University of Louisville
ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

5-2013

Real-time trajectory generation for dynamic systems with nonholonomic constraints using Player/Stage and NTG.

Ryan Frazier 1989-
University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

Recommended Citation
Frazier, Ryan 1989-, "Real-time trajectory generation for dynamic systems with nonholonomic constraints using Player/Stage and NTG." (2013). *Electronic Theses and Dissertations.* Paper 455.
https://doi.org/10.18297/etd/455

This Master's Thesis is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

REAL-TIME TRAJECTORY GENERATION FOR DYNAMIC SYSTEMS WITH NONHOLONOMIC CONSTRAINTS USING PLAYER/STAGE AND NTG

By

Ryan Frazier
Bachelors of Science in Electrical Engineering, University of Louisville, May 2012

A Thesis
Submitted to the Faculty of the
University of Louisville
J. B. Speed School of Engineering
as Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Electrical Engineering

May 2013

REAL-TIME TRAJECTORY GENERATION FOR DYNAMIC SYSTEMS WITH NONHOLONOMIC CONSTRAINTS USING PLAYER/STAGE AND NTG

Submitted by: ____________________________________

John Ryan Frazier

A Thesis Approved On

____

(Date)

by the Following Reading and Examination Committee:

____

Dr. Tamer Inanc, Thesis Director

____

Dr. Michael McIntyre

____

Dr. Christopher Richards

ACKNOWLEDGMENTS

Through Jesus, all things are possible, even this thesis. Thank you to Chuck Sites, a Linux guru, who helped me through the installation of all programs and Operating Systems. Thank you to Dr. Tamer Inanc for guiding me for the past year in a direction that would lead to a completed thesis, and for providing me with the tools to complete it. Thank you to Rick Paris, a fellow Electrical Engineering student at UofL, who helped me write text file code. Thank you to Yinan Cui, who is defending his dissertation at UofL this year in Electrical Engineering. He helped me learn how to program NTG. Finally, thank you to Kristina Frazier, my wife, who has supported my work at school and home and has made thesis bearable for the past year.

ABSTRACT

This thesis will present various methods of trajectory generation for various types of mobile robots. Then it will progress to evaluating Robot Operating Systems (ROS's) that can be used to control and simulate mobile robots, and it will explain why Player/Stage was chosen as the ROS for this thesis. It will then discuss Nonlinear Trajectory Generation as the main method for producing a path for mobile robots with dynamic and kinematic constraints. Finally, it will combine Player, Stage, and NTG into a system that produces a trajectory in real-time for a mobile robot and simulates a differential drive robot being driven from the initial state to the goal state in the presence of obstacles.

Experiments will include the following: Blobfinding for physical and simulated camera systems, position control of physical and simulated differential drive robots, wall following using simulated range sensors, trajectory generation for omnidirectional and differential drive robots, and a combination of blobfinding, position control, and trajectory generation. Each experiment was a success, to varying degrees. The culmination of the thesis will present a real-time trajectory generation and position control method for a differential drive robot in the presence of obstacles.

TABLE OF CONTENTS

REAL-TIME TRAJECTORY GENERATION FOR DYNAMIC SYSTEMS WITH NONHOLONOMIC CONSTRAINTS USING PLAYER/STAGE AND NTG ii
ACKNOWLEDGMENTS iii
ABSTRACT iv
LIST OF TABLES vii
LIST OF FIGURES viii
I. INTRODUCTION 1
II. LITERATURE REVIEW 5
III. ER-1 ROBOT REVIEW 20
IV. REVIEW ON ROBOT OPERATING SYSTEMS 23
V. DESCRIPTION OF PLAYER/STAGE 25
VI. REVIEW OF NONLINEAR TRAJECTORY GENERATION (NTG) 30
VII. THESIS PROCEDURE 33
A. Installation 33
B. ER-1 Preparation 33
C. Player/Stage Preparation 35
D. NTG Preparation 45
E. Interconnecting Player/Stage and NTG for a Real-Time Application 57
VIII. DISCUSSIONS AND CONCLUSIONS 63
IX. RECOMMENDATIONS 66
APPENDIX I. INSTALLING MAGEIA, PLAYER, STAGE, AND NTG 68
A. Installing Mageia 68
B. Installing Player/Stage 69
C. Installing NTG 75
APPENDIX II. PLAYER/STAGE PROGRAMS 77
A. Wall Following 77
B. Blobfinding 80
C. Proportional Feedback Control 84
APPENDIX III. PLAYER PROGRAMS 88

APPENDIX IV. NTG PROGRAM FOR DIFFERENTIAL DRIVE ROBOT................ 90
APPENDIX V. PLAYER/STAGE/NTG DIFF. DRIVE ROBOT CODE....................... 97
APPENDIX VI. NTG PROGRAMMING PARAMETERS 115
LIST OF REFERENCES 119
VITA 122

LIST OF TABLES

TABLE I: RESULTS OF BLACK BLOBFINDING ……………………………………43

TABLE II: NTG PARAMETERS FOR SIMPLE SCENARIO…………………….115

TABLE III: NTG PARAMETERS FOR THE DIFFERENTIAL DRIVE EXAMPLE…………………………………………………………………………...116

LIST OF FIGURES

FIGURE 1 - Simulation Results of the Conventional and Updated Potential Field Method [6] 7

FIGURE 2 - Simulation Results of the ERRT Method [4] 9

FIGURE 3 - Car-like Configuration with Parameters x, y, θ, ϕ, and *v* [1] 11

FIGURE 4 - Parallel Parking-(a) Initial and final destinations, (b) Geometric planner (.2s), (c) Nonholonomic path (.8s), (d) Optimized path (1.5s), (e) Traveled path [1] 13

FIGURE 5 - Complicated Case-(a) Geometric planner (5s), (b) Nonholonomic path (23s), (c) Optimized path (10s), (d) Traveled path [1] 14

FIGURE 6 - Result of Control Law Applied to Hilare to Stabilize about a Circle [9]..... 17

FIGURE 7 - Result of Control Law Applied to the Car-like Robot to Parallel Park [9].. 18

FIGURE 8 - Typical ER-1 Robot Build 21

FIGURE 9 - Player System Block Diagram [15] 28

FIGURE 10- The Player/Stage System Block Diagram [15] 28

FIGURE 11 - Two Degree of Freedom Control [8] 30

FIGURE 12 - Wall Following using Player/Stage 36

FIGURE 13 - Robot with Blobfinder in Stage Simulation 37

FIGURE 14 - getblob.cc Terminal Output 38

FIGURE 15 - Using Player's "Go To" Driver to Move from (0,0,0°) to (3,3,135°) 39

FIGURE 16 - Differential Drive Components of Motion [23] 40

FIGURE 17 - Using a Proportional Controller to Provide a Smooth Path to a Goal Point 41

FIGURE 18 - A, B: First Felt Location 43

FIGURE 19 - Second Felt Location 43

FIGURE 20 - Path Generated for the Omnidirectional Robot 49

FIGURE 21 - Trajectory of the Differential Drive Scenario 55

FIGURE 22 - Linear Velocity of the Differential Drive Robot 56

FIGURE 23 - Angular Velocity of the Differential Drive Robot 56

FIGURE 24 - Player/Stage/NTG Software Flowchart 58

FIGURE 25 - Player/Stage/NTG Simulation of Multi-obstacle Differential Drive Scenario 60

FIGURE 26 - NTG Waypoints Calculated for the Two Static Obstacle Problem 61

FIGURE 27 - NTG Waypoints Calculated for No Observed Obstacles Scenario 62

FIGURE 28 - Player running simple.cfg with Stage 74

I. INTRODUCTION

The problem of designing routes for mobile robotics is as old as the field of mobile robotics itself. It is a well-researched field with a basic underlying question: How can the robot move from point A to point B? More formally, techniques focus on converting a rigid robot body into a point moving through its configuration space. The configuration space includes all of the spatial points in which the robot can move [1]. In addition to developing paths, routes are also optimized to enhance energy, time, or other parameters. The copious varieties of mobile robots, situations, and constraints each play a specific role in developing techniques to guide vehicular robots. In an age when autonomous systems either aid or replace human work, path planning offers ample opportunities.

Autonomous driving, safety, combat, and recon are only a few of the major aspects mobile robot path planning can influence in modern life. One of the best examples for modern research in autonomous driving is the DARPA Urban Challenge (DUC). The DUC is hosted as an "autonomous vehicle research and development program, conducted as a series of qualification steps leading to a competitive final event" [2]. The most recent challenge was held on November 3, 2007 in Victorville, California. The DUC gave each vehicle that entered the responsibility to navigate in traffic and perform all of the tasks typical drivers must complete, such as "merging, passing, parking, and negotiating intersections" [3].

As is clear in the DUC, safety is of the upmost importance for any realistic autonomous vehicle. Vehicles would need to have the ability to classify images into

people, cars, debris, and so forth, and it would be necessary to recognize conditions such as four way stops, yielding, and erratic driver behavior. Image classification and situational recognition aside, each autonomous mobile robot must be able to generate fast and safe planned trajectories. A vehicle must not only plan the most efficient route to its destination, but must also be able to react, in real time, to obstacles such as pedestrians, road blocks, or crashes [4].

Mobile robotics' influence extends beyond civilian life into military and defense. "Application areas include grid searching by coordinating robots, surveillance using multiple unmanned air or ground vehicles, and synthetic aperture imaging with clusters of micro-satellites" [5]. Robots such as the Mars Curiosity can explore uninhabitable areas, while others may collect intelligence in war-zones. However, each of the applications listed above must have an underlying trajectory generation algorithm.

Researchers have studied various methodologies for path planning algorithms, and each has its benefits and disadvantages. Much of the research focuses on trajectory generation for nonholonomic robots as opposed to holonomic robots. In short, holonomic mobile robots have constraints that are integrable; therefore, any path that is allowed by the configuration space is also a feasible path. However, nonholonomic robots' constraints are nonintegrable. The main consequence of this is that the paths these robots can follow do not correspond to arbitrary paths in the configuration space. One attempt to remedy this situation is to plan a path using classical methods assuming a holonomic robot. This path may not be feasible for nonholonomic robots. However, feasible paths may be derived from the nonfeasible route [1].

Accordingly, a control method for car-like vehicles has been developed [2]. The vehicle's current geometry would be projected onto the desired trajectory such that its lateral dividing line was parallel to the tangent line of the path and the turning radius of the wheels would match the curvature of the path. This control strategy gave the car the ability to track curves with errors that converge to zero.

Another major research area focuses on multi-robot collaboration. With cooperating robots, teams may be able to accomplish tasks that are impossible individually. Such tasks include traversing inclined obstacles, forming world maps through image stitching, and data collection in physically separate locations. There have been attempts at a method in which there is no leader in the multi-robot strategy, which allows for tasks to be completed in the face of individual robot malfunction [5].

In addition to new areas of research, classical methods are being improved upon. For instance, the potential field method has been applied using new potential field functions [6]. These nontraditional functions allow mobile robots to reach goals that are near obstacles using the "mathematical elegance and simplicity" of the potential field method.

Two subjects under research form the building blocks of this thesis: Robot Operating Systems and Nonlinear Trajectory Generation. "A robot operating system (ROS) is a collection of programs which allow a user to easily control the mobile operations of a robot" [7]. Various ROS's have been developed, and the right choice can make mobile robotic research quick and efficient. Depending on the ROS, expensive robots and sensors can be simulated and/or physically controlled without the lower level knowledge of robotic hardware. An ROS simplifies the experimentation process, and it

can allow for faster code/compile/test cycles for methods such as Nonlinear Trajectory Generation (NTG). NTG is able to produce optimal trajectories numerically in real time for robots with dynamic constraints minimizing nonlinear cost functions [8]. Using the output spatial and temporal coordinates from NTG, a ROS may use a feedback controller to drive the mobile robot to those coordinates.

The previous situation is the main subject of this thesis. First, in Section II, a literature review is presented of current areas of research, such as control strategies and trajectory generation for mobile robots. Then, in Section III, the thesis will provide an overview of mobile robotic platforms. Specifically, the ER-1 robot will be described in detail. In Section IV, V, and VI, the software used in this thesis will be explained, including the ROS Player/Stage System and the NTG program.

The main purpose of this thesis is to present the results of using NTG as a trajectory generator in simulated and physical experiments in real time. To that end, the procedure to install the software and program the experiment will be described in detail in Section VII and the appendices. Finally, also in Section VII, the results will show that NTG is successful as a real time trajectory generator; Section VIII and IX will give future recommendations for this research to continue.

II. LITERATURE REVIEW

In order to familiarize oneself with current research in mobile robot control and trajectory generation, contemporary research articles should be studied. The following literature review discusses general topics pertaining to path planning and control. The most applicable articles will be reviewed in sections of this thesis covering Player/Stage and NTG.

The first article presents a novel function useful for potential field path planning [6]. Using the potential field method, an artificial potential field is simulated that will attract the robot to the goal. The goal emanates an attractive potential field, which is usually represented by

$$U_{att}(q) = .5 * \xi * \rho^m(q, q_{goal}) \quad (1)$$

where ξ is a positive scaling factor, ρ is the distance between the robot and the goal, *m* is the order of the field, and q is an (x,y) position. The attractive field is proportional to the ρ^m distance. Obstacles emanate repulsive potential fields, which typically take the form

$$U_{rep}(q) = \begin{cases} .5 * \eta * \left(\frac{1}{\rho(q,q_{obs})} - \frac{1}{\rho_0}\right)^2 & if\ \rho(q,q_{obs}) \leq \rho_0 \\ 0 & if\ \rho(q,q_{obs}) > \rho_0 \end{cases} \quad (2)$$

where η is a positive constant, ρ is the distance between the robot and the obstacle, and ρ_o is the threshold at which the obstacle's field loses affect. This field grows strong as the robot drives closer to the obstacle, and it decreases to zero at ρ_o as the distance increases. Each field projects a force in the direction of the negative gradient of the field. The

forces of each field are summed, and the resultant vector yields the direction the robot should travel.

While the fields generate path direction easily, there are a few disadvantages, including local minima traps, no passage between closely spaced obstacles, and oscillation between narrow passages and obstacles [6]. The problem that this article seeks to fix is "goals nonreachable with obstacles nearby," (GNRON). GNRON situations arise when a goal lies near an obstacle. For the potential field method to work, the global minimum of the sum of fields must be at the goal. However, with GNRON, the attractive force of the goal decreases, and the repulsive force of the obstacle increases such that the global minimum is not at the goal.

In order to solve this problem, the repulsive field is modified to decrease when the robot is near the goal.

$$U_{rep}(q) = \begin{cases} .5 * \eta * \left(\frac{1}{\rho(q,q_{obs})} - \frac{1}{\rho_0}\right)^2 * \rho^n(q, q_{goal}) & if\ \rho(q, q_{obs}) \leq \rho_0 \\ 0 & if\ \rho(q, q_{obs}) > \rho_0 \end{cases} \quad (3)$$

As long as n>0, a global minimum is guaranteed at the goal. However, a local minimum may appear near the goal so that the robot will stop at this point. To resolve this, proper choices of η and ξ will eliminate the local minimum due to a GNRON situation. As long as ξ/η is greater than

$$k_n' = \left(\frac{1}{p_m} - \frac{1}{p_0}\right) * \frac{(\rho_0 - r)^{n-1}}{\rho_0^2} \quad (4)$$

there will be no local minima. In the preceding equation, r is the distance between the robot and the obstacle and ρ_m is a simplifying term

$$\rho_m = \frac{2*r}{1-\frac{n}{2}+\sqrt{\left(1-\frac{n}{2}\right)^2+\frac{2*n*r}{\rho_0}}} \tag{5}$$

The results of simulating this updated repulsive equation were promising as can be seen in FIGURE 1 below. Even though, the GNRON problem is solved, there are still disadvantages. Local minima arising from other obstacles are still possible. Most importantly however, the easiest application of the potential field method is to an omnidirectional robot. In this case, the robot has no constraints and can follow any path generated by the method. However, a nonholonomic robot would not benefit directly from the generated path because there is no guarantee the path is feasible. For instance, a car-like robot can only move instantaneously in the direction that the wheels are pointing.

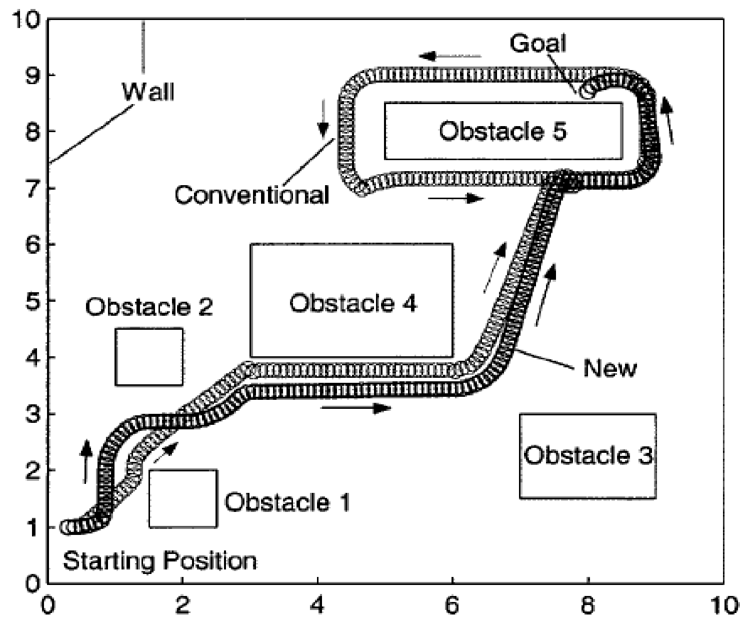


FIGURE 1 - Simulation Results of the Conventional and Updated Potential Field Method [6]

The second article discusses the Rapidly-Exploring Random Trees (RRT's) path planner and presents an improvement called the Execution Extended RRT (ERRT) [4]. RRT is a fast, continuous domain path planner that is designed to explore large state spaces efficiently. Its output is the basis for a probabilistically complete, but non-optimal, kinodynamic path planner.

A robot using the RRT method will begin a tree with a root at start and attempt to reach a goal through one of its branches. With probability p, it will extend a branch towards the goal within its configuration space that maintains its current kinematic constraints. With probability 1-p, it will extend a branch to a random state not necessarily towards the goal. In this way, it combines random exploration with a bias towards the goal. While planners such as potential field or motor schema do not have the ability to look ahead, the RRT method is able to scan potential states before it moves the robot through a branch.

The algorithm utilizes three general functions: Extend(), Distance(), and RandomState(). Extend() calculates the next state in which to move. If an obstacle blocks the movement, then another branch is calculated. Distance() uses the distance, time, or other parameter that the algorithm is trying to minimize to estimate the number of times Extend() would have to be called to reach the goal. Finally, RandomState() is called with probability 1-p and returns a state uniformly chosen from all possible states.

The improvement suggested in this article is the ERRT method. It adds a waypoint cache and adaptive cost penalty search to the RRT planner. With the time reduced by adding these revisions, the ERRT planner can be considered real time. When a state is chosen and planned, it is reasonable to search in the same area of states during

the next search cycle. This is due to the fact that the world situation would not change much in incremental steps. Each time a new state is chosen in the ERRT algorithm, the previous states are randomly placed in an array. With probability p, a target state is chosen towards the goal. With probability, r, a target state is chosen from the waypoint cache. Finally, with probability 1-p-r, a random state is chosen.

In addition, the adaptive cost penalty search allows the ERRT method to keep track of the distance between the root of the search tree and the target goal as opposed to only the distance between the current location and the target. A value, β, is initialized to 0. If a branch is extended, then β is incremented by .05, and if a branch is not found, then β is decremented by .05. As β increases between 0 and .65, more weight is placed on the distance between the root of the tree and the goal.

FIGURE 2 below shows two simulations run on custom software to show the ERRT method. The cached waypoints are modeled as black dots, and the best path is bolded.

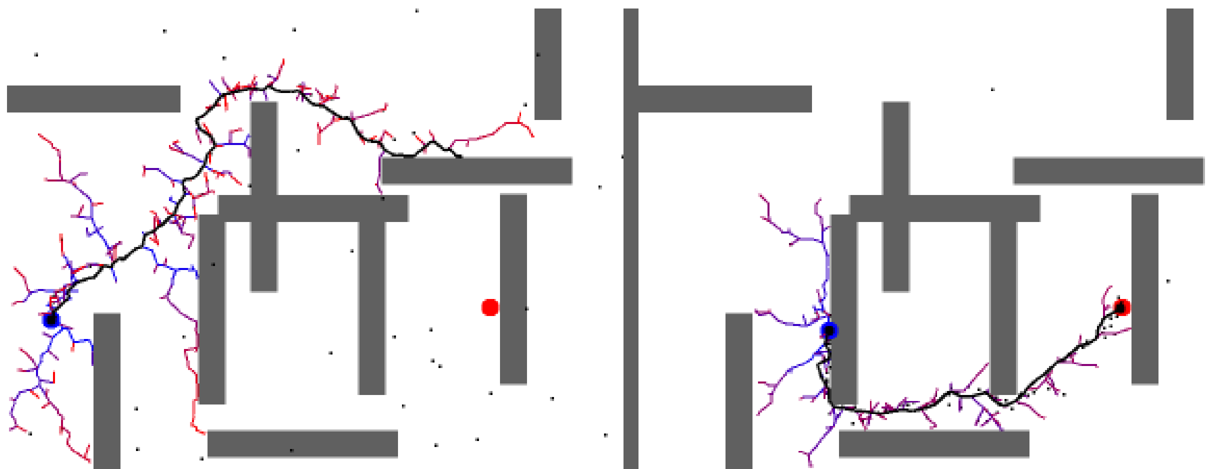


FIGURE 2 - Simulation Results of the ERRT Method [4]

The simulation results show that ERRT planner, minimizing time, is able to quickly plan paths, with knowledge of all obstacles at all times. Through experiment,

“the best combination of parameters that we were able to find, trading off physical performance and success with execution time was the following: 500 nodes, 200 waypoints, P[goal] = 0.1, P[waypoint] = 0.7, and a step size of 1/15sec” [4].

This work used the ERRT for holonomic robots. As used in the article, it ignores kinematic constraints until post processing can be done to modify the paths. In addition, if kinematics are not ignored, the paths chosen are even less optimal, especially if fast computation times are required. Finally, because the path tree is built with the knowledge of all obstacle locations (dynamic and static) in the world, the method can suffer from the frame problem. This issue, which increases computation time due to the amount of data in a world, has the potential to make the method slower than real time.

While the first two articles focus their published work on holonomic robots, the next article, co-authored by Richard Murray, attempts specifically to develop a nonholonomic path planner for a car-like robot [1]. Murray has produced plentiful and various articles, presentations, and software concerning mobile robotics, and his work on NTG is utilized heavily in this thesis.

When this article was published, most of the work of path planning had been completed for holonomic vehicles. Configuration spaces can be made for these robots based on the geometric constraints alone. Trajectory generation for a car-like robot adds the difficulty of a nonholonomic constraint and a curvature constraint (meaning that the robot has a minimal radius that it can track). As previously stated, nonholonomic constraints cannot be integrated; in addition, these constraints cannot be removed by reducing the configuration space. Therefore, a connected component within the

configuration space does not necessarily yield a path. The goal of this article is to produce an exact and fast path planner for a car-like robot.

In order for the presented algorithm to function, the robot must be locally controllable, and the shortest length paths must be characterized in terms of a car-like robot. The local controllability is tested by the Lie algebra rank condition (LARC). A system with r constraint equations and n derivatives can form the (n-r)-distribution Δ. "If the rank of LA(Δ) [Lie Algebra Delta distribution] is full at a given configuration c, then there exists a neighborhood of c, all of whose points are reachable by the system from c. In this case, the system is said to be locally controllable" [1]. If this condition is satisfied, then the existence of a trajectory is guaranteed, but the actual trajectory is not produced.

A car-like robot configuration studied in this article is shown below in FIGURE 3.

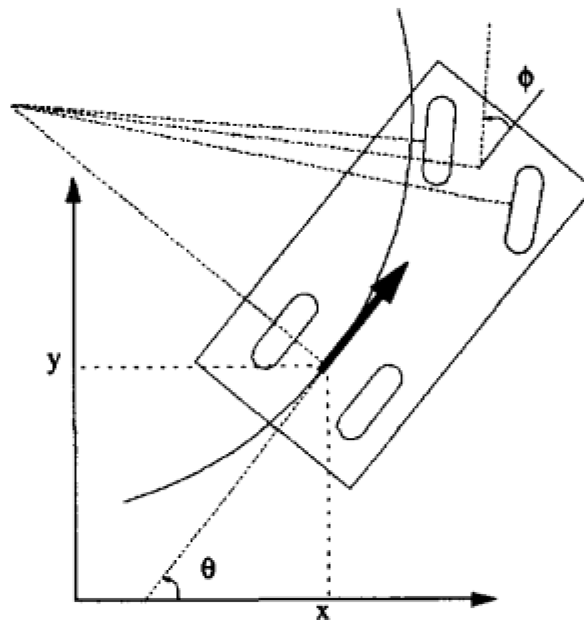


FIGURE 3 - Car-like Configuration with Parameters x, y, θ, ϕ, and *v* [1]

This yields the system

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} * v * \cos(\emptyset) + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} * v * \sin(\emptyset). \quad (6)$$

In addition, v is upper bounded and there is a maximum ϕ (minimum turning radius). The system and constraints can be analyzed using the LARC, and the car-like robot is found to be locally controllable.

In the absence of obstacles, the shortest length path for a car-like robot is made of arcs of minimal radius and straight lines. For robots that can move backwards and forwards, cusps can form; between two points, at most two cusps are necessary. The shortest length path possible between two points not separated by an obstacle is denoted d_{RS}. By utilizing the local controllability and the knowledge of shortest length paths, a corollary is formed as the basis of the algorithm: For the neighborhood N_1 of a robot's configuration, there exists another neighborhood N_2 such that for any configuration in N_2 the path between the two configurations is in N_1. This implies that if points from a holonomic path are connected in this way, then a car can travel the points of the path, too.

The algorithm is as follows. First, the minimum length path is determined from start to finish by a low level geometric planner that ignores constraints and obstacles. Second, if an obstacle is on the route, the path is divided in half. This is continued until all paths are free of obstacles and the endpoints of subpaths are concatenated. Finally, the paths that connect subpaths around obstacles are optimized using the minimal length path d_{RS}. In "big-O" notation, the algorithm runs on $O(\rho/\varepsilon^2)$, where ρ is the minimum turning radius and ε is the diameter containing the initial free space calculated by the geometric planner.

There is much manipulation of the initial path in this algorithm, so the path is stored using an array. Any changes to a subpath correspond to a replacement and/or modification of a section of the array. In order to detect collisions along the subpaths, the n robot vertices and the m obstacle vertices on each subpath are checked based on the fact that the paths are made of straight lines and arcs. These collision points provide locations in the array to subdivide. At these points the paths are locally optimized. While a global minimum path cannot be guaranteed, the individual subpaths are converted to minimal length paths based on the d_{RS} criteria.

The results of experimentation were promising. FIGURE 4 below shows the classic case of parallel parking. In all, the planning time took 2.5 seconds.

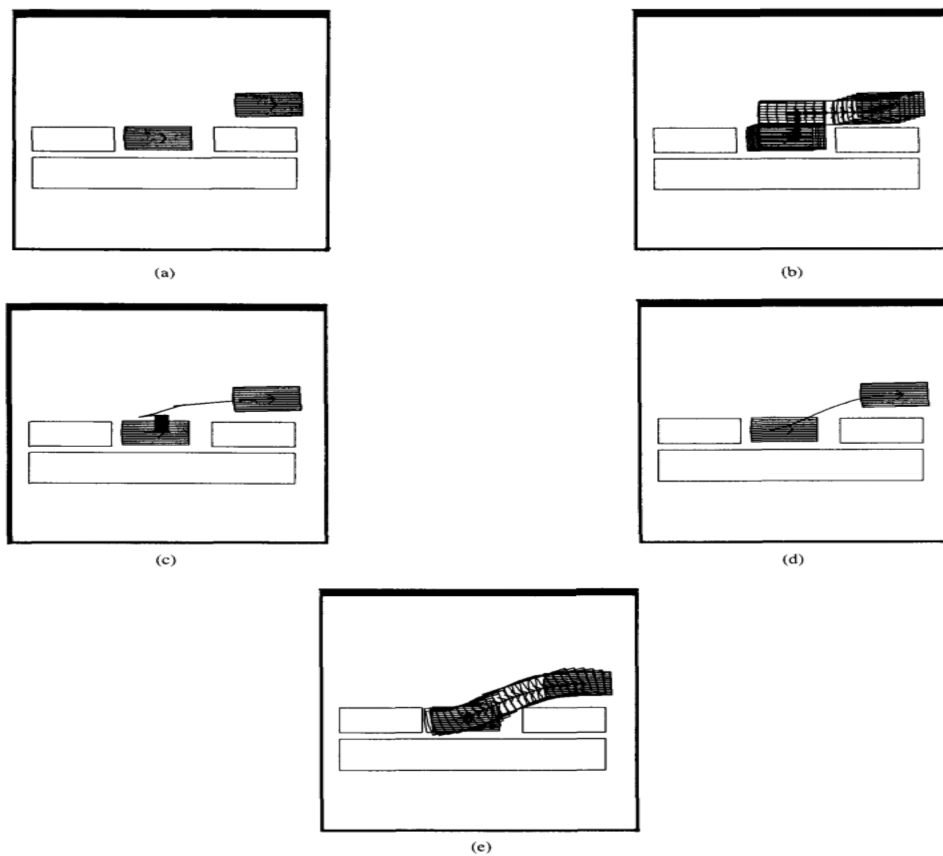


FIGURE 4 - Parallel Parking-(a) Initial and final destinations, (b) Geometric planner (.2s), (c) Nonholonomic path (.8s), (d) Optimized path (1.5s), (e) Traveled path [1]

A more complicated case is shown below in FIGURE 5. The total computation time was 38 seconds.

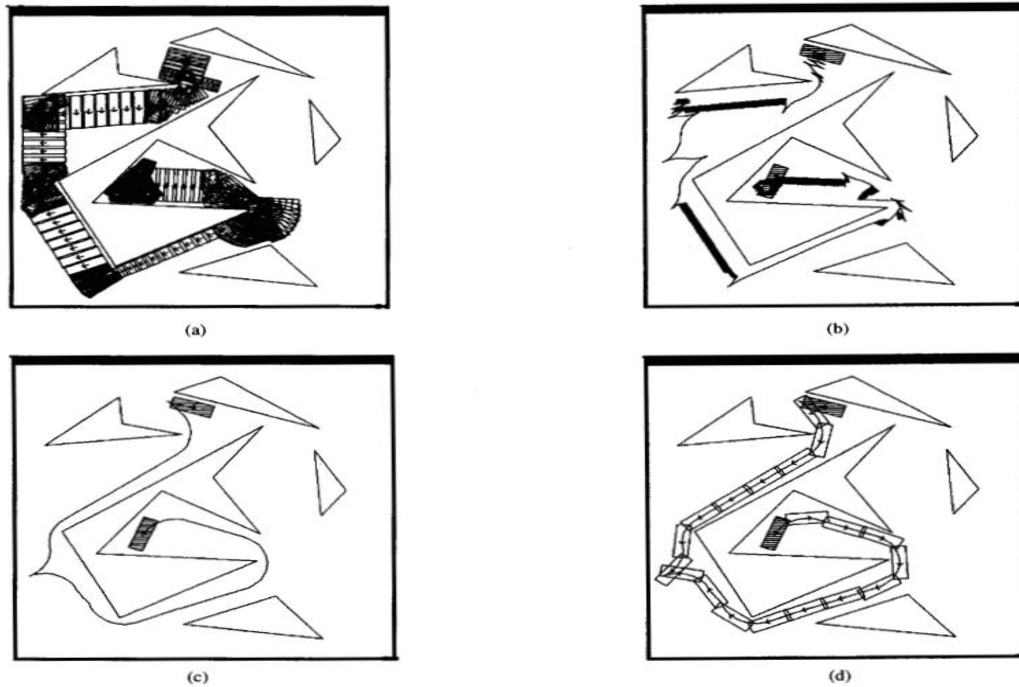


FIGURE 5 - Complicated Case-(a) Geometric planner (5s), (b) Nonholonomic path (23s), (c) Optimized path (10s), (d) Traveled path [1]

This work has potential to lead to path planning for other nonholonomic robots. Two problems are inherent, however. The first is that the entire path is planned at the beginning of the run, meaning that the world must not be dynamic. This, in addition to the possibility of the frame problem, increases the computation time.

While the previous article focused on trajectory generation, there have been other efforst to solve the other half of the problem: Controlling the robot to move about the desired trajectory [9]. The goal of the paper is: "Given a nonholonomic system, a feasible desired trajectory to follow, a known clearance between obstacles, and a measure

of accuracy of the sensors, find a control law which will stabilize the system to this path, avoiding the obstacles robustly in the face of disturbances." However, this work focuses only on finding a control law that smoothly and quickly converges to a given trajectory. The ability to integrate the sensor and spatial accuracy requirements was not completed numerically, but rather qualitatively.

At the time of this paper's publishing, most of the research in controlling nonholonomic robots focused on stabilization about a point. It has been shown that smooth static-feedback cannot do this. This work uses a linear time-varying feedback law based on robust stability to control a system such as the following:

$$\dot{x} = f(x) + g(x)u \tag{7}$$

$$x \in \boldsymbol{R}^n \ states$$

$$u \in \boldsymbol{R}^p \ inputs$$

$$x^0(t) \quad Desired\ Trajectory$$

$$u^0(t) \quad Nominal\ Inputs$$

The control law is developed, and proven to stabilize about a trajectory. The main step in this development is to linearize the system about the trajectory using a Taylor series and then ignore higher order terms. The following terms are defined:

$$A(t) := \left[\frac{\partial f}{\partial x}\left(x^0(t)\right) + \frac{\partial\left(gu^0(t)\right)}{\partial x}\left(x^0(t)\right)\right]$$

$$B(t) := g\left(x^0(t)\right)$$

$$H_c(t_0, t) = \int_{t_0}^{t} e^{6\alpha(t_0-\tau)}\Phi(t_0,\tau)B(\tau)B(\tau)^T\Phi(t_0,t)^T d\tau$$

$$P_c(t) \coloneqq H_c^{-1}(t, t+\delta)$$

where ϕ is the state transition matrix for A(t) and δ is chosen such that H_c is bounded away from singulairity. The following is the control law:

$$u = u^0 - \gamma(t) * B(t)^T * P_c(t) * (x - x^0) \tag{8}$$

This control law was applied to a differential drive and car-like robot. The axle of the differential drive robot, named Hilare, crossed through its center of mass. Its control inputs, $u=[u_1, u_2]^T$ are the linear and angular velocities of the robot. Its states, $x=[x_1, x_2, x_3]$, are the x-position, y-position, and orientation (θ), respectively. Hilare is modeled as follows:

$$\begin{aligned} \dot{x_1} &= \cos(x_3) * u_1 \\ \ddot{x_2} &= \sin(x_3) * u_1 \\ \dot{x_3} &= u_2 \end{aligned} \tag{9}$$

Using α=.1 and δ=1 and a desired control input of $u^0=[1,1]^T$ (corresponding to a circle), the control law was applied to Hilare. An initial error, (-.1, .2, .1) was given to Hilare. FIGURE 6 below shows the result of Hilare tracking this trajectory.

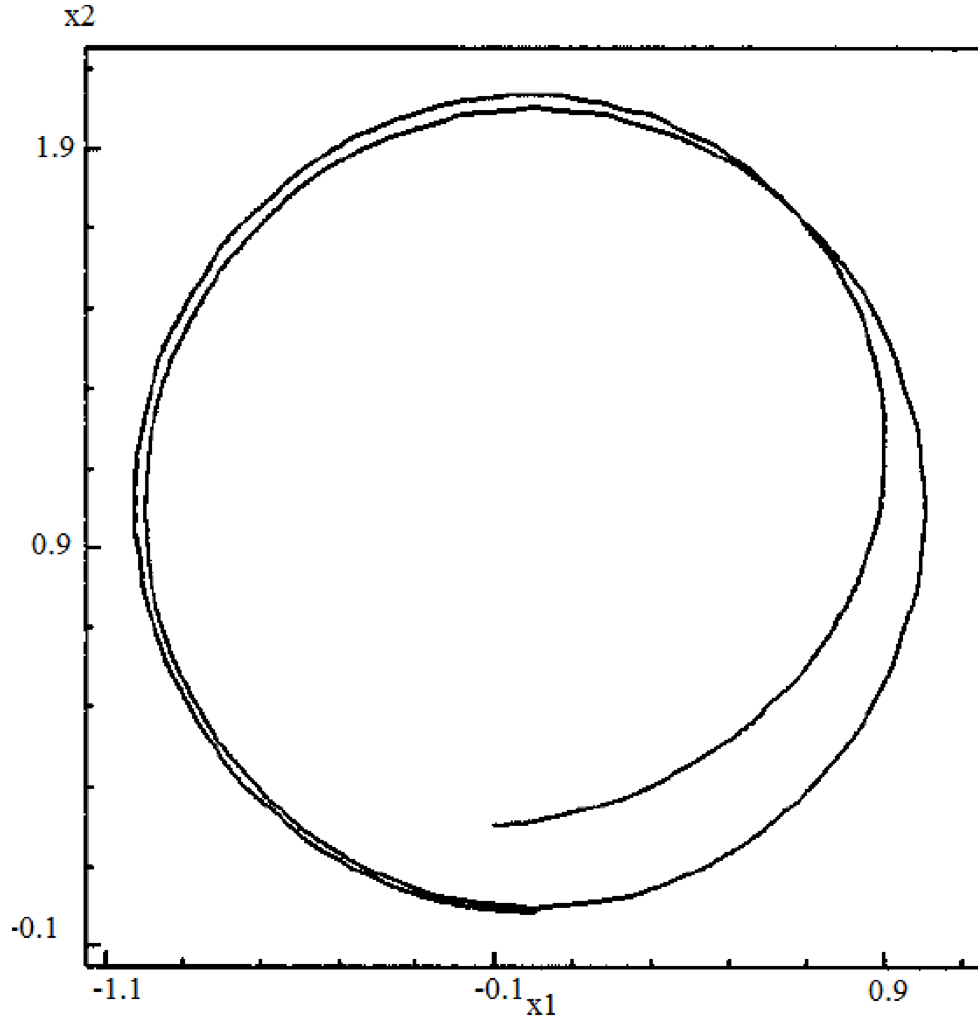


FIGURE 6 - Result of Control Law Applied to Hilare to Stabilize about a Circle [9]

These results show that Hilare smoothly and quickly converged to the circular path.

The car-like robot has a more complex system model. While u is the same, the states become x=[x_1, x_2, x_3, x_4], the x-position, y-position, angle of the front wheels from forwards, and the angle of the car from the x-axis, respectively. The system model is as follows:

$$\begin{aligned} \dot{x_1} &= \cos(x_3)\cos(x_4)\,u_1 \\ \dot{x_2} &= \cos(x_3)\sin(x_4)\,u_1 \\ \dot{x_3} &= u_2 \\ \dot{x_4} &= \frac{1}{L}\sin(x_3)\,u_1 \end{aligned} \tag{10}$$

Using α=.1, δ=1, and $u^0=[\sin(t), \cos(t)]^T$ (corresponding to a parallel parking maneuver), the control law was applied to the car-like robot. An initial error, (.1, -.1, .05, .2), was given to the robot. FIGURE 7 below shows the tracking of the parallel parking maneuver.

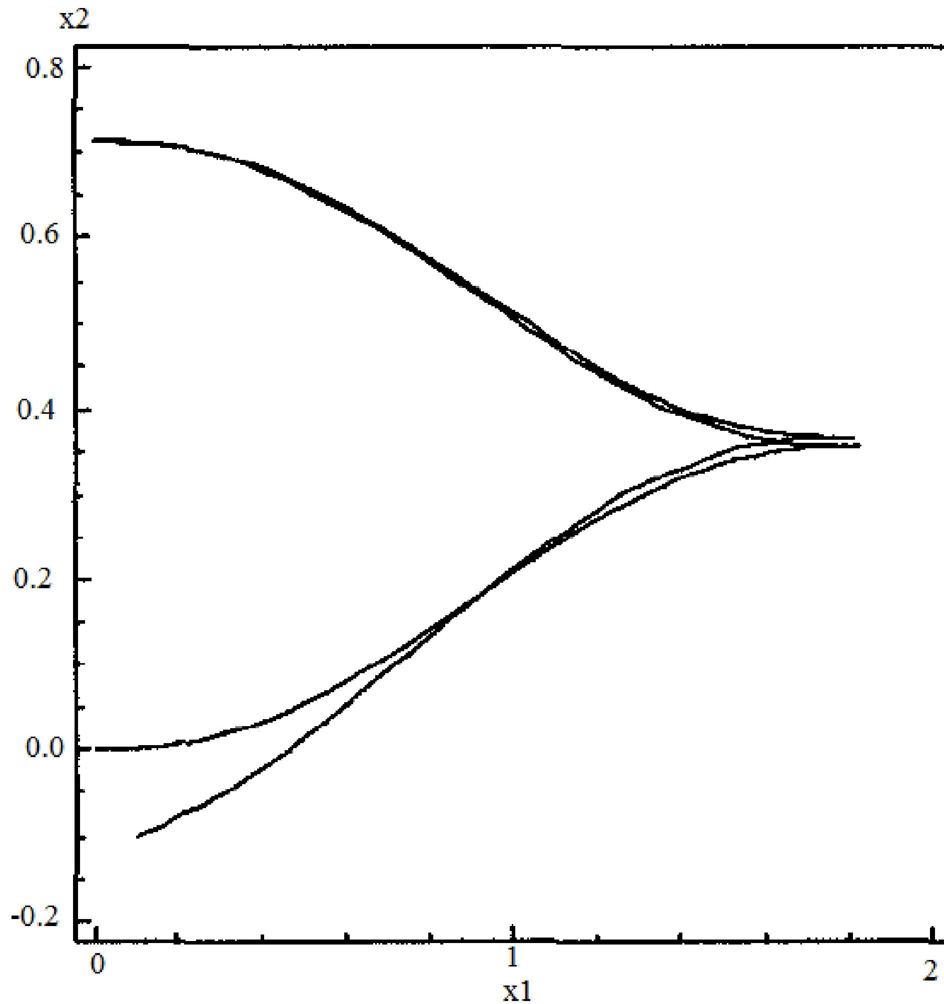


FIGURE 7 - Result of Control Law Applied to the Car-like Robot to Parallel Park [9]

The results show that, as long as a trajectory can be generated, there is a way to control the robot to follow the path. In addition, if α is increased, then faster convergence can be achieved. This method is able to converge despite disturbances and sensor inaccuracies.

Because this work focuses on stabilizing about a trajectory, the ability to create trajectories becomes the problem to be solved.

III. ER-1 ROBOT REVIEW

Each form of research tends to focus on a specific type of mobile robot. The vast majority of path planning research has been carried out with omnidirectional robots. Omnidirectional robots may travel in any direction instantly; thus, an omnidirectional robot is a holonomic robot. The implications on research are that an omnidirectional robot may move from one point to another without rotating first, and it may rotate while it is moving so that its orientation is correct when arriving at its destination [10]. This ability allows for an easier implementation of path generation algorithms because the algorithm can strictly focus on the minimization parameter and not the constraints imposed by the robot.

While holonomic robots are useful for experimentation on more theoretical or proof-of-concept algorithms, nonholonomic mobile robots are useful for experimentation that will apply practically. Cars, motorcycles, tricycles, and differential drive robots are all nonholonomic, so any trajectory generation algorithm that is to apply to these vehicles needs to take into account their nonholonomic nature. Nonholonomic mobile robots have differential degrees of freedom (DDOF) that are less than three [11]; these robots have a DDOF of three, minus the number of sliding constraints. This sliding constraint makes it impossible for nonholonomic robots to move sideways (without slipping), and, as stated before, feasible paths must take this constraint into account.

This thesis relies in part on the previous work and material procured by Dr. Tamer Inanc, Travis Riggs, and Derek Thomas. Two differential drive robots considered were the ER-1 robot and the Pioneer P3-DX robot [12]. The Pioneer P3-DX is manufactured by Adept Mobile Robots. This fully programmable robot is known for reliability,

versatility, and durability, and it is fully supported on the Player program. It is purchased fully assembled with motors with 500-tick encoders, providing odometry measurements. The P3-DX also has 8 forward-facing sonar sensors, and it is powered by three, hot-swappable batteries. An internal computer or laptop must be supplied for the robot's computation engine. This robot is one of the most popular research robots in the world.

The ER-1 is an out-of-production robot manufactured by Evolution Robotics, now owned by iRobot. The robot's base package was advertised to provide easy and quick behavioral programming based on "if-then" statements involving images, sounds, e-mails, and sensors. While much of the institutional research has been focused on human robot interaction, compatibility has been built (but not necessarily maintained) for programs such as Player and Robotics4.net.

FIGURE 8 below shows a typical setup of the ER-1. The base package comes with an iREZ Kritter Cam (320x240 pixels, 30 fps), a frame with two stepper motorized wheels and a front castor wheel, and aluminum core X-beams for customizing. Similarly to the P3-DX, a laptop must be supplied as the robot "brain."

FIGURE 8 - Typical ER-1 Robot Build

The laptop communicates to the Robot Control Module via USB to Serial interface. The Robot Control Module (RCM) controls the stepper motors. It readily accepts commands of linear and angular velocity. The RCM and motors are powered by a 12V rechargeable battery pack. In addition to the base package, and IR sensor pack and gripper arm are also available. Three robots were purchased with the extra IR sensors and gripper arms, and these robots have been controlled in Player 1.6.5 to test a robotic test-bed [13]. The availability of these robots is what caused the ER-1 to be chosen in this thesis rather than the Pioneer P3-DX.

IV. REVIEW ON ROBOT OPERATING SYSTEMS

To aid in the research of mobile robotics, robot operating systems (ROS's) have been developed. An ROS is a program or set of programs that allow for easy control of mobile robot functions. An ROS is to a robot as an operating system (OS) is to computer memory and computation. It should have a user-friendly environment that manages the "low-level details." In addition, code that is used to control robots should be written in a general language that does not make the code obsolete when there are (supported) hardware changes.

Sixteen various ROS's have been compared using four main criteria: ease of use, capability, adaptability, and ease of installation and maintenance [7]. A robot operating system should provide functions that allow basic movement functions without worrying the user with lower level code. In addition, the program should be well documented, and the documentation ought to provide examples. The ease of use should not affect the capability, however. The ROS must have the ability to simulate and control physical actuators, sensors, and robots. In addition, this control code should be easily transferrable from one robot or sensor to another, in any software language. Finally, the installation time must be kept to a minimum so that research is not hampered.

Microsoft Robotics Developer Studio (RDS) and Player/Stage scored highest [7]. In addition, the Willow Garage Robot Operating System (WG-ROS) was considered because it seemed to be the most widely used. RDS provides a simple user interface with various and meticulous tutorials. It can control eight different robots, and it can simulate this control on Visual Simulation Environment (VSE). Player/Stage on the other hand is not as simple, but its documentation is more copious and in depth. There are fewer

tutorials, but it includes a more comprehensive function library. In addition, it can control and simulate thirteen different robots. WG-ROS uses the same simulator as Player/Stage, but it can control sixty different robots. The largest disadvantage of WG-ROS is its complexity. The decision factor between these three systems was academic representation: since Player/Stage was the most widely cited in academia, it was chosen as the best ROS for a college student.

V. DESCRIPTION OF PLAYER/STAGE

The Player/Stage software system, originally developed at CalTech by Richard Vaughan et. al., is a useful way to solve the issues of the cost of research and the knowledge necessary for low level control of actuators and sensors. Using Stage, virtually any number of robots can be simulated with any number of sensors, and, using Player, these simulated robots can be controlled through simple commands and interfaces.

One of the obstacles facing robotics researchers is the cost of mobile robots and sensors. The Pioneer P3-DX is base-listed at $4918.03. This price is without many of the sensors that could be necessary, such as range finders or gyroscopes. 3-axis gyroscopes range around $79.95; a 1mm temperature compensated sonar could cost $109.95, or a Hokuyo R311 laser-range finder could cost $3950.00. Procuring the initial research material would therefore be difficult with a modest budget. In addition, once the materials are bought, the researcher must have the knowledge of how to use them with the robot. This could include creating customized drivers, designing software package interfaces, and learning how to control each device within a single program of control code. Finally, if multi-robot cooperation is desired, a knowledge of wireless communications may be needed, which adds another level of complexity.

In 1998 and 1999, at the California Institute of Technology, many contributors under the leadership of B. P. Gerkey, R. T. Vaughan, and A. Howard created two programs named Player and Stage to be run in the Linux environment. The programs were designed to aid in the research of mobile robotics. Stage is a virtual robot simulator. In Stage, a virtual world with virtual objects is filled with almost any number of virtual

robots and sensors that can interact with the world and its objects. Using Stage, the Pioneer P3-DX can be simulated with any number of sensors at no cost. Rather than learning each new sensor, Stage provides models of robot sensors; to create a sensor, the user must only make an instance of that model. These models include rangers (simulating sonars, lasers, and IR sensors), bumpers, position control, color-blob tracking, grippers, and WIFI [14]. In addition, unique models can be created and instanced any number of times. For example, a "banana peel" [15] model could be created by using a picture in the shape of a banana, scaling the size, and making the color of the banana yellow. Then these banana peels could be placed in various locations throughout the world.

Stage is a powerful tool, but it is computationally cheap. In the initial paper about Player and Stage, R. T. Vaughan states that Stage is written under a "Good-Enough Fidelity" [16]. The "Good-Enough Fidelity" philosophy implies that each of the models in Stage provides an estimate that is good-enough to the real-world device, but not exactly like it. In addition, the computational power increases linearly with the number of devices being simulated [16]. Because of this computationally cheap model, Stage can be run on personal computers. Although this requires control code to be written robustly, this robust code aligns with good practices of programming for real-world robots [16].

Stage utilizes two types of files, world and include files [15]. World files contain the virtual world and everything that is in it, including robots, sensors, and objects. Include files contain robots, sensors, and objects, and they can be included in the world file. Whatever is instantiated in the include file can then be used and possibly customized

in the world file. Using include files allows the user to refrain from retyping code multiple times.

Directly before Stage was developed, Player was created. “Player is a network server for robot control. It provides a clean and simple interface to the robot's sensors and actuators over the IP network” [17]. Each robot subscribes to Player as a client, and Player can control the client as a server. Player is most useful because if a driver is written for a device, then the driver can be linked to Player; then, Player abstracts each driver into device proxies so that common commands can be used for each type of interface. For example, any IR sensor can be controlled via common commands, such as “GetRange().” Player takes the control code and translates it into a command that each individual driver can use. Furthermore, Player provides copious pre-compiled drivers, such as various sonars, lasers, rangers, cameras, and motor drives. Player can be used to control any number of robot clients with actuators and sensors using any programming language that supports TCP sockets [18].

The process of Player controlling a robot is as follows. Hardware devices have drivers that Player can access. A robot containing these devices subscribes to Player. Proxies are formed, one for each device. These proxies transmit information and commands between the drivers for the devices and Player. Any device connected to Player has access to the information and commands of any other device. Therefore, multiple robots can interact freely as long as they are connected to Player. A block diagram of the Player system is presented below in FIGURE 9.

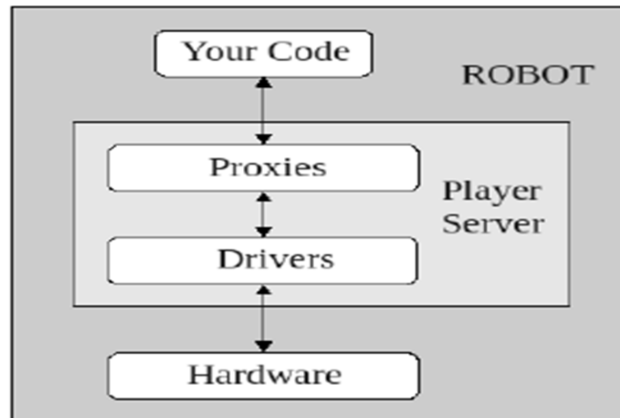


FIGURE 9 - Player System Block Diagram [15]

Player uses configuration files [15]. Each configuration file contains a driver (separate from the individual hardware driver) that contains information like device name, hardware driver plugin name, what interface the driver provides (such as camera, position2d, ranger), what interface the driver needs, and what the address of the device is.

Stage can provide simulated robots and drivers, and Player can control robots with hardware drivers linked to them. This immediately gives rise to the Player/Stage system, in which Player can control Stage simulated robots by using Stage as a plugin for Player. A block diagram of the Player/Stage system can be seen below in FIGURE 10.

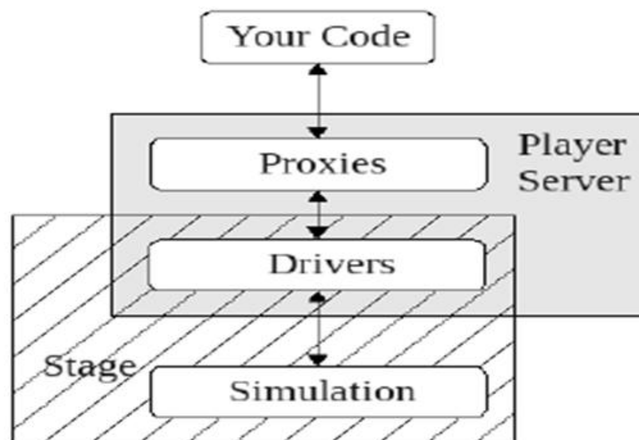


FIGURE 10- The Player/Stage System Block Diagram [15]

All that needs to be changed for a simulation of Player/Stage is to add Stage as a plugin in the configuration file.

The uses of Player/Stage are numerous. One can create any type of robot, and any number of robots, with any number of robot sensors in Stage and situate these objects in a virtual world. Player can then control these robots and sensors and communicate among all of the clients. Best of all, simulations can be run again and again, making changes as necessary, at no cost. Finally, if a robot is created in Stage that is modeled after an actual robot, the same control code that controlled the simulated robot can control the actual robot as well. These benefits have made Player/Stage a common robotics research tool.

VI. REVIEW OF NONLINEAR TRAJECTORY GENERATION (NTG)

The program called Nonlinear Trajectory Generation (NTG) was developed by M. Milam, K. Mushambi, and R. Murray at Caltech [8], [19]. Its goal is to provide a trajectory that is feasible for a nonholonomic vehicle to follow. Then, a feedback controller can be used to drive the vehicle along that trajectory. This scheme, called two degree of freedom control, can be seen below in FIGURE 11.

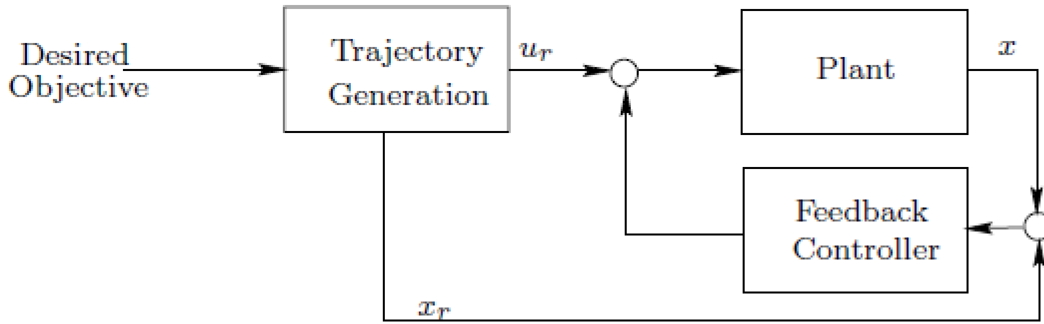


FIGURE 11 - Two Degree of Freedom Control [8]

The question of solving for the trajectory is a nonlinear optimal control problem. Using classical methods, solving this problem is too slow for real time use. Rather than solving for the entire trajectory, the path is split into intervals, p, and each interval is approximated by lower order polynomials. These polynomials are represented by B-splines,

$$z_1 = \sum_{i=1}^{l} B_{i,k}(t) * C_i \qquad (11)$$

where z_1 is the variable being approximated, B is the B-Spline basis function with polynomial order k, l is the number of coefficients, and C are the coefficients of the

polynomials. The coefficients of the B-spline functions are determined using the sequential quadratic program NPSOL. In addition, the B-spline derivatives will match at the knot points based on some smoothness. Furthermore, if the B-splines are calculated once, and then the output coefficients are used as initial guesses for a second B-spline calculation, then the second calculation is very quick. This method is called a “warm start.”

The problem statement is usually given as follows. Given a system (12) and initial, trajectory, and final cost functions (13) and constraints subject to boundary conditions (14), make a trajectory to minimize the cost function.

$$\dot{x} = f(x, u) \tag{12}$$

$$\varphi_o(t_i) + \int_{t_i}^{t_f} L(x, u)dt + \varphi_f(t_f) \tag{13}$$

$$\begin{gathered} lb_0 < \emptyset_0(x(t_i), u(t_i)) < ub_0 \\ lb_t < \emptyset_t(x(t), u(t)) < ub_t \\ lb_f < \emptyset_f\big(x(t_f), u(t_f)\big) < ub_f \end{gathered} \tag{14}$$

In general, the variables should be written in terms of the least number differentially flat outputs necessary to describe the system. However, this only aids in the speed of the calculation, not the ability of NTG to function. This translates into the vector variables x and u should be written in terms of $z=z_1, z_2, \ldots, z_q$.

There are both advantages and disadvantages of NTG [20]. It is very fast, especially under a warm start, it converges well, and it handles all nonholonomic constraints. This means it can generate meaningful trajectories for all mobile robots in

real-time. The disadvantages include that there is no convergence proof, and the constraints can only be guaranteed to be followed at each discretization point within each interval. This means that there is an assumption that the robot can move from one point to another within the generated trajectory feasibly. This is not an outlandish assumption, especially if the points are generated closely together.

VII. THESIS PROCEDURE

A. Installation

The previous research provoked the thesis project to use the Player/Stage system as an ROS, the ER-1 as the test robot, and NTG as the trajectory generator. The goal was to produce a system that provides a trajectory to a (nonholonomic) differential drive robot and drives the robot through the trajectory by combining Player/Stage and NTG in real time and using a receding horizon approach.

Since Player/Stage was developed for Linux, the free Mageia distribution of Linux was chosen. For most of the thesis, a 64-bit architecture was used, but a 32-bit architecture was also used equally successfully. However, the Ubuntu distribution provides much more documentation, aid, and tutorials for beginning users of Linux. The newest version of Ubuntu, 12.10 LTS, was originally installed for the thesis, but it was later found (after about five weeks of attempts) that this newest version was not compatible with both Player and Stage. The libraries for the Boost package could not be found in Ubuntu. Mageia, however, allowed for a quick (approximately three days) installation of Player/Stage.

Installing Player and Stage is non-trivial. Installation steps can be found in Appendix I. The instructions for installing NTG can also be found in the same Appendix.

B. ER-1 Preparation

At the beginning of this thesis, one complete ER-1 system with a 12VDC lead acid battery and power module, Robot Control Module (RCM), robot frame, gripper arm, digital camera, and a Dell Latitude laptop with the Robot Control Center (RCC) installed, was available for use. The RCC allows for simple behavioral programming of the ER-1

robot [21]. These behaviors are built on the reactive paradigm of robotic programming. A behavior is formed as such: If some combination of conditions occurs, then perform an action. "If conditions" include viewing a color, object, or motion, hearing a phrase or a sound level, sensing with an IR sensor, and acting at a certain time. Actions include moving, playing a sound, sending a message, or using the gripper arm.

While the RCC GUI is powerfully simple, an Application Programming Interface (API) is also provided. The API allows for Command Line arguments to be passed to the ER-1 robot either through its local laptop or through another laptop over the internet. Commands such as move, move rotate, set velocity, set angular velocity, close gripper arm, etc are available which give the user more access to lower level commands. In addition, Java Scripts can be written and compiled for the ER-1. Although the RCC gives complete access to the ER-1's sensors and motors, and it uses the powerful ERSP vision system for object recognition, it is useful only for the ER-1 robot with a 32-bit laptop, and it is expensive. Because of this, control of the robot was to be carried out with Player, which could control more types of robots and be written in more software languages.

Before the ER-1 could be used, however, it had to be able to move. The robot was seven years old at the inception of the thesis, and it had not moved in at least two years. While the RCC could connect to the RCM while the battery was charging, the RCC could not control the wheel motors. Through troubleshooting and contacting Deltran Battery Tender, the battery's manufacturer, it was found that the lead acid batteries had been completely depleted and needed to be replaced. The battery charger supplied enough current (750mA) to the RCM to allow communication to the RCC, but it

could not supply the current required by the stepper motors of the wheels. Upon battery replacement, the ER-1 moved easily, and the thesis could begin to move onto Player/Stage.

C. Player/Stage Preparation

The Player/Stage system is, at first, a daunting program. In order to become familiar with it, one must practice with each aspect of the system. The tutorial on how to use Player and Stage 3.2.X by Jenny Owen [15] of York, UK is useful to learn the basics of the system. The process of this thesis went through the tutorial and developed other programs in order to prepare to use Player/Stage with NTG.

According to the two-degree of freedom design depicted in FIGURE 11 above, the position of the robot must be measured in order to give feedback to the system. This can be measured through internal odometry, which is highly inaccurate, range sensors, cameras, or GPS systems. For this reason, most of the preparatory work on with Player and Stage was completed using range sensors, cameras, and blobfinders.

1. Wall Following with Player/Stage

The first program is an implementation of a wall following behavior. Using a robot with a range sensor pointed to its right, the robot can follow a wall on its right. The control program uses a three state control algorithm: If the robot is too far away from the wall, veer to the right. If the robot is too close to the wall, veer to the left. If it is within a threshold, continue straight. The Stage simulation window can be seen below in FIGURE 12. The grey box with the nose is the simulated robot, the small sliver of grey to its right is the range sensor, and the gray trail behind it is its previous path.

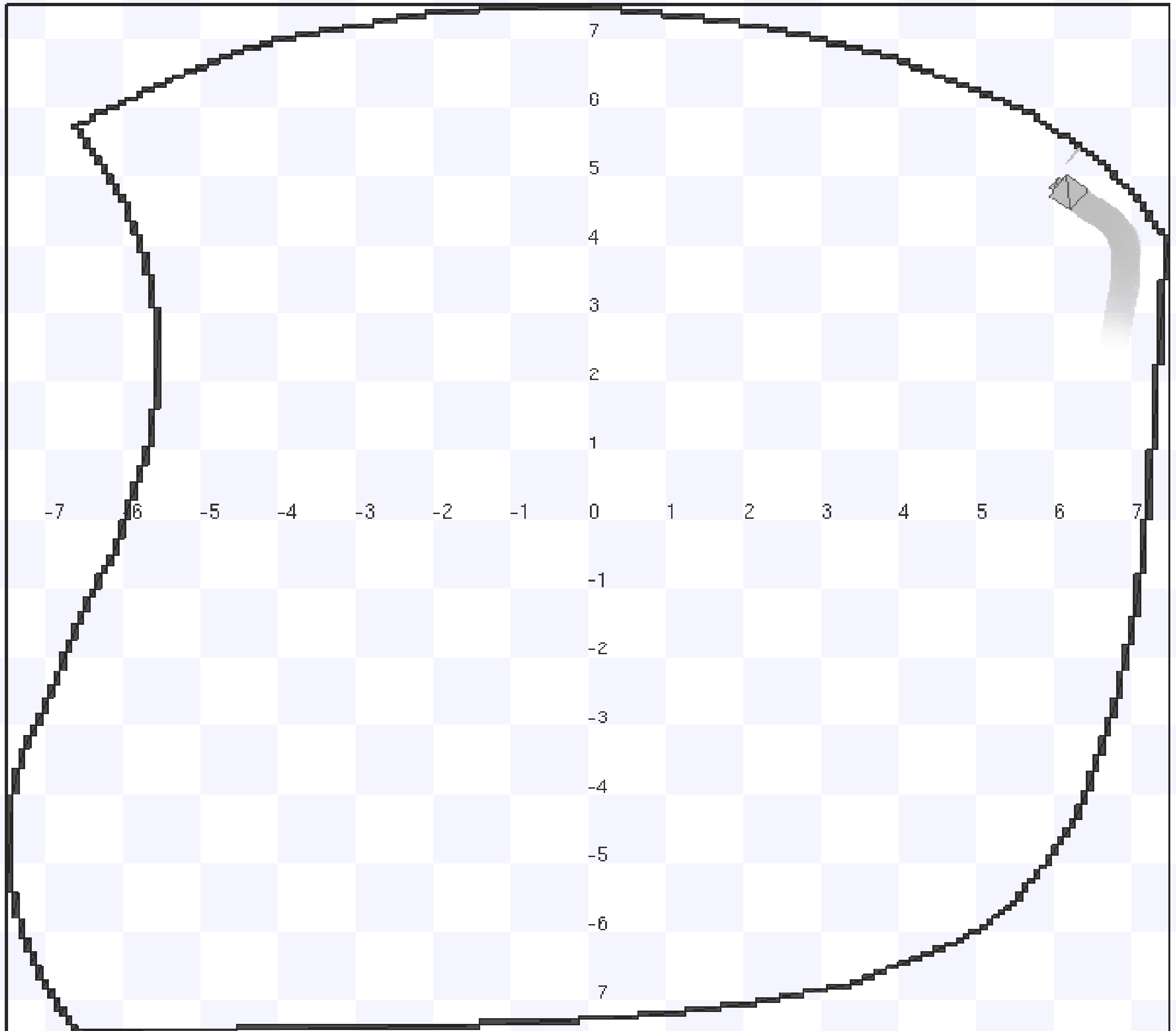


FIGURE 12 - Wall Following using Player/Stage

The code for this group of programs can be viewed in Appendix II. This algorithm set a constant forward speed of .1 m/s. The threshold used to determine if the robot is too far or too close to a wall is 3±.055 m.

2. <u>Blobfinding with Player/Stage</u>

While previous program shows the proof of concept of determining distances to obstacles through range sensing, the next program shows how the exact location of obstacles can be determined from blobfinders. A blobfinder is an algorithm that

determines the location of a color blob in an image. Stage allows the user to skip the camera image to blob conversion that would normally have to be computed and directly detect blobs. A blobfinder will detect any of the colors that the user specifies. In this program, the robot has a blobfinder that is set to find grey, green, and purple blobs. Once a blob is found, it will display the number of blobs it finds, and the location and bounding box of the first blob in the set. It should be noted that the source code of Stage had to be modified in order for the blobfinder to work. In p_blobfinder.cc, a patch had to be implemented as can be found in [22].

The world is set up initially with a purple robot facing a green robot in a cave-like world. Once the control code is run, the purple robot sees three blobs-two gray blobs and one green blob representing the walls and the other robot. This can be seen below in FIGURE 13.

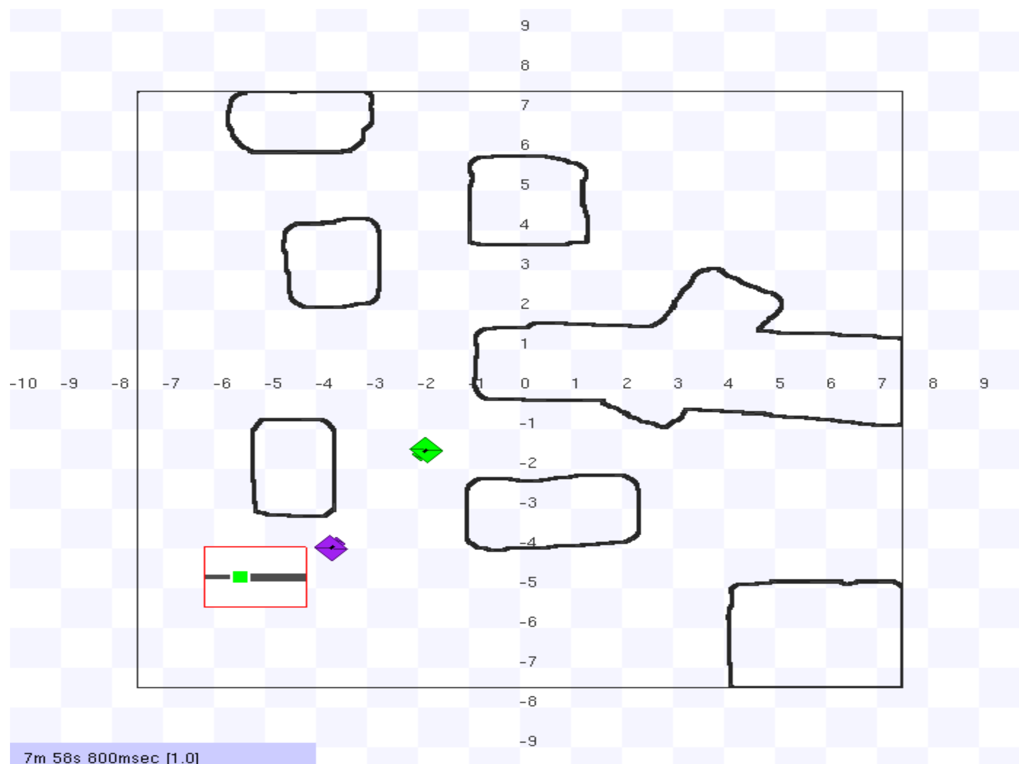


FIGURE 13 - Robot with Blobfinder in Stage Simulation

The output of the control code can be seen below in FIGURE 14. There are three blobs in the purple robot's field of view, and the blob that is being reported on is the right-most grey blob. Its centroid location is at the point (57, 30) as referenced by the purple robot. The bounding box positions show the blob to have an area of 344 pixels at a range of 4.233362 meters.

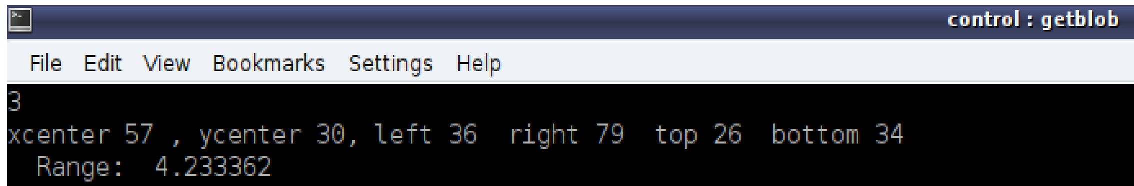


FIGURE 14 - getblob.cc Terminal Output

The uses of this type of program are clear: Objects can be detected and the information gathered can be used to avoid it. Furthermore, these objects can be dynamic in that no prior knowledge is necessary for blobfinders to function (other than the color of the object). The code for this group of programs can be seen in Appendix II.

3. Position Control with Player/Stage

The next program uses a built in Player virtual driver called "Go To." This driver takes in a point in terms of (x,y,θ), and it moves the robot to that point [18]. Given a differential drive robot, it will simply rotate the robot towards the point, drive in a straight line to the point, and then rotate to the desired pose. While this is good enough in some applications, the movements are not smooth and would not be easily transferrable to planes, bicycles or other vehicles that cannot hover in place. FIGURE 15 below shows how a differential drive robot is moved from the point (0,0,0°) to the point (3,3, 135°) within a circular arena.

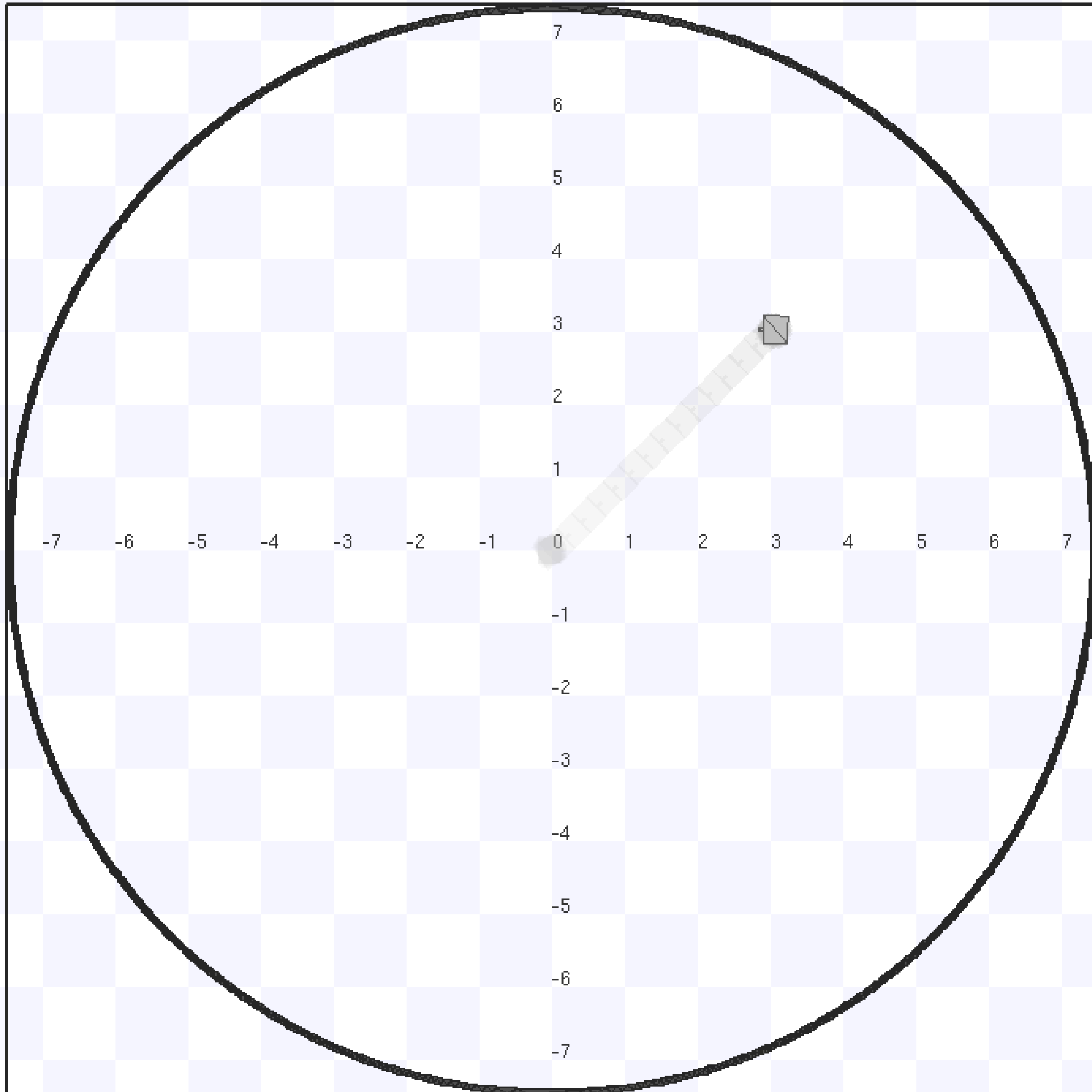


FIGURE 15 - Using Player's "Go To" Driver to Move from (0,0,0°) to (3,3,135°)

Rather than using this driver, it would be better to use a proportional feedback controller that forces a smooth path for a differential drive robot. According to [23], the kinematic model for a differential drive robot is:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{15}$$

The components of motion can be seen below in FIGURE 16.

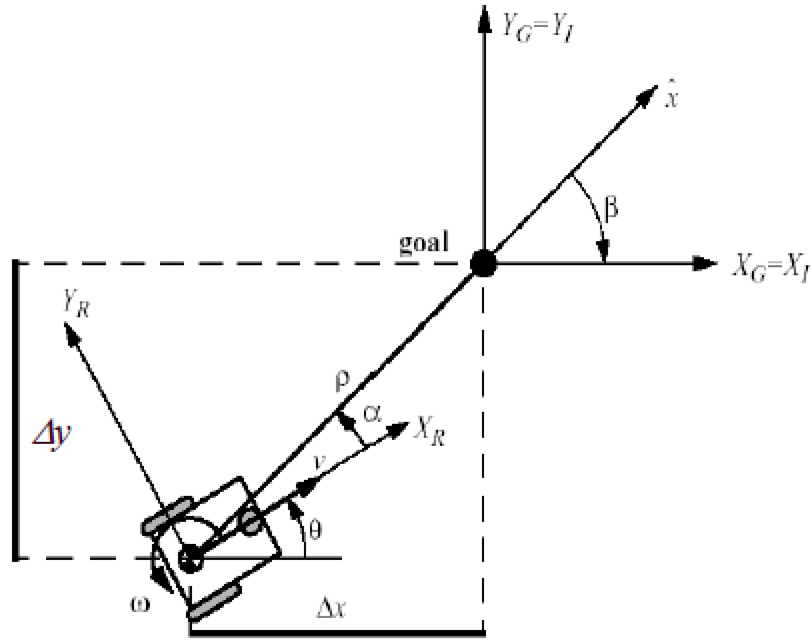


FIGURE 16 - Differential Drive Components of Motion [23]

Using the coordinate transformations,

$$\rho = \sqrt{\Delta x^2 + \Delta y^2} \tag{16}$$

$$\alpha = -\theta + \arctan\left(\frac{\Delta y}{\Delta \mathrm{x}}\right) \tag{17}$$

$$\beta = -\theta - \alpha, \tag{18}$$

the new constraints are

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} -\cos\alpha & 0 \\ \frac{\sin\alpha}{\rho} & -1 \\ \frac{-\sin\alpha}{\rho} & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{19}$$

if

$$v = k_\rho * \rho_{err} \tag{20}$$

$$\omega = k_\alpha * \alpha_{err} + k_\beta * \beta_{err} \,. \tag{21}$$

Then the system is locally exponentially stable, and the error will be driven to zero.

Using these equations, a program that drives a differential drive robot to a point (x,y,θ) was written. The simulation result of driving the robot to the point (3,3,135°) is shown below in FIGURE 17.

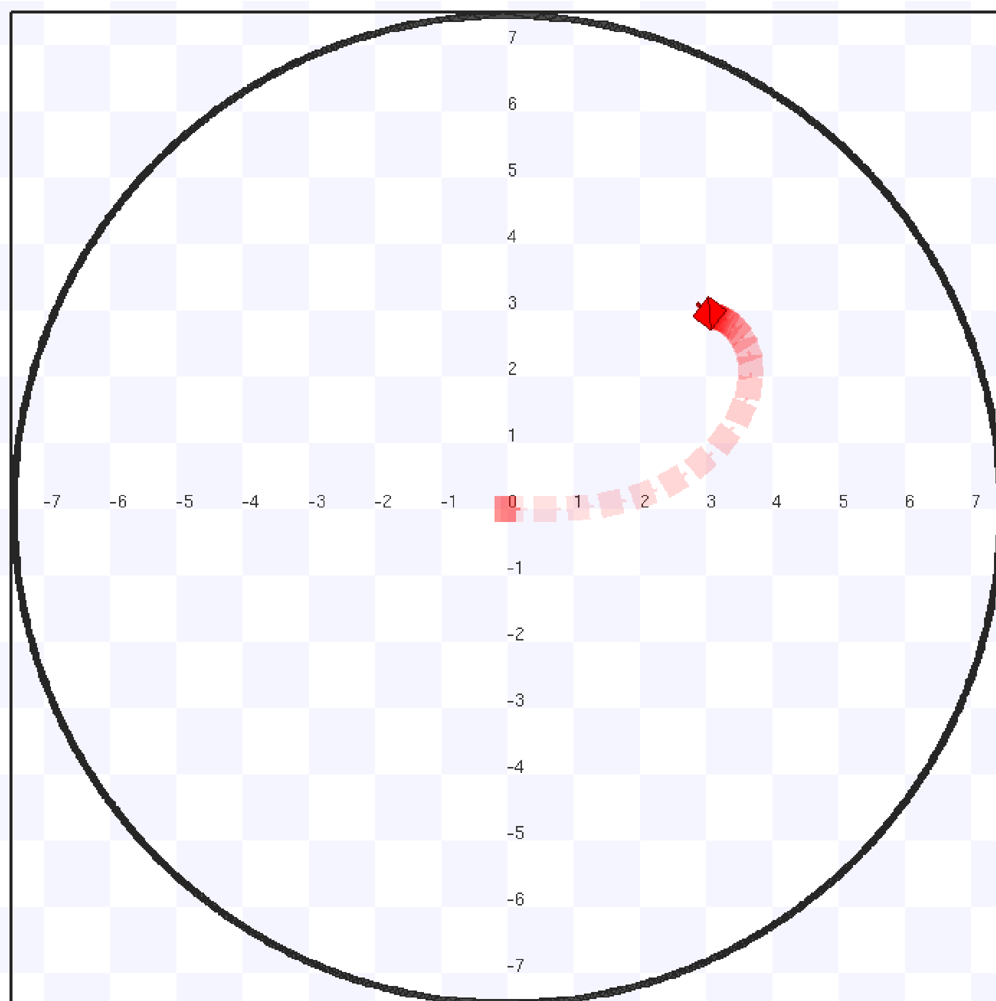


FIGURE 17 - Using a Proportional Controller to Provide a Smooth Path to a Goal Point

The robot must approach the goal point at the final goal angle. Therefore, rather than going in a straight line, it must overshoot the point in the x direction and curve up. This is more desirable than the Player driver "Go To." This control model takes into account more constraints than is necessary. A differential drive robot can move in place by spinning its wheels at equal speeds in opposite directions. However, the model shows that, given a path to follow that is feasible for the differential drive robot, it can follow the path using a method that is considerate of the robot's constraints. The code for this group of programs can be seen in Appendix II.

4. <u>Blobfinding with Player</u>

The next group of programs this thesis focuses on is the code using Player to interface with hardware rather than Stage. The first program uses a built-in HP Pavilion webcam and an algorithm to detect black blobs. To do this, two Player drivers must be instantiated: The physical driver "cvcam" and the virtual driver "CMVision" [18]. The driver "cvcam" uses the OpenCV framework to capture images using cameras like webcams that OpenCV can utilize. While "cvcam" works with a physical device, "CMVision" works with an algorithm to act as a blobfinder. "CMVision (Color Machine Vision) is a fast color-segmentation software library…The CMVision driver provides a stream of camera images to the CMVision code and assembles the resulting blob information into Player's data format" [18]. To determine what color blobs CMVision attempts to find, a "colors" file must be included in the directory. In this file, each color desired is listed in the following format [24]:

[colors]

(Red Value, Green Value, Blue Value) merge expected_number_of_blobs name

[thresholds]

(Y_min:Y_max, U_min:U_max, V_min:V_max)

where Y,U, and V values are from the YUV color spectrum.

In this program, black blobs were attempted to be discerned from an image. The RGB color value of black was ascertained by capturing an image of a black t-shirt and analyzing it using Matlab. The average value of the black t-shirt was found to be (27, 33, 20). In addition, the maximum and minimum values of the t-shirt were found, and these values were converted to YUV format using:

$$Y = .257 * R + .504 * G + .098 * B + 16 \quad (22)$$

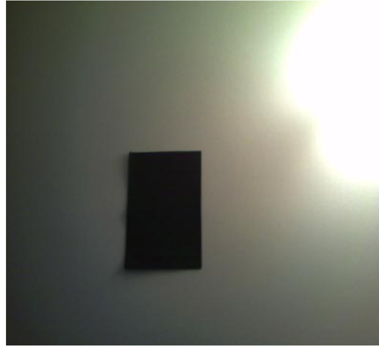
$$V = .439 * R - .368 * G - .071 * B + 128 \quad (23)$$

$$U = -.148 * R - .291 * G + .439 * B + 128 \quad (24)$$

The final Colors.txt file can be seen in Appendix III.

Using this information, a single black blob was detected accurately. In a short experiment, a square, black piece of felt was placed in the view of the camera in two locations. Its blob's center and area was tracked, and each time a blob was found, a picture was taken. This code can be seen in Appendix III.

The following three figures show the two situations in which the blobfinder must find a black blob. FIGURE 18 shows the same position of the felt, while FIGURE 19 shows a lower location of the felt.



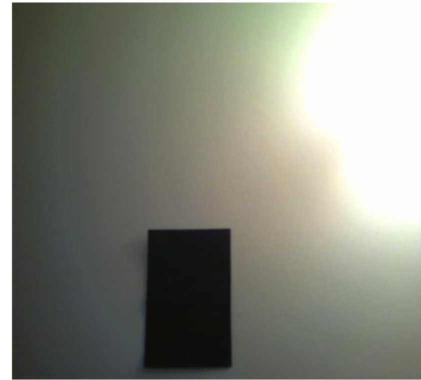


FIGURE 19 - Second Felt Location

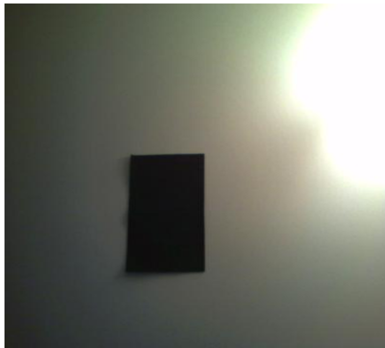


FIGURE 18 - A, B: First Felt Location

The table below shows the output of the program for each blob.

TABLE I

RESULTS OF BLACK BLOBFINDING

Picture Number	1	2	3
Number of Blobs	1	1	1
Centroid of Blob	(311,211)	(281,211)	(261,291)
Area of Blob (pixels^2)	42	42	40

As can be seen from the table, the first two blobs that were found correspond to almost the same blob: each blob contains the same area, and the position of the blob differs by

only 30 pixels in the x-direction. The third blob that was found had a similar area (40 pixels^2 instead of 42 pixels^2), and its position was lower in the camera field of view (pixel locations are measured from (0,0) in the top left corner of the image). This corresponds to a lower position of the blob.

This blobfinding technique shows that an obstacle can be represented as a blob and its two dimensional shape can be analyzed as long as the blobfinder is searching for the object's color. This could provide two purposes. It could be an obstacle detection device in which a path could be planned around the object. It could also be used as a localizer, in which a known object of known location could be used to judge the current position of the robot using the camera. In addition, using the "cmcam" driver, pictures can be saved and analyzed using customized digital image processing algorithms that can yield more information.

The final Player program was the most important but the most disappointing. The ER-1 robot must be controlled by Player in order to be used in experimentation for path planning. However, it seemed that the ER-1 hardware driver which was developed in 2006 for Linux distributions has lost functionality with the newest versions of Player. Using a laptop with the Mandriva Free 2006 operating system, Player version 1.6.5, and the same code described above in the proportional feedback control code, the ER-1 robot physically moved in the same way as in simulation. It used internal odometry instead of GPS-like localization, which suffers from accumulated error over time. In addition, the kinematic constraints of the vehicles are not ideal, in that slip of the wheels does occur. However, the same code used in Mageia with Player version 3.1.x did not move the robot. In addition, while the camera and blobfinding code physically functioned with

Player 3.1.x, attempting to use any camera in Mandriva Free 2006 caused a kernel crash. Because of this, there was not a way to provide location feedback to the actual robot. The thesis continued using the Player/Stage control and simulation scheme under the knowledge that this code can be easily transferred to a physical robot and hardware assuming the hardware drivers are up to date.

D. NTG Preparation

Because the Player/Stage system had been shown to be able to drive a robot to a given point (x,y,θ), the goal with NTG was to program it to produce a list of these points (called waypoints) for the robot to follow. NTG as a program function updates the initial B-Spline coefficients into coefficients for B-Splines that would follow the desired path. Therefore, the first step to producing a list of waypoints was to utilize a function that converts coefficients of B-Splines into waypoints on those B-Splines.

The function SplineInterp, provided by the NTG system, provides the variable outputs (e.g. x, $\dot{x}$, and $\ddot{x}$) given the number of points on the entire path, the number of intervals, the coefficients of the B-Splines, the order of the B-Splines, and the smoothness of the knot points. This result can then be printed to a file using the PrintVector command, also provided by the NTG system. Much work has also been completed using MATLAB and NTG. The example given with the NTG source code comes with a MATLAB file that takes the same information as SplineInterp and computes the variables. MATLAB, however, cannot easily be used in real-time, and therefore SplineInterp was the better choice of B-Spline interpreters for this thesis.

1. An Intuitive Example

As stated previously, NTG generates a trajectory given a cost function to be minimized, according to linear and nonlinear initial, trajectory, and final constraints. As an example, consider an omnidirectional point mass robot (this example is based on the University of Louisville's Yinan Cui's NTG example with two UAV's). In order to minimize its kinetic energy on a path, its linear velocity must be minimized, according to the following trajectory cost function

$$MIN\ J = \int_0^{t_f} (\dot{x}^2 + \dot{y}^2) dt. \tag{25}$$

Therefore, between 0 seconds and the final time, t_f, over the course of a path, the square of the linear velocity, $(\dot{x}^2 + \dot{y}^2)$, is minimized. The robot is subject to linear constraints such as

Initial Linear Constraints: (26)

$x(0) = 5$

$y(0) = 5$

Trajectory Linear Constraints: (27)

$0 \le x \le 100$

$0 \le y \le 100$

$0 \le \dot{x} \le 10$

$0 \le \dot{y} \le 10$

Final Linear Constraints: (28)

$95 \le x(t_f) \le 100$

$$95 \le y(t_f) \le 100.$$

This constraint enforces the robot to start at (5,5) and move to around (97.5,97.5) while keeping its x-direction velocity and y-direction velocity less than 10 m/s and staying within a 100m x 100m square. The robot is also subject to nonlinear constraints such as

Trajectory Nonlinear Constraint: (29)

$$r^2 \le (x - x_o)^2 + (y - y_o)^2 \le B^2.$$

This means that the robot must not share the same space as a circular obstacle at (x_o,y_o) with a radius of r, but it must not go outside the boundary from the center of the obstacle, B. In this case, B should be large enough to encase the entire space in which the obstacle can move. For this example, let (x_o,y_o) = (50,50), B = 100, and r=10m so that the robot must go around the obstacle to reach its goal, and let t_f = 30 seconds. The other parameters for programming NTG can be seen in Appendix VI.

The derivatives of the cost function and nonlinear trajectory constraint functions c are used by NTG, and are clculated in one of two ways. The first way is by passing the function itself into NTG and letting NTG calculate the derivates. The second way is by passing the gradients of the functions with respect to each variable and its derivative. While this is more complex from a programming perspective, it is quicker than the first method. Therefore, the gradients were passed into NTG as follows

Unintegrated Cost Function Gradient: (30)

$$\frac{\partial U}{\partial x} = 0$$

$$\frac{\partial U}{\partial \dot{x}} = 2\dot{x}$$

$$\frac{\partial U}{\partial \ddot{x}} = 0$$

$$\frac{\partial U}{\partial y} = 0$$

$$\frac{\partial U}{\partial \dot{y}} = 2\dot{y}$$

$$\frac{\partial U}{\partial \ddot{y}} = 0$$

Nonlinear Trajectory Constraint Gradient: (31)

$$\frac{\partial T}{\partial x} = 2(x - 50)$$

$$\frac{\partial T}{\partial \dot{x}} = 0$$

$$\frac{\partial T}{\partial \ddot{x}} = 0$$

$$\frac{\partial T}{\partial y} = 2(y - 50)$$

$$\frac{\partial T}{\partial \dot{y}} = 0$$

$$\frac{\partial T}{\partial \ddot{y}} = 0$$

It should be noted that, depending on the Linux distribution and compiler, the differentials that are zero may or may not have to be initialized as such. In this thesis, however, it was necessary for the differentials to be initialized at zero so that NTG functioned correctly.

By calling NTG with these parameters, the following path, shown in FIGURE 20, is generated. The figure shows how the robot avoids the obstacle while traveling in an energy saving pattern (two straight lines and an arc).

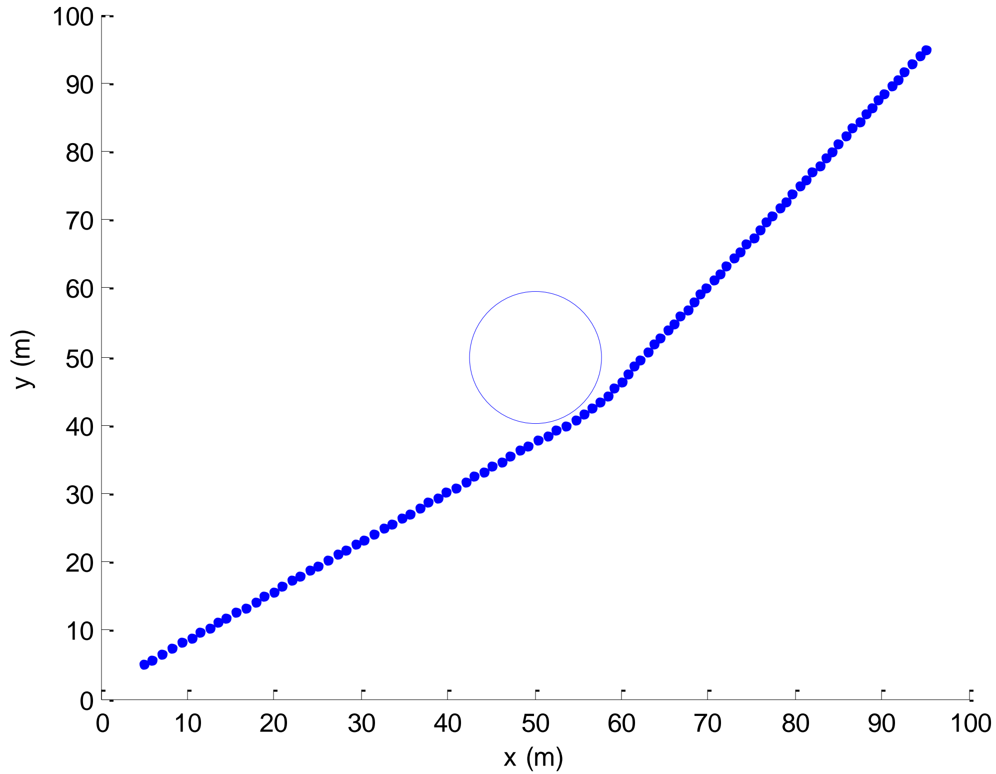


FIGURE 20 - Path Generated for the Omnidirectional Robot

2. <u>Differential Drive Robot Example</u>

The next example models a differential drive robot. First, the robot has kinematic constraints, as shown in (15). In order to minimize the kinetic energy as it travels from point to point, it must minimize the following cost function

$$MIN\, J = \int_0^{t_f} (\dot{x}^2 + \dot{y}^2 + \dot{\theta}^2) dt. \tag{32}$$

Therefore, between 0 seconds and the final time over the course of a path, the square of the linear velocity, $(\dot{x}^2 + \dot{y}^2)$, and the square of the angular velocity, $\dot{\theta}^2$, is minimized. The robot is subject to linear constraints such as

Initial Linear Constraints: (33)

$$x(0) = 5$$

$$y(0) = 5$$

$$\theta(0) = 0$$

Trajectory Linear Constraints: (34)

$$0 \leq x \leq 100$$

$$0 \leq y \leq 100$$

$$-\pi/2 \leq \dot{\theta} \leq \pi/2$$

Final Linear Constraints: (35)

$$95 \leq x(t_f) \leq 100$$

$$95 \leq y(t_f) \leq 100$$

$$\theta(t_f) = 0.$$

This means that the robot must stay within a 100m x 100m square as it travels from (5,5,0) to (97.5,97.5,0). It must also keep its angular velocity within the limits of the ER-1 robot. It is also subject to nonlinear constraints such as

Trajectory Nonlinear Trajectory Constraints: (36)

$$0 \leq \dot{x}^2 + \dot{y}^2 \leq .25$$

$$r^2 \le (x - x_o)^2 + (y - y_o)^2 \le B^2.$$

This means that the linear velocity must stay between 0 m/s and .5 m/s per the ER-1 robot specifications. In addition, the robot must avoid an obstacle of radius r, but stay within a bounding circle, radius B. For this example, let the r = 5m and let t_f=300 seconds, holding the other constants the same as in the previous example.

While these constraints and kinematics accurately describe the differential drive robot scenario, it is not the most efficient in terms of NTG programming. The previous setup yields three differentially flat outputs, (x,y,θ). However, as shown elsewhere, fewer differentially flat outputs cause NTG to generate the trajectory more quickly [25], [26]. In addition, by using the kinematic and dynamic constraints, fewer output variables may be used. From (15),

$$\tan\theta = \frac{\dot{y}}{\dot{x}} \tag{37}$$

This constraint states that the differential drive robot must be oriented tangent to the direction of motion. From (37), the equation for the angular velocity in terms of x and y is

$$\dot{\theta} = \frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^2 + \dot{y}^2} \tag{38}$$

Using (38), the cost function and constraints can be rewritten as

Trajectory Cost Function: (39)

$$MINIMUM: \int_0^{t_f} (\dot{x}^2 + \dot{y}^2 + \left\{\frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^2 + \dot{y}^2}\right\}^2)dt$$

Linear Initial Constraints: (40)

$$x(0) = 5$$

$$y(0) = 5$$

Trajectory Linear Constraints: (41)

$$0 \leq x \leq 100$$

$$0 \leq y \leq 100$$

Final Linear Constraints: (42)

$$95 \leq x(t_f) \leq 100$$

$$95 \leq y(t_f) \leq 100$$

Trajectory Nonlinear Constraints: (43)

$$0 \leq \dot{x}^2 + \dot{y}^2 \leq .25 \quad (43\text{ A})$$

$$-\pi/2 \leq \frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^2 + \dot{y}^2} \leq \pi/2 \quad (43\text{ B})$$

$$r^2 \leq (x - x_o)^2 + (y - y_o)^2 \leq B^2. \quad (43\text{ C})$$

This is equivalent as the previous constraints, except that it has only two flat outputs, x and y. The parameters for programming NTG can be seen in Appendix VI. Finally, the gradients of the cost function and the nonlinear constraints are as follows.

Unintegrated Trajectory Cost Function Gradient: (44)

$$\frac{\partial U}{\partial x} = 0$$

$$\frac{\partial U}{\partial \dot{x}} = 2\dot{x} - \frac{2(\ddot{x}\dot{y}-\dot{x}\ddot{y})(-\ddot{y}\dot{x}^2+2*\dot{x}\ddot{x}\dot{y}+\dot{y}^2\ddot{y})}{(\dot{x}^2+\dot{y}^2)^3}$$

$$\frac{\partial U}{\partial \ddot{x}} = \frac{2\dot{y}(\ddot{x}\dot{y}-\dot{x}\ddot{y})}{(\dot{x}^2+\dot{y}^2)^2}$$

$$\frac{\partial U}{\partial y} = 0$$

$$\frac{\partial U}{\partial \dot{y}} = 2\dot{y} - \frac{2(\ddot{x}\dot{y}-\dot{x}\ddot{y})(\ddot{x}\dot{x}^2+2\dot{x}\ddot{y}\dot{y}-\dot{y}^2\ddot{x})}{(\dot{x}^2+\dot{y}^2)^3}$$

$$\frac{\partial U}{\partial \ddot{y}} = \frac{2\dot{x}(-\ddot{x}\dot{y}+\dot{x}\ddot{y})}{(\dot{x}^2+\dot{y}^2)^2}$$

Nonlinear Trajectory Constraint Gradients: (45)

43 A:

$$\frac{\partial A}{\partial x} = 0$$

$$\frac{\partial A}{\partial \dot{x}} = 2\dot{x}$$

$$\frac{\partial A}{\partial \ddot{x}} = 0$$

$$\frac{\partial A}{\partial y} = 0$$

$$\frac{\partial A}{\partial \dot{y}} = 2\dot{y}$$

$$\frac{\partial A}{\partial \ddot{y}} = 0$$

43 B:

$$\frac{\partial B}{\partial x} = 0$$

$$\frac{\partial B}{\partial \dot{x}} = \frac{-\ddot{y}\dot{x}^2+2\dot{x}\ddot{x}\dot{y}+\dot{y}^2\ddot{y}}{(\dot{x}^2+\dot{y}^2)^2}$$

$$\frac{\partial B}{\partial \ddot{x}} = \frac{-\dot{y}}{\dot{x}^2+\dot{y}^2}$$

$$\frac{\partial B}{\partial y} = 0$$

$$\frac{\partial B}{\partial \dot{y}} = \frac{-\ddot{x}\dot{x}^2+2\dot{x}\ddot{y}\dot{y}+\dot{y}^2\ddot{x}}{(\dot{x}^2+\dot{y}^2)^2}$$

$$\frac{\partial B}{\partial \ddot{y}} = \frac{-\dot{x}}{\dot{x}^2+\dot{y}^2}$$

43 C:

$$\frac{\partial C}{\partial x} = 2(x-50)$$

$$\frac{\partial C}{\partial \dot{x}} = 0$$

$$\frac{\partial C}{\partial \ddot{x}} = 0$$

$$\frac{\partial C}{\partial y} = 2(y-50)$$

$$\frac{\partial C}{\partial \dot{y}} = 0$$

$$\frac{\partial C}{\partial \ddot{y}} = 0$$

It should be noted that when programming, a small value ε=.00001, must be added to the denominators to ensure numerical stability.

By calling NTG with the parameters above, the following path, shown in FIGURE 21, was generated.

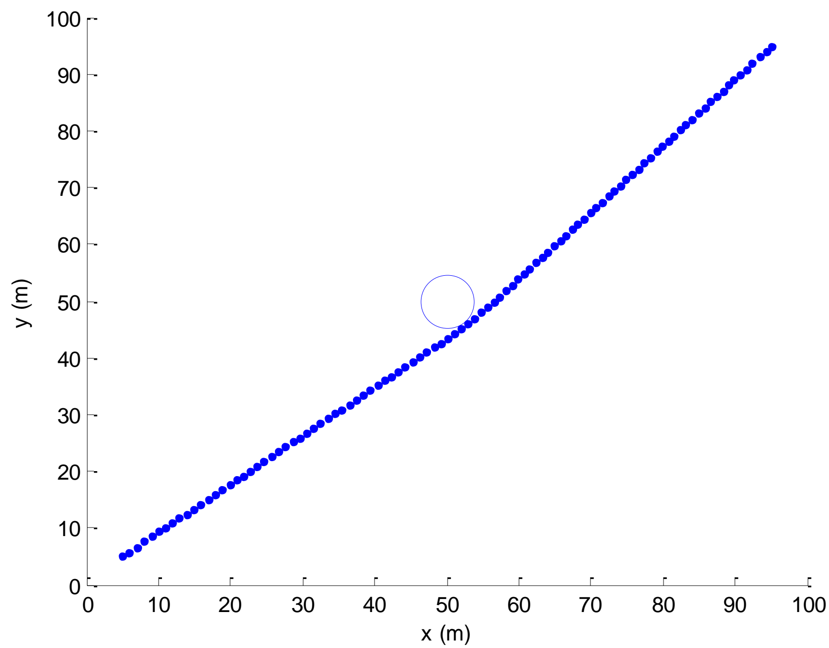


FIGURE 21 - Trajectory of the Differential Drive Scenario

In addition, the linear and angular velocity constraints can be checked for consistency. The following two figures, FIGURE **22** and FIGURE **23**, show the linear and angular velocities along this trajectory. Note that the linear velocity is always between 0m/s and .5m/s and the angular velocity is well within the –π/2rads/sec and π/2rads/sec.

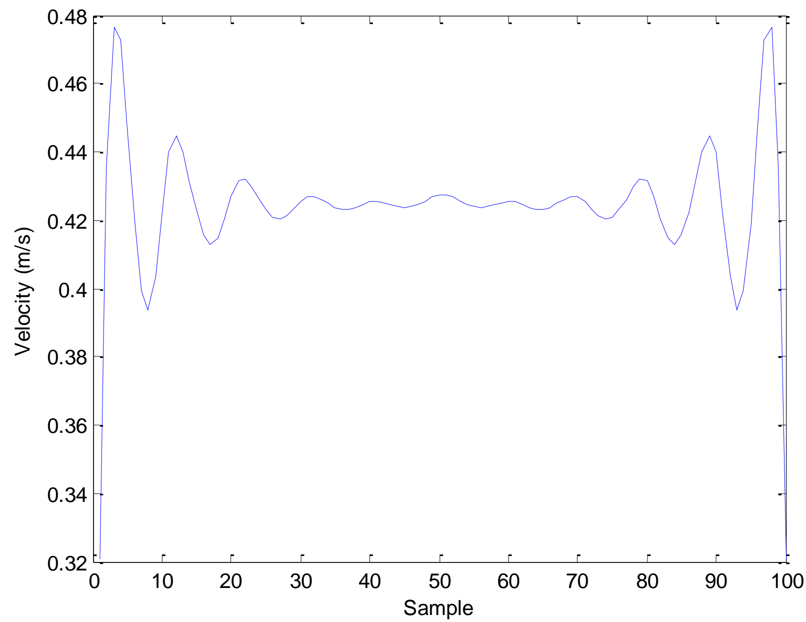


FIGURE 22 - Linear Velocity of the Differential Drive Robot

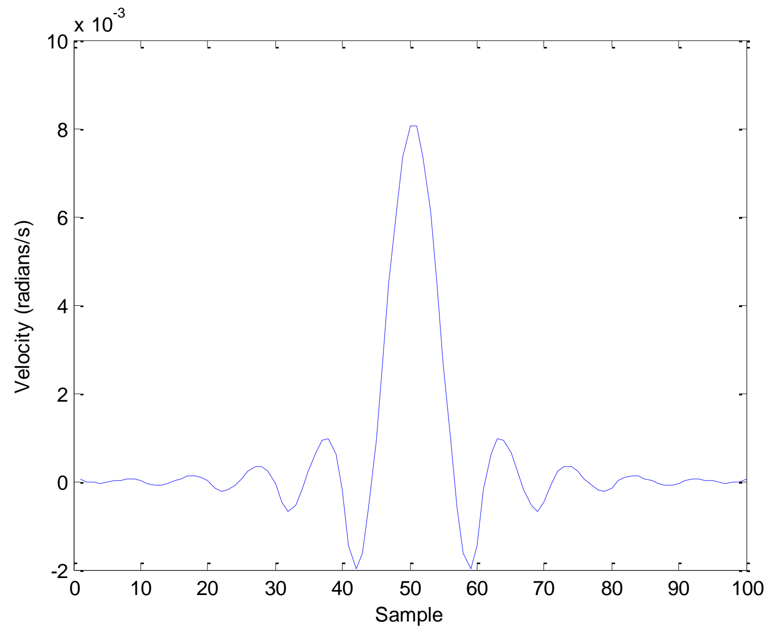


FIGURE 23 - Angular Velocity of the Differential Drive Robot

These results show that a successful path was planned for a differential drive robot with constraints and an obstacle. The NTG program for this trajectory can be seen in Appendix IV.

E. Interconnecting Player/Stage and NTG for a Real-Time Application

The previous examples show how trajectories can be generated that minimize kinetic energy, avoid obstacles, and are feasible for nonholonomic vehicles. However, the examples are not real-time. Indeed, the obstacle is pre-programmed into the trajectory generation, and the trajectory is not updated after the initial generation. If an obstacle were to dynamically show up in the path of the robot, the previous examples would not have the capability to deal with it.

In order to develop this capability, the following scheme was implemented. First, a simulated differential drive robot with a blobfinder is situated at the point (5,5,0). The blobfinder specifications are: Field of View=60°, image size=60x80 pixels, range=10m. The blobfinder gives the system the capability to discover obstacles that may appear within 10m of the robot. Using the blobfinder functions from Player/Stage, the approximate location and size of the robot could be found without prior knowledge of the obstacles existence. Then, a Player control program calls NTG to create a trajectory to move between any number of goal points. The way points for each trajectory to a goal are read via a text file by the Player program.

For this example, the user chose a path from (5,5,0) to (50, 95, 0) to (95, 5, 0) with no assumptions about the existence of obstacles in the trajectory. The Player control program drives the robot from point to point using the smooth path proportional controller program. If the blobfinder senses a dynamic obstacle, the program calculates the size and position of the obstacle using

$$x_o = r * \cos(\theta_c) + x_c \tag{46}$$

$$y_o = r * \sin(\theta_c) + y_c \tag{47}$$

$$radius = \frac{l * r * \tan 30}{60} \tag{48}$$

where x_o is the x-center point of the obstacle, y_o is the y-center point of the obstacle, (x_c, y_c, θ_c) is the current robot pose, r is the range to the object, $l/60$ is the pixel length of the blob divided by 60 (the image x-size), and tan(30) is the tangent of half of the field of view angle (60). Then, these seven pieces of information (x_o , y_o, radius, x_c, y_c, x_{goal}, y_{goal}) are fed into NTG via a text file, and NTG is called again, which generates the path around the obstacle. FIGURE 24 below shows a flowchart for the software.

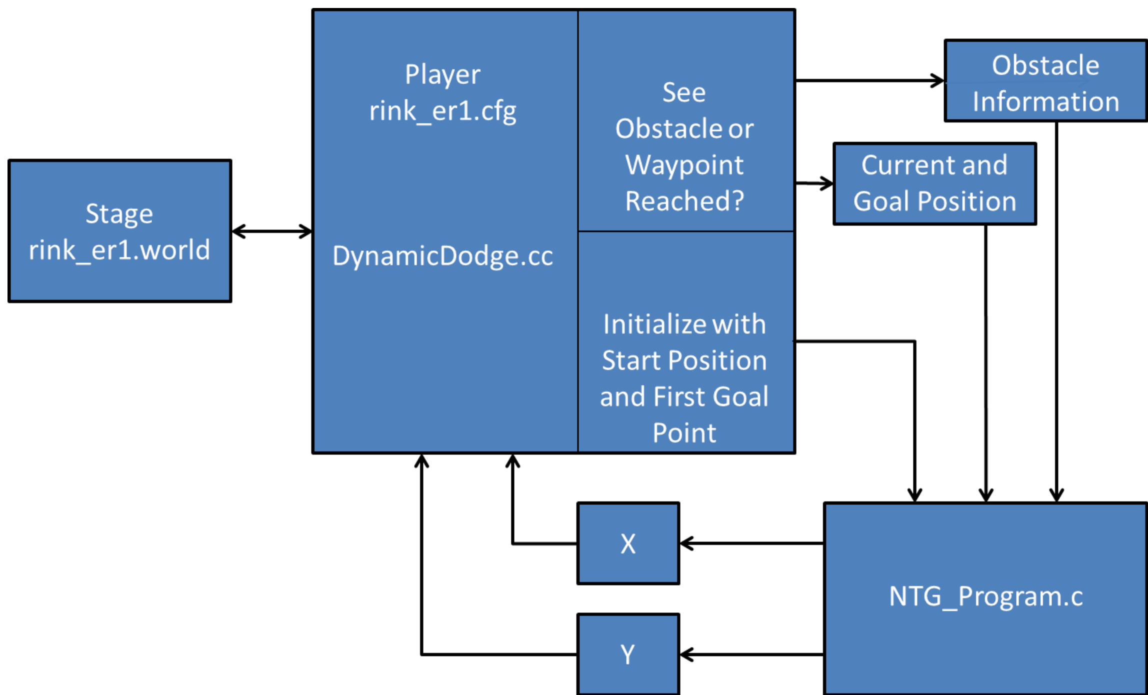


FIGURE 24 - Player/Stage/NTG Software Flowchart

It should be noted that the NTG program only outputs x, y, and their derivatives. While θ could be calculated by the derivatives of x and y, computation time is saved by estimating θ using Euler's forward approximation of a derivative

$$\theta = \tan^{-1} \frac{x(n+1) - x(n)}{y(n+1) - y(n)}. \quad (49)$$

1. Validation of the Approach

FIGURE 25 below shows the results of running this software system with an initial point (5,5), two goal points (50,95) and (95, 5), and two rectangular obstacles at (12, 20, 150) and (60, 75, 30) with a length of 5m. Note the red trail is the past position of the robot. The robot is equipped with a differential drive system, and a blobfinder, with the same specifications as above, that can view gray and green objects. The robot calculates the initial paths to the goal points with no knowledge of the obstacles; the robot sees the obstacles at the points (7.75915, 10.6796, 60) and (54.9231, 83.9231, -30) and senses the obstacles' locations to be (12.0342, 19.4781) and (58.9272, 74.9095) with a radii of about 2.9m each. It then recalculates trajectories around the obstacles. The

Player/Stage/NTG code can be seen in Appendix V.

FIGURE 25 - Player/Stage/NTG Simulation of Multi-obstacle Differential Drive Scenario

The path that the robot followed can be compared to the list of points given, shown in FIGURE 26 below.

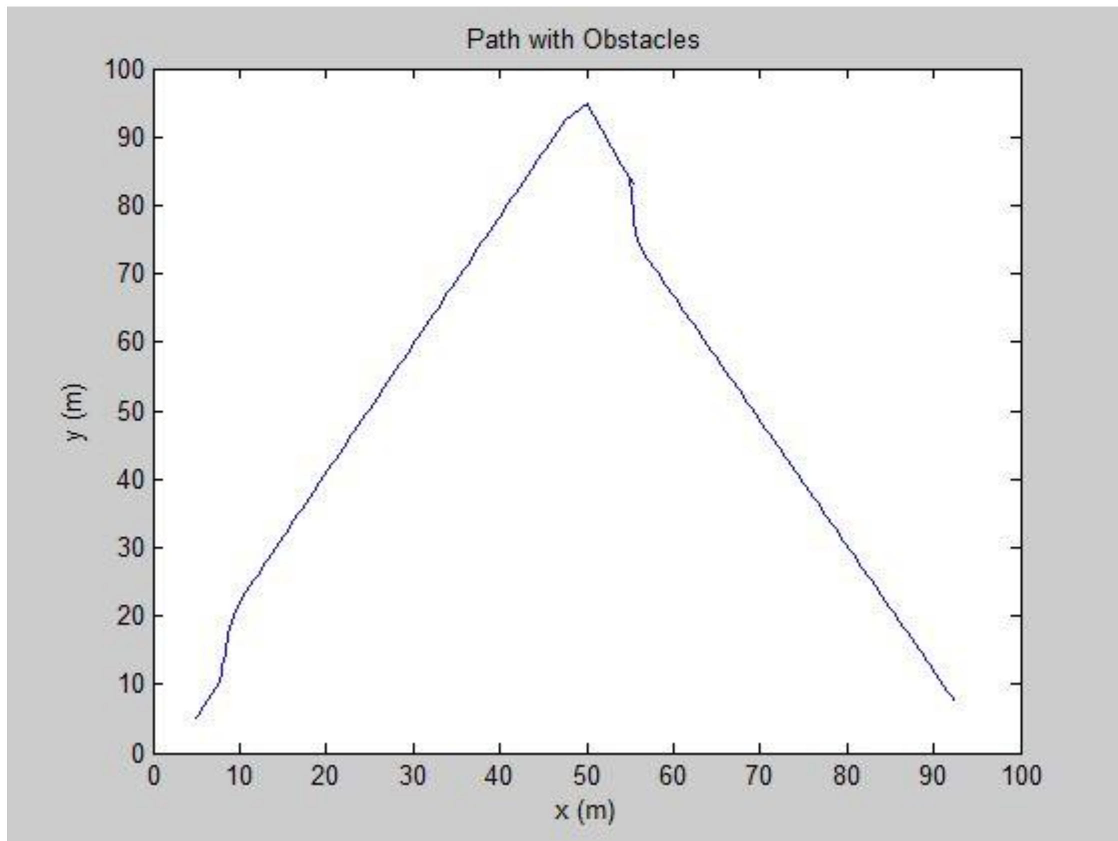


FIGURE 26 - NTG Waypoints Calculated for the Two Static Obstacle Problem

It can be clearly seen that the paths are almost identical, except that the actual traveled path is smooth and does not contain the corner at (50,95). This is due to the proportional feedback controller which drives the robot about smooth curves.

In contrast to this example, another example was run to generate a trajectory for the same initial and goal points without obstacles. In the following, FIGURE 27 shows the results of such a simulation. Note that the path generated moves completely through the obstacles' original positions which shows that the first simulation was not a predetermined path.

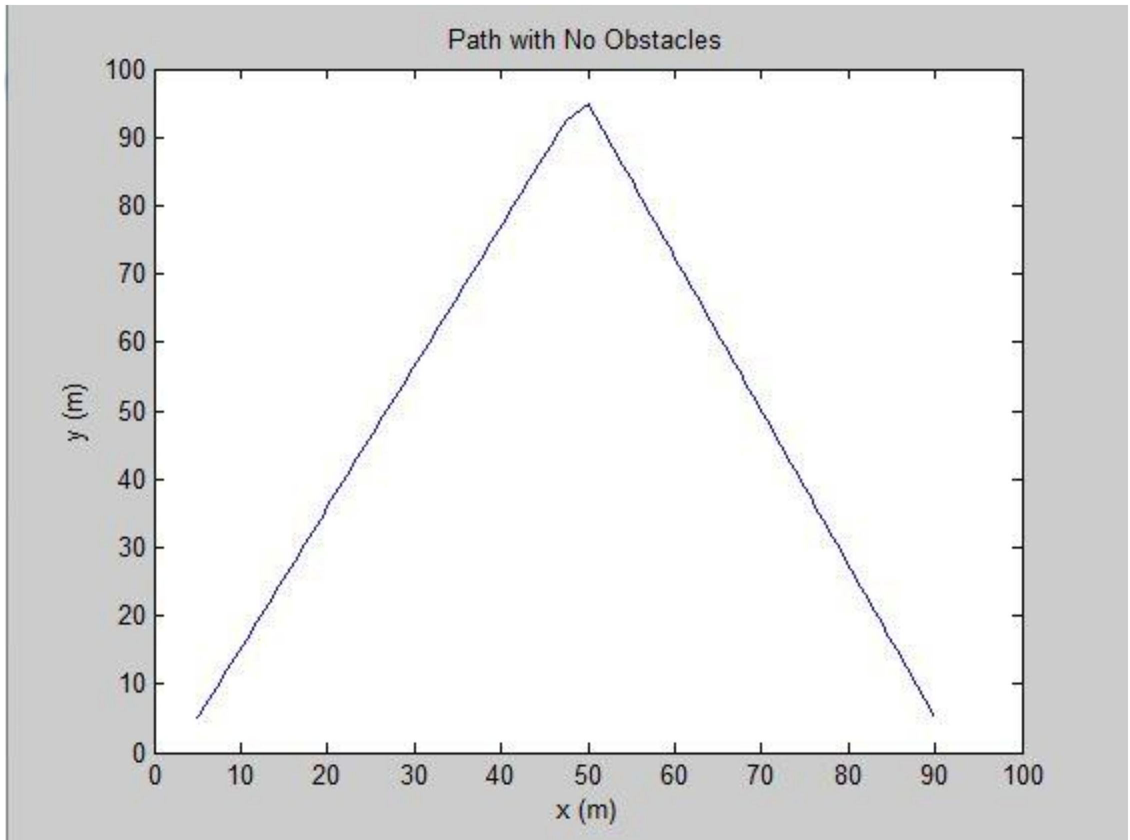


FIGURE 27 - NTG Waypoints Calculated for No Observed Obstacles Scenario

The previous two examples show that NTG can be used to generate a trajectory for a differential drive robot with dynamic obstacles in real time. In addition, this data can be used by Player/Stage to simulate and control robots. Furthermore, data that is captured through Player and Stage can be used by NTG to make accurate paths for dynamic situations.

VIII. DISCUSSIONS AND CONCLUSIONS

The previous results presented in this thesis show many important facts in terms of the capabilities of Player, Stage, and NTG. The work with Player showed the ability to easily capture information about blobs and control a physical robotic drive train. The research concerning the Player/Stage system revealed that algorithms involving potentially expensive sensors and robots could be implemented and tested for free. These algorithms, such as wall following with a range sensor, blobfinding, and proportional feedback position control were tested and shown to work on simulated systems (as well as blobfinding and position control with physical systems). Work pertaining to NTG showed how trajectories could be generated for systems with the following aspects: linear and nonlinear constraints, such as initial positions or speed limitations; holonomic and nonholonomic constraints, such as an omnidirectional or differential drive robot; and dynamic constraints, such as obstacles or energy minimization requirements. Finally a combination of Player, Stage, and NTG displayed the ability to generate real-time trajectories for a differential drive mobile robot.

Because routing robots from one point to another in the presence of dynamic and systematic constraints is a common problem, this thesis aids the solution by providing a way to generate meaningful, feasible paths for nonholonomic mobile robots. Autonomous vehicles such as cars or planes could make use of the basic idea laid forth in this thesis. Even though the culminating experiment of the thesis was simulation only, the other experiments revealed that all that is needed for a physical representation of the same experiment would be a machine and version of Player that would work with both the physical robot and the camera/blobfinding system, not just one or the other. This

does assume, however, that a GPS-like localization (either from a GPS or overhead camera system) is available to give the robot its location. If this is not the case, the software would have to be modified to localize the robot through some other means.

This implementation does have limitations, however. The first issue is the hard coded optimal time. The culminating experiment used a fixed trajectory time; the same time is used no matter where the robot starts on the map as it heads to the goal. In the example, 250 seconds would be allocated for the path from (5,5,0) to (50, 97.5, 0) as well as the recalculated path from (7.75915, 10.6796, 60) to (50, 97.5, 0). Rather than doing this mission, time could be optimized as well [25], [26] by letting

$$\tau = t/t_f \tag{50}$$

$$x(t) = z_1(\tau) \tag{51}$$

$$y(t) = z_2(\tau) \tag{52}$$

$$t_f = z_3. \tag{53}$$

Then, in addition to calculating the optimal path, the optimal time would be calculated as well. One of the most important problems with hard coded trajectory time is that it is possible to generate a trajectory that is feasible and starts and finishes at desired points, but it may have wasted movement, which is undesired according to the cost function. Rather than going in a straight line, the trajectory may take the "scenic route" and oscillate about the line on the way to a goal.

The second limitation pertained to obstacles. The obstacle position is determined by comparing the length of a blob when it gets close to the robot. However, this obstacle,

if long enough, would have a radius greater than that of the range of the blobfinder. Therefore, the robot could start within the region of the obstacle circle; NTG would then fail since there is no way to meet the constraints of the initial condition if the robot contradicts the constraints. A better way to determine the obstacle shape is therefore necessary. In addition, once an obstacle is found, a trajectory is generated. However, the obstacle is still in the blobfinder's field of view directly after the trajectory is generated. The program should not mistake this as a new obstacle and then recalculate a new trajectory. Therefore, a way to handle multiple dynamic obstacles without creating new trajectories for the same obstacle is necessary. In the culminating example, different obstacles are determined by color. If an obstacle is a different color than the last obstacle the robot saw, then it is counted as a different obstacle.

IX. RECOMMENDATIONS

There are many ways in which this thesis can be improved upon for future research. NTG and Player communicate to each other in the system presented through text files. There is one text file created for the obstacle information, and there are two other files created for x and y points. There are at least two possibilities that would lead to faster computation and less programming effort. The first is to use the extern keyword. By declaring variables in header files, and then using the variables in a c or c++ program, the variables can be shared rather than written to and read from a text file. A much more extensive project would be to try to build an NTG path planner driver for the path planner interface for Player. Then, only control code would need to be written for a single robot situation. It would save time, code, and memory during future implementations.

In addition to the passing of variables between Player and NTG, a different control law could be used for driving the robot. The first and easiest option would be to directly solve for the linear and angular velocities using the output of NTG and then set the speeds in Player for the duration of seconds between each waypoint. In this way, NTG is the planner and the controller. Secondly, another control law could be written using Player control code. The control law developed elsewhere, [9], would be more useful than the proportional feedback controller used in this thesis. The proportional feedback controller stabilizes the robot about a point, (x,y,θ) whereas this specific control law stabilizes the robot about a trajectory. Using this control law would be more consistent with the generated trajectory.

Finally, the generated trajectory could be even more efficient than it was in this thesis. By adding robot dynamic constraints, NTG could know exactly how much energy the robot is exerting under certain linear and angular velocities. The dynamics in [27] would suffice, specific to a differential drive robot. These dynamics would be programmed as two more nonlinear trajectory constraints.

This thesis has only begun to provide a comprehensive real-time trajectory generator for nonholonomic robots. However, it did succeed in doing so in a somewhat controlled environment using NTG and Player/Stage. Much more work can be carried out using the recommendations presented previously, and through other research avenues. As each year passes, though, technology progresses more closely to more fully autonomous mobile robots.

APPENDIX I. INSTALLING MAGEIA, PLAYER, STAGE, AND NTG

A. Installing Mageia

Live DVD and CD iso's of Mageia can be found in many places. Two options as of Sept. 2012 are http://www.mageia.org/en/ and http://distrowatch.com/table.php?distribution=mageia . Use these (or similar) websites to download an iso and burn the image to a CD or DVD. Users may install Mageia in parallel to an existing operating system (e.g. Windows and Mageia machine), or they may install Mageia by itself. The installation instructions are fairly simple and Mageia will prompt the user for any actions necessary; the main warning is to not delete any part of your hard drive on accident during installation.

When installing Mageia, the following notes should be followed:

- The user should ensure the correct iso image is downloaded for each machine's specifications (concerning 32-bit vs. 64-bit or i586 vs x86-64).
- Once Mageia is installed, the user should make sure the machine is wired to the internet and go to the Mageia Control Center. Then, under "Configure media sources for install and update", all the lines should be removed and then the Mirror list Core, Nonfree, and Tainted (not Debug or Testing) should be added. This will connect the machine to the online repositories and keep the user from having to insert the installation disc whenever updating is desired.
- After installation, it may be helpful to configure the machine's hardware (such as Wireless cards). The user should ensure that machine is wired to the internet during installation of Mageia. Also, the Mageia Control Center (and the hardware tab) should be used to configure hardware.

B. Installing Player/Stage

The main reason Player and Stage are so difficult to install is the amount of dependencies that are necessary to be installed before Player and Stage can be installed. As of Sept. 2012, a list of dependencies necessary for Player and Stage can be found on http://www.control.aau.dk/~tb/wiki/index.php/Installing_Player_and_Stage_in_Ubuntu. After Mageia is installed and is set to receive updates and packages from online repositories (see second bullet under Installing Mageia), the packages on this page should be available for installation. These packages are CMAKE, CPP, FLTK, FREEGLUT, GDK, GNOMECANVAS, GSL0, GTK, JPEG, BOOST, LIBTOOL, LTDL, OPENCV, PNG, SWIG, XMU, AUTOTOOLS, and BUILD-ESSENTIAL.

There are two options for installing these packages. Those comfortable with the Command Line interface should use the urpmi command. Beginners and those who like GUI's should use the Mageia Control Center. After navigating to the Install and Remove Software within the Center, the user should search for each package (making sure there are no filters, i.e. the tabs say "All" and "All" in the top left corner). The user should ensure to install the newest versions of each package and the packages that are for his/her system (x86-64 vs. I586). In addition, with a x86-64 machine, the packages will be named with a "lib64" prefix, and not just a "lib" prefix. The best way to search for each package is to search for the main body of the package name e.g. boost, gdk-pixbuf, etc.

Note that as of Sept. 2012 autotools and build-essential were not found (they are mainly listed for Ubuntu users). Finally, note that packages will require other packages to install. When the user is prompted to install these additional packages, allow them.

The next step is to download the actual Player and Stage files. The user should

follow these steps.

1. Open a terminal (for the Command Line) and navigate (using the cd command) to the directory with Downloads, Videos, Documents, etc. It should be the default directory. Then:

$ mkdir src /*for source code

$ cd src

The above are commands for the terminal. Do not type in the "$" or the comments listed with /*. Then:

$ mkdir player-svn

$ cd player-svn

2. To get the most up to date version (which is 3.1.0 as of Sept. 2012):

$ svn co https://playerstage.svn.sourceforge.net/svnroot/playerstage/code/player/trunk

This will download the latest version into a file within player-svn, named "trunk".

3. Change directory to src again, and this time run:

$ git clone git://github.com/rtv/Stage.git

4. If these websites are out of date, then do not do 2 and 3. Search for and download the tarballs (like a Windows zip file). Then extract them to the directory src:

$ tar -xvf name_of_tarball.tar path_to_file_src

In order to use Player with Stage, Player must be installed first (after installing all dependencies, including those for Stage). Starting in the /trunk directory (or the file that contains the extracted Player files):

$ mkdir build

$ cd build

$ ccmake ..

This last command will bring up a configuration screen. Press “c” to configure. Then press 't” to bring up all options. Find the option for cmake verbose mode and turn it on by pressing ENTER. Then, find the install prefix and make sure it says either usr/local/lib or usr/local/lib64 (again, this depends on the user’s machine). Press “c” again to configure, and then press “g” to generate the make files. Then:

$ cmake ..

If it returns no errors, then continue; otherwise read the errors carefully. Make sure they do not mention a missing package-if so, download and install the package using Mageia control Center. If it is another error, the best advice is to search for the error on the internet.

$ make

If it returns no errors, then continue.

$ su

$ /*enter password

$ make install

If it returns no errors, Player has been successfully installed.

$ exit /*to get out of su

Once Player has been installed, Stage may be installed. An important note here is to make sure Stage finds the installed version of Player. Type:

$ which player

If it does not return a directory, then:

$ export

PKG_CONFIG_PATH=/usr/local/lib64/pkgconfig:$PKG_CONFIG_PATH

Again, if the machine is 32 bit, replace “lib64” with “lib”. The user may want to run this command, even if “which player” returned a directory. Then:

$ which player /*just to check that it indeed returns a directory

Navigate into the rtv-Stage file. Then:

$ mkdir build

$ cd build

$ ccmake ..

Again, press “c” to configure, press “t” to toggle options, turn on verbose mode, and press “c” to configure again. Press “g” when it is ready to generate.

$ cmake ..

$ make

Directly after this command, make sure that Player version 3.1.0 is found, and that it does not say “Player not found”. If it says this, make sure to stop and allow Stage to find Player. There are many reasons that Stage may have not found Player. One reason may just be the search path to libstageplugin is incorrect. Another reason may be that Player did not install correctly, or that Player did not have the most up-to-date files when it was installed. A re-install of Player may be necessary if it is not a file path issue. If so, delete the build files and restart the installation. Without Stage finding Player, the stageplugin will not be built, and Stage will not be able to be used with Player. Then:

$ su

$ /*enter password

$ make install

$ exit

If there were no errors, then Stage was installed successfully.

Checking the Installation:

Many times the user may have thought his/her installation was successful, but it may not have been. First, the user should update a file called ld.so.conf. This may help errors in Stage not finding Player, and it is also necessary for running Player and Stage together. Run:

$ cd /

$ cd etc

$ su

$ /*enter password

$ vi ld.so.conf

Go to the end of the last line in the document that opens and press the INSERT key. Then add a line that says:

/usr/local/lib64

And another that says:

/usr/local/lib

Then press the ESCAPE key, and type:

:wq

In addition, run the following commands, and add them to the bashrc.

$ export LD_LIBRARY_PATH=/usr/local/lib64

$ export STAGEPATH=/usr/local/lib64

$ export PLAYERPATH=/usr/local/lib64

Remember to replace “lib64” with “lib” if the user is on a 32-bit machine.

Finally:

$ player /usr/local/share/stage/worlds/simple.cfg

This should open a window of a robot wandering (see below in FIGURE 28).

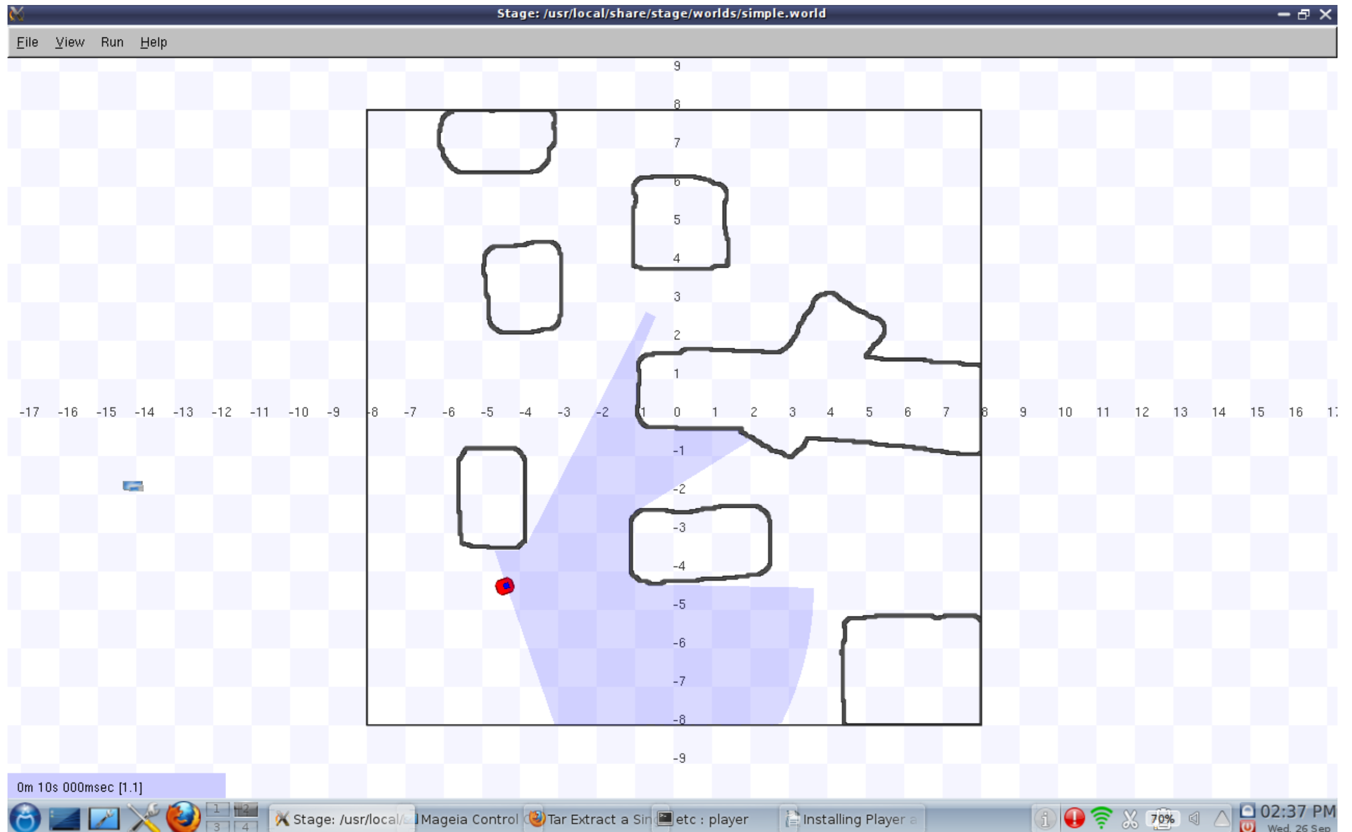


FIGURE 28 - Player running simple.cfg with Stage

As a word of encouragement, the errors users may (probably) encounter by many encountered by many other users. Search for them carefully on the internet to solve them. However, this instruction set has been tested on a 64-bit and 32-bit machine with both Mageia 1 and 2, so these instructions should work.

C. Installing NTG

To install NTG2.2+, the tarbal should be downloaded from http://www.cds.caltech.edu/~murray/software/2002a_ntg.html. Then NTG should be extracted in the home directory. The user should ensure that the folders named "doc," "examples," "lib," "pgs," and "src" exist within this extraction. Before installing NTG, gcc-gfortran must be installed for Mageia using either the command line or Mageia Control Center.

PGS must be installed first:

```
$ cd NTG/pgs
$ make
```

If there are no errors:

```
$ make install
```

If there are errors, they will be listed as Fortran code errors. Common errors that were encountered in this thesis included the files l2err.f, l2main.f, and setdatx3.f that were variable or array type errors. After editing and reconciling the existing source code:

```
$ make
$ make install
```

Note that if the machine is 64 bit, the user will need to compile pgs in 32 bit mode. This can be done by modifying the makefile within pgs to say -m32 where it says -O3.

Next:

```
$cd ..
$make
```

$make install

Again note that that if the machine is 64 bit, the user will need to compile pgs in 32 bit mode. This can be done by modifying the makefile within pgs to say -m32 where it says -O3.

Next, the user should check to make sure these four files are in the NTG/lib directory: libg2c.a, libnpsol.a, libpgs.a, and libntg.a . libnpsol.a was given to the thesis by the thesis director.

Next:

$cd examples

The user should modify the makefile and add "-lgfortran" to the line that contains "LIB= -lm -lntg -lpgs -lnpsol -lg2c". Then:

$make vanderpol

Run vanderpol by

$./vanderpol

APPENDIX II. PLAYER/STAGE PROGRAMS

A. Wall Following

WallFollow.cc:

```
#include <libplayerc++/playerc++.h>
#include <iostream>
#include <stdio.h>
#include <math.h>

#include "args.h"

int main (int argc, char **argv) {
  parse_args(argc, argv);

  try {
    using namespace PlayerCc;

    PlayerClient robot(gHostname, gPort);
    Position2dProxy p2d(&robot, gIndex);
    RangerProxy sr(&robot,gIndex);

    p2d.SetMotorEnable (true);

    double fSpeed, tSpeed;
    double goal, range, thresh;

    robot.Read();
    goal = .3;
    thresh = .055;
    p2d.SetSpeed(.1,0);
    while (1) {
      robot.Read();
      range=sr.GetRange(0);
      if (range < goal -thresh) {
      p2d.SetSpeed(.1, .17);
      }
      if (range > goal - thresh && range < goal + thresh) {
      p2d.SetSpeed(.1,0);
      }
      if (range > goal + thresh) {
      p2d.SetSpeed(.1,-.17);
      }
    }

    return 0;
  }
  catch (PlayerCc::PlayerError & e) {
    std::cerr << e << std::endl;
    return -1;
  }
}
```

rink_er1_ranger.cfg:

```
driver
(
  name "stage"   #there is a driver named stage
  plugin "stageplugin"     #the driver is in the stageplugin library

  provides ["simulation:0"]

  #Load the named file into the simulator
  worldfile "rink_er1_ranger.world"
)

driver (
  name "stage"
  provides ["6665:position2d:0" "6665:ranger:0"]
  model "erone1"
)
```

rink_er1_ranger.world

```
include "/usr/local/share/stage/worlds/map.inc"  #this gets me the
include file map
include "erone_ranger.inc"

#configure the GUI window
window (
  size [700.000 700.000]    #size of the window in pixels
  scale 46        #pixels/meter  this value is (window size)/(floorplan
size)

  show_data 1
)

#load an environment bitmap
floorplan (
  bitmap "random.png"   #just a big circular room
  size [15 15 1.5]    #size of the room [x y z] in meters
)

#make an instance of the er1 robot
erone(
  name "erone1"
  pose [6.5 0 0 90]   #starts at origin on the floor facing right
  color "grey"
)
```

erone_ranger.inc

```
#make the model for the er1 with a sonar (named erone)

#make a position model for the er1 with wheels and shape
define erone position (
  #actual size of the er1 robot (not trailer)
  size [0.4064 0.381 0.6096]  #erone is scaled to fit in this box
                              #center is [.2032 .1905 .3048]
  #shape of erone
  block(
    points 8
    point[0] [0 10]
    point[1] [15 10]
    point[2] [15 7.5]
    point[3] [17.5 7.5]
    point[4] [17.5 2.5]
    point[5] [15 2.5]
    point[6] [15 0]
    point[7] [0 0]
    z [0 10]
  )

  #positional descriptions for erone wheels
  drive "diff"    #differential drive robot
  localization "gps"  #knows its location perfectly
  localization_origin [0 0 0 0]

  #attach sensors
  erone_sonar()

  #what can erone sense
  ranger_return 1
  obstacle_return 1
)


define erone_sonar ranger (
  #eronesonar( pose [.25 -.25 -.25 -90] )
  #eronesonar( pose [.25 0 -.25 0] )
  sensor (
    size [0.1 0.1 0.1]
    range [0 7.5]
    fov 10
    samples 1
    color "gray"
  )
  pose [.25 -.25 -.25 -90]
)
```

B. Blobfinding

getblob.cc:

```
#include <stdio.h>
#include <iostream>
#include <time.h>
#include <libplayerc++/playerc++.h>
#include "args.h"


int main(int argc, char **argv) {
  parse_args(argc, argv);

  try {
  using namespace PlayerCc;

  PlayerClient bug1(gHostname, gPort);

  Position2dProxy p2d(&bug1, gIndex);
  BlobfinderProxy blb(&bug1, gIndex);

  int Depth, Width, Height;
  int NumBlobs;
  int xCent, yCent;
  int left, right, top, bottom;
  playerc_blobfinder_blob_t MyBlob;


  bug1.Read();

  while(1) {
    bug1.Read();
    if (blb.GetCount() > 0) {
      MyBlob = blb.GetBlob(0);
      xCent = MyBlob.x;
      yCent = MyBlob.y;
      printf("xcenter %d , ycenter %d, left %d  right %d  top %d
bottom %d \n", xCent, yCent, MyBlob.left, MyBlob.right, MyBlob.top,
MyBlob.bottom);
      printf("  Range:  %f  \n", MyBlob.range);
      std::cout << blb.GetCount() << std::endl;
    }
    else
      std::cout << blb.GetCount() << std::endl;
  }


  return 0;
  }
  catch (PlayerCc::PlayerError & e)
  {
    std::cerr << e << std::endl;
    return -1;
  }
}
```

Scenery.cfg

```
driver
(
  name "stage"   #there is a driver named stage
  plugin "stageplugin"     #the driver is in the stageplugin library

  provides ["simulation:0"]

  #Load the named file into the simulator
  worldfile "scenery.world"
)

driver
(
  name "stage"   #there is a driver named stage
  provides ["6665:position2d:0" "6665:blobfinder:0"]  #all the devices
on the model
  model "bug1"  #needed by Player/Stage to show that anything done with
this drive will affect this model
)

driver
(
  name "stage"   #there is a driver named stage
  provides ["6666:position2d:0" "6666:blobfinder:0"]  #all the devices
on the model
  model "bug2"  #needed by Player/Stage to show that anything done with
this drive will affect this model
)
```

Scenery.world:

```
include "/usr/local/share/stage/worlds/map.inc"  #this gets me the
include file map
include "photobug.inc"

window (
  size [700.000 700.000]
  scale 35

  show_data 1
)

floorplan (
  bitmap "/usr/local/share/stage/worlds/bitmaps/cave.png"
  size [15 15 1.5]
  blob_return 1
)
```

```
photobug (
  name "bug1"
  pose [-4 -5 0 45]
  color "purple"
  blob_return 1
)

photobug (
  name "bug2"
  pose [-3 -3 0 225]
  color "green"
  blob_return 1
)
```

Photobug.inc:

```
#make the model for the photobug

#position model
define photobug position (
  size [.5 .5 .5]

  #shape
  block (
    points 8
    point[0] [0 10]
    point[1] [15 10]
    point[2] [15 7.5]
    point[3] [17.5 7.5]
    point[4] [17.5 2.5]
    point[5] [15 2.5]
    point[6] [15 0]
    point[7] [0 0]
    z [0 10]
  )

  drive "diff"
  localization "gps"
  localization_origin [0 0 0 0]

  photobug_blob()
  photobug_cam()
  blob_return 1
  obstacle_return 1
)

#blobfinder model

define photobug_blob blobfinder (
    colors_count 3
    colors ["gray30" "green" "purple"]
    range 10.0
    image [80 60]
    size [0.1 0.07 0.05]
```

```
    color "black"
    #alwayson 1
)

#camera model
define photobug_cam camera(
  resolution [32 32]
  range [0.2 8.0]
  fov [70.0 40.0]
  pantilt [0.0 0.0]  #left right and down up

  size [0.1 0.07 0.05]
  color "black"
  watts 100.0
  #alwayson 1
)
```

C. Proportional Feedback Control

ProportionalGoTo.cc:

```
#include <stdio.h>
#include <iostream>
#include <time.h>
#include <math.h>
#include <libplayerc++/playerc++.h>

#include "args.h"

#define RAYS 32
#define pi 3.141592653589793238

int main(int argc, char **argv) {
  parse_args(argc, argv);

  try {
  using namespace PlayerCc;

  PlayerClient er1(gHostname, gPort);

  Position2dProxy p2d(&er1, gIndex);

  p2d.SetMotorEnable(1);

  float xerr, yerr, terr, perr, aerr, berr;
  float Xg, Yg, Tg;   //goal vars
  float Xc, Yc, Tc;   //current vars
  float thresh = .1;
  float offset = .2032;
  float kp, ka, kb;
  kp=.5;
  ka=1;
  kb=-.5;  //these values work!!


  /*kp=.3;
  ka=.5;
  kb=-.15;   //these values work */


  double fSpeed, tSpeed;

  printf("Enter the x coordinate ");
  std::cin >> Xg;
   printf("Enter the y coordinate ");
  std::cin >> Yg;
   printf("Enter the final angle in degrees ");
  std::cin >> Tg;

  Tg=Tg*3.14159/180;   //convert to radians
  //Xg=Xg+offset*cos(Tg);   //cos and sin is in radians
  //Yg=Yg+offset*sin(Tg);
```

```
  printf("Goal Pose is  (%f ,  %f ,  %f )", Xg, Yg, Tg);

  while (1) {
    er1.Read();
    Xc=p2d.GetXPos();
    Yc=p2d.GetYPos();
    Tc=p2d.GetYaw();

    xerr = Xg-Xc;
    yerr = Yg-Yc;
    terr = Tg-Tc;

    if(abs(xerr/thresh) < 1) {
      if(abs(yerr/thresh) < 1) {
      if(abs(terr/thresh) < 1) {
        break;
      }
      }
    }

    perr=sqrt(xerr*xerr+yerr*yerr);
    aerr=atan2(yerr,xerr)-Tc;
    berr=Tg-Tc-aerr;

    if (aerr < -3.14159)
      aerr=aerr+2*3.14159;
    if (aerr > 3.14159)
      aerr=aerr-2*3.14159;

    if (berr < -3.14159)
      berr=berr+2*3.14159;
    if (berr > 3.14159)
      berr=berr-2*3.14159;

    fSpeed=kp*perr;
    tSpeed=ka*aerr+kb*berr;

    p2d.SetSpeed(fSpeed,tSpeed);

  }
  return 0;
  }
  catch (PlayerCc::PlayerError & e)
  {
    std::cerr << e << std::endl;
    return -1;
  }

}
```

```
rink_er1.cfg:

driver
(
  name "stage"   #there is a driver named stage
```

```
  plugin "stageplugin"     #the driver is in the stageplugin library

  provides ["simulation:0"]

  #Load the named file into the simulator
  worldfile "rink_er1.world"  #this is a world file stored in the same
place as empty.cfg
)

driver (
  name "stage"
  provides ["odometry::6665:position2d:0"]
  model "erone1"
)
```

rink_er1.world:

```
include "/usr/local/share/stage/worlds/map.inc"  #this gets me the
include file map
include "erone.inc"

#configure the GUI window
window (
  size [700.000 700.000]    #size of the window in pixels
  scale 46        #pixels/meter   this value is (window
size)/(floorplan size)

  show_data 1
)

#load an environment bitmap
floorplan (
  bitmap "/usr/local/share/stage/worlds/bitmaps/rink.png"   #just a big
circular room
  size [15 15 1.5]    #size of the room [x y z] in meters
)

#make an instance of the er1 robot
erone(
  name "erone1"
  pose [0 0 0 0]   #starts at origin on the floor facing right
  color "red"
)
```

erone.inc

```
#make the model for the er1  (named erone)

#make a position model for the er1 with wheels and shape
define erone position (
  #actual size of the er1 robot (not trailer)
  size [0.4064 0.381 0.6096]  #erone is scaled to fit in this box
                        #center is [.2032 .1905 .3048]
```

```
  #shape of erone
  block(
    points 4
    point[3] [0 0]
    point[2] [0 16]
    point[1] [15 16]
    point[0] [15 0]
    z [0 24]  #how tall he is
  )

  block (
    points 4
    point[3] [15 7]
    point[2] [18 7]
    point[1] [18 9]
    point[0] [15 9]
    z [0 24]  #how tall he is
  )

  #positional descriptions for erone wheels
  drive "diff"    #differential drive robot
  localization "gps"  #knows its location perfectly
  localization_origin [0 0 0 0]

  obstacle_return 1
)
```

APPENDIX III. PLAYER PROGRAMS

CamAndBlob.cfg

```
driver
(
  name "cvcam"
  provides ["camera:0"]
)

driver
(
  name "cmvision"
  provides ["blobfinder:0"]
  requires ["camera:0"]
  colorfile ["colors.txt"]
  minblobarea 40  #min number of pixels to qualify as a blob
  maxblobarea 400  #max number of pixels to qualify as a blob
)
```

Colors.txt:

```
[Colors]
(27,33,20)  0.00000 2 BlackShirt

[Thresholds]
(16:77,132:161,127:129)
```

Getblob.cc:

```
#include <stdio.h>
#include <iostream>
#include <time.h>
#include <math.h>
#include <libplayerc++/playerc++.h>

#include "args.h"

#define RAYS 32
#define pi 3.141592653589793238

int main(int argc, char **argv) {
  parse_args(argc, argv);

  try {
  using namespace PlayerCc;
```

```
  PlayerClient robot(gHostname, gPort);

  CameraProxy cam(&robot, gIndex);
  BlobfinderProxy blb(&robot, gIndex);

  int Depth, Width, Height;
  int NumBlobs;
  int xCent, yCent;
  int area;
  playerc_blobfinder_blob_t MyBlob;

  robot.Read();

  Width=cam.GetWidth();
  Height=cam.GetHeight();
  Depth=cam.GetDepth();
  printf("Width %d ,Height %d , Depth %d \n",Width, Height, Depth);
  while(1) {
    robot.Read();
    if(blb.GetCount() > 0) {
      MyBlob = blb.GetBlob(0);
      xCent=MyBlob.x;
      yCent=MyBlob.y;
      area=MyBlob.area;
    printf("Number of Blobs %d \n",blb.GetCount());
    printf("Blob 0 center is %d  %d \n Blob 0 Area is %d pixels
\n",xCent,yCent,area);
    sleep(5);
    cam.SaveFrame("blob", 0);
    }
  }

  /*for (int i=0; i<1; i++) {
    robot.Read();
    cam.SaveFrame("camera");
  }*/


  return 0;
  }
  catch (PlayerCc::PlayerError & e)
  {
    std::cerr << e << std::endl;
    return -1;
  }

}
```

APPENDIX IV. NTG PROGRAM FOR DIFFERENTIAL DRIVE ROBOT

```
#include <stdlib.h>
#include <math.h>
#include "ntg.h"

#define NOUT            2
#define nINTERV         10
#define NINTERV         nINTERV, nINTERV
#define mULT            4
#define MULT            mULT, mULT
#define oRDER           6
#define ORDER           oRDER, oRDER
#define mAXDERIV  3
#define MAXDERIV        mAXDERIV, mAXDERIV

#define nCOEF           24          //NINTERV * (ORDER - MULT) + MULT
#define NCOEF           2 * nCOEF
#define NBPS            (NCOEF + 6) //not a necessary relationship
#define NVAR            3 * 2 * 1
#define TIMESPAN  300         // time span of 300s

// Label outputs and their derivatives

                              // active variable sequence
#define x             zp[0][0]    // 0         horizontal location of
robot
#define xd            zp[0][1]    // 1   horizontal speed of robot
#define xdd       zp[0][2]    // 2

#define y             zp[1][0]    // 3   vertical location of robot
#define yd            zp[1][1]    // 4   vertical speed of robot
#define ydd       zp[1][2]    // 5


/* number of linear constraints */
#define NLIC        2          // linear initial constraints
#define NLTC        2        // linear traj. constraints
#define NLFC        2        // linear final constraints

/* number of nonlinear constraints */
#define NNLIC        0       // nonlinear initial constraints
#define NNLTC          3           // nonlinear trajectory constraints
#define NNLFC        0       // nonlinear final constraints

//nonlinear constraints function active variables
#define NINITIALCONSTRAV     0
#define NTRAJECTORYCONSTRAV  6
#define NFINALCONSTRAV       0

//number of cost functions
#define NICF      0           // initial cost function
#define NUCF      1           // unintegrated trajectory cost function
#define NFCF      0           // final cost function
```

```
//active variables for cost function
#define NINITIALCOSTAV     0
#define NTRAJECTORYCOSTAV  4
#define NFINALCOSTAV       0

// enum x, y are used to divide the coefficients table into 2 portions
enum {xx1 = nCOEF, yy1 = 2*nCOEF};


/* declare all of the active variables required for the nonlinear
constraints and the cost functions */
static AV trajectoryconstrav[NTRAJECTORYCONSTRAV]={{0,0}, {0,1}, {0,2},
{1,0}, {1,1}, {1,2}}; // x, xd, xdd, y, yd, ydd
static AV trajectorycostav[NTRAJECTORYCOSTAV]={{0,1}, {0,2}, {1,1},
{1,2}}; // xd, xdd, yd, ydd


/* declare the function used to compute the unintegrated cost */
void ucf(int *mode, int *nstate, int *i, double *f, double *df, double
**zp);

/* declare nonlinear traj. constraint function */
void nltcf(int* mode, int* nstate, int* i, double* f, double** df,
double** zp);

// declare spline interp function, this function returns an ncps*3
array
double* callSplineInterp(int xxyy, double* coefficients, double**
knots,
  int* ninterv, int* order, int* mult, int* maxderiv, double* result,
int ncps);


  float xob = 50;
  float yob = 50;
  float radius = 5;
int main(void)
{


    double* knots[NOUT];
    int ninterv[NOUT] = {NINTERV};
    int mult[NOUT] = {MULT};
    int maxderiv[NOUT] = {MAXDERIV};
    int order[NOUT] = {ORDER};

    // result is used to store the interpretted splines results
    int ncps = 100;      // 100 points per variable, e.g. 100 x1 values
    double* result;      // global variable used to store interpretted
splines
    result = malloc(ncps * 3 * sizeof(double));



    int nbps;           // number of break points
    double* bps;
```

```
    double** lic;
    double** lfc;
    double** ltc;

    int ncoef;
    double* coefficients;
    int* istate;
    double* clambda;
    double* R;

    // upper and lower bounds
    double lowerb[NLIC + NLTC + NLFC + NNLIC + NNLTC + NNLFC];
    double upperb[NLIC + NLTC + NLFC + NNLIC + NNLTC + NNLFC];
    int inform =0;
    double objective;

    // allocate space and initialize the knot points
    int index;
    for(index = 0; index < NOUT; ++index){
        knots[index] = malloc((ninterv[index] + 1) * sizeof(double));
        linspace(knots[index], 0, TIMESPAN, ninterv[index] + 1);
    }

    // WARNING, specific for this case

    ncoef = NCOEF;
    coefficients = malloc(ncoef * sizeof(double));

    // initial guess for coefficients
    linspace(coefficients, 1, 1, ncoef);

    // allocate space for breakpoints and initialize
    nbps = NBPS;
    bps = malloc(nbps * sizeof(double));
    linspace(bps, 0, TIMESPAN, nbps);     // 300 is fixed optimal traj.
time

    /*
     * ntg internal memory
     * these calculations do not need to be changed
     */

    istate = malloc((ncoef + NLIC + NLFC + NLTC * nbps
            + NNLIC + NNLTC * nbps + NNLFC) * sizeof(int));

    clambda = malloc((ncoef + NLIC + NLFC + NLTC * nbps
            + NNLIC + NNLTC * nbps + NNLFC) * sizeof(double));

    R = malloc((ncoef + 1) * (ncoef + 1) * sizeof(double));



    lic = DoubleMatrix(NLIC, NVAR);
    ltc = DoubleMatrix(NLTC, NVAR); // 6 = 3 * 2 * number of robots
    lfc = DoubleMatrix(NLFC, NVAR);
```

```
    /* define the constraints
     * for linear constraints:
     * indexed by the active variable number
     * [constraint number][a.v. number]
     */

    // 2 lic
    lic[0][0] = 1;            // first 0 is the sequence number of the
constraint
                              // second 0 is the sequence number of the
a.v.(x here)
                              // 1 is the factor/coefficient of this
a.v.
    lowerb[0] = upperb[0] = 5;      // 5 is the boundary value of the
constraint
    lic[1][3] = 1; lowerb[1] = upperb[1] = 5;

    // 2 ltc
    ltc[0][0] = 1; lowerb[2] = 0; upperb[2] = 100;
    ltc[1][3] = 1; lowerb[3] = 0; upperb[3] = 100;


    // 2 lfc, location constraints on x1,y1
    lfc[0][0] = 1; lowerb[4] = 95; upperb[4] = 100;
    lfc[1][3] = 1; lowerb[5] = 95; upperb[5] = 100;


    //Nonlinear Initial Constraints
    //None

    //Nonlinear Trajectory Constraints
    lowerb[6] = 0; upperb[6] = .25;
    lowerb[7] = -3.14/2; upperb[7] = 3.14/2;
    lowerb[8] = pow(radius,2); upperb[8] = pow(100,2);


    //Nonlinear Final Constraints
    //None

    // the ntg subroutine:


    npsoloption("summary file = 0");


          ntg(NOUT, bps, nbps, ninterv, knots, order, mult, maxderiv,
            coefficients,
            NLIC,                   lic,
            NLTC,                   ltc,
            NLFC,                   lfc,
            NNLIC,                  NULL,
            NNLTC,                  nltcf,
            NNLFC,                  NULL,
            NINITIALCONSTRAV,       NULL,
            NTRAJECTORYCONSTRAV,      trajectoryconstrav,
            NFINALCONSTRAV,         NULL,
```

```
            lowerb,                 upperb,
            NICF,                   NULL,
            NUCF,                   ucf,
            NFCF,                   NULL,
            NINITIALCOSTAV,         NULL,
            NTRAJECTORYCOSTAV,      trajectorycostav,
            NFINALCOSTAV,           NULL,
            istate, clambda, R, &inform, &objective
            );



    printf("Out from ntg call, now using the PrintVector \n");
    printf("Inform is %d \n", inform);

    // call spline interp and print out to file for x and y
    // e.g. the file x contains 100 x values followed by 100 xd
    // followed by 100 xdd

    PrintVector("x", callSplineInterp(xx1, coefficients, knots,
    ninterv, order, mult, maxderiv, result, ncps), 3*ncps);

    PrintVector("y", callSplineInterp(yy1, coefficients, knots,
    ninterv, order, mult, maxderiv, result, ncps), 3*ncps);


    // print coefficients table
    PrintVector("coef1", coefficients, ncoef);


    FreeDoubleMatrix(lic);
    FreeDoubleMatrix(ltc);
    FreeDoubleMatrix(lfc);
    free(istate);
    free(clambda);
    free(R);
    free(bps);
    free(coefficients);
    int index2;
    for(index2 = 0; index2 < NOUT; ++index2) {
        free(knots[index2]);
    }
    free(result);

    return 0;
}




void ucf(int *mode, int *nstate, int *i, double *f, double *df, double
**zp){
      float lit = .00001;
    if (*mode == 0 || *mode == 2){
```

```
        *f = pow(xd, 2.0) + pow(yd, 2.0) + pow((xd*ydd-
yd*xdd)/(pow(xd,2)+pow(yd,2) +lit),2);
    }
    if (*mode == 1 || *mode == 2){

        df[0] = 0;
        df[1] = 2*xd-2*(xdd*yd-xd*ydd)*((-
ydd)*(pow(xd,2))+2*xd*xdd*yd+pow(yd,2)*ydd)/(pow(pow(xd,2)+pow(yd,2),3)
+lit);      // df/dx1d
        df[2] = 2*yd*(xdd*yd-xd*ydd)/(pow(pow(xd,2)+pow(yd,2),2)+lit);
      df[3] = 0;
        df[4] = 2*yd+2*(xdd*yd-xd*ydd)*(xdd*pow(xd,2)+2*xd*yd*ydd-
xdd*pow(yd,2))/(pow(pow(xd,2)+pow(yd,2),3)+lit);      // df/dy1d
        df[5] = 2*xd*(xd*ydd-xdd*yd)/(pow(pow(xd,2)+pow(yd,2),2)+lit);
    }
}
void nltcf(int* mode, int* nstate, int* i, double* f, double** df,
double** zp){
    float lit = .00001;
    if (*mode == 0 || *mode == 2){

            f[0] = pow(xd,2) + pow(yd,2);
      f[1] = (xd*ydd-yd*xdd)/(pow(xd,2) + pow(yd,2)+lit);
      f[2] = pow(x - xob, 2.0) + pow(y - yob, 2.0);

    }
    if (*mode == 1 || *mode == 2) {

            df[0][0] = 0;
        df[0][1] = 2*xd;
      df[0][2] = 0;
        df[0][3] = 0;
        df[0][4] = 2*yd;
      df[0][5] = 0;

      df[1][0] = 0;
        df[1][1] = ((-
ydd)*pow(xd,2)+2*xd*xdd*yd+pow(yd,2)*ydd)/(pow(pow(xd,2)+pow(yd,2),2)+l
it);
      df[1][2] = (-yd)/(pow(xd,2)+pow(yd,2)+lit);
        df[1][3] = 0;
        df[1][4] = ((-xdd)*pow(xd,2)-
2*xd*yd*ydd+xdd*pow(yd,2))/(pow(pow(xd,2)+pow(yd,2),2)+lit);
      df[1][5] = (xd)/(pow(xd,2)+pow(yd,2)+lit);

      df[2][0] = 2 * x - 2 * xob;   // /dx1
        df[2][1] = 0;
      df[2][2] = 0;
        df[2][3] = 2 * y - 2 * yob; // /dy1
        df[2][4] = 0;
      df[2][5] = 0;



    }
}
```

```
double* callSplineInterp(int xxyy, double* coefficients, double**
knots,
    int* ninterv, int* order, int* mult, int* maxderiv,
    double* result, int ncps) {

    double coefs[nCOEF];  // array to contain 15 coefficients for one
variable
    int i1, i2;
    double fz[3];
    double* time = malloc(ncps * sizeof(double));


    for (i1 = 0; i1 < nCOEF; ++i1) {            // take 15 coefficients
        coefs[i1] = coefficients[i1 + xxyy - nCOEF];
    }

    // calling the SplineInterp function
    for (i2 = 0; i2 < ncps; ++i2) {

        time[i2] = TIMESPAN * (double)(i2) / (double)(ncps - 1);
        SplineInterp(fz, time[i2], knots[0], ninterv[0], coefs, nCOEF,
order[0],
        mult[0], maxderiv[0]);    // compute the values of interpretted
outputs

        // merging all outputs in one table
        result[i2] = fz[0];                 // position values e.g. x
        result[i2 + ncps] = fz[1];          // 1st derivative e.g. xd
        result[i2 + ncps + ncps] = fz[2];   // 2nd derivative e.g. xdd
    }

    free(time);

    return result;

}
```

APPENDIX V. PLAYER/STAGE/NTG DIFF. DRIVE ROBOT CODE

Obstacles2.c

```
#include <stdlib.h>
#include <math.h>
#include "ntg.h"

#define NOUT            2
#define nINTERV         10
#define NINTERV         nINTERV, nINTERV
#define mULT            4
#define MULT            mULT, mULT
#define oRDER           6
#define ORDER           oRDER, oRDER
#define mAXDERIV  3
#define MAXDERIV        mAXDERIV, mAXDERIV

#define nCOEF           24          //NINTERV * (ORDER - MULT) + MULT
#define NCOEF           2 * nCOEF
#define NBPS            (NCOEF + 6)
#define NUAV            1           //number of robots
#define NVAR            3 * 2 * NUAV

// Label outputs and their derivatives

                                    // active variable sequence
#define x              zp[0][0]    // 0   horizontal pose of robot
#define xd             zp[0][1]    // 1   horizontal speed of robot
#define xdd       zp[0][2]    // 2

#define y              zp[1][0]    // 3   vertical location of robot
#define yd             zp[1][1]    // 4   vertical speed of robot
#define ydd       zp[1][2]    // 5


/* number of linear constraints */
#define NLIC         2          // linear initial constraints
#define NLTC         2        // linear traj. constraints
#define NLFC         2        // linear final constraints

/* number of nonlinear constraints */
#define NNLIC         0       // nonlinear initial constraints
#define NNLTC           2     // no obstacle
#define NNLTC_OB     3        // if an obstacle appears
#define NNLFC         0       // nonlinear final constraints

//nonlinear constraints function active variables
#define NINITIALCONSTRAV     0
#define NTRAJECTORYCONSTRAV  4    /* xd, xdd, yd, ydd*/
#define NTRAJECTORYCONSTRAV_OB  6 /* x, xd, xdd, y, yd, ydd*/
#define NFINALCONSTRAV       0

//number of cost functions
#define NICF      0           // initial cost function
```

```
#define NUCF      1          // unintegrated (trajectory) cost
function
#define NFCF      0          // final cost function

//active variables for cost function
#define NINITIALCOSTAV     0
#define NTRAJECTORYCOSTAV  4   /* xd, xdd, yd, ydd */
#define NFINALCOSTAV       0

// enum x, y are used to divide the coefficients table into 2 portions
enum {xx1 = nCOEF, yy1 = 2*nCOEF};



/* declare all of the active variables required for the nonlinear
constraints and the cost functions */
static AV trajectoryconstrav[NTRAJECTORYCONSTRAV]={{0,1}, {0,2}, {1,1},
{1,2}}; // xd, xdd, yd, ydd
static AV trajectoryconstrav_OB[NTRAJECTORYCONSTRAV_OB]={{0,0}, {0,1},
{0,2}, {1,0}, {1,1}, {1,2}}; // x, xd, xdd, y, yd, ydd
static AV trajectorycostav[NTRAJECTORYCOSTAV]={{0,1}, {0,2}, {1,1},
{1,2}}; // xd, xdd, yd, ydd


/* declare the function used to compute the unintegrated cost */
void ucf(int *mode, int *nstate, int *i, double *f, double *df, double
**zp);

/* declare nonlinear traj. constraint function */
void nltcf(int* mode, int* nstate, int* i, double* f, double** df,
double** zp);
void nltcf_OB(int* mode, int* nstate, int* i, double* f, double** df,
double** zp);

// declare spline interp function, this function returns an ncps*3
array
double* callSplineInterp(int xxyy, double* coefficients, double**
knots,
  int* ninterv, int* order, int* mult, int* maxderiv, double* result,
int ncps);


  float xob = 0;
  float yob = 0;
  float radius = 0;
  double goalxlow = 0;
  double goalxhigh = 0;
  double goalylow = 0;
  double goalyhigh = 0;
  double xStart = 0;
  double yStart = 0;
  int TIMESPAN = 250;
  int TIMESPAN_OB = 250;
int main(void)
{
```

```
/* get obstacle information*/
  int i;
  char * file_name =
"/home/ryanfrazier/Player_Stage_Projects/ER1Projects/Player_NTG/Dynamic
Dodge2/control/Obstacle_Info.txt";
  float ObsInfo[7];
  int IsObs;

  FILE *fp;
  fp=fopen(file_name,"r");
  if (fp == NULL) {
    printf("Fail \n");
    exit(0);
  }
  //in the form xob, yob, radius, startx, starty
  while (fscanf(fp, "%f %f %f %f %f %f %f\n", &ObsInfo[0], &ObsInfo[1],
&ObsInfo[2], &ObsInfo[3], &ObsInfo[4], &ObsInfo[5], &ObsInfo[6]) ==7)
  fclose(fp);
    xStart = (double)(ObsInfo[3]);
    yStart = (double)(ObsInfo[4]);
    goalxlow = (double)(ObsInfo[5])-2.5;
    goalxhigh = (double)(ObsInfo[5]) + 2.5;
    goalylow = (double)(ObsInfo[6]) - 2.5;
    goalyhigh = (double)(ObsInfo[6]) + 2.5;


  IsObs = 1;
  if (ObsInfo[2] == 0) {  //we know that if the radius of the obstacle
is 0, there is no obstacle
    IsObs = 0;
  }
  printf("Is there an Obs?  %d \n", IsObs);

  if (IsObs) {
    xob = ObsInfo[0];
    yob = ObsInfo[1];
    radius = ObsInfo[2];
  }



    double* knots[NOUT];
    int ninterv[NOUT] = {NINTERV};
    int mult[NOUT] = {MULT};
    int maxderiv[NOUT] = {MAXDERIV};
    int order[NOUT] = {ORDER};

    // result is used to store the interpretted splines results
    int ncps = 100;      // 100 points per variable, e.g. 100 x values
    double* result;      // global variable used to store interpreted
splines
    result = malloc(ncps * 3 * sizeof(double));
```

```
    int nbps;           // number of break points
    double* bps;

    double** lic;
    double** lfc;
    double** ltc;

    int ncoef;
    double* coefficients;
    int* istate;
    double* clambda;
    double* R;

    // upper and lower bounds
    double lowerb[NLIC + NLTC + NLFC + NNLIC + NNLTC + NNLFC];
    double upperb[NLIC + NLTC + NLFC + NNLIC + NNLTC + NNLFC];

    double lowerb_OB[NLIC + NLTC + NLFC + NNLIC + NNLTC_OB + NNLFC];
    double upperb_OB[NLIC + NLTC + NLFC + NNLIC + NNLTC_OB + NNLFC];


    int inform =0;
    double objective;

    // allocate space and initialize the knot points
    int index;

    if (!IsObs) {
    for(index = 0; index < NOUT; ++index){
        knots[index] = malloc((ninterv[index] + 1) * sizeof(double));
        linspace(knots[index], 0, TIMESPAN, ninterv[index] + 1);
    }
    }

    if (IsObs) {
    for(index = 0; index < NOUT; ++index){
        knots[index] = malloc((ninterv[index] + 1) * sizeof(double));
        linspace(knots[index], 0, TIMESPAN_OB, ninterv[index] + 1);
    }
    }

    // WARNING, specific for this case

    ncoef = NCOEF;
    coefficients = malloc(ncoef * sizeof(double));

    // initial guess for coefficients
    linspace(coefficients, 1, 1, ncoef);

    // allocate space for breakpoints and initialize
    nbps = NBPS;
    bps = malloc(nbps * sizeof(double));

    if (!IsObs) {
    linspace(bps, 0, TIMESPAN, nbps);     // 20 is fixed optimal traj.
time
    }
```

```
    if (IsObs) {
    linspace(bps, 0, TIMESPAN_OB, nbps);  // 20 is fixed optimal traj.
time
    }

    /*
     * ntg internal memory
     * these calculations do not need to be changed
     */

    if (!IsObs) {
    istate = malloc((ncoef + NLIC + NLFC + NLTC * nbps
            + NNLIC + NNLTC * nbps + NNLFC) * sizeof(int));

    clambda = malloc((ncoef + NLIC + NLFC + NLTC * nbps
            + NNLIC + NNLTC * nbps + NNLFC) * sizeof(double));
    }

    if (IsObs) {
    istate = malloc((ncoef + NLIC + NLFC + NLTC * nbps
            + NNLIC + NNLTC_OB * nbps + NNLFC) * sizeof(int));

    clambda = malloc((ncoef + NLIC + NLFC + NLTC * nbps
            + NNLIC + NNLTC_OB * nbps + NNLFC) * sizeof(double));
    }

    R = malloc((ncoef + 1) * (ncoef + 1) * sizeof(double));




    lic = DoubleMatrix(NLIC, NVAR);
    ltc = DoubleMatrix(NLTC, NVAR); // 6 = 3 * 2 * number of uavs
    lfc = DoubleMatrix(NLFC, NVAR);


    /* define the constraints
     * for linear constraints:
     * indexed by the active variable number
     * [constraint number][a.v. number]
     */
   if (!IsObs) {
    // 2 lic
    lic[0][0] = 1;            // first 0 is the sequence number of the
constraint
                              // second 0 is the sequence number of the
a.v.(x here)
                              // 1 is the coefficient of this a.v.
    lowerb[0] = upperb[0] = xStart;
    lic[1][3] = 1; lowerb[1] = upperb[1] = yStart;

    // 2 ltc
    ltc[0][0] = 1; lowerb[2] = 0; upperb[2] = 100;
    ltc[1][3] = 1; lowerb[3] = 0; upperb[3] = 100;
```

```
    // 2 lfc
    lfc[0][0] = 1; lowerb[4] = goalxlow; upperb[4] = goalxhigh;
    lfc[1][3] = 1; lowerb[5] = goalylow; upperb[5] = goalyhigh;


    //Nonlinear Initial Constraints
    //None

    //Nonlinear Trajectory Constraints
    lowerb[6] = 0; upperb[6] = .25;
    lowerb[7] = -3.14/2; upperb[7] = 3.14/2;
   }

    if (IsObs) {
        // 2 lic
    lic[0][0] = 1;            // first 0 is the sequence number of the
constraint
                              // second 0 is the sequence number of the
a.v.(x here)
                              // 1 is the coefficient of this a.v.
    lowerb_OB[0] = upperb_OB[0] = xStart;       value of the constraint
    lic[1][3] = 1; lowerb_OB[1] = upperb_OB[1] = yStart;

    // 2 ltc
    ltc[0][0] = 1; lowerb_OB[2] = 0; upperb_OB[2] = 100;
    ltc[1][3] = 1; lowerb_OB[3] = 0; upperb_OB[3] = 100;


    // 2 lfc
    lfc[0][0] = 1; lowerb_OB[4] = goalxlow; upperb_OB[4] = goalxhigh;
    lfc[1][3] = 1; lowerb_OB[5] = goalylow; upperb_OB[5] = goalyhigh;


    //Nonlinear Initial Constraints
    //None

    //Nonlinear Trajectory Constraints
    lowerb_OB[6] = 0; upperb_OB[6] = .25;
    lowerb_OB[7] = -3.14/2; upperb_OB[7] = 3.14/2;

    lowerb_OB[8] = pow(radius,2); upperb_OB[8] = pow(100,2);
    }


    //Nonlinear Final Constraints
    //None

    //Nonlinear Final Constraints
    //None




    // the ntg subroutine:


    npsoloption("summary file = 0");
```

```
    if (!IsObs) {
    ntg(NOUT, bps, nbps, ninterv, knots, order, mult, maxderiv,
            coefficients,
            NLIC,                   lic,
            NLTC,                   ltc,
            NLFC,                   lfc,
            NNLIC,                  NULL,
            NNLTC,                  nltcf,
            NNLFC,                  NULL,
            NINITIALCONSTRAV,       NULL,
            NTRAJECTORYCONSTRAV,      trajectoryconstrav,
            NFINALCONSTRAV,         NULL,
            lowerb,                 upperb,
            NICF,                   NULL,
            NUCF,                   ucf,
            NFCF,                   NULL,
            NINITIALCOSTAV,         NULL,
            NTRAJECTORYCOSTAV,      trajectorycostav,
            NFINALCOSTAV,           NULL,
            istate, clambda, R, &inform, &objective
            );
    }
    if (IsObs) {
          ntg(NOUT, bps, nbps, ninterv, knots, order, mult, maxderiv,
            coefficients,
            NLIC,                   lic,
            NLTC,                   ltc,
            NLFC,                   lfc,
            NNLIC,                  NULL,
            NNLTC_OB,                  nltcf_OB,
            NNLFC,                  NULL,
            NINITIALCONSTRAV,       NULL,
            NTRAJECTORYCONSTRAV_OB,      trajectoryconstrav_OB,
            NFINALCONSTRAV,         NULL,
            lowerb_OB,                 upperb_OB,
            NICF,                   NULL,
            NUCF,                   ucf,
            NFCF,                   NULL,
            NINITIALCOSTAV,         NULL,
            NTRAJECTORYCOSTAV,      trajectorycostav,
            NFINALCOSTAV,           NULL,
            istate, clambda, R, &inform, &objective
            );
    }



    printf("Out from ntg call, now using the PrintVector \n");
    printf("xStart:  %f, yStart:  %f,  xob:  %f,  yob:  %f,
radius:  %f, xGoal: %f, yGoal:  %f \n",xStart, yStart, xob, yob,
radius, goalxlow, goalylow);

    // call spline interp and print out to file for x, y
    // e.g. the file x contains 100 x values
    PrintVector("x", callSplineInterp(xx1, coefficients, knots,
    ninterv, order, mult, maxderiv, result, ncps), ncps);
```

```
    PrintVector("y", callSplineInterp(yy1, coefficients, knots,
    ninterv, order, mult, maxderiv, result, ncps), ncps);


    // print coefficients table
    PrintVector("coef1", coefficients, ncoef);


    FreeDoubleMatrix(lic);
    FreeDoubleMatrix(ltc);
    FreeDoubleMatrix(lfc);
    free(istate);
    free(clambda);
    free(R);
    free(bps);
    free(coefficients);
    int index2;
    for(index2 = 0; index2 < NOUT; ++index2) {
        free(knots[index2]);
    }
    free(result);

    return 0;
}




void ucf(int *mode, int *nstate, int *i, double *f, double *df, double
**zp){
      //printf("In the ucf loop, i = %i, nstate = %i \n", *i, *nstate);

    float lit = .00001;
    if (*mode == 0 || *mode == 2){
        *f = pow(xd, 2.0) + pow(yd, 2.0) + pow((xd*ydd-
yd*xdd)/(pow(xd,2)+pow(yd,2) +lit),2);
    }
    if (*mode == 1 || *mode == 2){

      df[0] = 0;
        df[1] = 2*xd-2*(xdd*yd-xd*ydd)*((-
ydd)*(pow(xd,2))+2*xd*xdd*yd+pow(yd,2)*ydd)/(pow(pow(xd,2)+pow(yd,2),3)
+lit);      // df/dx1d
        df[2] = 2*yd*(xdd*yd-xd*ydd)/(pow(pow(xd,2)+pow(yd,2),2)+lit);
      df[3] = 0;
        df[4] = 2*yd+2*(xdd*yd-xd*ydd)*(xdd*pow(xd,2)+2*xd*yd*ydd-
xdd*pow(yd,2))/(pow(pow(xd,2)+pow(yd,2),3)+lit);      // df/dy1d
        df[5] = 2*xd*(xd*ydd-xdd*yd)/(pow(pow(xd,2)+pow(yd,2),2)+lit);
    }
}

void nltcf_OB(int* mode, int* nstate, int* i, double* f, double** df,
double** zp){
    float lit = .00001;
    if (*mode == 0 || *mode == 2){
```

```
            f[0] = pow(xd,2) + pow(yd,2);
      f[1] = (xd*ydd-yd*xdd)/(pow(xd,2) + pow(yd,2)+lit);
      f[2] = pow(x - xob, 2.0) + pow(y - yob, 2.0);

    }
    if (*mode == 1 || *mode == 2) {

            df[0][0] = 0;
        df[0][1] = 2*xd;
      df[0][2] = 0;
        df[0][3] = 0;
        df[0][4] = 2*yd;
      df[0][5] = 0;

      df[1][0] = 0;
        df[1][1] = ((-
ydd)*pow(xd,2)+2*xd*xdd*yd+pow(yd,2)*ydd)/(pow(pow(xd,2)+pow(yd,2),2)+l
it);
      df[1][2] = (-yd)/(pow(xd,2)+pow(yd,2)+lit);
        df[1][3] = 0;
        df[1][4] = ((-xdd)*pow(xd,2)-
2*xd*yd*ydd+xdd*pow(yd,2))/(pow(pow(xd,2)+pow(yd,2),2)+lit);
      df[1][5] = (xd)/(pow(xd,2)+pow(yd,2)+lit);

      df[2][0] = 2 * x - 2 * xob;   // /dx1
        df[2][1] = 0;
      df[2][2] = 0;
        df[2][3] = 2 * y - 2 * yob; // /dy1
        df[2][4] = 0;
      df[2][5] = 0;
    }
}

void nltcf(int* mode, int* nstate, int* i, double* f, double** df,
double** zp){
    float lit = .00001;
    if (*mode == 0 || *mode == 2){

            f[0] = pow(xd,2) + pow(yd,2);
      f[1] = (xd*ydd-yd*xdd)/(pow(xd,2) + pow(yd,2)+lit);

    }
    if (*mode == 1 || *mode == 2) {

            df[0][0] = 0;
        df[0][1] = 2*xd;
      df[0][2] = 0;
        df[0][3] = 0;
        df[0][4] = 2*yd;
      df[0][5] = 0;

      df[1][0] = 0;
        df[1][1] = ((-
ydd)*pow(xd,2)+2*xd*xdd*yd+pow(yd,2)*ydd)/(pow(pow(xd,2)+pow(yd,2),2)+l
it);
      df[1][2] = (-yd)/(pow(xd,2)+pow(yd,2)+lit);
```

```
        df[1][3] = 0;
        df[1][4] = ((-xdd)*pow(xd,2)-
2*xd*yd*ydd+xdd*pow(yd,2))/(pow(pow(xd,2)+pow(yd,2),2)+lit);
      df[1][5] = (xd)/(pow(xd,2)+pow(yd,2)+lit);
    }
}

double* callSplineInterp(int xxyy, double* coefficients, double**
knots,
    int* ninterv, int* order, int* mult, int* maxderiv,
    double* result, int ncps) {

    double coefs[nCOEF];  // array to contain 15 coefficients for one
variable
    int i1, i2;
    double fz[3];
    double* time = malloc(ncps * sizeof(double));


    for (i1 = 0; i1 < nCOEF; ++i1) {            // take 15 coefficients
        coefs[i1] = coefficients[i1 + xxyy - nCOEF];
    }

    // calling the SplineInterp function
    for (i2 = 0; i2 < ncps; ++i2) {

        time[i2] = TIMESPAN * (double)(i2) / (double)(ncps - 1);
        SplineInterp(fz, time[i2], knots[0], ninterv[0], coefs, nCOEF,
order[0],
        mult[0], maxderiv[0]);    // compute the values of interpretted
outputs

        // merging all outputs in one table
        result[i2] = fz[0];                 // position values e.g. x1
        result[i2 + ncps] = fz[1];          // 1st derivative e.g. x1d
        result[i2 + ncps + ncps] = fz[2];   // 2nd derivative e.g. x1dd
    }

    free(time);

    return result;

}
```

DynamicDodge2.cc

```
#include <stdio.h>
#include <iostream>
#include <time.h>
#include <math.h>
#include <fstream>
#include <libplayerc++/playerc++.h>
#include "/home/ryanfrazier/NTG2.2+/examples/globals.h"

#include "args.h"

#define RAYS 32
#define pi 3.141592653589793238

using namespace std;

void ReadAndDisplay(char * file_name, int NCPS, int numLength, double *
Result);
void RecalculatePath( double * Resultx, double * Resulty, double *
Resultth, int length, float range, float xc, float yc, float tc, int
NCPS, int numLength, float xGoal, float yGoal);

int main(int argc, char **argv) {
  parse_args(argc, argv);

  try {
  using namespace PlayerCc;

  PlayerClient er1(gHostname, gPort);
  Position2dProxy p2d(&er1, gIndex);
  BlobfinderProxy blb(&er1, gIndex);

  p2d.SetMotorEnable(1);


  //get points from user
  int NumGoals;
  int GoalIncrement = 0;
  float InitialX, InitialY;
  printf("Enter the number of goals (and press ENTER) ");
  std::cin >> NumGoals;
  printf("Enter the inital x position (and press ENTER) ");
  std::cin >> InitialX;
  printf("Enter the inital y position (and press ENTER) ");
  std::cin >> InitialY;
  float XGOALS[NumGoals];
  float YGOALS[NumGoals];
  for (GoalIncrement = 0; GoalIncrement < NumGoals; GoalIncrement++) {
    printf("Enter the x coordinate of the %d  goal \n", GoalIncrement);
    std::cin >> XGOALS[GoalIncrement];
    printf("Enter the y coordinate of the %d  goal \n", GoalIncrement);
    std::cin >> YGOALS[GoalIncrement];
  }
  GoalIncrement = 0;
  //end initialization from user
```

```
  //make a file with xob, yob, radius, xccurrent, ycurrent, current x
goal, current y goal
  //initialize with no obstacle (0 0 0 * * * *)
  char * file_name = "Obstacle_Info.txt";
  std::ofstream out(file_name);
  out << 0 << " " << 0 << " " << 0 << " " << InitialX << " " <<
InitialY << " " << XGOALS[0] << " " << YGOALS[0] << endl;
  system("(/home/ryanfrazier/NTG2.2+/examples/Obstacles2/multipoint)");
  //end calculating initial points


  playerc_blobfinder_blob_t Obstacle;
  int NCPS;
  int i; //for counting through the file
  int numLength;
  numLength=25;
  NCPS=100;
  double Resultx[NCPS];
  double Resulty[NCPS];
  double Resultth[NCPS];

  //read in initial points
  ReadAndDisplay("x", NCPS, numLength, Resultx);
  ReadAndDisplay("y", NCPS, numLength, Resulty);
  //Estimate Theta
  for (i=0; i < NCPS; i++) {
    Resultth[i]=atan2((Resulty[i+1]-Resulty[i]),(Resultx[i+1]-
Resultx[i]));
  }
  Resultth[0]=0;
  //end reading inital points


  //set up the proportional go to driver
  float xerr, yerr, terr, perr, aerr, berr;
  float Xg, Yg, Tg;   //goal vars
  float Xc, Yc, Tc;   //current vars
  float thresh = .1;
  float offset = .2032;
  float kp, ka, kb;
  int color = 0;
  kp=.5;
  ka=1;
  kb=-.5;  //these values work!

  double fSpeed, tSpeed;
  Xg=Resultx[0];
  Yg=Resulty[0];
  Tg=Resultth[0];
  //end set up


  i=0;
  while (1) {
    er1.Read();
```

```
    Xc=p2d.GetXPos();
    Yc=p2d.GetYPos();
    Tc=p2d.GetYaw();


    if (blb.GetCount() > 0 ) {
      Obstacle = blb.GetBlob(0);
      if (color != Obstacle.color && Obstacle.area > 200){
      color = Obstacle.color;
        //sleep(1);
        p2d.SetSpeed(0,0);
        er1.Read();
        Xc=p2d.GetXPos();
        Yc=p2d.GetYPos();
        Tc=p2d.GetYaw();
        i = 0;
        RecalculatePath(Resultx, Resulty, Resultth, Obstacle.bottom,
Obstacle.range,Xc, Yc, Tc, NCPS, numLength, XGOALS[GoalIncrement],
YGOALS[GoalIncrement]);
      }
    }



    xerr = Xg-Xc;
    yerr = Yg-Yc;
    terr = Tg-Tc;


    //if(abs(xerr/thresh) < 1) {
      //if(abs(yerr/thresh) < 1) {
      //if(abs(terr/thresh) < 1) {
        //break;
      //}
      //}
    //}

    perr=sqrt(xerr*xerr+yerr*yerr);
    aerr=atan2(yerr,xerr)-Tc;
    berr=Tg-Tc-aerr;

    if (aerr < -3.14159)
      aerr=aerr+2*3.14159;
    if (aerr > 3.14159)
      aerr=aerr-2*3.14159;

    if (berr < -3.14159)
      berr=berr+2*3.14159;
    if (berr > 3.14159)
      berr=berr-2*3.14159;

    fSpeed=kp*perr;
    tSpeed=ka*aerr+kb*berr;
```

```
    if (fSpeed < .5) {
      i++;
      if (i == NCPS) {
      GoalIncrement++;
      if (GoalIncrement == NumGoals) { //break if there are no more
goal points
        printf("breaking");
        break;
      }
      std::ofstream out(file_name);
      out << 0 << " " << 0 << " " << 0 << " " << XGOALS[GoalIncrement-
1] << " " << YGOALS[GoalIncrement-1] << " " << XGOALS[GoalIncrement] -
2.5 << " " << YGOALS[GoalIncrement] -2.5 << endl; //if there are more
goal points, run NTG again with the new points
      system("(/home/ryanfrazier/NTG2.2+/examples/Obstacles2/multipoint
)");
      ReadAndDisplay("x", NCPS, numLength, Resultx);
      ReadAndDisplay("y", NCPS, numLength, Resulty);
      //Estimate Theta
      for (i=0; i < NCPS; i++) {
        Resultth[i]=atan2((Resulty[i+1]-Resulty[i]),(Resultx[i+1]-
Resultx[i]));
      }
      Resultth[NCPS-1]=Resultth[NCPS-2];
      Resultth[0]=0;
      i = 0;
      }
      Xg=Resultx[i];
      Yg=Resulty[i];
      Tg=Resultth[i];

      printf(" %f  %f  %f  \n",Xg, Yg, Tg);
    }

    p2d.SetSpeed(fSpeed,tSpeed);

  }

  return 0;
  }
  catch (PlayerCc::PlayerError & e)
  {
    std::cerr << e << std::endl;
    return -1;
  }

}

void ReadAndDisplay(char * file_name, int NCPS, int numLength, double *
Result) {
  int i;
  int numChar;
  char * pch;
  numChar=NCPS*numLength;
  char input_line[numChar];

  i=0;
```

```
  ifstream file_in(file_name);
  file_in.getline(input_line,numChar);
  file_in.close();
  pch = strtok (input_line," ");
  while (pch != NULL) {
    istringstream os(pch);
    os >> Result[i];
    //printf("Result in loop %f \n",Result[i]);
    pch = strtok (NULL, " ");
    i++;
  }

}

void RecalculatePath(double * Resultx, double * Resulty, double *
Resultth, int length, float range, float xc, float yc, float tc, int
NCPS, int numLength, float xGoal, float yGoal) {
  float xob, yob, radius;
  int i;
  char * file_name = "Obstacle_Info.txt";
  int numChar = 9;  //change if each number is not just 2 digits
  char * pch;
  char input_line[numChar];
  float newInfo[3];


  //find the center and radius of the obstacle
  xob=range*cos(tc)+xc;
  yob = range*sin(tc) + yc;
  /*FIGURE OUT A WAY TO FIND RADIUS!!!!*/
  radius = length*range*.57735/60;


  //write to a file
  std::ofstream out(file_name);
  out << xob << " " << yob << " " << radius << " " << xc << " " << yc
<< " " << xGoal << " " << yGoal << endl;

  system("(/home/ryanfrazier/NTG2.2+/examples/Obstacles2/multipoint)");

  ReadAndDisplay("x", NCPS, numLength, Resultx);
  ReadAndDisplay("y", NCPS, numLength, Resulty);

   for (i=0; i < NCPS; i++) {

    Resultth[i]=atan2((Resulty[i+1]-Resulty[i]),(Resultx[i+1]-
Resultx[i]));
    //printf("(%f ,  %f ,  %f ) \n", Resultx[i], Resulty[i],
Resultth[i]);
    }
    Resultth[NCPS-1] = Resultth[NCPS - 2];
    Resultth[0]=0;

}
```

Configuration File

```
driver
(
  name "stage"   #there is a driver named stage
  plugin "stageplugin"     #the driver is in the stageplugin library

  provides ["simulation:0"]

  #Load the named file into the simulator
  worldfile "rink_er1.world"  #this is a world file stored in the same
place as empty.cfg
)

driver (
  name "stage"
  provides ["odometry::6665:position2d:0" "6665:blobfinder:0"]
  model "erone1"
)
```

World File

```
include "/usr/local/share/stage/worlds/map.inc"  #this gets me the
include file map
include "erone.inc"

#configure the GUI window
window (
  size [700.000 700.000]    #size of the window in pixels
  scale 7         #pixels/meter   this value is (window
size)/(floorplan size)

  show_data 1
)

obstacle(
  pose [12 20 0 150]
  #color "grey"   #so the robot can't see it
  color "gray30"  # so that the robot can see it
)
obstacle(
  pose [60 75 0 30]
  color "green"
  #color "grey" #so the robot won't see the obstacle
)

#make an instance of the er1 robot
erone(
  name "erone1"
  pose [4 4 0 0]
  color "red"
)
```

Inc file

```
#make the model for the er1  (named erone)
```

```
#make a position model for the er1 with wheels and shape
define erone position (
  #actual size of the er1 robot (not trailer)
  size [0.4064 0.381 0.6096]  #erone is scaled to fit in this box
                        #center is [.2032 .1905 .3048]
  #shape of erone
  block(
    points 4
    point[3] [0 0]
    point[2] [0 16]
    point[1] [15 16]
    point[0] [15 0]
    z [0 24]  #how tall he is
  )

  block (
    points 4
    point[3] [15 7]
    point[2] [18 7]
    point[1] [18 9]
    point[0] [15 9]
    z [0 24]  #how tall he is
  )

  #positional descriptions for erone wheels
  drive "diff"    #differential drive robot
  localization "gps"  #knows its location perfectly
  localization_origin [0 0 0 0]

  #attach sensors
  erone_blob()

  #what can erone sense

  blob_return 1
  #obstacle_return 1
  obstacle_return 0
)

define erone_blob blobfinder (
    colors_count 2
    colors ["gray30" "green"]
    range 10.0
    image [80 60]
    size [0.1 0.07 0.05]
    color "black"
)


define obstacle position (
size [5 1 1]
block (
  points 4
  point[3] [0 0]
  point[2] [0 16]
  point[1] [15 16]
  point[0] [15 0]
```

```
  z [0 24]
)
)
```

APPENDIX VI. NTG PROGRAMMING PARAMETERS

TABLE II

NTG PARAMETERS FOR SIMPLE SCENARIO

Parameter	**Value**	**Meaning**
NOUT	2	Two Flat Outputs (x,y) = (z_1,z_2)
NINTERV	10	Each interval is 3 seconds
MULT	4	The derivatives of each B-Spline at the intersection of the intervals must match to the 4th order
ORDER	6	6th Order B-Splines are used
MAXDERIV	3	At least the maximum derivative used (1)
NCOEF	48	Number of coefficients used for the B-Splines. This is equal to: NINTERV*(ORDER-MULT)+MULT
NBPS	54	Number of breakpoints over the course of 30 seconds
NVAR	6	Number of variables possibly used (x, y, and their first and second derivatives)
NLIC	2	Two linear initial constraints
NLTC	4	Four linear trajectory constraints
NLFC	2	Two linear final constraints

NNLIC	0	No nonlinear initial constraints
NNLTC	1	One nonlinear trajectory constraint
NNLFC	0	No nonlinear final constraints
NINITIALCONSTRAV	0	No active variables for nonlinear initial constraints
NTRAJECTORYCONSTRAV	2	Two active variables for nonlinear trajectory constraints (x and y)
NICF	0	No initial cost function
NUCF	1	One trajectory cost function
NFCF	0	No final cost function
NINITIALCOSTAV	0	No active variables for the initial cost function
NTRAJECTORYCOSTAV	2	Two active variables for the trajectory cost function ($\dot{x}$ and $\dot{y}$)
NFINALCOSTAV	0	No active variables for the final cost function
NCPS	100	100 collocation points per variable

TABLE III

NTG PARAMETERS FOR THE DIFFERENTIAL DRIVE EXAMPLE

Parameter	**Value**	**Meaning**
NOUT	2	Two Flat Outputs $(x,y)=(z_1,z_2)$

NINTERV	10	Number of intervals over the course of 300 seconds
MULT	4	The derivatives of the B-Spline functions must match to the fourth derivative at the interval points
ORDER	6	The B-Spline polynomials are sixth order
MAXDERIV	3	Must be at least the highest derivative used (2)
NCOEF	48	Number of coefficients used for the B-Spline
NBPS	54	Number of breakpoints in the course of 300 seconds
NVAR	6	Number of variables (x, y, and their first and second derivatives)
NLIC	2	Two linear initial constraints
NLFC	2	Two linear trajectory constraints
NLTC	2	Two linear final constraints
NNLIC	0	No nonlinear initial constraints
NNLTC	3	Three nonlinear trajectory constraints
NNLFC	0	No nonlinear final constraints
NINITIALCONSTRAV	0	No active variables for nonlinear initial constraints

NTRAJECTORYCONSTRAV	6	Six active variables for nonlinear trajectory constraints (all variables)
NFINALCONSTRAV	0	No active variables for nonlinear final constraints
NICF	0	No initial cost function
NUCF	1	One trajectory cost function
NFCF	0	No final cost function
NINITIALCOSTAV	0	No active variables for initial cost function
NTRAJECTORYCOSTAV	4	Four active variables for trajectory cost function $(\dot{x}, \dot{y}, \ddot{x}, \ddot{y})$
NFINALCOSTAV	0	No final cost function active variables
NCPS	100	100 collocation points per variable

LIST OF REFERENCES

[1] J.-P. Laumond, P. E. Jacobs, M. Taix and R. M. Murray, "A Motion Planner for Nonholonomic Mobile Robots," *IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION,* vol. 10, no. 5, pp. 557-574, October 1994.

[2] M. Linderoth, K. Soltesz and R. M. Murray, "Nonlinear Lateral Control Strategy for Nonholonomic Vehicles," in *American Control Conference*, Seattle, 2008.

[3] DARPA, "DARPA Urban Challenge," [Online]. Available: http://archive.darpa.mil/grandchallenge/. [Accessed 4 February 2013].

[4] M. Veloso and J. Bruce, "Real-Time Randomized Path Planning for Robot Navigation," Carnegie Mellon University, Pittsburgh.

[5] W. B. Dunbart and R. M. Murray, "Model predictive control of coordinated multi-vehicle formations," in *41st Coneference on Decision and Control*, Las Vegas, 2002.

[6] S. S. Ge and Y. J. Cui, "New Potential Functions for Mobile Robot Path Planning," *TRANSACTIONS ON ROBOTICS AND AUTOMATION,,* vol. 16, no. 5, pp. 1042-1047, 2000.

[7] J. Kerr and K. Nickels, "Robot Operating Systems: Bridging the Gap between Human and Robot," in *44th IEEE Southeastern Symposium on System Theory*, Jacksonville, 2012.

[8] M. B. Milam, K. Mushambi and R. M. Murray, "A New Computational Approach to Real-Time Trajectory Generation for Constrained Mechanical Systems," in *Decision and Control*, 2000.

[9] G. Walsh, D. Tilbury, S. Sastry, R. Murray and J. P. Laumond, "Stabilization of Trajectories for Systems with Nonholonomic Constraints," *Transactions on Automatic Control,* vol. 39, no. 1, pp. 216-222, January 1992.

[10] R. Rojas and A. Forster, "Holonomic Control of a Robot with Omnidirectional Drive," *Kunstliche Intelligenz,* 2006.

[11] S. Siegwart, I. Nourbakhsh and D. Skaramuzza, Introduction to Autonomous Mobile Robotics, 2nd ed., Cambridge, Massachusetts: The MIT Press, 2004.

[12] D. Thomas, "Review on Small Mobile Robotics," Louisville, 2012.

[13] R. Riggs, "A ROBOTICS TESTBED: THE DESIGN & IMPLEMENTATION WITH," University of Louisville, Louisville, 2005.

[14] R. T. Vaughan and contributors, "The Stage Robot Simulator," 1998-2011. [Online]. Available: http://rtv.github.com/Stage/index.html ; http://rtv.github.com/Stage/group__model.html. [Accessed 21 2 13].

[15] J. Owen, "How to Use Player/Stage," York, UK, 2010.

[16] R. T. Vaughan and contributors, "The Player/Stage Project: Tools for Multirobot Distributed Sensor Systems," in *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, Coimbra, Portugal, 2003.

[17] B. P. Gerkey and contributors, "The Player Robot Device Server," 1999-2011. [Online]. Available: http://playerstage.sourceforge.net/doc/Player-svn/player/group__tutorial__config.html. [Accessed 21 2 13].

[18] B. P. Gerkey and contributors, "Player," 2006. [Online]. Available: http://playerstage.sourceforge.net/index.php?src=player. [Accessed 21 2 13].

[19] M. Milam, "Real-Time Optimal Trajectory Generation for Constrained Dynamical Systems," California Institute of Technology, Pasadena, 2003.

[20] R. M. Murray, "CDS 270-2: Lecture 3-1 Real-Time Trajectory Generation," Pasadena, 2006.

[21] Evolution Robotics, "Index of /~hamblen/4006/er1," 2004. [Online]. Available: http://users.ece.gatech.edu/~hamblen/4006/er1/er1man12.pdf. [Accessed 14 March 2013].

[22] Contributors, "p_blobfinder.cc not working well, patch attached," March 2012. [Online]. Available: https://github.com/rtv/Stage/issues/21. [Accessed 22 March 2013].

[23] R. Siegwart, "Mobile Robot Kinematics," ETH Zurich, 2008.

[24] Multiple Contributors, "cmvision," 9 May 2011. [Online]. Available: http://www.ros.org/wiki/cmvision. [Accessed 15 March 2013].

[25] M. Flores, "Real-Time Trajectory Generation for Constrained Nonlinear Dynamical SystemsUsing Non-Uniform Rational B-Spline Basis Functions," Pasadena, 2007.

[26] R. Bhatthacharya, M. Flores and S. I. T. Fraga, "Nonlinear Trajectory Generation Workshop," Pasadena , 2003.

[27] J. R. Asensio and L. Montano, "A KINEMATIC AND DYNAMIC MODEL-BASED MOTION CONTROLLER FOR MOBILE ROBOTS," in *15th Triennia World Congress*, Barcelona, 2002.

VITA

Ryan Frazier is a Master's student in the Electrical Engineering Department at UofL. He enjoys robotics, mathematics, and control in academics, and he enjoys running, basketball, studying the Bible in his free time. He was born in 1989 in Louisville, KY and married to Kristina Frazier in 2011. In 2013, he will be working for C&I Engineering in Louisville, KY and will begin researching adoption.