex to each path in the list along with the label for which child is used in this path.

Within the path generation procedure, the costs are stored in a separate array as well. For a list of paths returned by the path generation procedure, there is a corresponding list of costs with one cost per path. The cost of a path is defined as the final arrival time at the final vertex. This is calculated as the sum of costs between each vertex in the path. The cost between two vertices is defined as the arrival time at the second vertex. The arrival time at a vertex is retrieved from the trajectory parameters which are stored within each vertex objects.

Every time paths are generated, the paths and costs are not only returned by the function but stored in the corresponding vertex object. This way, any time a new path is being generated and

the algorithm comes across a vertex where the paths have already been calculated, it can simply use those stored paths instead of re-generating them. This seemed to generate positive results in computational time, although a more thorough analysis of storage impacts could be done.

3.4.2 Inter-robot Collision Checking with BetterResponse Algorithm

When robot i calculates which of its paths are collision free with other robots, it does so by testing a fast-forwarded simulation. Unlike in the motion simulator at the end of the code where there is a time delay to visualize the robots moving real-time, during the path collision procedure there is no time delay introduced. At each point along the fast simulation, the distance between robot i and the other robots is checked. If the distance is smaller than some collision distance value, then this is not considered a collision free path. That collision distance value is roughly the diameter of the robots, or slightly larger for extra caution. Since the user interface is not displaying these inter-robot collision checking procedures, the resolution can be decreased. This means that in a given trajectory, for example, rather than checking for collisions among Q discrete points, $K < Q$ points could be checked. The collision checking procedure is still accurate enough for this to work successfully. This allows for less computation during every path collision checking procedure.

3.4.3 i-Nash-Connect Procedure

In the i-Nash extend procedure, there is no re-wiring behavior like there is in RRT*. There are also only connections added in one direction upon new vertices, unlike the extend procedure of RRG. Although the extend procedure was chosen carefully in its design in order to allow for proper path generation and Nash Equilibrium convergence, these differences from the globally optimal algorithms point to a big limitation in the i-Nash extend procedure. This is true because the extend procedures are core to RRT* and RRG achieving global optimality. In i-Nash, there is only addition of paths to a path list upon more iterations of the extend procedure, and choosing of the lower cost one in the BetterResponse procedure.

It was observed that many of the new paths upon new iterations of the algorithm were often

formed as branches from an existing path, rather than a completely new path from the root. This is partially because vertices were only ever being added as children, unlike in the globally optimal single robot algorithms that add connections in different directions. At some point in time, a path would be found. There was a strong likelihood that upon new iterations a new branch would form away from that path nearby the goal-set and also collide with the goal set. For example, see Figure 3.4. This figure displays a few paths that end at the goal set. Notice how each path is about the same for most of the path, then branches form out from the path towards the end. This was a common occurrence for the algorithm. This is similar to something that would be observed for RRT, a non-optimal algorithm.

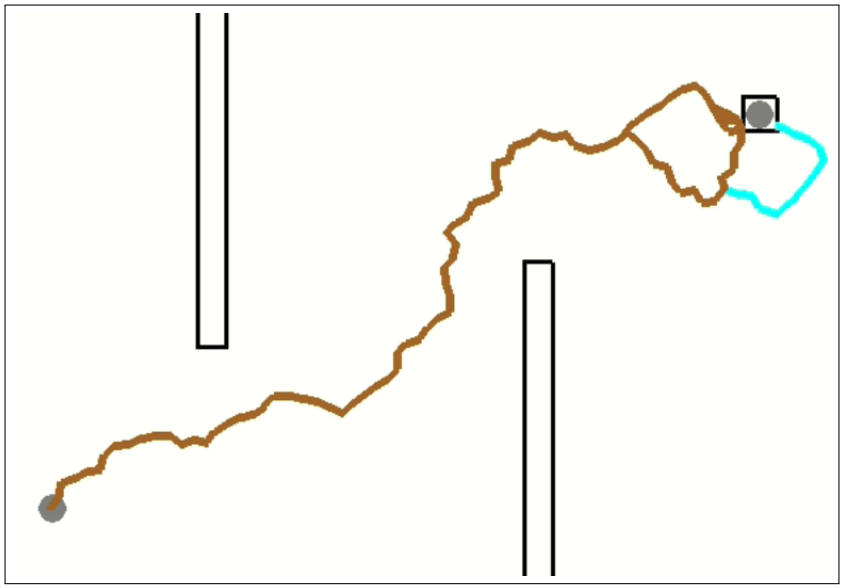

Figure 3.4: Example of typical paths that would form. Bottom left is robot starting point, top right is goal set. Paths frequently formed as small branches from existing ones, rather than completely new paths from the root.

This is an issue in optimality but also potentially in avoiding inter-robot collisions. If robot i were to have an inter-robot collision early in the path displayed in Figure 3.4, then none of its path options would avoid colliding with another robot. If the robot has increased path variety, then it could check those very different paths for collisions with other robots.

The approach to increasing optimality and improving possible path selection was to incorporate the RRT-Connect concept from [12]. In that algorithm, RRTs were formed from both the starting

point and from the goal set. It decreased time taken to find feasible paths, and also made some improvements to path costs. For the case of applying this Connect procedure to the i-Nash trajectory algorithm, it had additional benefits. For example, see Figure 3.5. Notice differences in path formation as compared to Figure 3.4. In Figure 3.5, paths don't just branch away from existing ones; they also can branch back into existing ones connecting to the goal set. This is a feature that is not able to happen in a single graph with the i-Nash extend procedure.

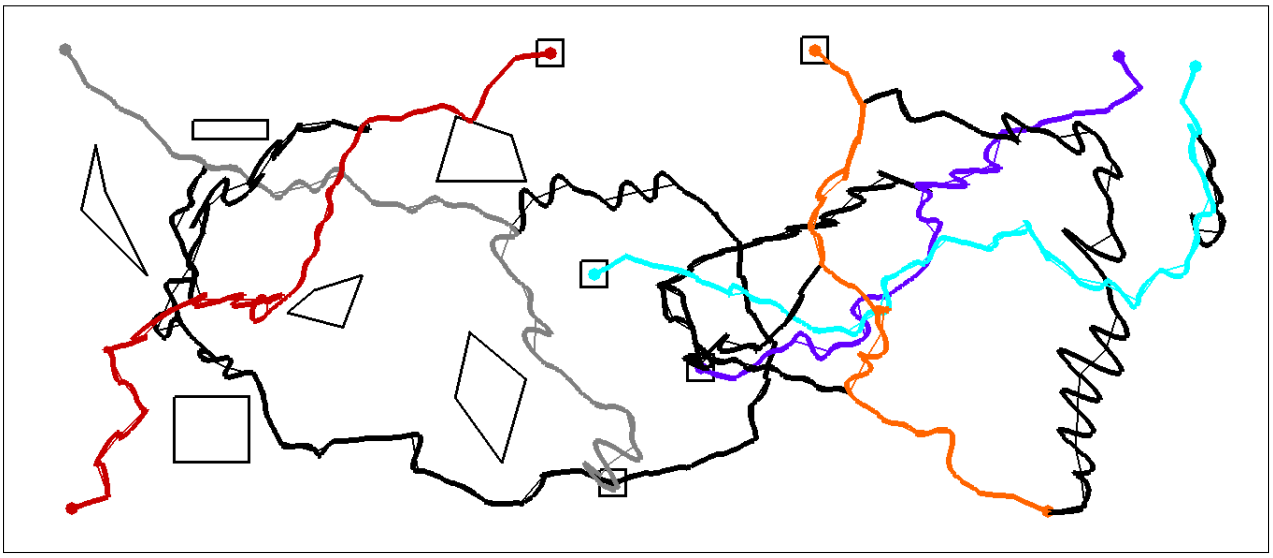

Figure 3.5: Display of some path results utilizing the i-Nash Connect procedure, an application of RRT-Connect within i-Nash. Note that when more paths are found, they now can consist of branches away from existing paths but also branches back into existing paths. This is not a display of all paths found, but just a few.

In [21] it was very specifically explained that avoiding cycles in the graph is critical for guaranteeing that the path generation procedure is well defined. It was important to be careful of this in the details of designing the i-Nash-Connect procedure. The tree originating from the robot starting point will be referred to as tree 1. The tree originating from the robot goal set will be referred to as tree 2. Every robot had a tree 1 and a tree 2. On even numbers of the overall iteration of the algorithm, a vertex was added to tree 1. On odd numbers, a vertex was added to tree 2. The procedure for tree 1 was essentially unchanged from the original algorithm. The procedures for tree 2 were roughly inverted; new vertices were added as parents rather than as children. Near vertices were only searched for the corresponding tree. If a vertex was being added to tree 1, it only checked

for near vertices in tree 1. If a vertex was being added to tree 2, it only checked for near vertices in tree 2. This was to avoid adding edges in the incorrect direction. Then, instead of checking for every near vertex in the opposing tree, it finds the nearest one in the opposing tree and checks only if that one is within the near radius. If it is, then an edge will be added with the parent vertex being the one in tree 1, and the child vertex being the one in tree 2. It was observed that the number of total paths increased. In order to avoid an overly large number of paths, after X number iterations (around 200 in these simulations), tree 2 would stop expanding.

It is known whether a new path has been found before the path generation procedure because a new path is found in two specific cases. The first case is if tree 1 collides with the goal set, in which that goal collision vertex is the one passed into the path generation procedure. The second case is if a connection is added between tree 1 and tree 2, in which the original goal vertex is the one passed into the path generation procedure. It was mentioned before that paths leading to a specific vertex are stored in that vertex class. With i-Nash-Connect, there are now two occasions when the paths must be calculated for a given vertex (this is not in regards to the goal set vertex). Case 1 is if no paths have been calculated for that vertex, as described before. Case 2 is if that vertex is in tree 2, since any tree 2 vertex can have new parent vertices at any new iteration. Tree 1 can never have any new parent vertices.

3.5 Robot Description and Two Point Boundary Value Problem

A very important feature of this algorithm is the integration with robot dynamics. It is valuable to abstract out robot dynamics in many cases when the goal of a motion planning project is to rigorously analyze probabilities or path generation procedures, and to design the fundamental algorithm. The integration of robot dynamics is then compartmentalized and integrated into the discrete state space sampling problem. This means the algorithm can work for different types of robot dynamics, however the optimal control solutions are not easily calculated and integrated for a very large variety of robot dynamics. In the case of this thesis, all robots were given the same

dynamics.

3.5.1 State Space

As has been discussed, sampling occurred in x position, y position, x velocity, and y velocity. This is the 4 dimensional state space of the second order disc robot, often referred to as the double integrator system. The robots had some mass that can be defined in the settings class. The robot was treated as a point-mass, where there were two inputs. Input 1 was the force in the x direction, and input 2 was the force in the y direction. See Figure 3.6 for a visual. Essentially, in order to follow a given velocity and position trajectory, the accelerations were controlled. Hard constraints were placed on the control magnitudes.

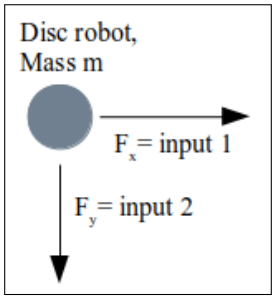

Figure 3.6: Double Integrator Disc Robot used in the simulation of i-Nash.

The linear state equation of this double integrator robot can then be written as:

$$\begin{bmatrix} \dot{x} \\ \dot{v_x} \\ \dot{y} \\ \dot{v_y} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ v_x \\ y \\ v_y \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{1}{m} & 0 \\ 0 & 0 \\ 0 & \frac{1}{m} \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix} = \begin{bmatrix} v_x \\ \frac{u_x}{m} \\ v_y \\ \frac{u_y}{m} \end{bmatrix} \tag{3.3}$$

Another common and more realistic vehicle used in motion planning simulations is the Dubins vehicle. The idea in Dubins vehicle is having a set of wheels that cannot roll in a direction that they are not pointing. This would add a new constraint to the dynamics that couples the directions of motion, which has been rigorously analyzed in various literature such as in [26] and [27]. Ideas

in radius constrained movement have been applied in more elaborate ways as well, including flight vehicles in [28]. The Dubins vehicle is not implemented in this thesis. This could be a potential next-step.

3.5.2 Control Solution

Since every vertex added to the robot's graph is a well-defined state space sample, that means every connection added between two vertices can be defined as a fixed initial state, fixed final state optimal control problem. This breaks down to a constrained optimization problem, and can be solved as a minimum fuel or minimum time problem. For this thesis, the minimum time problem was solved.

Necessary conditions for an optimal controller can generally be determined from the Hamiltonian. Within the Hamiltonian equation, x is the state, u is the input, t is time, f is the state equation defined in equation 3.3, p is the co-state in the dual problem, and g is the function integrated within the cost function. For minimum-time problems, this Hamiltonian comes out to be:

$$H(x, u, p, t) = g(x, u, t) + p^T f(x, u, t) = 1 + p^T f(x, u, t) \tag{3.4}$$

For the case of a two dimensional state space of position and velocity, the solution would be simple. It would be a bang-bang solution with one switching time. It becomes a bit more complex for the four dimensional state space problem, which is not easily calculated with the Hamiltonian method.

For this specific problem with simpler robot dynamics, it can be noticed that the x and y directions of this robot are uncoupled. The following approach was used.

For step one, solve the optimal control problem in the x and y directions separately, each being their own two-dimensional state-space bang-bang solution. This means calculating the switch time, given the constraint of the input force. See equations 3.5 to 3.10 for this analytical solution. Notice that this would mean the two directions would have different arrival times, which doesn't make

sense.

Step two is to keep the solution for the dimension that takes longer, since this dimension cannot get to the desired state within the time-frame of the other dimension. For example, if the x dimension has a calculated smaller arrival time, then keep the y arrival time and y-switching times, and do not use these initial x trajectory calculations.

For step three, re-calculate what was initially the faster dimension, but as a fixed final time problem. For the example given in step two, this would mean now re-calculating a different solution for the x trajectory. That means that this re-calculation is now a fixed initial state, fixed final state, fixed final time problem. It is no longer an optimal control problem in this dimension, since the cost has to be equal to the fixed final time. There could be multiple solutions to this problem, but the approach taken was a sub-bang-bang problem. This meant calculating the magnitude of control input and the switching time required to achieve the desired final state and final time. See equations 3.11 to 3.17 for this analytical solution.

Analytical solutions are solved via creating a system of equations describing the motion of the robot defined previously, assuming bang-bang solutions. The subscripts "bb" are added to represent bang-bang from step one, and "sub" to represent sub-bang-bang from step three.

The beginning equations for step one are:

$$V_f = V_0 + u_m t_{s,bb} - u_m(t_{f,bb} - t_{s,bb}) \tag{3.5}$$

and

$$x_f = x_0 + v_0 t_{f,bb} + 2u_m t_{f,bb} t_{s,bb} - u_m t_{s,bb}^2 - \frac{1}{2} u_m {t_{f,bb}}^2 \tag{3.6}$$

where u_m is the maximum control magnitude normalized to the robot mass, t_s is the switch time, and t_f is the final time. Equations 3.5 and 3.6 solve to the following results:

$$t_{s,bb} = \frac{-v_0 \pm \sqrt{v_0^2 - u_m C}}{u_m} \tag{3.7}$$

$$t_{f,bb} = 2t_{s,bb} - Q \tag{3.8}$$

where C and Q are just constants defined as:

$$C = x_0 - x_f - v_0 Q - \frac{1}{2} u_m Q^2 \tag{3.9}$$

$$Q = \frac{v_f - v_0}{u_m} \tag{3.10}$$

The beginning equations for step 3 are:

$$v_f = v_0 + u_m t_{s,sub} - u_m (t_{f,bb} - t_{s,sub}) \tag{3.11}$$

$$x_f = x_0 + v_0 t_{f,bb} + 2u_m t_{f,bb} t_{s,sub} - u_m t_{s,sub}^2 - \frac{1}{2} u_m t_{f,bb}^2 \tag{3.12}$$

Equations 3.11 and 3.12 solve to the following results:

$$t_{s,sub} = \frac{-B \pm \sqrt{B^2 - 4AD}}{2A} \tag{3.13}$$

$$u_{sub} = \frac{v_f - v_0}{2t_{s,sub} - t_{f,bb}} \tag{3.14}$$

where A, B, and D are just constants defined as:

$$A = v_0 - v_f \tag{3.15}$$

$$B = 2v_f t_{f,bb} + 2x_0 - 2x_f \tag{3.16}$$

$$D = x_f t_{f,bb} - x_0 t_{f,bb} - \frac{1}{2} v_0 t_{f,bb}^2 - \frac{1}{2} v_f t_{f,bb}^2 \tag{3.17}$$

This overall design method from steps one, two, and three is guaranteed to find a solution in software. When calculating the switch times, it is assumed that the first control input is the maximum in the direction of the next state and the second control input is the maximum in the opposite direction of the next state. This is equivalent to saying that the fastest time to get to a location at a specific speed is to accelerate as quickly as possible and then slam on the brakes at

the last second to ensure arrival at the proper velocity. If this has no solution, then the reverse is assumed, where the accelerator is first applied in the reverse direction. The "no solution" case is when the term within the square root returns less than zero. See equations 3.18 to 3.20 for more details on the proof that a solution is always returned.

There is no solution if 3.7 is imaginary, or when:

$$v_0^2 - u_m C < 0 \tag{3.18}$$

Equation 3.18 solves to:

$$v_f^2 + v_0^2 < 2u_m(x_0 - x_f) \tag{3.19}$$

Since u_m is the maximum control magnitude normalized to the robot mass, equation 3.19 tells us that there is only no solution if the sum of initial and final kinetic energies is less than the work applied. If this is the case, then as mentioned, the acceleration is chosen to have negative of the initial assumption. When this is done, then equation 3.19 is adjusted to find there is no solution if:

$$v_f^2 + v_0^2 < 2u_m(x_f - x_0) \tag{3.20}$$

Since the only difference between equations 3.20 and 3.19 is a negation, then it is known that if one assumption does not return a result, the other assumption will. The only occasion where this won't have a solution is if the required trajectory collides with a static obstacle, which can be easily checked with procedures already described and is no concern for error.

3.6 Velocity Biasing for Path Smoothing

Sampling in the two position dimensions for a first order robot is much simpler than sampling in four dimensions for a second order robot. One issue is that the position sampling may form a reasonable path, whereas the velocities associated with those positions may be abnormal. For example, a robot moves from vertex 1 to the right until it is at vertex 2. However, maybe the velocity

at vertex 1 is left, and the velocity at vertex two is left. When this type of process repeats out of random chance, the result is a robot that oscillates in one or two dimensions a lot. The algorithm still works, but in a very non-globally-optimal manner even with locally optimal conditions.

The method of solving this issue was to bias the velocity sampling. This meant increasing the likelihood that a velocity sample has a value in the direction of the goal set. It was important to not give zero likelihood of velocity in the reverse direction, since avoiding obstacles might require moving in multiple directions. This solution yielded positive results without noticeable drawbacks. See Figure 3.7 and observe the x-velocity plot. As the robot moved to the right, there were many cases where the x velocity sampling and x value sampling were positive due to the biasing. This resulted in less cases where the robot had to turn around and move left briefly for no reason other than random chance.

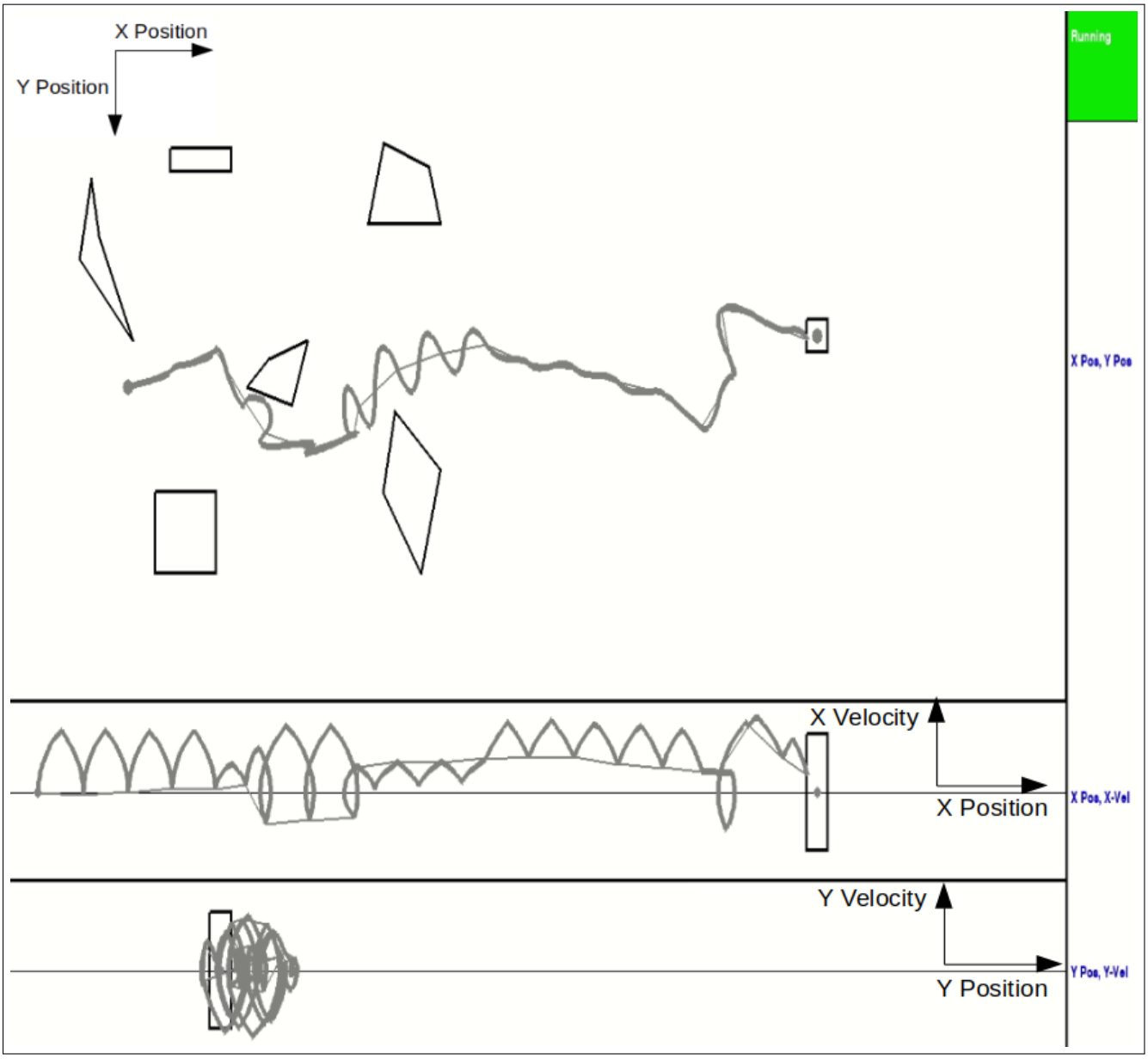

Figure 3.7: Display of entire user-interface. Button on the top right is used to stop path-planning and start moving simulations. X velocity plot here is a great example of the purpose of biased velocity sampling. X velocity plot is also strong visualization of bang-bang control solutions. Note that the y-velocity plot is messy by chance that the robot's starting point and goal point have similar y position values; this also relates to the points discussed next.

The next observation was that in the case where the position of the start point and goal point were similar in one dimension - such as having similar y values, like in Figure 3.7 - the resulting path would often still unnecessarily oscillate in that dimension. To fix this, the solution was to bias velocity samples in that dimension to be closer to zero. Again, this method yielded positive results.

See Figure 3.8 and notice the lack of y-direction oscillations due to the biased y velocity sampling. Also notice the smaller circulations in the y velocity plot for another perspective.

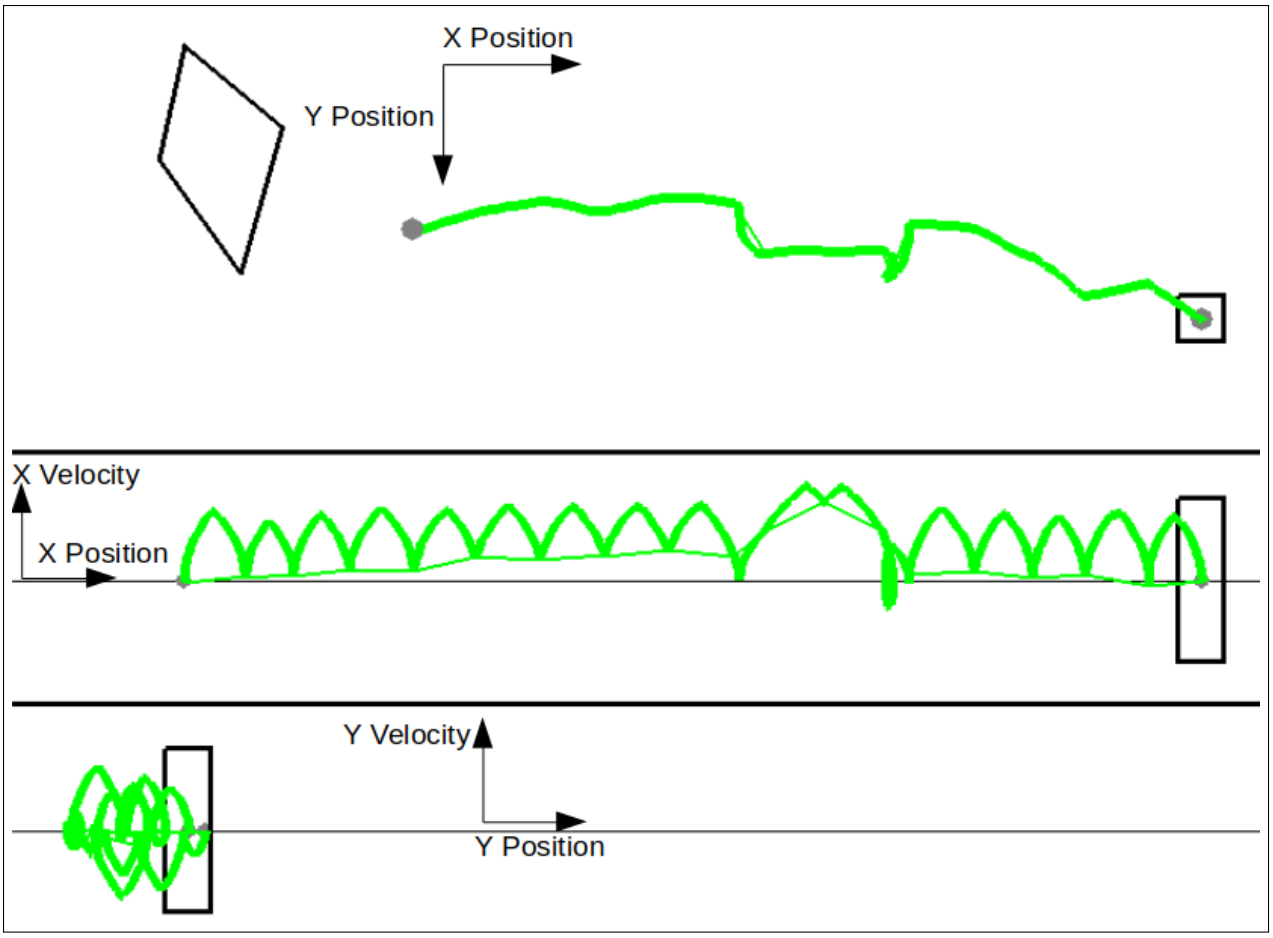

Figure 3.8: More biased velocity sampling. Still includes the positive x velocity bias, but also includes a y velocity bias to be closer to zero. Notice the lack of oscillations in the y position. Also notice the smaller number of circulations in the y velocity plot. Code automatically adjusts these biases corresponding to the relative directions between start and goal set.

The simple sampling bias can be summarized as such:

```
If |GoalPosition_x - StartPosition_x| > (arbitrary A)
    Then If sampling iteration K is a multiple of (arbitrary B)
        Then sample uniform in values towards GoalPosition_x
    Else sample uniform in entire velocity region
Else If sampling iteration K is a multiple of (arbitrary C)
```

```
Then sample uniform in much smaller region centered at zero
```

This procedure remains the same for the y-direction as well. We see easily that regardless of the extent to which this smooths paths, regular uniform samples remain for most iterations. This leaves all theorems of the rapidly exploring random graphs still intact to properly explore through the space, though this is not much of a concern considering the position space is still being searched fully. There is some exploration in the literature of more thorough sample biasing as mentioned previously. A potentially more thorough method of bias sampling here would be implementing neural networks to learn the most useful regions of sampling over many simulations.

3.7 The Motion Simulator

As mentioned previously, there is a button on the interface that allows the user to stop the path-planning and start the motion simulation. This means that the trees stop forming and the robots start following their trajectories in real-time, and the user can watch the robots avoid the obstacles and avoid each other on their way to the goal set. This all works successfully. Full videos can be viewed at the Github repository [25].

When the main algorithm loop is finished upon the user clicking that button, the paths that are currently chosen for that robot are stored and sent to the motion simulator function. For the motion simulator, a time loop is used. This means some time step is defined, in the tens of milliseconds range. Then upon each loop, the time is defined as the time step multiplied by the loop iteration number. Next, the robot displays are updated to where they are in the trajectory at that time. In each loop, there is a time delay introduced so that the simulation occurs close to real-time. In order to avoid making unnecessary calculations during this loop and delaying the simulation away from real time, position arrays with equal time step are calculated prior to this simulator loop. This means the paths (lists of discrete vertices) for each robot are converted into larger arrays of position values incorporating the continuous trajectory information between vertices; each consecutive element is the position at the next time step. This must be done since from vertex to vertex in robot paths,

the arrival times vary greatly. Note that this results in a series of vertices that do not necessarily include the exact discrete vertices that define the graph. See Figure 3.9 for a visual.

Trajectories were previously defined by the set of parameters switch times, sign of control, magnitude of control, and arrival time. The position of a robot at some time t can be calculated directly from those parameters. Since every connection between vertices in a graph is defined with independent switch and arrival times (one vertex treated as time zero, one vertex treated as arrival time, regardless of where this pair is within the graph), it was important to treat each of these parameters not as global times, but delta times.

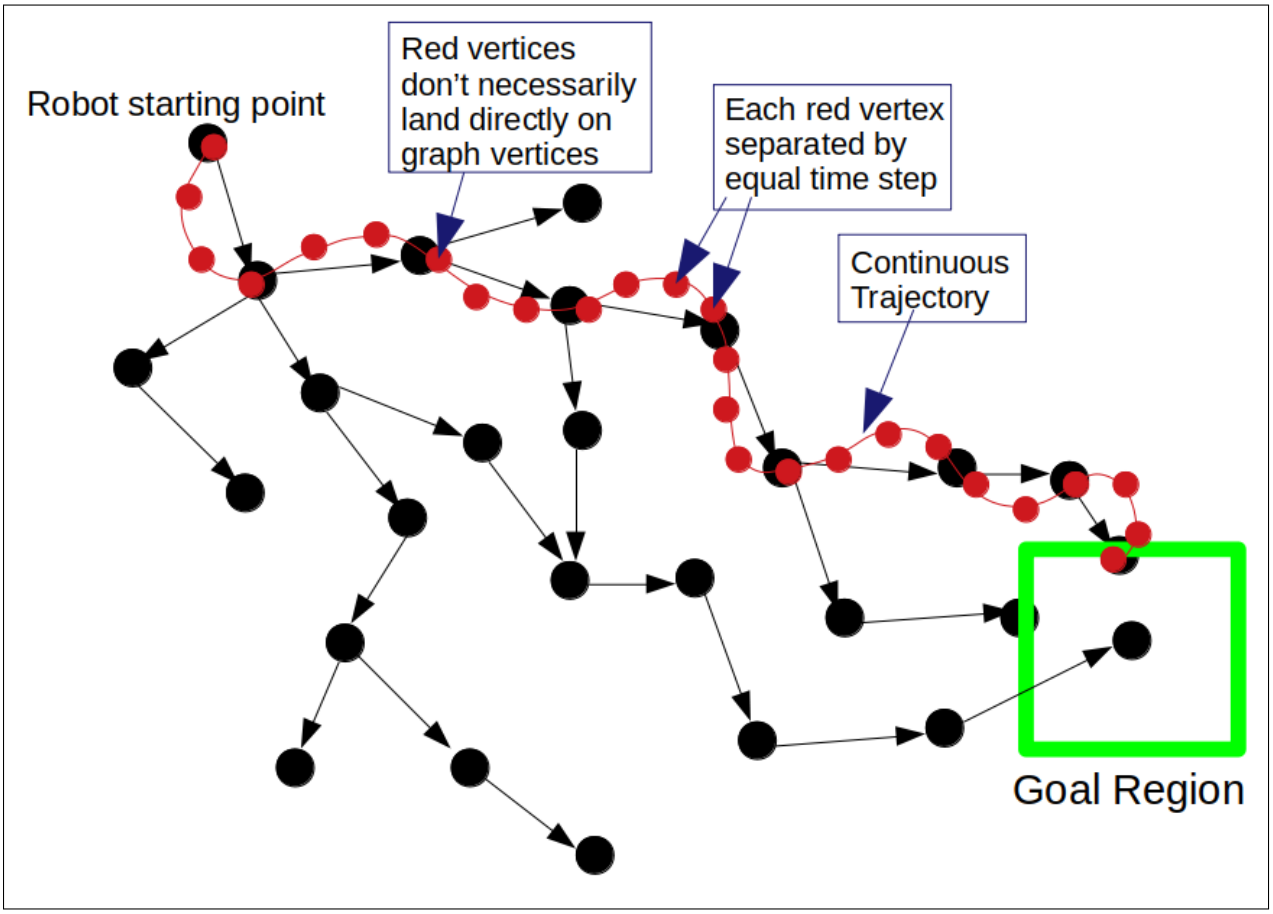

Figure 3.9: Equal Time Step Vertices for Video Simulator. Entire graph is shown here, though simulator only calculates equal time step vertices for the one path selected by some robot i.

All of these simulations were tested in Python 2.7. They use Pygame, as Pygame is a simple method of simulation that does not require a powerful computer. The algorithm itself still uses a lot

of resources, however Pygame is adding negligible computation to the algorithm and simulation. Utilizing Gazebo and ROS (Robotic-Operating-System) would be the next step towards a more realistic implementation of these algorithms, and possibly the better method of implementing the policy based method of i-Nash as defined in [23]. More details were provided in the Algorithms Description section.

3.8 Automation of Simulation Test Cases

After the simulation was confirmed to be working correctly, it was important to collect data regarding the computational complexity for various inputs. All data on computational complexity is provided in the results section.

The Python profiler called cprofiler was used to fix computational concerns. At the end of running code, this returns the cumulative time of running code within every function of the code. This is how the computational issues were found that were discussed in the path definition previously.

A more manual timing procedure was required for recording time taken to reach a Nash Equilibrium. In python, the current time can be obtained easily. During every iteration of the loop, it was checked to see if every robot had chosen a path yet. Once every robot had selected as a path, the current time was saved. Due to the anytime nature of the algorithm, choosing the first point in time that all robots had selected a path is the most accurate way to determine the amount of time taken to reach the first Nash Equilibrium. Due to the fact that this simulator ran on a single processor, calculations were completed to adjust Nash times for the more realistic scenarios of distributed computing for a non-cooperative planning algorithm. This means dividing the recorded Nash Equilibrium time by the number of robots for that simulation case. The timing data was saved to a csv file, along with a column for the number of robots for the corresponding simulation.

It was noticed that between simulation cases, timing could vary a lot with different inputs (starting and goal set locations, amount that the paths were likely to cross with each other, et cetera), and the law of large numbers explains that a larger number of simulation cases was required. Rather

than manually running many cases, this procedure was automated. Essentially, all of the existing code was placed within a loop, and whenever a Nash Equilibrium was reached, that data was saved to the csv file, any vertex and path storage for that simulation was deleted, and the next simulation would start. There was also a timer checking for bad Nash Equilibriums; if any given simulation was taking longer than a specified value (a value large enough to allow accurate timing data for a large number of simulation cases, but small enough to allow for all of the simulations to actually finish), then the simulation would reset. This prevented a bad Nash simulation from ruining the automation procedure. When the user runs the code, they input the number of robots and the number of Nash Equilibria data desired for that number of robots. Any number of simulations can be run for each robot number, and the MATLAB script reading in the csv data would still parse data properly before plotting and calculating averages.

Chapter 4

Results

A new successful Python simulator was created for i-Nash Trajectory. For double-integrator disc robots, the constrained minimum time problem was solved as a two-control input problem with bang-bang solutions. All simulation results and videos can be viewed at the Github repository [25]. See Figure 4.1 for an image displaying a single frame from the video of a successful motion simulation.

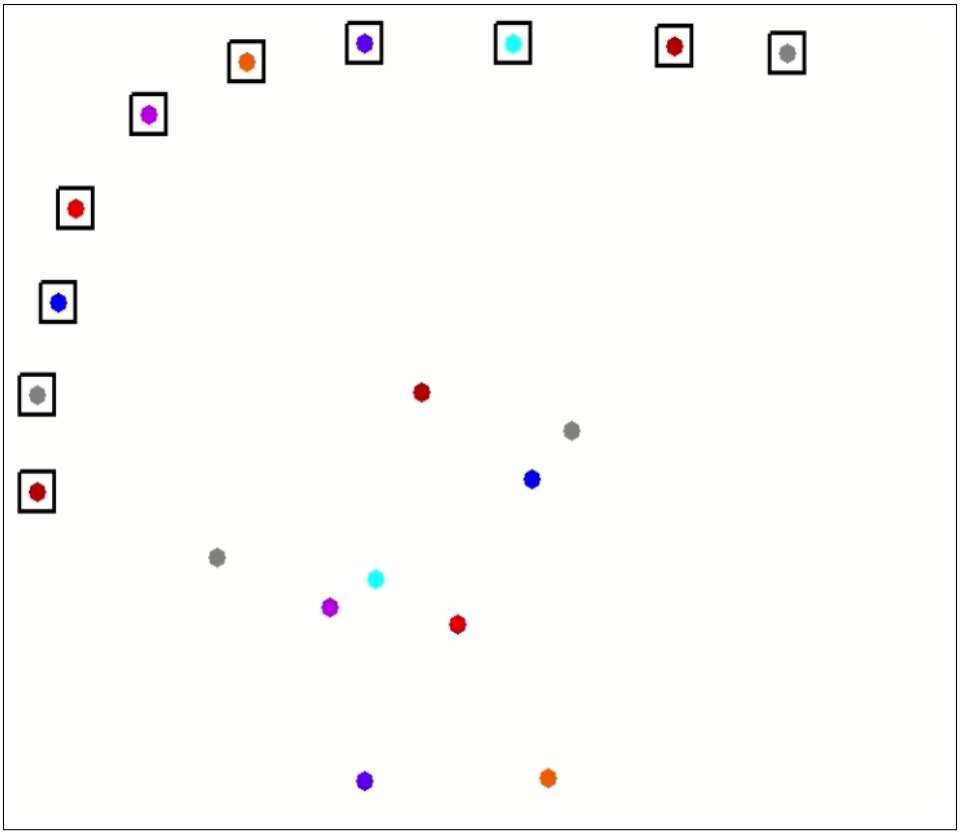

Figure 4.1: Single frame of a motion simulation video. Simulator worked successfully for many robots, even in environments as condensed as this. See the Github repository [25] for full videos.

4.1 i-Nash General Performance Observations

The focus of the results section will be on computational complexity, but some comments are necessary regarding observations on the general performance of the algorithm.

The algorithm was successful and proved impressive in many ways. The k-d tree (k-dimensional tree) implementation for spatial sorting has a major impact on speed of the algorithm; inter-robot collision checking is the dominating computational procedure, but near and nearest neighbor searching could potentially dominate computationally for a large enough number of vertices if not using a sorting algorithm. Without any re-wiring behavior (as in RRT*) or edges in both directions (as in RRG) position paths are frequently far from optimal; with the addition of velocity dimensions in the state space, the trajectories are even more-so non-optimal because of the discussed

oscillation issues. This is all true even though formally a Nash Equilibrium has been reached. The RRT-connect dual tree method can be implemented successfully, resulting in more path options for some robot i meaning a better likelihood of quickly finding a feasible path option without collision. Biased velocity sampling can be used to smooth path oscillation issues and improve path cost. This is a simple but effective solution; more thorough solutions may be possible, such as training neural networks to learn biased regions or using an artificial potential field to control bias of velocity sampling.

4.2 Robot Number Complexity

This section provides additional data regarding the computational performance of the algorithm.

The algorithm design states that the complexity should only increase linearly with the number of robots. This is because the inter-robot path-collision checking procedure is the dominating computational procedure for large numbers of robots, and that procedure is at worst-case theoretically linear. This is stated in Lemma 3.2 of [21], and is explained by the fact that robot A has a finite number Q of paths within iteration k; within that iteration, robot A only calls the collision free path procedure Q or less times per robot.

This linearity was verified with a large amount of simulation data. For one to fifteen robots, simulations were run fifteen times each. This makes for a total of 225 simulations ran. At each simulation, once a Nash Equilibrium was reached, this data was saved and sent to MATLAB, and the simulations repeated. A full description of the automation of these simulation test cases was given in the Implementation section. See Figure 4.2 for the display of all raw data on when N robots reached their Nash Equilibrium, as well as the mean. It should be noted that individual test cases can have a larger Nash time than the linear line limit would “allow” but this does not mean the algorithm is nonlinear; it just means that for different inputs (robot number, robot start positions, robot goal sets, and obstacles) the number Q paths can change between test cases. This

is why so many tests were run. In order to confirm that this data was linear, averages were taken between robot numbers, which is shown in Figure 4.3; the second difference plot is also shown in Figure 4.4 to confirm that there is no concavity in the average plot.

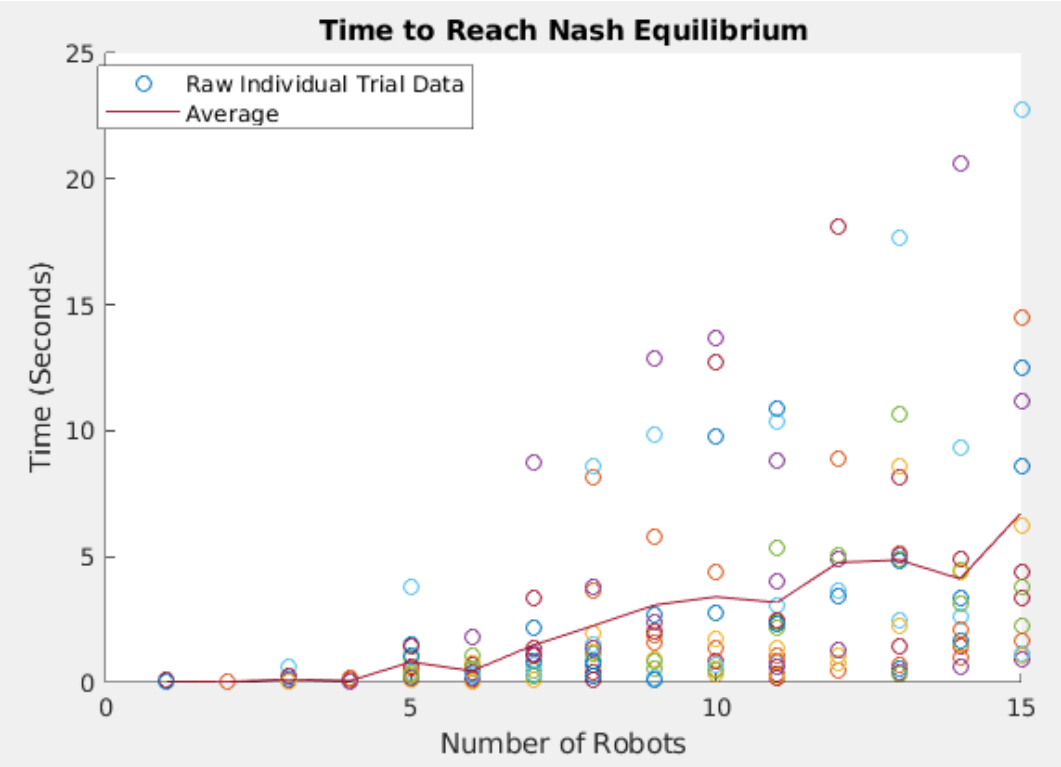

Figure 4.2: 225 measurements of time taken for a set of robots to reach a Nash Equilibrium. "Bad" Nash Equilibria were excluded (situations with no Equilibrium was reached). All circles are individual Nash times, and the line is the average for that number of robots.

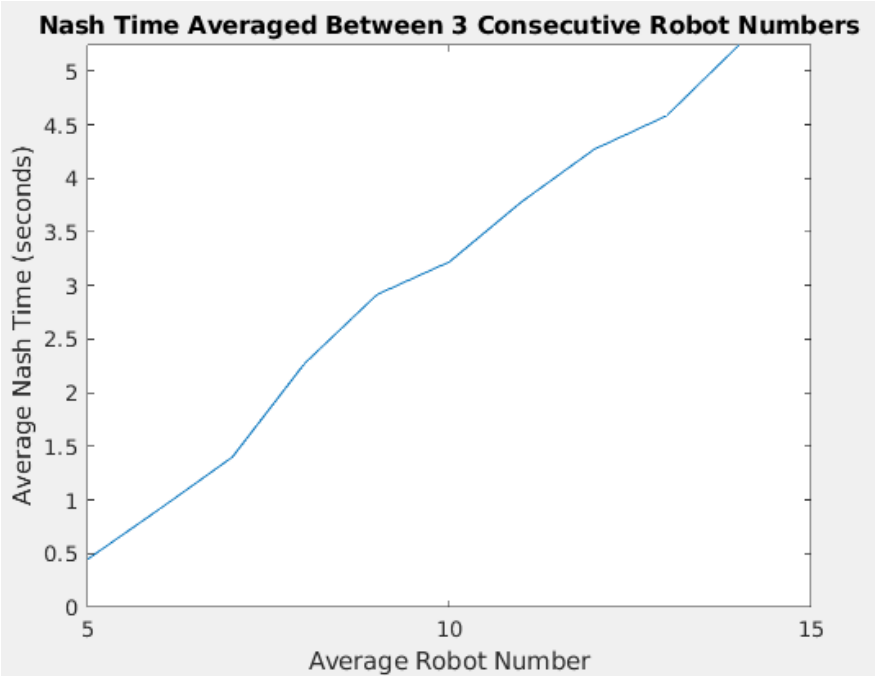

Figure 4.3: Averages of data from Figure 4.2 to better show the linearity. For x-axis value i, the average Nash time is the average time between i-1, i, and i+1 robots, hence why the x-axis is labeled as the average robot number.

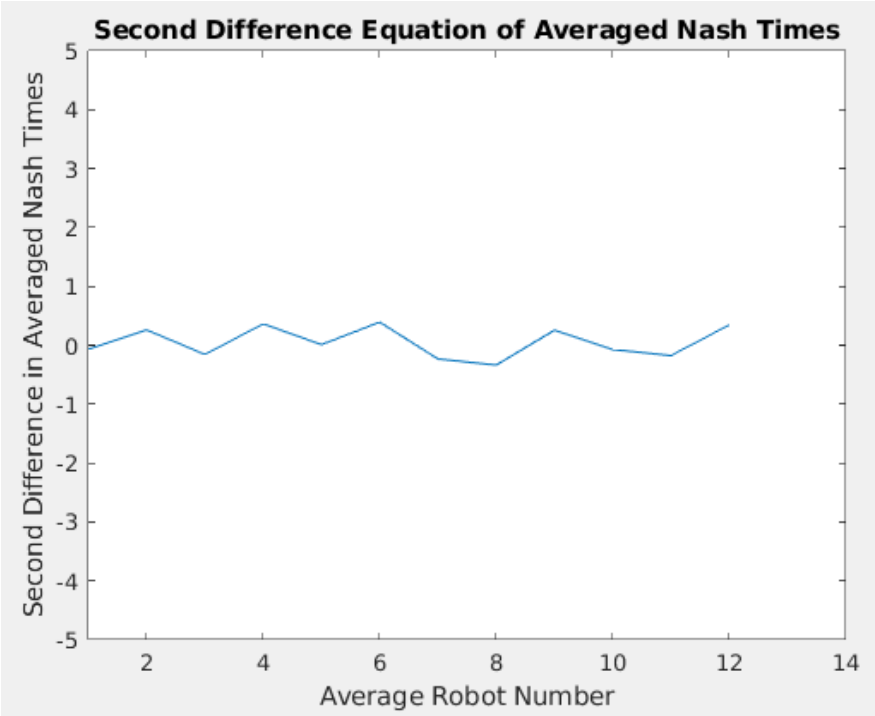

Figure 4.4: Second difference (discrete second derivative) of the plot from Figure 4.3, coming out to be near zero for all values of N robots, showing that there is no concavity, further verifying that the Nash times are only increasing linearly with the robot number.

Chapter 5

Conclusions and Next Steps

In this paper, a new Python simulator was successfully created, many environments were simulated for the i-Nash Trajectory algorithm, and additional data was provided regarding time studies and general performance of the algorithm. This was a great expansion of the data provided in [21]. Previously, [21] simulated the algorithm for first order robots, and confirmed that for a large number of N robots this algorithm performed well; it was the first distributed, anytime algorithm to compute open-loop Nash equilibrium for non-cooperative robots. Here, simulations were run for second order robots with higher-dimension state spaces, a more realistic scenario, complicating the two-point boundary value solution. These local boundary problems were solved as minimum-time problems. Known computational improvement methodologies were implemented successfully, such as the k-d tree for near and nearest neighbor searching. The i-Nash algorithm was integrated with the concept of RRT-connect dual trees, which increased the number of path options for robots and improved speed of finding feasible paths. Data was provided for 225 simulation cases and the corresponding Nash Equilibrium times. It was verified that the algorithm's complexity increases linearly with the robot number on average, although individual simulations are not guaranteed to be have proportional differences from a completely different simulation since there are new inputs

(random chance, starting point and goal sets). It was also noted that although the algorithm formally reaches a Nash Equilibrium, the extend procedure is a limiting factor in global optimality since it does not include re-wiring (as in RRT*) or edges in multiple directions (as in RRG).

There are a few possible next steps. Some involve moving the simulations towards physical application. Others are more theoretical. These possible steps are listed below.

Add uncertainty and disturbances to the robot model. In this disc robot problem, the robot model is treated under the assumption that it is known perfectly. For example, uncertainty in the robot mass could be introduced. In many practical applications, the mass of a vehicle will decrease with fuel usage.

Increase complexity of robot dynamics to something such as a Dubins vehicle, i-Robot ground robot, or Ardrone flight vehicle. Adjust the local optimization solutions accordingly. Update the user interface to display a different robot shape accordingly, if desired.

Use i-Robot or Ardrone trajectory solutions to command position trajectories using ROS (Robotic Operating System) and 3D robotics simulator Gazebo. See Appendix A for a visual of what Gazebo is. This could be the simpler first step in integrating with Gazebo by only controlling to discrete positions. This could involve sending the path vertices to the existing ROS Python controller already designed at the Github Repository [29].

Adjust the static collision checking procedure. Rather than checking for a collision with every line in the environment, use a sorting algorithm to find the near obstacle lines. Then, rather than checking for an exact collision between two lines, calculate the closest distance between the lines. This allows the algorithm to be more realistic in that it considers the physical size of the robot, rather than treating it as a small point in space.

More rigorously analyze sample velocity biasing for path smoothing. There is literature in utilizing neural networks for sample biasing for position, but not in a multi-robot Nash Equilibrium seeking algorithm. Neural Networks could potentially learn biases for velocity to improve path costs. Additionally, it is possible that artificial potential fields could be used to bias regions of sampling (compared to artificial fields being used to directly control robots in continuous space).

Rigorously analyze whether it is possible to change the i-Nash extend procedure and integrate a re-wiring behavior (such as in RRT*) or edges added in multiple directions as in (RRG) without sacrificing Nash Equilibrium convergence or path generation definition.

Expand the i-Nash Trajectory simulation into i-Nash Policy and implement within Gazebo. This means rather than just commanding discrete positions as just suggested, there would need to be a much more involved process involving feedback controls - such as PID. See the Algorithm Description section for a more thorough description on i-Nash Policy and how it connects with the i-Nash Trajectory algorithm.

Integrate the i-Nash Policy simulation in Gazebo with on-line conflict resolution as described in [15].

Appendix
Gazebo Simulator

The work done in this thesis was mainly scoped in Python and Pygame. Gazebo is a 3D simulator that was mentioned a few times, which incorporates 3D hardware simulation and more thorough controlling. See Figure 5.1 for a screenshot of Gazebo.

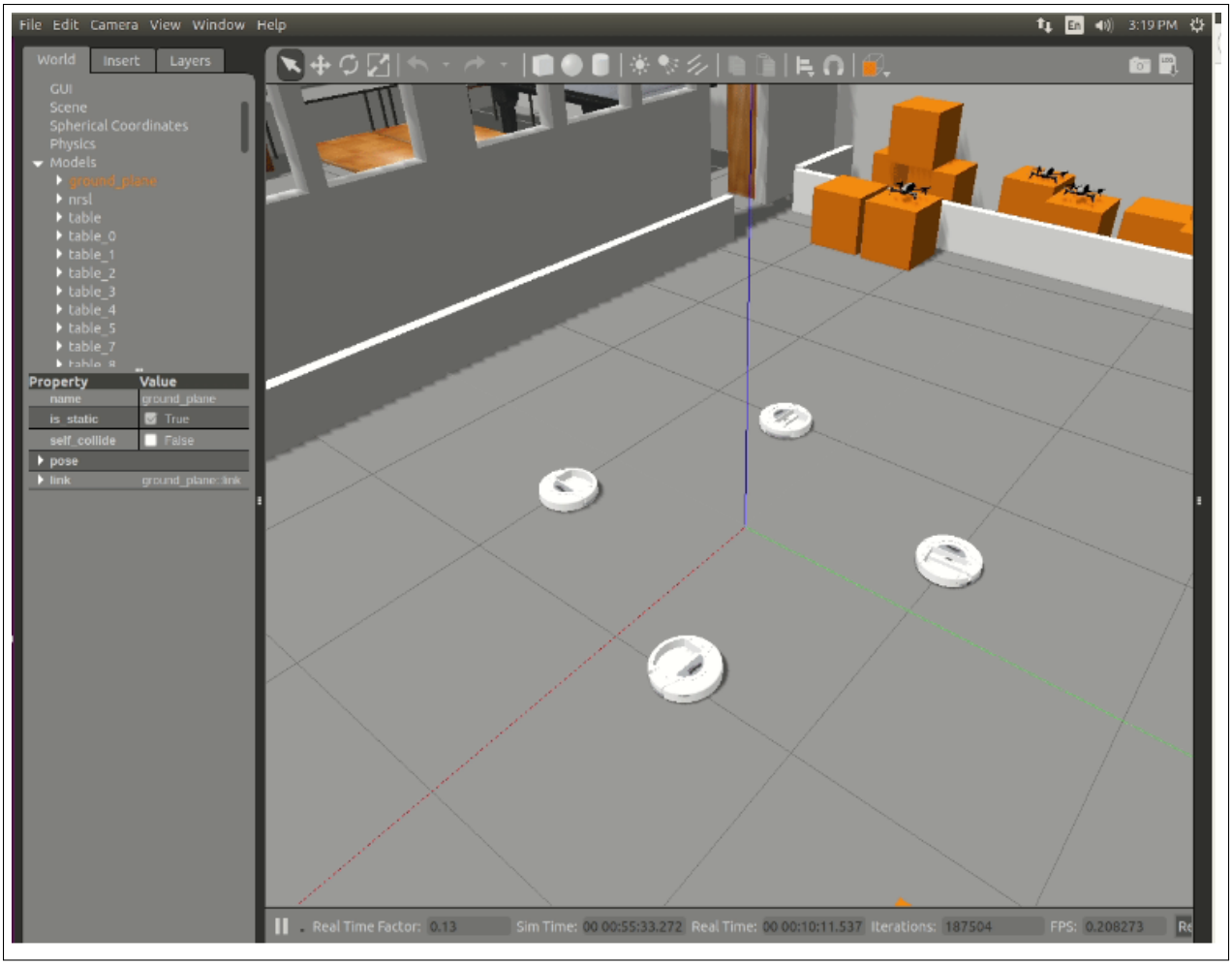

Figure 5.1: Screenshot of the Gazebo 3D simulator with four iRobot ground robots. Utilizing Gazebo was described in next-steps.

Bibliography

[1] S.M. LaValle. Planning algorithms. *Cambridge University Press*, 2006.

[2] The new hire: How a new generation of robots is transforming manufacturing. *PricewaterhouseCooper's LLP*, 2014.

[3] M. Johnson. Space station robotic arms have a long reach. *National Aeronautics and Space Administration*, 2019.

[4] C. Murphy. Legally operating a drone in the agriculture industry. *Iowa State University*, 2020.

[5] S. Karaman. Sampling-based algorithms for optimal path planning problems, ph.d dissertation. *Massachusetts Institute of Technology*, 2012.

[6] D. González, J. Pérez, V. Milanés, and F. Nashashibi. A review of motion planning techniques for automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17:1135–1145, 2015.

[7] C. Saranya, K. Koteswara Rao, M. Unnikrishnan, V. Brinda, V.R. Lalithambika, and M.V. Dhekane. Real time evaluation of grid based path planning algorithms: A comparative study. *IFAC*, pages 766–772, 2014.

[8] M. Sharir. Algorithmic motion planning, handbook of discrete and computational geometry. *CDC Press*, pages 733–754, 1997.

[9] B. Kovacs, G. Szayer, F. Tajti, M. Burdelis, and P. Korondi. A novel potential field method for path planning of mobile robots by adapting animal motion attributes. *Robotics and Autonomous Systems*, 82:24–34, 2016.

[10] A. Ravankar, Y. Hoshino, A. Ravankar, Y. Kobayashi, and C. Peng. Path smoothing techniques in robot navigation: State-of-the-art, current and future challenges. *Sensors*, 2018.

[11] B. Ichter, J. Harrison, and M. Pavone. Learning sampling distributions for robot motion planning. *International Conference on Robotics and Automation*, 2019.

[12] J.J. Kuffner and S.M. LaValle. Rrt-connect: An efficient approach to single-query path planning. *IEEE Conference on Robotics and Automation*, pages 995–1001, 2000.

[13] S.M. LaValle and J.J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, pages 20(5):378–400, 2001.

[14] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research*, pages 30(7):846–894, 2011.

[15] M. Zhu G. Zhao. Scalable distributed algorithms for multi-robot near-optimal motion planning. *IEEE 58th Conference on Decision and Control*, 2019.

[16] Q. Wu, T. J. Koo, and Y. Susuki. Dynamic security analysis of power systems by a sampling-based algorithm. *ACM Transactions on Cyber-Physical Systems*, 2, 2018.

[17] Y. Lu and M. Zhu. A control-theoretic perspective on cyber-physical privacy: Where data privacy meets dynamic systems. *Annual Reviews in Control*, 47:423–440, 2019.

[18] S. J. Buckley. Fast motion planning for multiple moving robots. *IEEE International Conference on Robotics and Automation*, pages 322–326, 1989.

[19] G. Sanchez and J.C. Latombe. On delaying collision checking in prm planning – application to multi-robot coordination. *International Journal of Robotics Research*, pages 21:5–26, 2002.

[20] M. Zhu and S. Martinez. Distributed optimization-based control of multi-agent networks in complex environments. *Springer*, 2015.

[21] M. Zhu, M. Otte, P. Chaudhari, and E. Frazzoli. Game theoretic controller synthesis for multi-robot motion planning part i: Trajectory based algorithms. *IEEE International Conference on Robotics Automation*, pages 1646–1651, 2014.

[22] D. Dimarogonas, S. Loizou, K. Kyriakopoulos, and M. Zavlanos. A feedback stabilization and collision avoidance scheme for multiple independent non-point agents. *Automatica*, 42 no.2, 2006.

[23] D. K. Jha, M. Zhu, and A. Ray. Game theoretic controller synthesis for multi-robot motion planning-part ii: Policy-based algorithms. *International Federation of Automatic Control PapersOnLine 48-22*, pages 168–173, 2015.

[24] C. Urmson and R. Simmons. Approaches for heuristically biasing rrt growth. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2:1178–1183, 2003.

[25] C. Dillinger. Motion planning github repository. https://github.com/codymdillinger/robotics.

[26] D.P. Bertsekas. Dynamic programming and optimal control. *Athena Scientific*, 2, 2000.

[27] L.E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, pages 497–516, 1957.

[28] H. Choi. Time-optimal paths for a dubins car and dubins airplane with a unidirectional turning constraint, ph.d dissertation. *University of Michigan*, 2014.

[29] S. Liu. Single-robot motion planning github repository. https://github.com/jackyuan18.

Cody Dillinger

Academic Vita

Education

The Pennsylvania State University, Schreyer Honors College **Graduation: December 2020**

University Park, Pennsylvania

- B.S in **Electrical Engineering**, with Honors
- M.S in **Electrical Engineering**, with Honors
  - Primary focus: Control Theory

The National University of Singapore **May-June 2016**

Mechanical Engineering Product Design Program

- Studied problem scoping and definition, concept generation, iteration, financial constraints, customer feedback

Research Experience

Networked Robotics Systems Lab **- Control Algorithm Engineer - University Park, PA 2020**

- Created a new Python simulator for the state-sampling-based i-Nash Trajectory algorithm, the first distributed anytime algorithm to compute an open-loop Nash equilibrium for non-cooperative robots
- Integrated locally time-optimal trajectory solutions to approach a more realistic simulation
- Verified computational linearity in relation to the number of robots by automating 225 simulation cases

Work Experience

Penn State **- Lead Electromagnetics Teaching Assistant - University Park, PA 2020**

- Organized lab plans for other Assistants and led weekly labs for 10-20 junior undergraduates
- Built upon the concepts of transmission lines, high frequency (RF) circuits, Maxwell's equations, capacitor design, antennas, waveguides, EM waves, and radars, via Matlab, HFSS, and vector network analyzers
- Assisted with additional sessions in the Circuits and Devices Lab, helping students connect circuit theory with the real-world via oscilloscopes, breadboards, and PCBs

Eaton Corporation **- Power Controls and Software Intern - Southfield, MI Summer 2018**

- Created and documented proof-of-concept for automotive Simulink embedded controls on TI microprocessor
- Modified Simulink field-oriented 3-phase controls for brushless servos, with PID, torque and speed driven methods
- Troubleshot power electronics - ADC, DAC, MOSFETs - via circuit analysis and oscilloscopes

Eaton Corporation **- Engineering Leadership Intern - Beltsville, MD Summer 2019**

- Designed data-logging LabVIEW code for a multi-axial strain-rate controlled test rig
- Drove decisions of test-rig actuators and methods of simulating those actuators

SpaceX **- Software Engineering Intern - McGregor, TX Spring 2019**

- Programmed and reviewed object-oriented LabVIEW used by various rocket engine and component test teams
- Added features and fixed bugs in frameworks for GUIs, performance monitors, data relays, and software unit tests

Penn State at The Navy Yard **- Battery Storage Intern - Philadelphia, PA Summer 2016**

- Wrote background material documents for a training course in distributed energy storage installation safety
- Analyzed data from Battery Management System case studies for Penn State course development
- Completed course-work in solar PV design and implementation, energy storage, and solar site evaluation

GE Transportation **- Product Lifecycle Management Intern - Erie, PA Summer 2017**

- Structured BOMs for service part tracking, 3D scanned engine frames, improved validation of CAD data migrations

Tools & Skills

MATLAB, Python, Simulink, LabVIEW, MultiSim, SPICE, SolidWorks, Linux, Git, C++, Oscilloscopes, Power Electronics, RF, Servos, Controls, Simulation, Testing, Hardware, Digital Signal Processing (DSP), Robotics, Motion Planning, Integration, Systems

Volunteering and Extracurriculars

- *Mentor Collective* - Engineering Mentor to 4 Undergraduate Mentees
- *Springfield FTK* - THON - Fundraising for Hershey Medical Center
- Club Table Tennis, Intramural Ultimate Frisbee