A Virtual Reality Visualization Ofan Analytical Solution Tomobile Robot Trajectory Generationin The Presence Of Moving Obstacles

2007

Ricardo Elias
University of Central Florida

University of
Central Florida

STARS
Showcase of Text, Archives, Research & Scholarship

A VIRTUAL REALITY VISUALIZATION OF
AN ANALYTICAL SOLUTION TO
MOBILE ROBOT TRAJECTORY GENERATION
IN THE PRESENCE OF MOVING OBSTACLES

by

RICARDO ELIAS
B.S. Texas A&M University, 1986

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2007

ABSTRACT

Virtual visualization of mobile robot analytical trajectories while avoiding moving obstacles is presented in this thesis as a very helpful technique to properly display and communicate simulation results. Analytical solutions to the path planning problem of mobile robots in the presence of obstacles and a dynamically changing environment have been presented in the current robotics and controls literature. These techniques have been demonstrated using two-dimensional graphical representation of simulation results. In this thesis, the analytical solution published by Dr. Zhihua Qu in December 2004 is used and simulated using a virtual visualization tool called VRML.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE: INTRODUCTION

This thesis documents the implementation of an analytical path planning algorithm that takes into consideration collision avoidance of stationary and moving obstacles. The simulation is performed using the Virtual Reality Modeling Language toolbox of Matlab and Simulink.

A car that drives without a human driver has been the subject of fiction movies since the beginning of film making. People dream of the moment when you can just tell the car where you want to go and then sit back and relax because the auto pilot will take you there quickly and safely. This auto pilot will probably make better route selection than its human counterpart taking into consideration traffic reports and construction status, thus getting there faster. It will not be distracted by a passenger talking to the driver or by interesting scenery and will keep its many eyes and sensors on the road and surroundings at absolutely all time, thus getting there safer. Technology is not quite at that level, but at the pace of recent advances it should soon be within reach.

In order to accomplish a completely autonomous vehicle, many systems must be integrated into one huge collaborative effort. Systems such as vision, radar, sonar, lidar, GPS, proximity sensors, wireless communications, database, mapping, sensor fusion, software development, fault tolerance, context switching, vehicle dynamics, simulations, engine control, obstacle avoidance, path planning, and others must work together to accomplish the task at hand. These systems have been progressing towards thess goals for the last 15 years and are now beginning to be integrated for the auto pilot function. In October 2005, five vehicles completed a 131 miles course in the Mojave desert without a driver in the DARPA Grand Challenge competition. In

November 2007, several teams will attempt to finish a 60 mile urban course in the DARPA Urban Challenge. These competitions are forcing researchers and designers to push the state of the art of robotics and to think outside the box to overcome current difficulties in the field.

One of these areas that have advanced considerably in the recent years is generation of valid trajectories for the motion of the vehicle. The field of trajectory generation, also known as path Planning, has been the subject of a lot of attention, research, and publications in the past 20 years and it has proven to be a very challenging problem. Many considerations must be taken into account before a trajectory can be generated and by the time the trajectory is actually generated most of these considerations have changed since vehicles operate in a dynamic environment. In order to make the best choice of trajectory, a lot of information about the current state of the vehicle and the surroundings must be known, but also information about the future state of the surroundings should be known. Since the world does not seem to like to follow nice structured rules and laws for changes, it is impossible to completely know ahead of time the changes that are going to occur which would affect the trajectory generation, so we end up ignoring these changes or making estimates of future states. The more realistic approaches should not depend on apriori knowledge of the environment, but should estimate near states with current data and react to changes in the environment.

Trajectory generation is also closely coupled with the obstacle avoidance problem. Simply expressed, all obstacles, static and dynamic, must be avoided even when their status is changing. A good path planner should generate optimal trajectories while avoiding obstacles and react to changes in the environment. Many approaches have been developed, all with their own merits,

to solve the path planning and obstacles avoidance problem, some numerically intensive and others more analytical in nature. Chapter 2 provides a brief summary of several methods that have been developed. In this thesis, we look at one analytical approach developed by Dr. Zhihua Qu that simultaneously solves the path planning and collision avoidance problems. As a proof of concept, simulations with a virtual reality visualization are performed.

Path planning is a very specific task of an autonomous vehicle. The term path planning should not be confused with map routing, which takes a mission goal and divides it into waypoints that are closer together and are located in well defined and understood locations. That first level of path planning is more of a mapping problem without regard to vehicle kinematics and dynamics or obstacles. It is more concerned with solving a trajectory problem by finding a route to go from point A to point B. Once this route has been determined and the proper way points have been generated, the path planner's task is to plan a way of moving from current location to the next waypoint by making a well behaved motion profile (following the vehicle's kinematics and dynamics), staying within boundaries (lane or drivable terrain) and avoiding obstacles (moving or static). This motion profile is passed to lower level control devices that will actuate the vehicle to generate desired motion. To better understand the role of path planning, Figure 1 shows a sample autonomous vehicle control hierarchy. This hierarchy applies to all kinds of vehicles; ground, aerial, or underwater, as well as holonomic and non-holonomic vehicles. The type of vehicle considered in this thesis is non-holonomic.
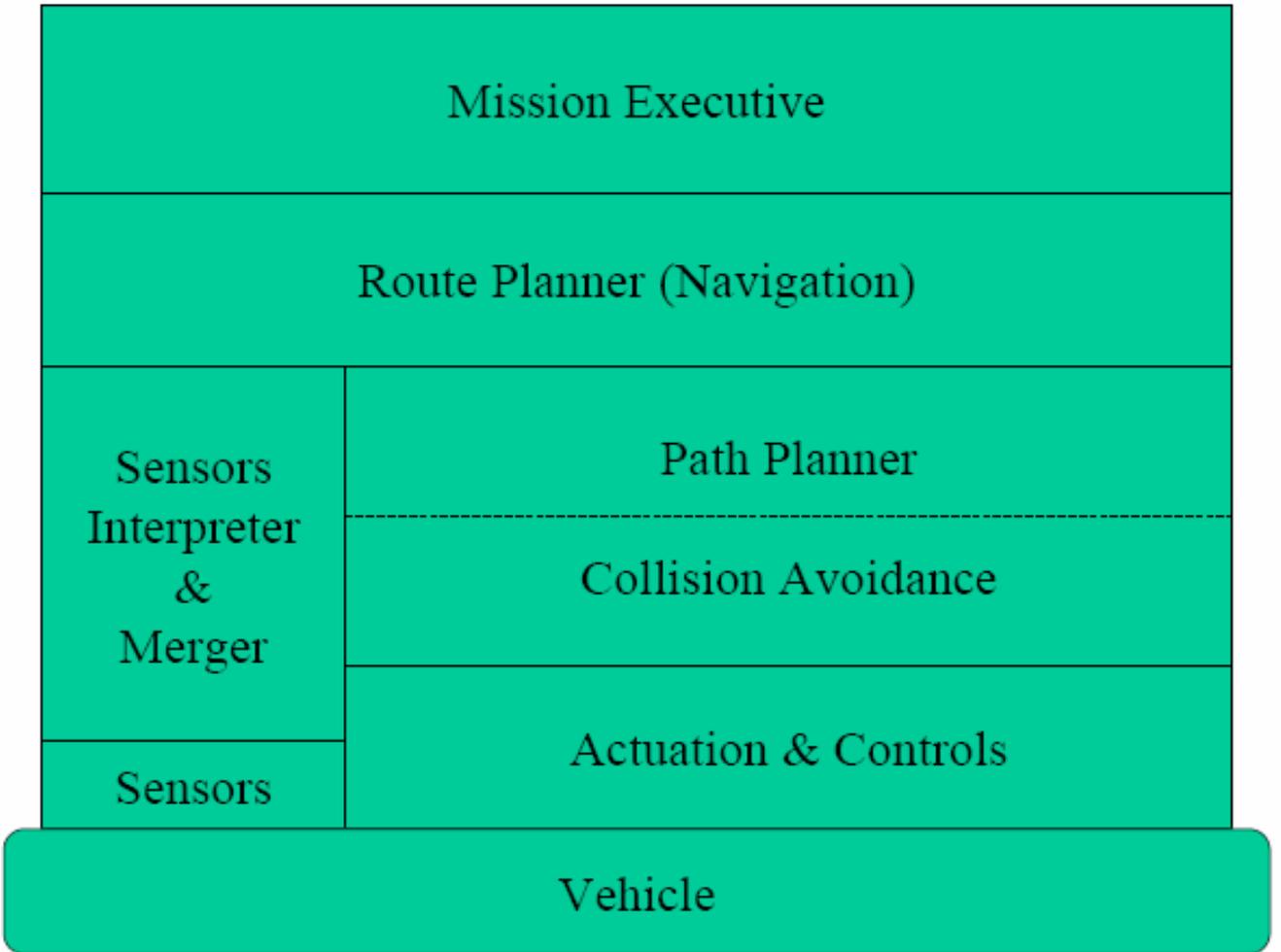
Figure 1 - Autonomous Vehicle Control Hierarchy

Nonholonomic vehicles are vehicles with constrained motion within their degrees of freedom. For example, a car has 2 degrees of freedom, but it cannot move sideways. It can move backward and forward and it can turn, but if you want to move the vehicle a few inches to the side, it is going to take at least two maneuvers to get there: move forward while turning the wheel in one direction, move backwards and turn the wheel in the other direction. Properly executed, the vehicle should stop in the desired location and the desired orientation. The constraint that did not allow the vehicle to make the sideways motion in one step, is called a

nonholonomic constraint and the vehicle is referred to as a nonholonomic vehicle. In terms of controls, a nonholonomic vehicle would have a larger number of generalized coordinates required to represent a system completely than the control degrees of freedom A holonomic vehicle has no such constraints within the degrees of freedom of the vehicle.

CHAPTER TWO: CURRENT STATE OF PATH PLANNING

2.1 Introduction

There has been many approaches developed in the field of robotics which attempt to solve the path planning problem in the presence of obstacles. Some of these approaches deal with holonomic systems such as potential field and vector field histogram. Since this thesis is only dealing with real-time mobile vehicles, this survey will only include the approaches developed for nonholonomic systems.

Table 1 - Summary of Path Planning Methodology

Numerical Approaches	Description	Comments
Latombe, Barraquand	Graph search on discretized space	
Wen, Divelbiss	Nonlinear least-squares in augmented space	
Donald, Xavier, Canny, Reif	Kino-dynamic planning algorithm to search for minimum time trajectories taking into account kinematics constraints	Static objects avoidance only
Lavalle, Kuffner	Random tree search algorithm to find inputs to 1st order ODE.	Static objects avoidance only
Eerdmann, Lozano-Perez	Recasting dynamic problem as a static problem, time is treated as a state variable.	Moving obstacles avoidance but trajectories must be known apriori
Hsu, Kindel, Latombe, Rock	Probabilistic roadmaps filled with local trajectories by integrating equation of motion	
Kant, Zucker	Divides problem in two: static path avoiding static obstacles, velocity planning avoiding moving obstacles	Solution is not guaranteed, requires knowing apriori trajectory of obstacles
Fiorini, Shiller	Velocity cone avoidance of constant velocity obstacles	
Analytical Methods	**Description**	**Comments**
Sussmann, Liu	Differential geometry approach	No obstacles avoidance
Fliess, Levine, Martin, Rouchon	Differential Flatness	No obstacles avoidance
Murray, Sastry, Tilbury	Input parameterization	No obstacles avoidance
Fernandes, Gurvits, Lit	Optimal control	No obstacles avoidance
Reeds, Shepp, Sussmann, Tang	Concatenation of simple pieces	No obstacles avoidance
Sundar, Shiller	hamilton-Jacobi-Bellman equation in suboptimal obstacle avoidance	Static objects avoidance only, but trajectory is holonomic so feasibility needs to be verified by optimalpath segments.
Qu, Wang, Plaisted	Input parameterization with dynamic obstacles avoidance	Nonholonomic with dynamic collision avoidance

2.2 Numerical Approaches

Methods based on numerical iteration are frequent in the literature. Below is a sample of some of the best known methods in the recent past. Most of these methods tackle the static obstacles problem while some actually include dynamic obstacles in their problem formulation.

Latombe and Barraquand [18] proposed a search algorithm involving the searching of a graph built after discretization of the configuration space. The nodes of this graph are small axis parallel cells. In this scheme, two cells are said to be adjacent if there is a feasible path segment between. The path segments are constructed by discretizing the controls and integrating the equations of motion.

Wen and Divelbiss [19] proposed to use a nonlinear least-squares problem in an augmented space to formulate nonholonomic motion planning. Introducing inequality constraints for obstacle avoidance, a feasible path trajectory is found numerically.

Donald, Xavier, Canny and Reif [20] proposed a dynamic programming algorithm to search for minimum time trajectories. This search is based on an approximation and takes into account the kinematics constraints for avoiding static and dynamic obstacles. These constraints are expressed in terms of bounds on velocity, acceleration, and force.

Lavalle and Kuffner [21] proposed the use of a random tree search algorithm to find appropriate inputs for a set of first order differential equations that contains the static obstacles. They called this approach kinodynamic planning.

Eerdmann and Lozano-Perez [22] proposed to treat the dynamic obstacles problem by recasting it as a static problem. To achieve this recasting, time is treated as a state variable in the n-dimensional configuration-time space. This approach requires the entire trajectories of moving obstacles to be known a priori.

Hsu, Kindel, Latombe, and Rock [23] proposed to use a search method using probabilistic roadmaps that are filled with local trajectories resulting from the integration of the equation of motion. This integration is performed by a randomly chosen controller from the set of admissible values.

Kant and Zucker [24] proposed to decompose the dynamic motion planning problem into two separate problems. The first problem tackles the static path planning problem by finding a path that avoids all static obstacles. The second problem tackles the velocity planning problem by determining the velocity of the vehicle along the path so that there will be no collision with moving obstacles. The short coming of this approach is that it requires complete information of obstacles current and future states. Solution is not guaranteed even if all future trajectories are know.

Fiorini and Shiller [25] proposed a method that deals with obstacles moving at a constant velocity. In this case, the concept requires the definition of a velocity cone and the evaluation of the robot's velocities relative to the obstacles. If this relative velocity does not enter the obstacle's velocity cone, no collision will occur.

2.3 Analytical Methods

Before Dr. Qu's analytical methods, there had been no comprehensive results on analytical motion planning for nonholonomic systems operating in a dynamical and uncertain environment. The kinematics constraints of nonholonomic systems make time derivatives nonintegrable for certain configurations. Therefore, it is not always feasible to determine a collision free path in the configuration space. In other words, a collision free path may not always be achievable by steering controls [4], [5].

Typically, there are two ways to approach the problem of nonholonomic systems and object avoidance. Some have concentrated exclusively on motion planning under nonholonomic constraints without considering obstacles. Others take holonomic results and modify them until the resulting path satisfy the nonholonomic constraints (making it kinematically feasible)

In the first group of nonholonomic motion planning without obstacles, Sussmann and Liu [6] proposed to use differential geometry. Fliess, Levine, Martin, and Rouchon [7] proposed to use differential flatness, Murray and Sastry [8], Monaco and Normand-Cyrot [9], and Tilbury, Murray and Sastry [10] proposed to use input parameterization. Fernandes, Gurvits, and Li [11] proposed to use optimal control. This last one is of particular interest because it proves that the nonholonomic motion problem can be recast as an optimal control problem and the Pontryagin's Maximum Principle can be applied. Reeds and Shepp [12] and Sussmann and Tang [13] show that the feasible shortest path for a point robot under two boundary conditions is a concatenation of simple pieces.

In the second group of results by modifying holonomic path planners and making the resulting path feasible Sundar and Shiller [14] proposed to use the Hamilton-Jacobi-Bellman equation in an algorithm called the suboptimal obstacle avoidance which allows static obstacles. The generated trajectory is holonomic, so its feasibility has to be verified (adapted) for nonholonomic mobile vehicle. Laumond and Jacobs [17] also proposed to generate trajectories using holonomic systems. However, they make the trajectories feasible to the nonholonomic vehicle by using a sequence of optimal path segments.

2.4 A Case for Analytical Path Planning With Collision Avoidance

The problem with a numerically based algorithm is that the computations are very processor (time) intensive and the results are not always guaranteed. Ideally, a path planner should find a feasible path if one exists. Some of the numerical methods have a very high percent of success, but none is 100% guaranteed to be successful at all times. For a path planner to be reliable it has to be 100% successful if a path exists. There may be cases where the path is a physical impossibility, and the only way to know that is if the path planner fails to produce a feasible path. Moreover, the path planner should be able to determine if a path exists before it starts to attempt the trajectory. Some of the numerical methods presented require the vehicle to perform certain motions before it can determine if the path exists.

An analytical (closed form) method should always yields a solution, if one exists. If one does not exist, this should be known even before the path is attempted. Of course, if the environment changes and the feasible paths are removed in the future, no real method can pre-determine that.

But even in that case, an analytical method should be able to tell as soon as the feasible paths are removed that a solution does not exists.

An ideal analytical method should take into consideration the kinematics constraints of the system automatically. In other words, no feasible path should be generated that is kinematically impossible. Only paths that are feasible in both the current environment and the kinematics constraints should be produced by the analytical method. Dynamic (moving) obstacles should also be considered regardless of the velocity profile of such obstacles without requiring a priori knowledge of future obstacles trajectories. Additionally, the path planner should have a pre-determined amount of computing resources and time to yield its results.

With these ideal characteristics in mind, the method proposed by Dr. Qu [1] is the only one that comes even close to achieving the goal: a rugged path planner with deterministic processing time which inherently takes kinematic constraints and dynamic obstacles into consideration. The following chapter explains this technique in complete details.

CHAPTER THREE: ANALYTICAL TRAJECTORY GENERATION

3.1 Introduction

This chapter explains the new analytical solution to mobile robot trajectory generation in the presence of moving obstacles developed by Dr. Zhihua Qu in [28]. Most figures and results are extracted from Dr. Qu's paper with permission. The trajectory generation technique will be formulated and simulated for a two-dimensional environment with one car like robot and a combination of up to three moving obstacles. Changes in the environment, as perceived by the robot, can be classified as obstacles changing velocity vectors or new obstacles appearing in view due to the limited range of sensors

In order to analyze the problem, it must be formulated mathematically. Figure 2 offers the frame of reference for the mathematical formulation:

- The robot is represented by a 2-D circle centered at point $O(t) = (x,y)$ and of radius R.

- The velocity of the robot is represented by the velocity vector $v_r(t)$.

- The range of the robot's sensors is a circle centered at $O(t)$ of radius R_s.

- The obstacles are represented by circles centered at point $O_i(t)$ and of radius r_i, where $i = 1,2,...,n_o$.

- The velocity of the robot is represented by the velocity vector $v_i(t)$.

- The robot moves from initial position $O_o=(x_o,y_o)$ and initial orientation θ_o to final position $O_f=(x_f,y_f)$ and final orientation θ_f.
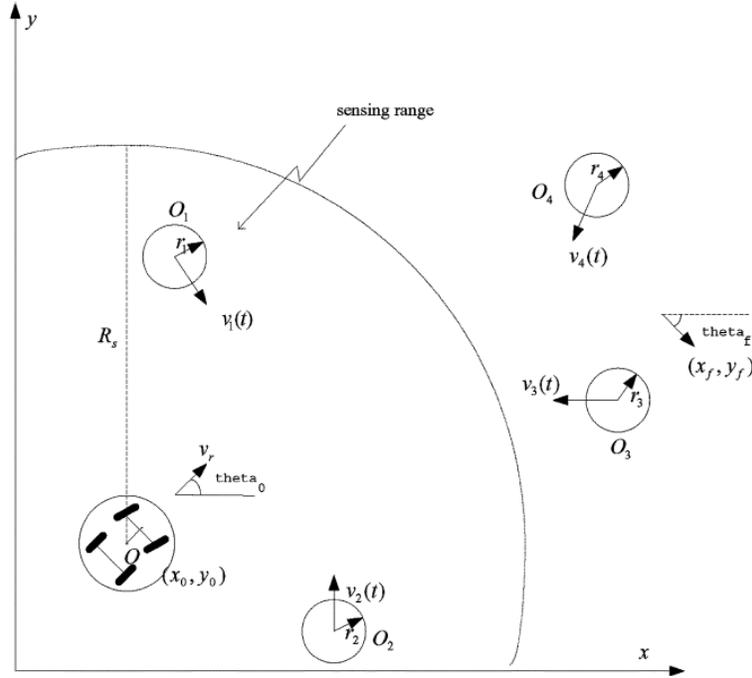
Figure 2 - General setting of trajectory planning in the presence of moving obstacles

The time it takes for the robot to move from O_o to O_f can be specified as $T_f - t_o$, or a velocity profile can be given and T_f left unspecified. If T_f is specified, the velocity must be adjusted to account for path deviations while avoiding obstacles. If T_f is not specified, which means that an arrival time is not important to the mission, the velocity profile must be provided giving velocity and acceleration maximum and minimum limits. Since the future velocity of the obstacles is unknown, the algorithm must take into account changes of the environment when they occur. In other words, the entire path is planned for the current environment, but as soon as a change is detected, the path that is left to traverse is automatically re-planned. With this in mind, the trajectory planning problem can be divided into smaller segments of time, T_s, with the following conditions:

- the velocity of all obstacles within the time period T_s is constant and linear

- the number of obstacle detected within the sensor range is constant

13

As soon as one of those conditions is violated, a new time segment must begin and a new trajectory must be generated. Mathematically, the time period is defined as $t \in [t_0 + kT_s, t_0(k+1)T_s)$ where k is the period number. The snapshot of the environment as seen by the robot, and the current state of robot and obstacles can be simplified as shown in Figure 3. The trajectory for this time segment can be generated using the velocity planning method developed by Fiorini and Shiller in [25].
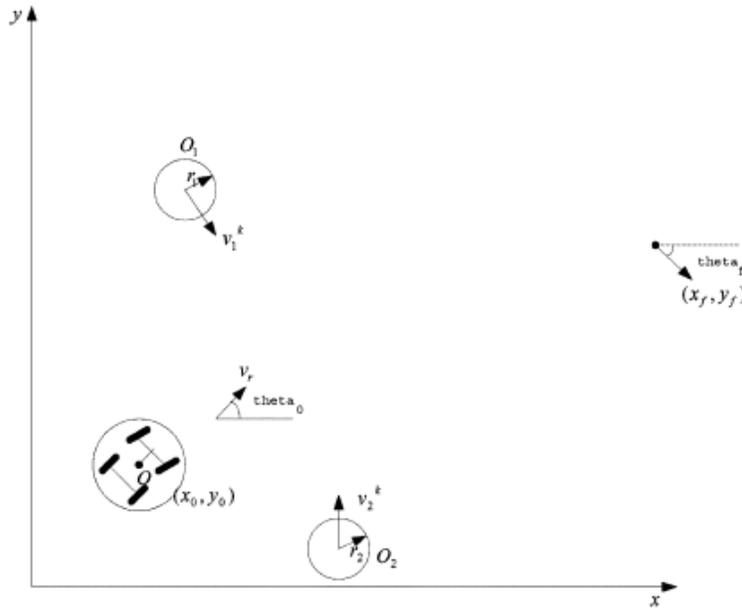
Figure 3 - Simplified setting of path planning with two moving obstacles

The velocity cone method considers the two obstacles to be moving with linear and constant velocities v_1 and v_2, respectively. With this approach, the radius of the obstacles is enlarged by the radius of the robot making the new obstacle radius $r_i + R$ and the radius of the robot is reduced to zero making it a guide point (GP). With the robot moving at velocity $v_r = [\dot{x}, \dot{y}]^T$, a relative velocity of each obstacle with respect to the robot, $v_{r,i} = v_r - v_i$ can be defined. Also, a velocity cone with its vertex at the robot GP, pointing toward the obstacle and touching the perimeter of

14

the circle around the obstacle with the new radius (r_i+R) can be defined. These geometrical definitions can easily be seen in Figure 4. Graphically, it is easy to see that if the relative velocities do not enter the velocity cone, collision will never happen. In Figure 4, $v_{r,1}$ is in the cone of obstacle 1 and $v_{r,2}$ is not in the cone of obstacle 2, which means that the robot will collide with obstacle 1 and not with obstacle 2.
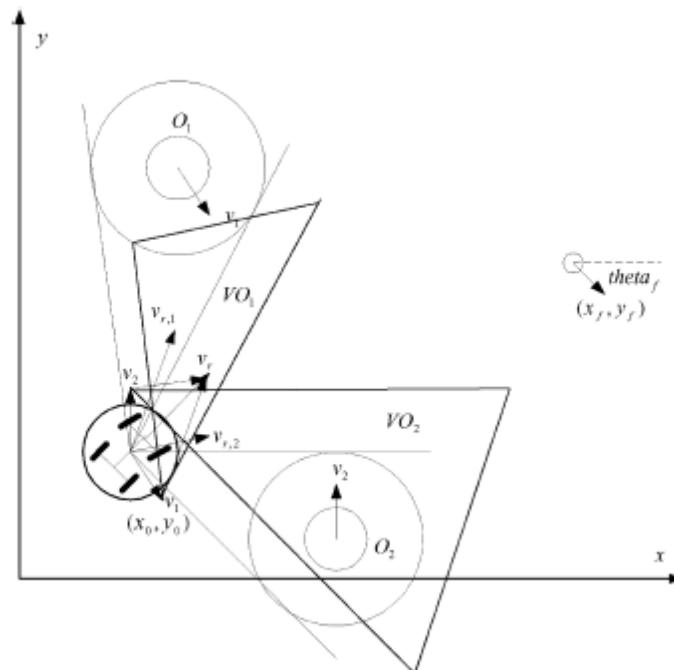
Figure 4 - Velocity cone method with two obstacles.

Considerations must be taken to overcome shortcomings of the velocity cone method:

- Velocities of robots and obstacles are not always constants and linear so a snapshot cone does not accurately reflect the future state of the environment and collision can occur even if velocities are outside the cone.

- Tackling the trajectory problem of path planning and velocity planning separately is not adequate for a truly dynamic environment because it would impose a priori knowledge of all obstacles trajectories.

- Kinematics constraints and dynamic model of the robot must be considered togother in trajectory planning.

3.2 Mobile Robot Modeling

The kinematics model of the robot is explicitly considered in this new path planning paradigm. The dynamic model is not currently considered but could also be included. The paradigm works for any kind of robot style as long as an accurate kinematic model is known.

The car like robot has front steering wheel and rear driving wheels (front wheel drive and all wheel drive are simple modifications of this model) with fixed orientation. Figure 5 shows a car like robot with a distance between the front and rear axles of l and centered at GP (the midpoint between the front and rear axles). For this case, the complete state of the vehicle is defined by $q = \begin{bmatrix} x & y & \theta & \phi \end{bmatrix}^T$, where (x,y) are the Cartesian coordinates of the GP, θ is the orientation of the robot body, and ϕ is the steering angle. The orientation angle θ is defined as the slope angle of the line passing through the GP and the center of the back axle.
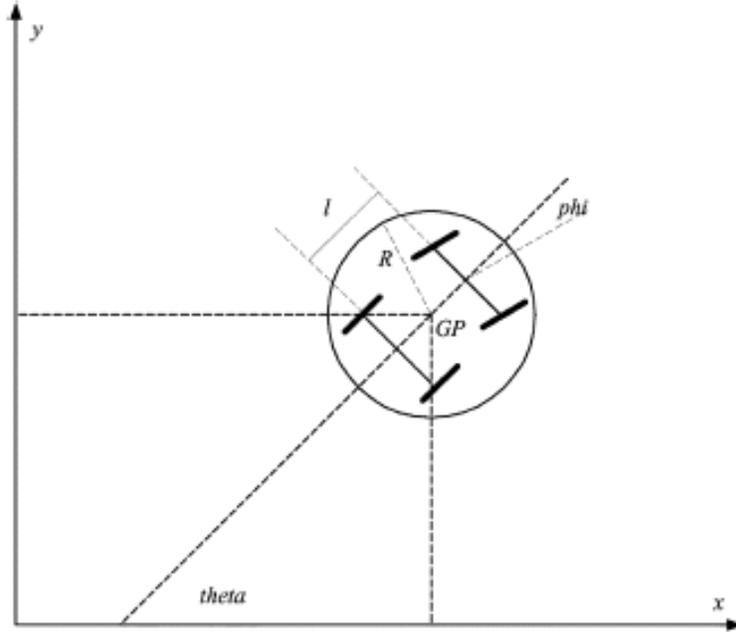
Figure 5 - Car like robot.

Letting ρ be the radius of the rear driving wheels, u_1 the angular velocity of the driving wheels, and u_2 the steering rate of the front wheels, the kinematic model for the car like robot is:

$$
\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \rho\cos(\theta) - \dfrac{\rho}{2}\tan(\phi)\sin(\theta) & 0 \\ \rho\sin(\theta) + \dfrac{\rho}{2}\tan(\phi)\cos(\theta) & 0 \\ \dfrac{\rho}{l}\tan(\phi) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{1}
$$

Equation (1) has a mathematical singularity at $\phi = \pm\pi/2$, which is a physical limitation of a car-like vehicle, but which does not occur in practice.

For a differential driven vehicle (also known as skid steering), such as a tank or as the experimental robot known as ATRV-Jr, the kinematic model must be changed to account for the

17

physical configuration shown in Figure. 6. Letting v_r be the linear velocity of the right wheel, v_l be the linear velocity of the left wheel, u_1 the vehicle's linear velocity, and u_2 the vehicle's angular velocity, the kinematic model for this vehicle is given by:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$u_1 = \frac{v_r + v_l}{2}, \quad u_2 = \frac{v_r - v_l}{l}$$

(2)
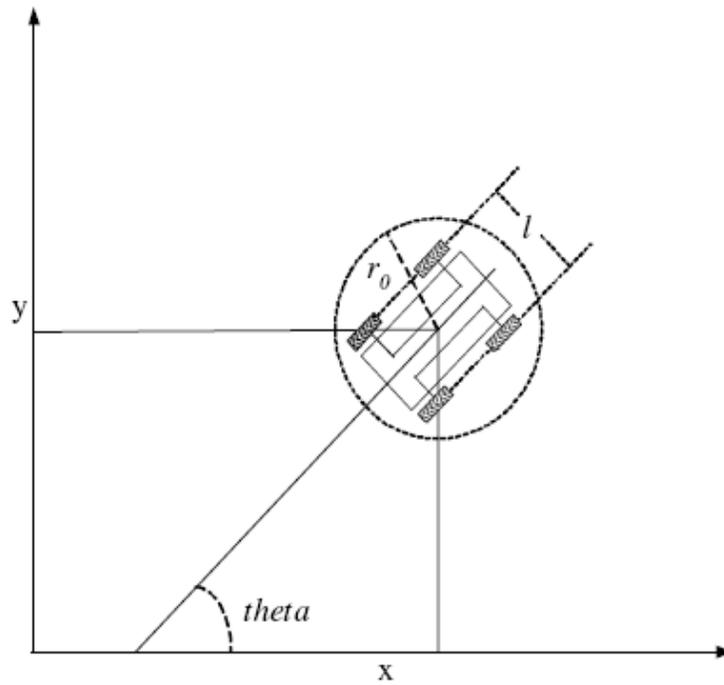
Figure 6 - Differential drive vehicle

Three wheeled vehicles with no steering wheel control and two wheeled vehicles such as Figure 7 have the same kinematics model [27] as in equation (2).

18
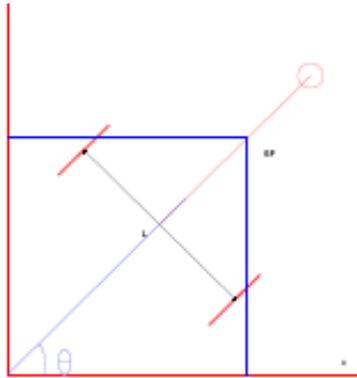
Figure 7 - Two wheel vehicle

For the remainder of this thesis, the car-like vehicle was selected to prove the concept and to perform the virtual reality simulations.

3.3 Chained Form

It is convenient to utilize a canonical form to represent the kinematics model which will standardize the process and facilitate the application to other kinematics models. In this thesis, the car like kinematics model, represented by equation (1), will be converted to a canonical form called the chained form. The chained form approach was introduced by Murray and Sastry in [8].

The first step is to develop the set of transformation equations to convert the system's variable from the world coordinates to the chained form coordinates. By properly selecting the transformation parameters, the following transformation can be obtained:

$$z_1 = x - \frac{l}{2}\cos(\theta)$$

$$z_2 = \frac{\tan(\phi)}{l\cos^3(\theta)}$$

$$z_3 = \tan(\theta) \tag{3}$$

$$z_4 = y - \frac{l}{2}\sin(\theta)$$

and consequently,

$$u_1 = \frac{v_{c1}}{\rho\cos(\theta)}$$

$$u_2 = -\frac{3\sin(\theta)}{l\cos^2(\theta)}\sin^2(\phi)v_{c1} + l\cos^3(\theta)\cos^2(\phi)v_{c2} \tag{4}$$

The state model for the kinematics model (1) under the above defined transformations is:

$$\dot{z}_1 = v_{c1}$$

$$\dot{z}_2 = v_{c2}$$

$$\dot{z}_3 = z_2 v_{c1} \tag{5}$$

$$\dot{z}_4 = z_3 v_{c1}$$

3.4 Steering Paradigm

The steering paradigm consists of three basic steps

1. Find physically achievable trajectories based on the kinematics model

2. Develop a collision avoidance criterion to avoid collisions with moving obstacles.

3. Parameterize the achievable trajectories into a specific class that meets the avoidance criterion.

3.4.1 Feasible Trajectories

Feasible trajectories have to satisfy both the boundary conditions and the dynamics of the kinematics model. Using the chained form equations in (5), a class of trajectories must be generates which automatically guarantees that all boundary conditions and the kinematics of the model will be satisfied. The initial and final conditions can be expressed as $z(t_0) = z^0 = \begin{bmatrix} z_1^0 & z_2^0 & z_3^0 & z_4^0 \end{bmatrix}^T$ and $z(t_f) = z^f = \begin{bmatrix} z_1^f & z_2^f & z_3^f & z_4^f \end{bmatrix}^T$. Defining a function $F(\ )$ as $z_4 = F(z_1)$, the boundary conditions can be expressed as:

$$
\begin{aligned}
z_4^0 &= F(z_1^0) \\
z_4^f &= F(z_1^f) \\
z_3^0 &= \frac{dF(z_1^0)}{dz_1^0} \\
z_3^f &= \frac{dF(z_1^f)}{dz_1^f} \\
z_2^0 &= \frac{d^2F(z_1^0)}{d(z_1^0)^2} \\
z_2^f &= \frac{d^2F(z_1^f)}{d(z_1^f)^2}
\end{aligned}
\tag{6}
$$

Letting $z_4 = F(z_1)$ conform to the boundary conditions $(x_0, y_0, \theta_0, \phi_0)$ and $(x_f, y_f, \theta_f, \phi_f)$, the steering problem can be solved. Assuming $\phi_0 = \phi_f = 0$:

$$
\begin{cases}
z_1^0 = x_0 - \dfrac{l}{2}\cos(\theta_0) \\[2mm]
F(z_1^0) = y_0 - \dfrac{l}{2}\sin(\theta_0) \\[2mm]
\left.\dfrac{dz_4}{dz_1}\right|_{z_1=z_1^0} = \tan(\theta_0) \\[2mm]
\left.\dfrac{d^2z_4}{d(z_1)^2}\right|_{z_1=z_1^0} = \dfrac{\tan(\phi_0)}{l\cos^3(\theta_0)} = 0
\end{cases}
\tag{7}
$$

21

$$
\begin{cases}
z_1^f = x_f - \dfrac{l}{2}\cos(\theta_f) \\[2ex]
F(z_1^f) = y_f - \dfrac{l}{2}\sin(\theta_f) \\[2ex]
\left.\dfrac{dz_4}{dz_1}\right|_{z_1=z_1^f} = \tan(\theta_f) \\[2ex]
\left.\dfrac{d^2 z_4}{d(z_1)^2}\right|_{z_1=z_1^f} = \dfrac{\tan(\phi_f)}{l\cos^3(\theta_f)} = 0
\end{cases}
\tag{8}
$$

3.4.2 Criterion For Avoiding Dynamic Objects

To develop the criterion in the proposed steering paradigm, lets consider the robot and the ith obstacle. The robot position in world coordinates is $(x(t),y(t))$ and the ith obstacle position in world coordinates is $(x_i(t),y_i(t))$ as shown in Figure 8 for the period $t \in [t_0 + kT_s, t_0(k+1)T_s)$. The robot is moving at a vector velocity $v_r \overset{\Delta}{=} [\dot{x}(t) \quad \dot{y}(t)]^T$ with an initial location $O_i = (x_i^k, y_i^k)$ where $x_i^k = x_i(t_0 + kT_s)$ and $y_i^k = y_i(t_0 + kT_s)$ and point O_i is moving at a known constant velocity $v_i^k \overset{\Delta}{=} [v_{i,x}^k, v_{i,y}^k]^T$. The robot velocity relative to the velocity of the ith obstacle is:

$$
v_{r,i}^k \overset{\Delta}{=} v_r - v_i^k = \begin{bmatrix} v_{r,i,x}^k \\ v_{r,i,y}^k \end{bmatrix} = \begin{bmatrix} \dot{x} - v_{i,x}^k \\ \dot{y} - v_{i,y}^k \end{bmatrix}
\tag{9}
$$
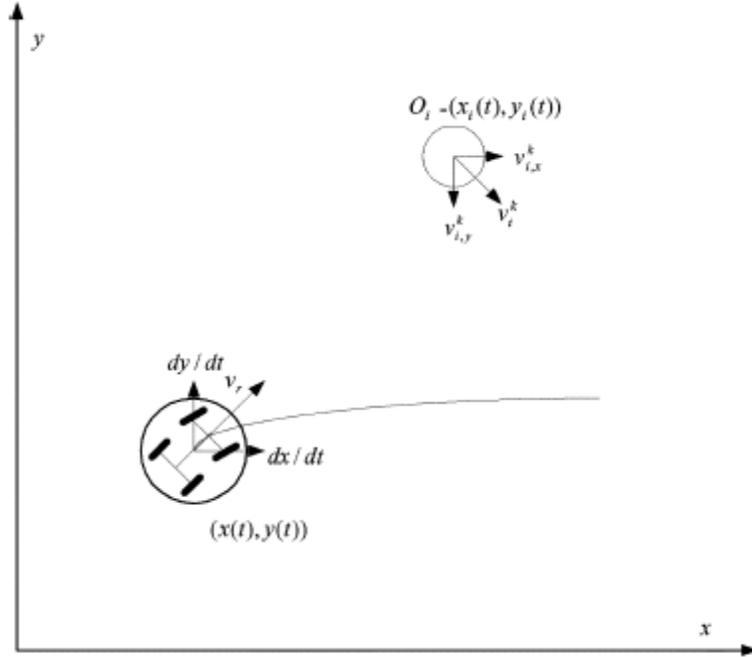
Figure 8 - Steering paradigm: robot and the ith obstacle.

By using relative velocity (9), Figure 8 can be transformed into Figure 9 where the obstacle is static. From Figure 9, the range of possible collision is limited to $\underline{x'}_i = x_i^k - r_i - R$ and $\overline{x'}_i = x_i^k + r_i + R$ for $x'_i \in \left[\underline{x'}_i, \overline{x'}_i\right]$. Given this limitation, the following inequality can be defined:

$$\left(y'_i - y_i^k\right)^2 + \left(x'_i - x_i^k\right)^2 \geq \left(r_i + R\right)^2 \tag{10}$$

where $x'_i = x - v_{i,x}^k \tau$, $y'_i = y - v_{i,y}^k \tau$, and $\tau = t - \left(t_0 + kT_s\right)$ for $t \in \left[t_0 + kT_s, t_f\right]$
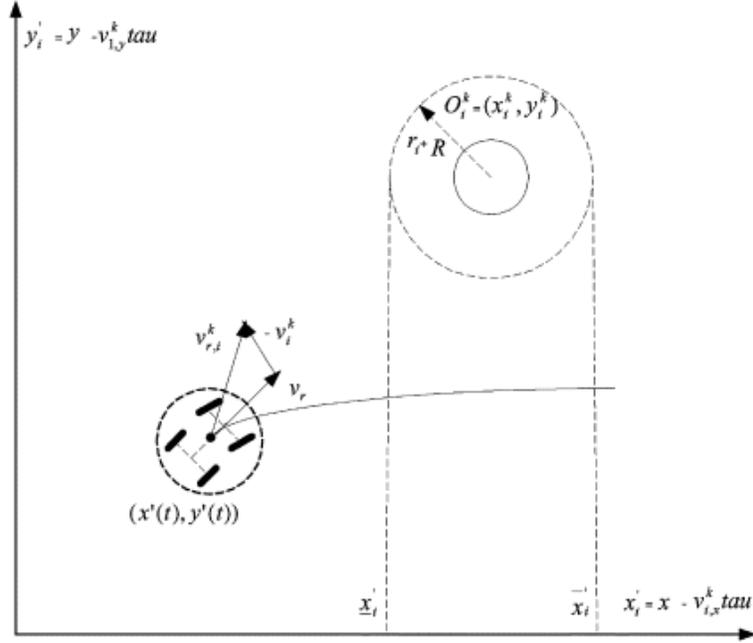
Figure 9 - Relative velocity of robot wrt. the ith obstacle

From state transformation (3), for any steerable path $z_4 = F(z_1)$, the corresponding feasible path in the *x-y* plane is:

$$y = F(x - 0.5l\cos(\theta)) + 0.5l\sin(\theta) \tag{11}$$

In the chained form transformed space z_4-z_1, the corresponding collision avoidance criterion, whenever $x_i^k \in \left[z_{1,i}' + 0.5l\cos(\theta) - r_i - R, \ z_{1,i}' + 0.5l\cos(\theta) + r_i + R \right]$, is:

$$\left(z_{4,i}' + \frac{l}{2}\sin(\theta) - y_i^k \right)^2 + \left(z_{1,i}' + \frac{l}{2}\cos(\theta) - x_i^k \right)^2 \geq (r_i + R)^2 \tag{12}$$

where $z_{1,i}' = z_i - v_{i,x}^k \tau$ and $z_{4,i}' = z_4 - v_{i,y}^k \tau$.

24

In order to find an analytical solution, a new criterion can be developed in terms of z_1 and z_4, without determining θ from z_3 numerically. All possible locations of point (x_i', y_i') are on the right semicircle centered at $(z_{1,i}', z_{4,i}')$ and of radius $l/2$ for $\theta \in [-\pi/2, \pi/2]$. Plotting a family of circles of radius $(r_i + R)$ along the right semicircle renders the region from which the center of the ith obstacle must stay clear as shown in Figure 10. The proposed collision avoidance criterion in the z_4-z_1 plane is:

$$\left(z_{4,i}' - y_i^k\right)^2 + \left(z_{1,i}' - x_i^k\right)^2 \geq \left(r_i + R + \frac{l}{2}\right)^2 \tag{13}$$

provided that

$$x_i^k \in \left[z_{1,i}' - r_i - R, \; z_{1,i}' + 0.5l + r_i + R\right] \tag{14}$$
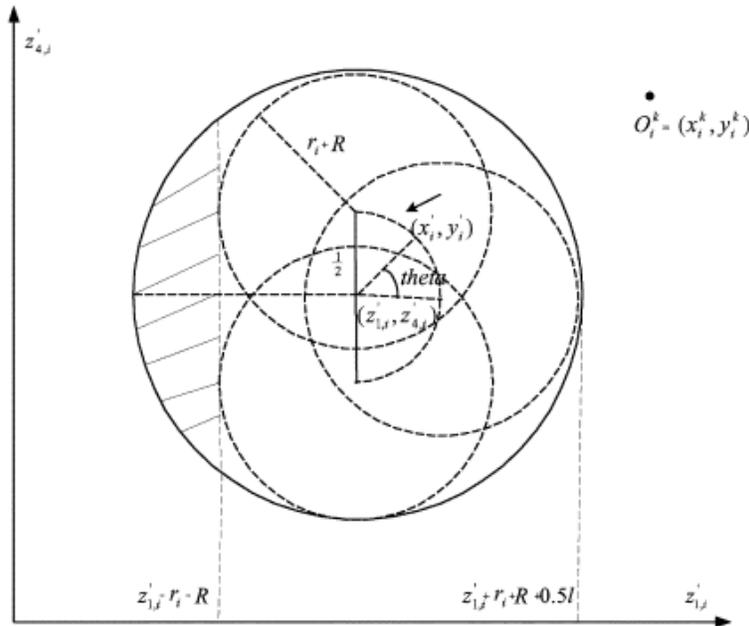
Figure 10 - Collision avoidance criterion in the transformed plane.

3.4.3 A Feasible Collision-Free Trajectory Parameterization

Defining $a^k = [a_0^k, a_1^k, a_2^k, a_3^k, a_4^k, a_5^k, a_6^k]$ as a constant vector and the vector composed of basis functions of $z_1(t)$ as $f(z_1) = [1, z_1(t), (z_1(t))^2, ..., (z_1(t))^6]^T$, the feasible trajectories are parameterized as

$$z_4(z_1) = F(z_1) = a^k f(z_1) \tag{15}$$

The sixth coefficient a_6^k makes the class a sixth order polynomial, one higher than needed for feasible trajectories, to take into account obstacles avoidance. This sixth parameter is determined by the collision avoidance criteria in (13). To account for obstacles changing velocities and for the appearance of new obstacles within the sensor's range, the solution is solved once within the time interval $t \in (t_0 + kT_s, t_0 + (k+1)T_s]$ and updated with respect to k, which is updated when obstacle velocities change or new obstacles appear in view. The analytical solution of a feasible collision free trajectory is guaranteed based on the following assumptions:

- Boundary conditions, $q_0 = [x_0, y_0, \theta_0, \phi_0]^T$ and $q_f = [x_f, y_f, \theta_f, \phi_f]^T$ with $\phi_0 = \phi_f = 0$, are defined by (7) and (8), and they satisfy the conditions $x_0 - \frac{l}{2}\sin(\theta_0) \neq x_f - \frac{l}{2}\sin(\theta_f)$ and $|\theta_0 - \theta_f| < \pi$

- Let $t_f = t_0 + T$ and T be the time for the mobile robot to complete its maneuver and T_s be the sampling period such that $\bar{k} = T/T_s$ is an integer, that centers of obstacles O_i are

26

located at (x_i^k, y_i^k) at $t = t_0 + kT_s$, and that these objects are all moving with known

constant velocities $v_i^k \overset{\Delta}{=} [v_{i,x}^k \quad v_{i,y}^k]^T$ for $t \in (t_0 + kT_s, t_0 + (k+1)T_s]$.

- For any given $k \in \{0,..., \bar{k} - 1\}$, the free space is connected in the presence of unshaded

  circular regions given by that in Figure 10 but located at O_i^k and of radius $r_i + R + 0.5l$,

  and the connectivity is with respect to "initial condition" (z_1^k, z_4^k) and "terminal

  condition" (z_1^{k+1}, z_4^{k+1}), where $z_i^k = z_i(t_0 + kT_s)$. Also, in relation to the free space and

  robot's sensing range, the robot's speed can be made faster than those of the objects.

Given these assumptions, a collision-free path can be generated analytically by undertaking the

following steps:

1. Select coordinates *(x,y)* of the working space such that $\theta \neq \pi/2$, apply state and input

   transformations (3) and (4), determine the corresponding boundary conditions

   $z^0 = [z_1^0, z_2^0, z_3^0, z_4^0]^T$ and $z^f = [z_1^f, z_2^f, z_3^f, z_4^f]^T$, and obtain the dynamics in chained form

   (5).

2. For $k = 0,..., \bar{k} - 1$ determine recursively constants a_6^k by ensuring the following second

   order inequality (or inequalities): $\forall i \in \{1,..., n_0^k\}$ where n_0^k is the number of obstacles

   within the sensing range during the time interval $t \in (t_0 + kT_s, t_0 + (k+1)T_s]$ and $n_0^k \leq n_0$

   as follows:

   $$\min_{t \in [\underline{t}_i^*, \bar{t}_i^*]} \left(g_2(z_1(t), k)(a_6^k)^2 + g_{1,i}(z_1(t), k, \tau)a_6^k + g_{0,i}(z_1(t), k, \tau) \Big|_{\tau = t - t_0 - kT_s} \right) \geq 0 \quad (16)$$

   where $[\underline{t}_i^*, \bar{t}_i^*] \subset [t_0 + kT_s, t_f]$ is the time interval (if it exists) during which

27

$$x_i^k \in [z_1(t) - v_{i,x}^k \tau - r_i - R, \, z_1(t) - v_{i,x}^k \tau + 0.5l + r_i + R] \tag{17}$$

In (15), functions $z_1(t)$, $g_2()$, $g_{1,i}()$, and $g_{0,i}()$ are defined as follows:

$$z_1(t) = z_1^k + \frac{z_1^f - z_1^0}{T}(t - t_0 - kT_s), \quad \forall t \in [t_0 + kT_s, t_f] \tag{18}$$

$$\underline{f}(z_1(t)) = [1 \quad z_1(t) \quad (z_1(t))^2 \quad (z_1(t))^3 \quad (z_1(t))^4 \quad (z_1(t))^5] \tag{19}$$

$$g_2(z_1(t), k) = [(z_1(t))^6 - \underline{f}(z_1(t))(B^k)^{-1} A^k]^2 \tag{20}$$

$$g_{1,i}(z_1(t), k, \tau) = 2[(z_1(t))^6 - \underline{f}(z_1(t))(B^k)^{-1} A^k] \\ \times [\underline{f}(z_1(t))(B^k)^{-1} Y^k - y_i^k - v_{i,y}^k \tau] \tag{21}$$

$$g_{0,i}(z_1(t), k, \tau) = [\underline{f}(z_1(t))(B^k)^{-1} Y^k - y_i^k - v_{i,y}^k \tau]^2 + \ldots \\ + (z_1(t) - x_i^k - v_{i,x}^k \tau)^2 - (r_i + R + 0.5l)^2 \quad \text{where,} \tag{22}$$

$$x_i^0 = x_i(t_0), \qquad y_i^0 = y_i(t_0)$$
$$x_i^k = x_i^0 + T_s \sum_{j=0}^{k-1} v_{i,x}^j, \qquad y_i^k = y_i^0 + T_s \sum_{j=0}^{k-1} v_{i,y}^j \tag{23}$$

$$z_4^0 = y_0 - \frac{l}{2}\sin(\theta_0)$$
$$z_3^0 = \tan(\theta_0) \tag{24}$$
$$z_2^0 = 0$$

$$z_1^k = z_1^0 + \frac{k(z_1^f - z_1^0)}{\bar{k}}$$
$$z_2^k = z_2^{k-1} + \int_{t_0+(k-1)T_s}^{t_0+kT_s} v_{c2}^{k-1}(\lambda) d\lambda$$
$$z_3^k = z_3^{k-1} + \frac{z_1^f - z_1^0}{\bar{k}} z_2^{k-1} + \frac{z_1^f - z_1^0}{T} \times \int_{t_0+(k-1)T_s}^{t_0+kT_s} \int_{t_0+(k-1)T_s}^{s} v_{c2}^{k-1}(\lambda) d\lambda ds \tag{25}$$
$$z_4^k = z_4^{k-1} + \frac{z_1^f - z_1^0}{\bar{k}} z_3^{k-1} + \frac{T_s^2}{2}\left(\frac{z_1^f - z_1^0}{T}\right)^2 \times z_2^{k-1} + \ldots$$
$$+ \left(\frac{z_1^f - z_1^0}{T}\right)^2 \int_{t_0+(k-1)T_s}^{t_0+kT_s} \int_{t_0+(k-1)T_s}^{\tau} \int_{t_0+(k-1)T_s}^{s} v_{c2}^{k-1}(\lambda) d\lambda ds d\tau$$

$$Y^k = \begin{bmatrix} z_4^k \\ z_3^k \\ z_2^k \\ y_f - \dfrac{l}{2}\sin(\theta_f) \\ \tan(\theta_f) \\ 0 \end{bmatrix}, \qquad A^k = \begin{bmatrix} (z_1^k)^6 \\ 6(z_1^k)^5 \\ 30(z_1^k)^4 \\ (z_1^f)^6 \\ 6(z_1^f)^5 \\ 30(z_1^f)^4 \end{bmatrix} \qquad (26)$$

$$B^k = \begin{bmatrix} 1 & z_1^k & (z_1^k)^2 & (z_1^k)^3 & (z_1^k)^4 & (z_1^k)^5 \\ 0 & 1 & 2z_1^k & 3(z_1^k)^2 & 4(z_1^k)^3 & 5(z_1^k)^4 \\ 0 & 0 & 2 & 6z_1^k & 12(z_1^k)^2 & 20(z_1^k)^3 \\ 1 & z_1^f & (z_1^f)^2 & (z_1^f)^3 & (z_1^f)^4 & (z_1^f)^5 \\ 0 & 1 & 2z_1^k & 3(z_1^f)^2 & 4(z_1^f)^3 & 5(z_1^f)^4 \\ 0 & 0 & 2 & 6z_1^f & 12(z_1^f)^2 & 20(z_1^f)^3 \end{bmatrix} \qquad (27)$$

3. A feasible, collision free path of form (15) in the transformed state is found by solving a^k

   according to:

$$a^k = \begin{bmatrix} a_0^k & a_1^k & a_2^k & a_3^k & a_4^k & a_5^k & a_6^k \end{bmatrix}$$
$$\begin{bmatrix} a_0^k & a_1^k & a_2^k & a_3^k & a_4^k & a_5^k & a_6^k \end{bmatrix}^T = (B^k)^{-1}(Y^k - A^k a_6^k) \qquad (28)$$

4. For $t \in (t_0 + kT_s, t_0 + (k+1)T_s]$, the steering inputs to achieve path (15) are given by

$v_{c1}(t) = v_{c1}^k(t)$, and $v_{c1}(t) = v_{c2}^k(t)$, where

$$v_{c1}^k(t) = \frac{z_1^f - z_1^0}{T} \tag{29}$$

$$
\begin{aligned}
v_{c2}^k(t) = &\, 6[a_3^k + 4a_4^k z_1^k + 10a_5^k (z_1^k)^2 + 20a_6^k (z_1^k)^3]v_{c1}^0 + ... \\
&+ 24[a_4^k + 5a_5^k z_1^k + 15a_6^k (z_1^k)^2](t - t_0 - kT_s)(v_{c1}^0)^2 + ... \\
&+ 60[a_5^k + 5a_6^k z_1^k](t - t_0 - kT_s)^2 (v_{c1}^0)^3 + 120a_6^k (t - t_0 - kT_s)^3 (v_{c1}^0)^4
\end{aligned}
\tag{30}
$$

5. The corresponding feasible, collision free Cartesian trajectory is given by $y = F(x - 0.5l\cos(\theta)) + 0.5l\sin(\theta)$, where θ can be found in closed form from state transformation (3) under steering inputs () and () together with control mapping (4)

30

CHAPTER FOUR: VIRTUAL REALITY SIMULATION

4.1 Introduction

The analytical technique explained in Chapter Three is implemented as part of this thesis utilizing Matlab and Simulink as the simulation tools. In particular, the VRML toolbox is utilized to present the simulation in a virtual reality environment. This virtual realization allows for easy perception of the results of the path planning technique. By actually moving a vehicle in a simulated open terrain, avoiding moving obstacles, the entire algorithm is corroborated. This chapter deals with the methodology used for simulation.

4.2 Overview of Virtual Reality Simulation Methodology

Virtual Reality traces its roots back to the radar screen technology. The first attempts to use a computer to generate graphics were done by Douglas Engelbart, a naval radar technician in the late 1950's who is better known for inventing the first mouse. The first actual implementations of simulations of real world were done by Naval radar developers and aircraft designers while trying to express computer results in a more human friendly form during the 1960's. During the 1970's computer simulations became a popular flight training technique for defense and space programs. At the same time, the entertainment industry started venturing into the virtual reality world as a means of displaying video games and movies. During the 1980's, graphics and simulations systems went in many different directions, but one thing remained common, the drive to communicate in pseudo real environment the results of an analysis.

The unexpected exponential growth that the World Wide Web had in the 1990's contributed in great way to the growth of virtual reality. During the first Internationl Conference on the World Wide Web, Mark Pesce and Tony Parisi presented a 3-D interface to the Web. At the end of the conference, there was consensus that a common language to specify 3-D scene descriptions was needed In 1995, the first two meetings on VRML took place which eventually led to the standard called VRML97. Since then, some other standards have been developed, mainly as spin-offs of VRML97 such as Web3D's X3D, Kismet 3D, Breve, Simul8, Modelica, OpenEaagles, and others.

All of the virtual modeling languages provide basically the same functionality. Given a time dependant data set, the virtual reality system will display the results graphically. The actual computational technique of the simulation is outside the scope of the virtual reality environment. The simulation tool needs to interface to the virtual reality environment in order to send the coordinates of the objects within the world coordinates.

The first step in a virtual reality simulation is to create a virtual reality world. Many worlds have been created that can be used as a starting point when creating a new world. Most toolboxes provide a world builder with some graphical tools to create and manipulate objects in the world. A world in 3-D is the 3-D Euclidean space where the simulation results will be displayed. It is a closed set with soft boundaries, which means that the simulation can go outside the world's space, but no more 3-D visualization would be displayed (unless an infinite world is created by applying a repeat function). The world can have textures and static objects that are pre-

positioned. The virtual reality engine will calculate the correct perspective to make the views look proportional, relative to created objects in the world.

The second step in the world creation process is to place movable objects. There is no actual difference between a movable object and a fixed object, except that the translation and rotation coordinates are either available to the simulation interface or are only internal to the world, respectively. In other words, any object that can be place stationary, can also be made to move and/or rotate from an external input.

The third step is to create viewpoints. A viewpoint is the location and orientation of a camera that can be used to display the status of the world. Viewpoints can be stationary or they can also move, rotate, or zoom based on external inputs. For example, a viewpoint can be like a bird's eye always looking at the center of the world from above. Another viewpoint is take the same bird's eye and connected to a moving object, so the center of the screen is not the center of the world, but a moving object. Objects can be made to move not only on external inputs but also on internal inputs using a routing technique, which basically duplicates and connects internal objects together.

Of course, the steps included in this thesis are an over-simplification of the actual steps needed to implement a particular world. Moreover, the detailed steps are environment specific, different for VRML than for X3D. We have chosen to use the VRML toolbox available for Simulink, the simulation environment for Matlab. This is a toolbox which is readily available and is completely compatible with all of the other simulation toolboxes available from Matlab.

To simulate a virtual reality environment with Matlab, a block diagram based on Simulink must be generated. In the block diagram, one of the final blocks will be the 3-D Visualization block from the VRML toolbox. In the block dialog box, the interface can be defined from any of the externally available links within the world. Typically, the translation and rotation references are used to control the positioning and orientation of the moving objects. For every moving object, translation and rotation information must be provided.

The translation reference is an obvious three parameter interface: (x,y,z). In most worlds, the y-axis is the elevation axis, x-axis and z-axis are axes parallel to the ground (the right hand rule is always observed). The rotation reference is a not-so-obvious four parameter interface (x,y,z,θ). The way to understand how these parameters work is to think of the (x,y,z) part as a flag that will tell the virtual reality environment to which axes to apply the value of θ. For example, if you want to rotate a vehicle 40 degrees about the y-axis (elevation axis, as in the case of a car on a flat road), you would specify (0,1,0,40), again, the right hand rule applies. By constantly updating translation and rotational parameters, the object changes location and orientation in the virtual reality world. If the updates are done at a small enough interval, the motion will appear continuous to the human eye.

4.3 Simulating Mobile Robots and Obstacles in VRML

The path planner application requires some knowledge of the obstacles position and velocity. Most virtual reality environments will provide an output interface which can be routed to any internal object. The output can then be used by the path planner to get current information of the

world. However, to increase simulation efficiency, most implementations actually do not use the output feature of the virtual reality environment. Since the simulation is generating the motion of all obstacles, the actual position and velocity can be obtained directly from the obstacle simulation block without having to go to the virtual reality environment. The coordinates will be provided to the world regardless, so that obstacles position and orientation can be updated.

The virtual reality simulation environment has no information of where the coordinates are coming from. The actual source could be a pre-calculated file with a time stamped sequence of coordinates for every moving object or it could be real-time position and orientation coordinates generated by dynamic and kinematics simulation of the robot. For a simulation to be considered "true", it should generate the coordinate information reacting to changes in the world, so a pre-calculated set would fall more in the 3-D graphical display arena than in the 3-D virtual reality simulation arena. The simulation results supporting this thesis use a real-time virtual reality simulation including vehicle kinematics. Therefore, the simulation block that generates the motion for the obstacles and the robot must know and apply the kinematics constraints of these vehicles. The path planner should never violate these constraints, so this is a valid approach. Figure 11 shows the high level block diagram of the virtual reality path planner simulator.

Autonomous Ground Vehicle with Dynamic Obstacle Avoidance
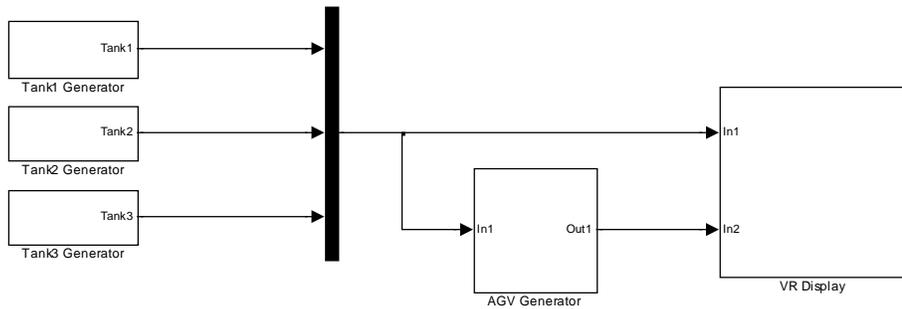
Figure 11 - Virtual Reality Path Planner Simulator

4.4 Obstacles Generation

Obstacles can be generated using simpler approaches than the path planner technique of the robot. For the purpose of this simulation, the method chosen to generate the obstacles path is not important and does not have to be the path planning technique discussed in Chapter Three. As a matter of fact, the obstacle should have the flexibility to move in many ways, so that different scenarios can be simulated. In other words, the obstacle should not be limited to a car like, nonholonomic vehicle.

It is convenient to keep the initial conditions and the velocity profile of the obstacle very accessible. These parameters would typically be constantly changing to test different scenarios. In this case, these parameters are represented as "source" block in Simulink instead of hard coded values in the Matlab s-function. Figure 12 shows the Obstacle Trajectory Generation Block Diagram.
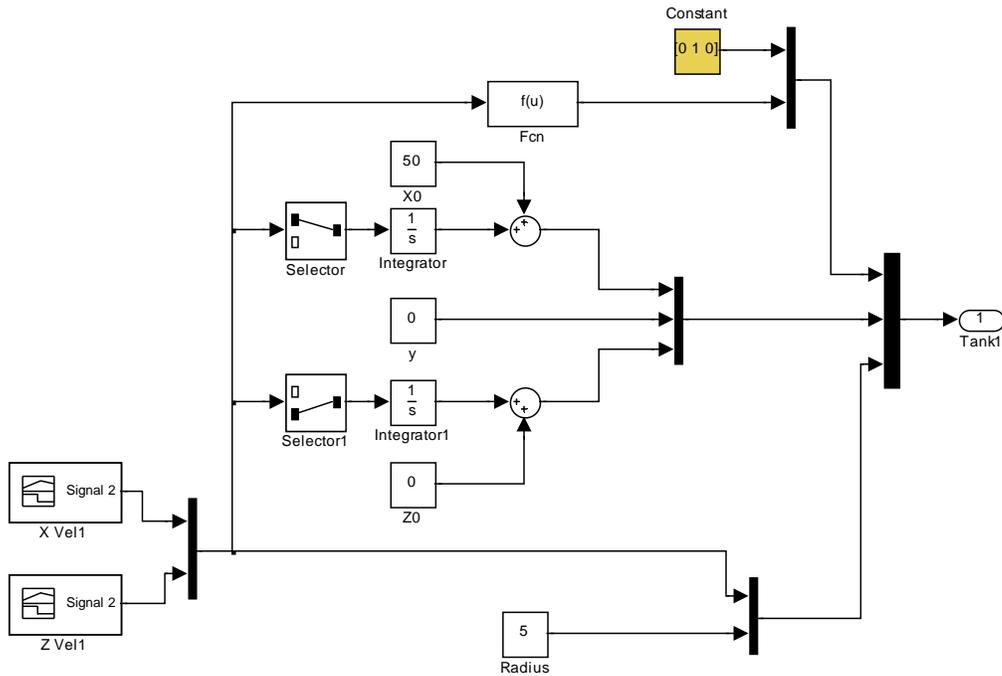
Figure 12 - Obstacle Trajectory Generation Block Diagram

4.5 AGV Generation

The details on how to implement the autonomous ground vehicle (AGV) generator are embedded into the Matlab s-function. However, like the obstacle generator, it is convenient to define certain initial conditions at the block diagram level so that the values are easily changed for different scenarios, instead of having them hardcoded. The main difference between obstacle generator and the AGV generator is that the AGV requires knowledge of the obstacle, while the obstacles move independently of everything else. Therefore, in the AGV block diagram, the proper feedback is provided for the AGV to get position and velocity information from the obstacles.

For testing purposes, it is convenient to have a quick method of turning collision avoidance on and off. By using the switch block in Simulink and providing the on/off status as an input to the AGV s-function, the path planner can simply ignore obstacles by the "flick" of a switch. Figure 13 shows the complete AGV Generator block diagram. The Matlab code of the AGVM s-function is provided in Appendix A.
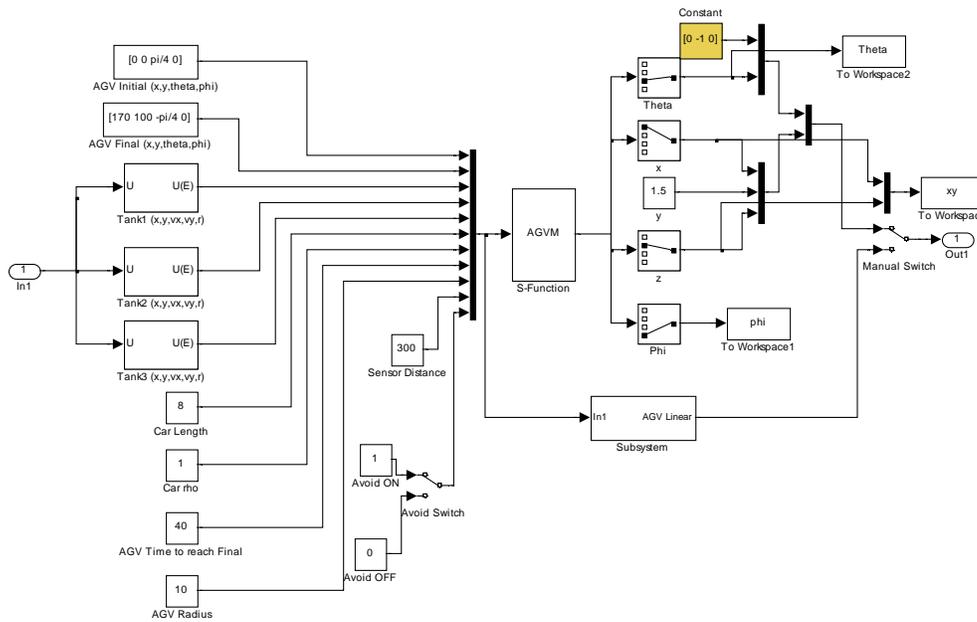
Figure 13 - AGV Generator

4.6 World Realization

The virtual reality world consists of an open environment simulating desert conditions with one road and a few bushes to provided depth of field and some level of reality. The obstacles are represented by military tanks. Since the obstacles kinematics are not important for this simulation, the differential drive kinematics of the tank are not simulated. Only linear translation

and rotation is handled, which is independent of the actual object representation chosen to be the obstacles. The virtual world is connected to the simulation via a dedicated block to handle interface. This block is shown in Figure 14.
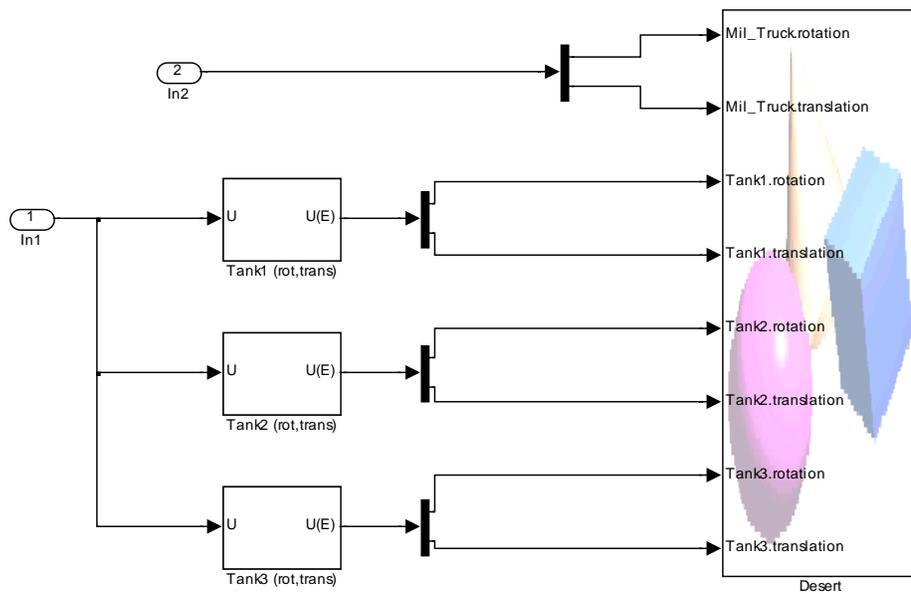
Figure 14 - Virtual Reality World Interface

CHAPTER FIVE: FINDINGS

5.1 Introduction

The results of the simulations are presented in this chapter. The obvious end product is the virtual reality 3-D visualization of AGV moving in the open terrain avoiding dynamic obstacles. In the process of investigating the proper virtual reality visualization, the tool was used to look at the effect of sensor radius to the performance of the algorithm. The plots in the next section, document the different paths the AGV used to reach the destination while avoiding moving obstacles. This translate in a change of the value of a_6 in different circumstances.

5.2 Simulation Results

The simulation results presented are for the cases of sensor radius of 25, 50, 75, 100, 125, and 300. There is an optimum path generated between sensor radius of 50 and 75, but no further effort was made to precisely locate this sensor radius. Every case was able to reach the endpoint while successfully avoiding all obstacles and moving within the boundaries and nonholonomic constraints. One interesting result is that once the sensor radius reached 150, increasing it made very little effect on the trajectory generated.
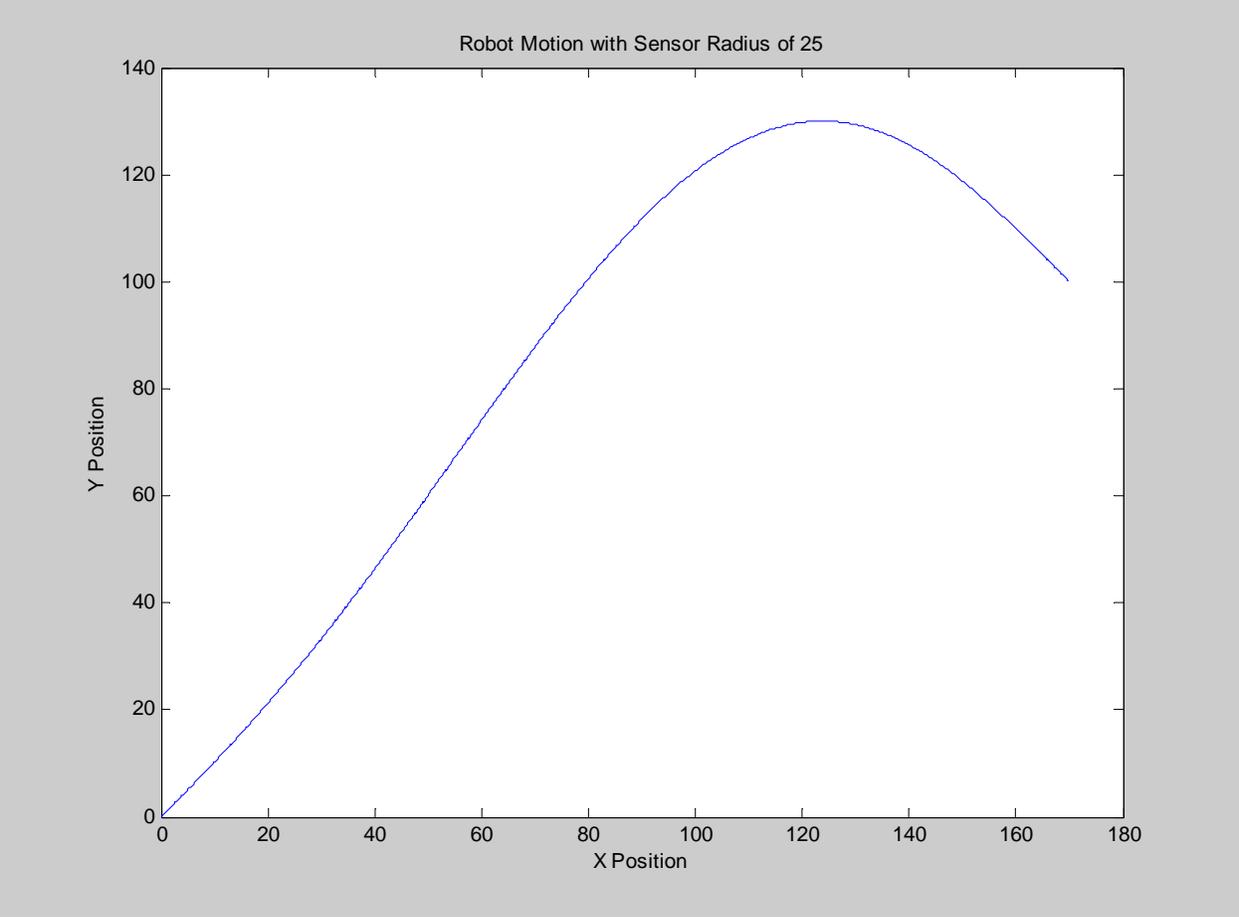
Figure 15 – Robot Motion with Sensor Radius of 25
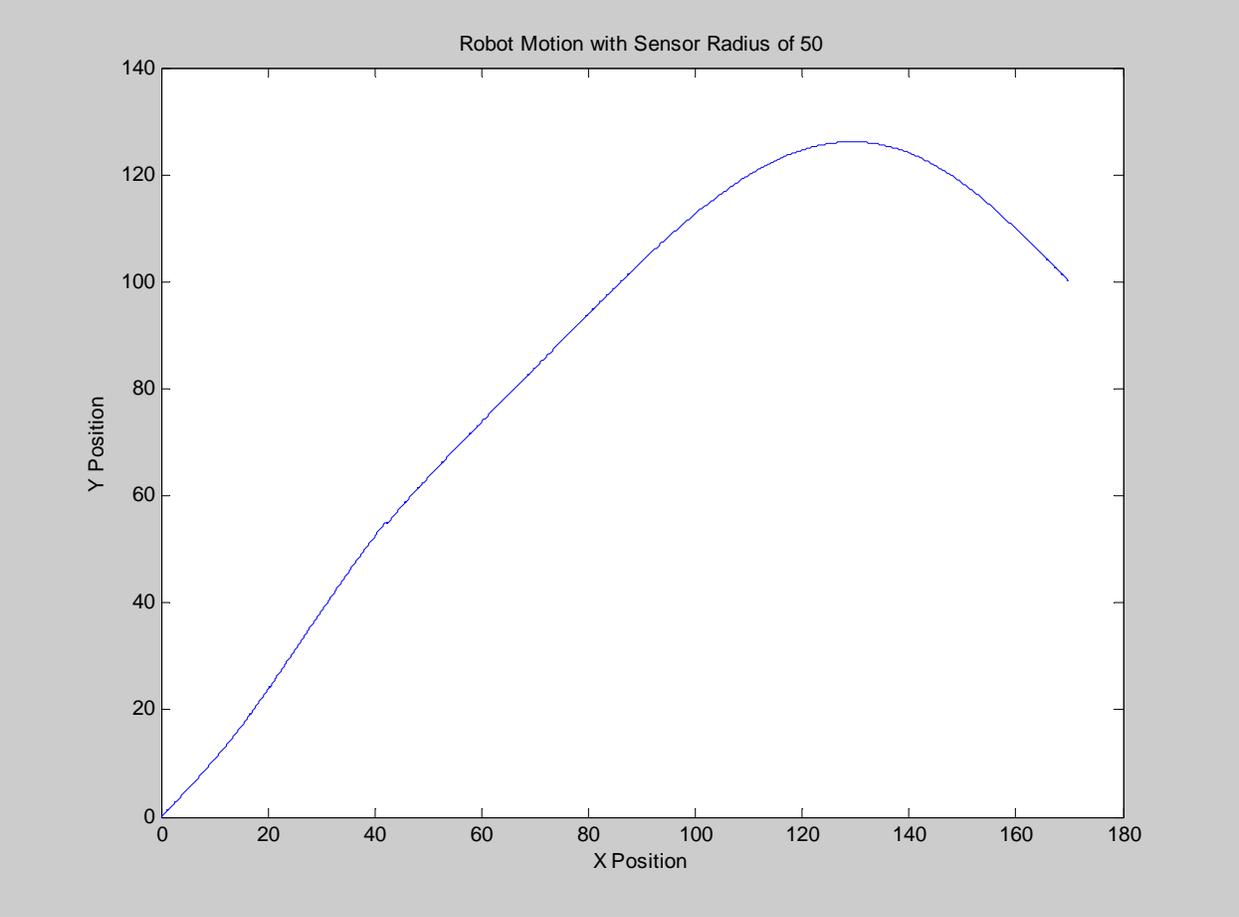
Figure 16 – Robot Motion with Sensor Radius of 50
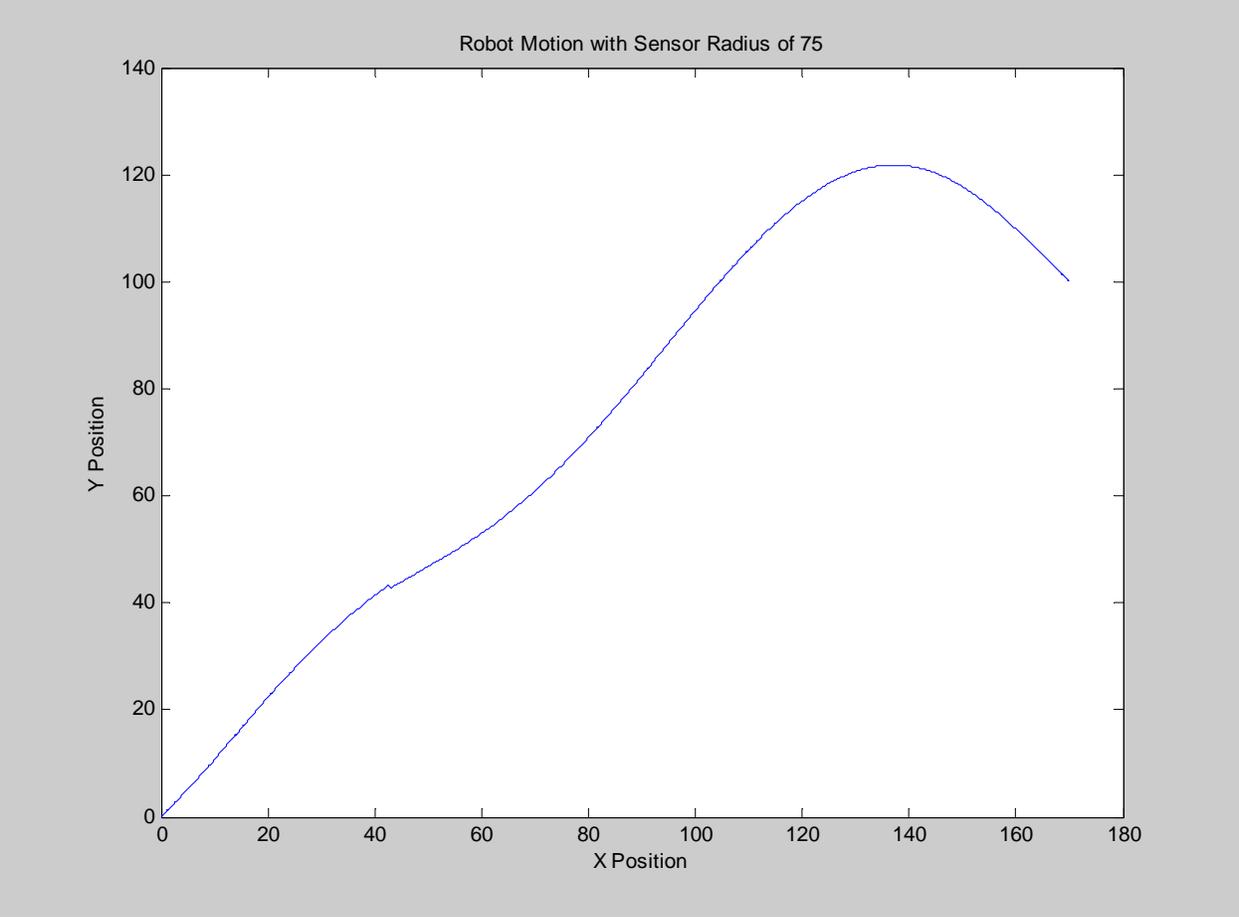
Figure 17 – Robot Motion with Sensor Radius of 75
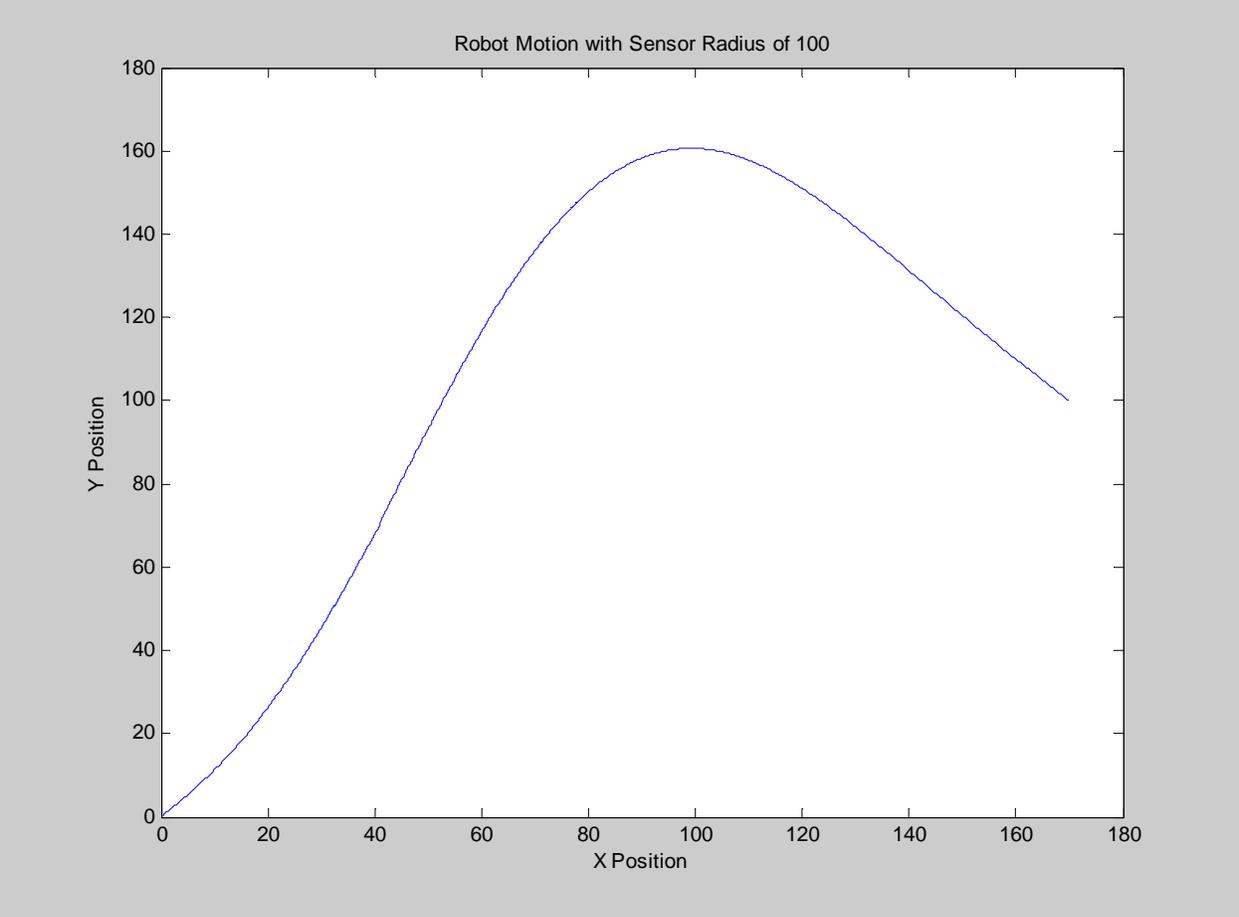
Figure 18 – Robot Motion with Sensor Radius of 100
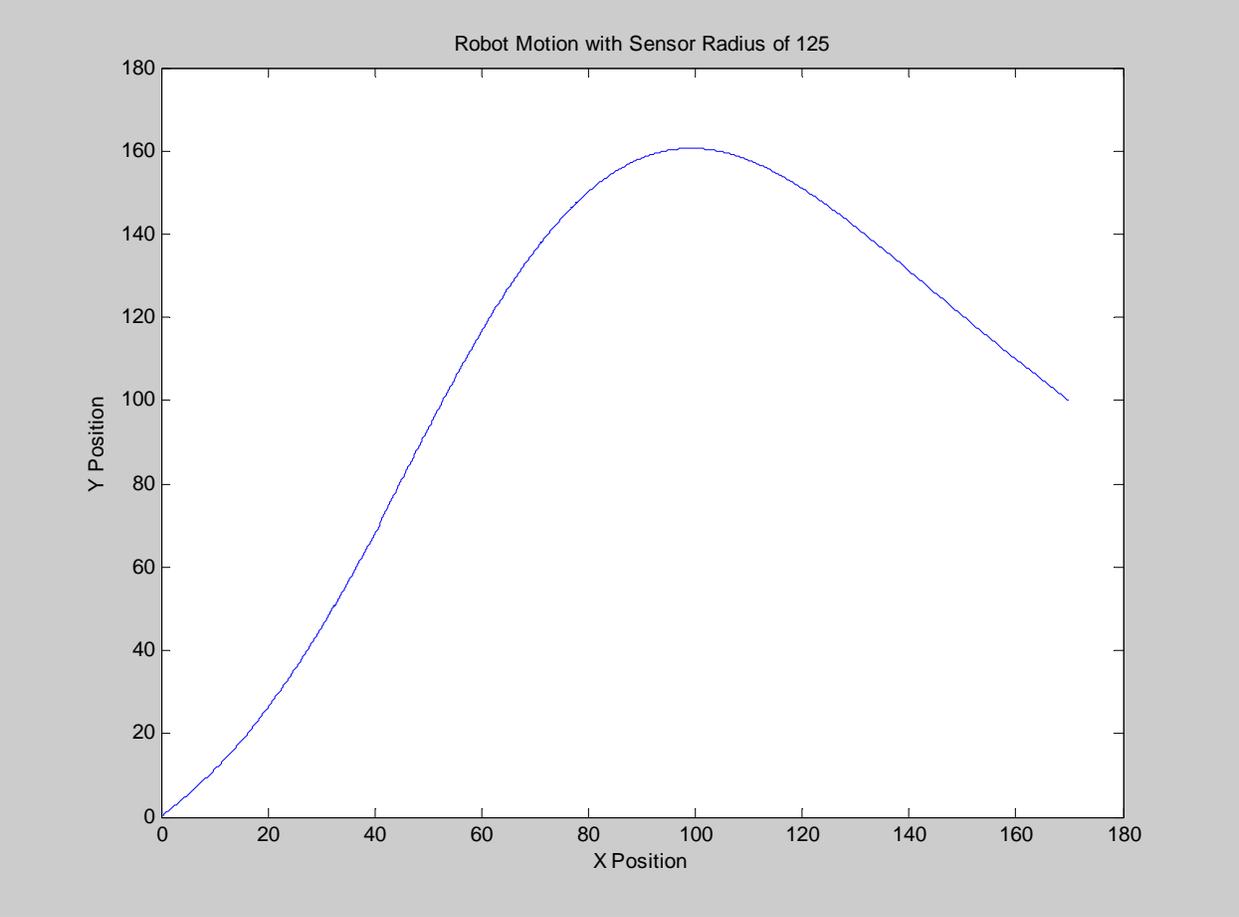
Figure 19 – Robot Motion with Sensor Radius of 125
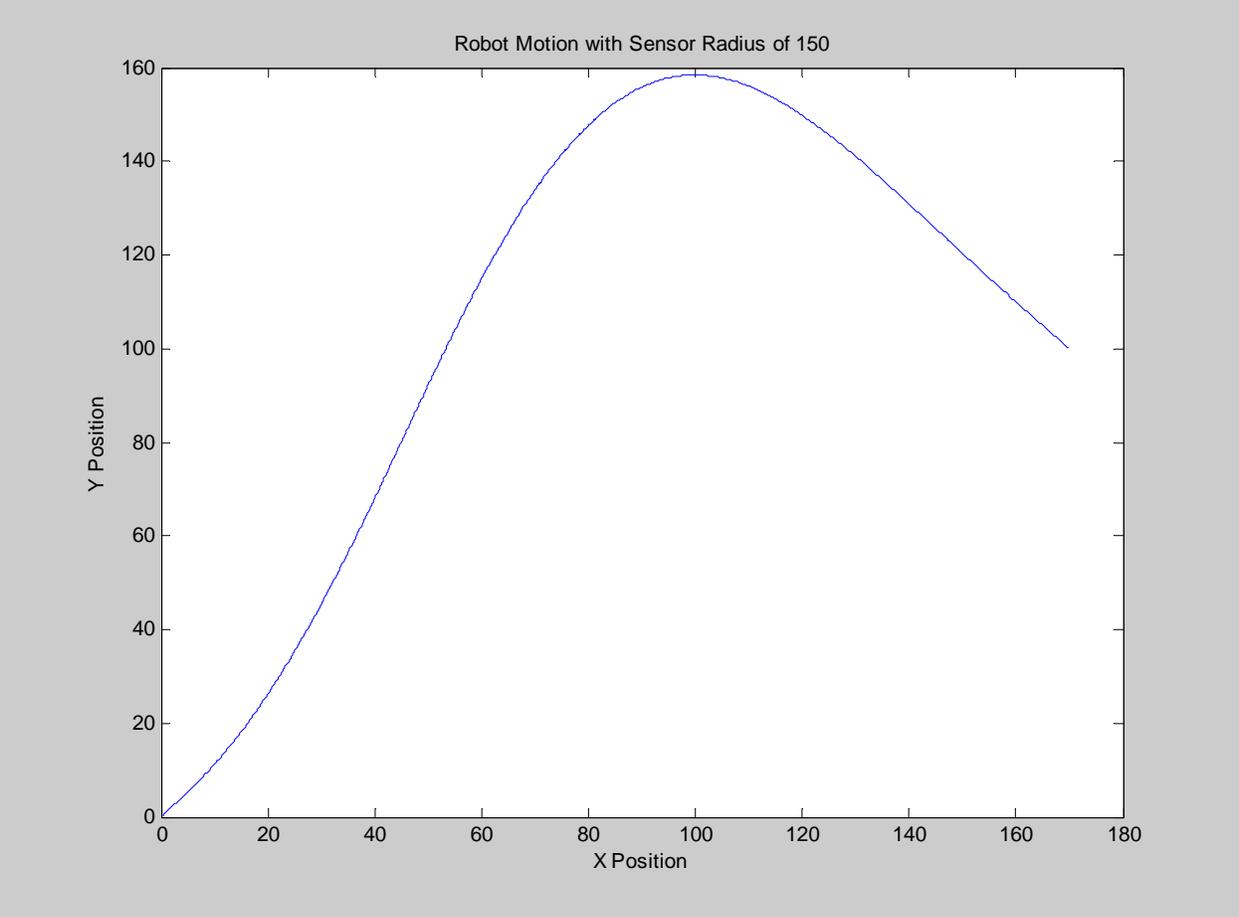
Figure 20 – Robot Motion with Sensor Radius of 150
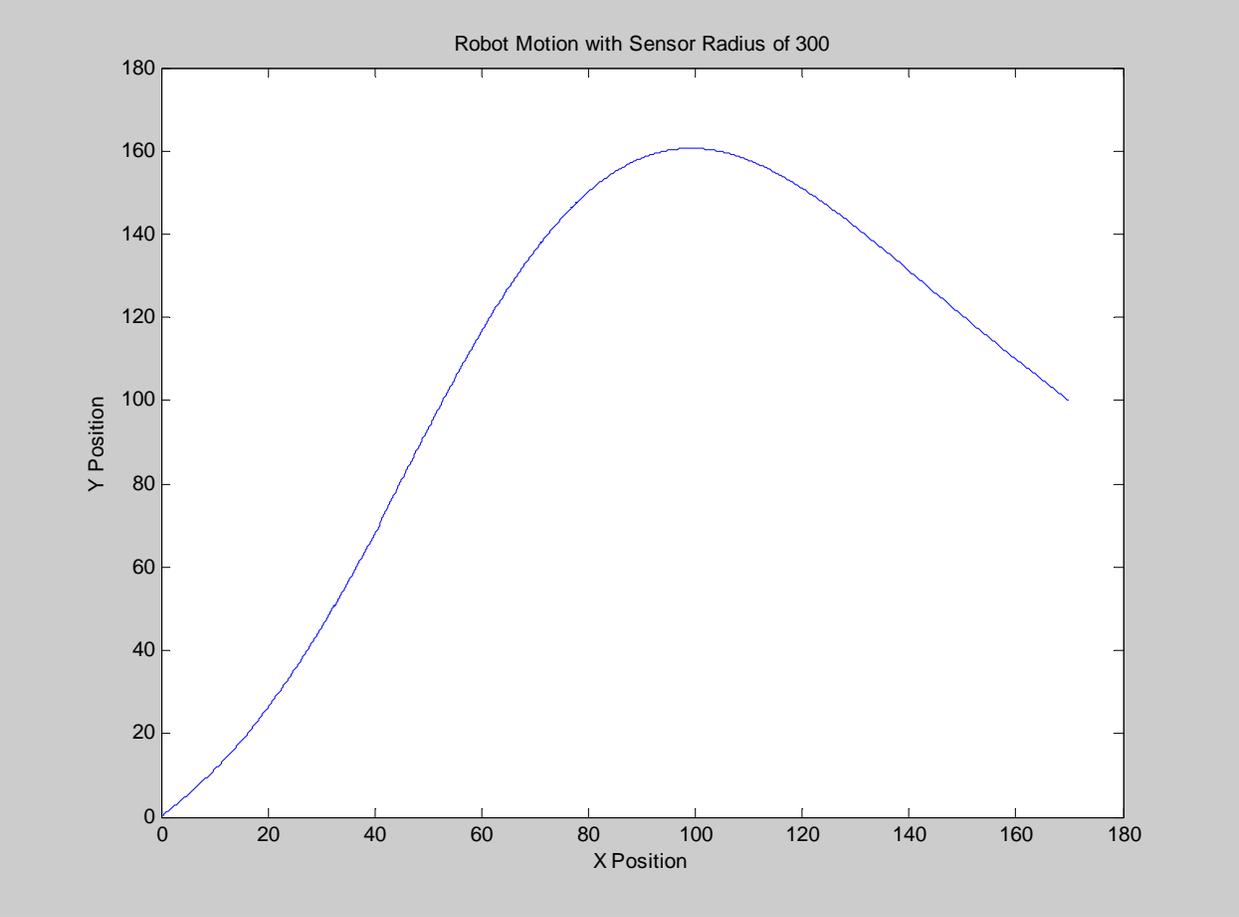
Figure 21 – Robot Motion with Sensor Radius of 300

CHAPTER SIX: CONCLUSION

The simulation proves that the analytical solution to the path planning problem in the presence of moving obstacles presented by Dr. Qu [28] works for the case of open terrain with three moving obstacles. The algorithm is independent of the number of obstacles, which can be verified by examination, so we can conclude that it works for any number of obstacles.

Regarding fixed obstacles, it is easy to see that they would affect the generation of a_6 at all times, since the velocity is not changing.

The following topics could be part of future research to expand on the results of this thesis:

- The selection of the sensor range is very important. Too small a range will not provide sufficient time for the AGV to perform an avoidance maneuver. Too large a range will make avoidance maneuver too large. This sensor distance needs to be optimized for the kinematics and dynamics constraints of the vehicle. This would make a good follow up study of this simulation.

- The actual values of a_6 are very small, making it very susceptible to small changes in the environment. When evaluating the inequality (15), careful considerations must be taken to avoid a numerical rounding error, since we are dealing with very large numbers in place of the variables g_2, g_1, and g_0 and very small numbers for the a coefficients. Perhaps this inequality could be better behaved if some further mathematical manipulation is performed to avoid the extremely large values. This is offer as another possible follow study.

APPENDIX: MATLAB CODE OF AGV.M

```
function [sys,x0,str,ts] = AGV(t,x,u,flag)

%SFUNTMPL General M-file S-function template
%   With M-file S-functions, you can define you own ordinary differential
%   equations (ODEs), discrete system equations, and/or just about
%   any type of algorithm to be used within a Simulink block diagram.
%
%   The general form of an M-File S-function syntax is:
%       [SYS,X0,STR,TS] = SFUNC(T,X,U,FLAG,P1,...,Pn)
%
%   What is returned by SFUNC at a given point in time, T, depends on the
%   value of the FLAG, the current state vector, X, and the current
%   input vector, U.
%
%   FLAG   RESULT          DESCRIPTION
%   -----  ------          -----------------------------------------------
%   0      [SIZES,X0,STR,TS]  Initialization, return system sizes in SYS,
%                          initial state in X0, state ordering strings
%                          in STR, and sample times in TS.
%   1      DX              Return continuous state derivatives in SYS.
%   2      DS              Update discrete states SYS = X(n+1)
%   3      Y               Return outputs in SYS.
%   4      TNEXT           Return next time hit for variable step sample
%                          time in SYS.
%   5                      Reserved for future (root finding).
%   9      []              Termination, perform any cleanup SYS=[].
%
%
%   The state vectors, X and X0 consists of continuous states followed
%   by discrete states.
%
%   Optional parameters, P1,...,Pn can be provided to the S-function and
%   used during any FLAG operation.
%
%   When SFUNC is called with FLAG = 0, the following information
%   should be returned:
%
%       SYS(1) = Number of continuous states.
%       SYS(2) = Number of discrete states.
%       SYS(3) = Number of outputs.
%       SYS(4) = Number of inputs.
%               Any of the first four elements in SYS can be specified
%               as -1 indicating that they are dynamically sized. The
%               actual length for all other flags will be equal to the
%               length of the input, U.
%       SYS(5) = Reserved for root finding. Must be zero.
```

```
%    SYS(6) = Direct feedthrough flag (1=yes, 0=no). The s-function
%            has direct feedthrough if U is used during the FLAG=3
%            call. Setting this to 0 is akin to making a promise that
%            U will not be used during FLAG=3. If you break the promise
%            then unpredictable results will occur.
%    SYS(7) = Number of sample times. This is the number of rows in TS.
%
%
%    X0     = Initial state conditions or [] if no states.
%
%    STR    = State ordering strings which is generally specified as [].
%
%    TS     = An m-by-2 matrix containing the sample time
%             (period, offset) information. Where m = number of sample
%             times. The ordering of the sample times must be:
%
%             TS = [0     0,     : Continuous sample time.
%                   0     1,     : Continuous, but fixed in minor step
%                                   sample time.
%                 PERIOD OFFSET, : Discrete sample time where
%                                   PERIOD > 0 & OFFSET < PERIOD.
%                  -2     0];    : Variable step discrete sample time
%                                   where FLAG=4 is used to get time of
%                                   next hit.
%
%             There can be more than one sample time providing
%             they are ordered such that they are monotonically
%             increasing. Only the needed sample times should be
%             specified in TS. When specifying than one
%             sample time, you must check for sample hits explicitly by
%             seeing if
%                abs(round((T-OFFSET)/PERIOD) - (T-OFFSET)/PERIOD)
%             is within a specified tolerance, generally 1e-8. This
%             tolerance is dependent upon your model's sampling times
%             and simulation time.
%
%             You can also specify that the sample time of the S-function
%             is inherited from the driving block. For functions which
%             change during minor steps, this is done by
%             specifying SYS(7) = 1 and TS = [-1 0]. For functions which
%             are held during minor steps, this is done by specifying
%             SYS(7) = 1 and TS = [-1 1].

%  Copyright 1990-2002 The MathWorks, Inc.
%  $Revision: 1.18 $
```

```matlab
%
% The following outlines the general structure of an S-function.
%
switch flag,

  %%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes(u);

  %%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys=mdlUpdate(t,x,u);

  %%%%%%%%%%%
  % Outputs %
  %%%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u);

  %%%%%%%%%%%%%%%%%%%%%%%%
  % GetTimeOfNextVarHit %
  %%%%%%%%%%%%%%%%%%%%%%%%%
  case 4,
    sys=mdlGetTimeOfNextVarHit(t,x,u);

  %%%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%%%
  case 9,
    sys=mdlTerminate(t,x,u);

  %%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);
```

end

% end sfuntmpl

%
%==
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%==
%
function [sys,x0,str,ts]=mdlInitializeSizes(u)

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded. This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%
sizes = simsizes;

sizes.NumContStates = 4;
sizes.NumDiscStates = 0;
sizes.NumOutputs   = 4;
sizes.NumInputs   = 29;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%

x0 = [0 0 pi/4 0];

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%

```matlab
ts  = [0 0];

% end mdlInitializeSizes

%
%=====================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=========================================================
%
function sys=mdlDerivatives(t,x,u)

   fprintf ('\nIn Derivatives @ Time = %6.2f\n',t);

   persistent x0init y0init theta0init phi0init;
   persistent vo1xlast vo1ylast;
   persistent vo2xlast vo2ylast;
   persistent vo3xlast vo3ylast;
   persistent T1 T2 T3;
   persistent x0last y0last theta0last phi0last;
   persistent xo1last xo2last xo3last;
   persistent yo1last yo2last yo3last;
   persistent obs1 obs2 obs3;
   persistent z1last z2last z3last z4last;

   %  Read initial inputs
   xf=u(5);    %  Final X Pos of AGV
   yf=u(6);    %  Final Y Pos of AGV
   thetaf=u(7);%  Final Theta (body orientation ) of AGV
   phif=u(8);  %  Final Phi (Steering angle)of AGV

   xo1=u(9);   %  X Pos of Obstacle 1
   yo1=u(10);  %  Y Pos of Obstacle 1
   vo1x=u(11); %  X Vel of Obstacle 1
   vo1y=u(12); %  Y Vel of Obstacle 1
   r1=u(13);   %  Radius o Obstacle 1

   xo2=u(14);  %  X Pos of Obstacle 2
   yo2=u(15);  %  Y Pos of Obstacle 2
   vo2x=u(16); %  X Vel of Obstacle 2
   vo2y=u(17); %  Y Vel of Obstacle 2
   r2=u(18);   %  Radius of Obstacle 2

   xo3=u(19);  %  X Pos of Obstacle 3
   yo3=u(20);  %  Y Pos of Obstacle 3
   vo3x=u(21); %  X Vel of Obstacle 3
```

54

```matlab
vo3y=u(22); %  Y Vel of Obstacle 3
r3=u(23);   %  Radius of obstacle 3

if t==0
    %fprintf ('First time:  Initializing initial state of AGV and obstacles\n',t);
    x0init=u(1);   %  Initial X Pos of AGV
    y0init=u(2);   %  Initial Y Pos of AGV
    theta0init=u(3);%  Initial Theta (body orientation) of AGV
    phi0init=u(4);  %  Initial Phi (Steering angle) of AGV

    vo1xlast = vo1x;
    vo1ylast = vo1y;
    vo2xlast = vo2x;
    vo2ylast = vo2y;
    vo3xlast = vo3x;
    vo3ylast = vo3y;

    xo1last = xo1;
    yo1last = yo1;
    xo2last = xo2;
    yo2last = yo2;
    xo3last = xo3;
    yo3last = yo3;

    T1 = 0;
    T2 = 0;
    T3 = 0;

    obs1=0;
    obs2=0;
    obs3=0;
end


l=u(24);    %  Length between axles of AGV
rho=u(25);
T=u(26);    %  Time to complete mission
R=u(27);    %  Radius of AGV
sensor=u(28); %  Sensor distance to detect obstacles
avoid=u(29); %  Flag to determine if AGV should avoid obstacle or not.

%fprintf      ('x0init=%6.2f      y0init=%6.2f      theta0init=%6.2f      phi0init=%6.2f\n',
x0init,y0init,theta0init,phi0init);
%fprintf ('xf=%6.2f yf=%6.2f thetaf=%6.2f phif=%6.2f\n', xf,yf,thetaf,phif);
%fprintf ('l=%6.2f rho=%6.2f T=%6.2f R=%6.2f avoid=%d\n',l,rho,T,R,avoid);
```

```matlab
    % The boundary conditions in the transformed space:
    % Initial point: z0
    z10=x0init-l/2*cos(theta0init);
    z20=1/l*tan(phi0init)/(cos(theta0init))^3;
    z30=tan(theta0init);
    z40=y0init-l/2*sin(theta0init);
    %fprintf ('z10=%6.2f z20=%6.2f z30=%6.2f z40=%6.2f\n', z10,z20,z30,z40);

    % Final point: zf
    z1f=xf-l/2*cos(thetaf);
    z2f=1/l*tan(phif)/(cos(thetaf))^3;
    z3f=tan(thetaf);
    z4f=yf-l/2*sin(thetaf);
    C=(z1f-z10)/T;
    %fprintf ('z1f=%6.2f z2f=%6.2f z3f=%6.2f z4f=%6.2f C=%6.2f\n', z1f,z2f,z3f,z4f, C);


% Read states
  z1=x(1);
  z2=x(2);
  z3=x(3);
  z4=x(4);

  fprintf ('States as read from system:\n');
  fprintf ('    z1=%6.2f z2=%6.2f z3=%6.2f z4=%6.2f \n', z1,z2,z3,z4);


  if t==0  % Initialize states
    %fprintf ('First time:  Initializing States @ Time=%6.2f to \n',t);
     % Initial states
     z1=z10;
     z2=z20;
     z3=z30;
     z4=z40;

     %fprintf ('          z1=%6.2f z2=%6.2f z3=%6.2f z4=%6.2f\n', z1,z2,z3,z4);

  end



% Initialization of a6

a61max=0;
```

```matlab
a61min=0;

a62max=0;
a62min=0;

a63max=0;
a63min=0;

a6min=0;
a6max=0;


  if (t==0)
    z1last = z1;
    z2last = z2;
    z3last = z3;
    z4last = z4;
  end;


  % the matrix B, Y, A in the boundary conditions
  B = [   1 z1last z1last^2 z1last^3   z1last^4   z1last^5; ...
        0 1   2*z1last 3*z1last^2 4*z1last^3  5*z1last^4;...
        0 0   2    6*z1last   12*z1last^2 20*z1last^3;...
        1 z1f z1f^2 z1f^3   z1f^4   z1f^5; ...
        0 1   2*z1f 3*z1f^2 4*z1f^3  5*z1f^4;...
        0 0   2    6*z1f   12*z1f^2 20*z1f^3   ];

  Y=[z4last; z3last; z2last; z4f; z3f; 0];

  A=[ z1last^6; 6*z1last^5; 30*z1last^4; z1f^6; 6*z1f^5; 30*z1f^4 ];

  theta = atan(z3);
  xr = z1+l/2*cos(theta);
  y = z4+l/2*sin(theta);
  phi = atan(z2*l*cos(theta)^3);

  fprintf ('xr=%6.2f y=%6.2f theta=%6.2f phi=%6.2f\n', xr,y,theta,phi);




  if avoid==1    %  avoid is external switch not to avoid obstacles
    %fprintf ('Avoid is on, compute obstacles distance\n');
```

```
    %  Calculate if obstacles are within sensor range
    distance1 = sqrt((yo1-y)^2 + (xo1-xr)^2);
    distance2 = sqrt((yo2-y)^2 + (xo2-xr)^2);
    distance3 = sqrt((yo3-y)^2 + (xo3-xr)^2);

    fprintf ('Dist1=%6.2f  Dist2=%6.2f  Dist3=%6.2f  Sensor=%f\n',  distance1, distance2,
distance3, sensor);
```

```
    if distance1 <= sensor    %  obstacle 1 within sensor range
      fprintf ('Obstacle 1 within sensor range \n');
       if (vo1xlast ~= vo1x) | (vo1ylast ~= vo1y) | obs1==0 % Obs vel changed or new within
sensor
            fprintf   ('         Obstacle   1    changed    velocity    or    new    obstacle
**********************************************************\n');
            vo1xlast = vo1x;
            vo1ylast = vo1y;
            xo1last = xo1;
            yo1last = yo1;
            x0last=xr;
            y0last=y;
            T1=t;
            theta0last=theta;
            phi0last=phi;
            z1last = z1;
            z2last = z2;
            z3last = z3;
            z4last = z4;
            fprintf ('     Changing Init Conditions because of Obs 1\n');
            fprintf ('              x0last=%6.2f  y0last=%6.2f  theta0last=%6.2f  phi0last=%6.2f
T1=%f\n',x0last,y0last,theta0last,phi0last,T1);
            fprintf ('              xo1last=%6.2f  yo1last=%6.2f  vo1xlast=%6.2f  vo1ylast=%6.2f
\n',xo1last,yo1last,vo1xlast, vo1ylast);
%       end
       %fprintf ('xo1last=%f z10=%f l=%f r1=%f \n', xo1last, z10, l, r1);
       %fprintf ('yo1last=%f vo1ylast=%f \n', yo1last, vo1ylast);
       %fprintf ('   R=%f vo1xlast=%f C=%f T1=%f \n', R, vo1xlast, C, T1);

       obs1=1;
```

```
% Time for object 1 (checking interval)
tm11=(xo1last-z10-0.5*l-r1-R-vo1xlast*T1)/(C-vo1xlast);
tm12=(xo1last-z10+r1+R-vo1xlast*T1)/(C-vo1xlast);
fprintf (' tm11=%6.2f tm12=%6.2f\n',tm11,tm12);

if tm12>tm11
   if tm11<T1
      tmin1=T1;
   else
      tmin1=tm11;
   end
   if tm12>T
      tmax1=T;
   else
      tmax1=tm12;
   end
else
   if tm12<T1
      tmin1=T1;
   else
      tmin1=tm12;
   end
   if tm11>T
      tmax1=T;
   else
      tmax1=tm11;
   end
end
fprintf (' tmin1=%6.2f tmax1=%6.2f\n',tmin1,tmax1);


% the possible a6 for obstacle 1
tau1=tmin1:0.01:tmax1;
for i=1:length(tau1)

   % the possible a6 for obstacle 1
   z=z10+C*(tau1(i)-T1);
   %fprintf (' z=%f C=%f tau1(%d)=%f T1=%f\n',z,C,i,tau1(i),T1);
   g2=(z^6-[1 z z^2 z^3 z^4 z^5]*inv(B)*A)^2;
   g1=2*(z^6-[1 z z^2 z^3 z^4 z^5]*inv(B)*A)*...
      ([1 z z^2 z^3 z^4 z^5]*inv(B)*Y-vo1ylast*(tau1(i)-T1)-yo1last);
   g0=([1 z z^2 z^3 z^4 z^5]*inv(B)*Y-vo1ylast*(tau1(i)-T1)-yo1last)^2+...
      (z-xo1last-vo1xlast*(tau1(i)-T1))^2-(r1+R+l/2)^2;
   b4ac=g1^2-4*g2*g0;
   %fprintf (' z=%6.2f g0=%6.3f g1=%6.2f g2=%6.2f b4ac=%6.2f\n',z,g0,g1,g2,b4ac);
```

```matlab
        if b4ac>=0
           %fprintf (' Inside if b4ac>=0\n');
           if sign((-g1-sqrt(b4ac))/(2*g2))~=sign((-g1+sqrt(b4ac))/(2*g2))
              %fprintf ('  Inside if sign:   (-)=%f (+)=%f \n',((-g1-sqrt(b4ac))/(2*g2)), ((-
g1+sqrt(b4ac))/(2*g2)) );
                 if (-g1-sqrt(b4ac))/(2*g2)<a61min
                    a61min=(-g1-sqrt(b4ac))/(2*g2);
                    %fprintf (' a61min=%f\n',a61min);
                 end
                 if (-g1+sqrt(b4ac))/(2*g2)>a61max
                    a61max=(-g1+sqrt(b4ac))/(2*g2);
                    %fprintf (' a61max=%f\n',a61max);
                 end
           end    % end of if sign
        else
           %fprintf (' b4ac less than 0\n');
        end    % end of if b4ac>=0
      end  % end of for loop
      pause;

end %  Testing to do loop only once per velocity change
      fprintf (' a61min=%f a61max=%f \n',a61min, a61max);

    else   %  Obstacle 1 outside of sensor range
       obs1=0;
       fprintf ('Obs 1 outside of sensor range\n');
    end    %  end of if obstacle 1 within sensor range
```

```matlab
    if distance2 <= sensor    %  obstacle 2 within sensor range
       fprintf ('Obstacle 2 within sensor range \n');
       if (vo2xlast ~= vo2x) | (vo2ylast ~= vo2y) | obs2==0 % Obs vel changed or new within
sensor
          fprintf ('       Obstacle    2    changed    velocity    or    new    obstacle
*****************************************\n');
          vo2xlast = vo2x;
          vo2ylast = vo2y;
          xo2last = xo2;
          yo2last = yo2;
```

60

```matlab
            x0last=xr;
            y0last=y;
            T2=t;
            theta0last=theta;
            phi0last=phi;
            z1last = z1;
            z2last = z2;
            z3last = z3;
            z4last = z4;
            fprintf ('       Changing Init Conditions because of Obs 2\n');
            fprintf ('               x0last=%6.2f y0last=%6.2f theta0last=%6.2f phi0last=%6.2f
T2=%f\n',x0last,y0last,theta0last,phi0last,T2);
            fprintf ('               xo2last=%6.2f yo2last=%6.2f vo2xlast=%6.2f vo2ylast=%6.2f
\n',xo2last,yo2last,vo2xlast, vo2ylast);
%        end
        %fprintf (' xo2last=%f z10=%f l=%f r2=%f \n', xo2last, z10, l, r2);
        %fprintf (' yo2last=%f vo2ylast=%f \n', yo2last, vo2ylast);
        %fprintf ('    R=%f vo2xlast=%f C=%f T2=%f \n', R, vo2xlast, C, T2);

        obs2=1;

        % Time for object 2 (checking interval)
        tm21=(xo2last-z10-0.5*l-r2-R-vo2xlast*T2)/(C-vo2xlast);
        tm22=(xo2last-z10+r2+R-vo2xlast*T2)/(C-vo2xlast);
        fprintf (' tm21=%6.2f tm22=%6.2f\n',tm21,tm22);

        if tm22>tm21
          if tm21<T2
             tmin2=T2;
          else
             tmin2=tm21;
          end
          if tm22>T
             tmax2=T;
          else
             tmax2=tm22;
          end
        else
          if tm22<T2
             tmin2=T2;
          else
             tmin2=tm22;
          end
          if tm21>T
             tmax2=T;
          else
```

61

```matlab
            tmax2=tm21;
          end
        end
        fprintf ('  tmin2=%6.2f tmax2=%6.2f\n',tmin2,tmax2);


        % the possible a6 for obstacle 2
        tau2=tmin2:0.01:tmax2;
        for i=1:length(tau2)

          % the possible a6 for obstacle 2
          z=z10+C*(tau2(i)-T2);
          %fprintf ('  z=%f C=%f tau2(%d)=%f T2=%f\n',z,C,i,tau2(i),T2);
          g2=(z^6-[1 z z^2 z^3 z^4 z^5]*inv(B)*A)^2;
          g1=2*(z^6-[1 z z^2 z^3 z^4 z^5]*inv(B)*A)*...
             ([1 z z^2 z^3 z^4 z^5]*inv(B)*Y-vo2ylast*(tau2(i)-T2)-yo2last);
          g0=([1 z z^2 z^3 z^4 z^5]*inv(B)*Y-vo2ylast*(tau2(i)-T2)-yo2last)^2+...
             (z-xo2last-vo2xlast*(tau2(i)-T2))^2-(r2+R+l/2)^2;
          b4ac=g1^2-4*g2*g0;
          %fprintf ('  z=%6.2f g0=%6.3f g1=%6.2f g2=%6.2f b4ac=%6.2f\n',z,g0,g1,g2,b4ac);

          if b4ac>=0
            %fprintf ('  Inside if b4ac>=0\n');
            if sign((-g1-sqrt(b4ac))/(2*g2))~=sign((-g1+sqrt(b4ac))/(2*g2))
              %fprintf ('   Inside if sign:   (-)=%f (+)=%f \n',((-g1-sqrt(b4ac))/(2*g2)), ((-g1+sqrt(b4ac))/(2*g2)) );
              if (-g1-sqrt(b4ac))/(2*g2)<a62min
                a62min=(-g1-sqrt(b4ac))/(2*g2);
                %fprintf ('  a62min=%f\n',a62min);
              end
              if (-g1+sqrt(b4ac))/(2*g2)>a62max
                a62max=(-g1+sqrt(b4ac))/(2*g2);
                %fprintf ('  a62max=%f\n',a62max);
              end
            end    % end of if sign
          else
            %fprintf ('  b4ac less than 0\n');
          end    % end of if b4ac>=0
        end  % end of for loop
        pause;

end % Testing to do loop only once per velocity change
        fprintf ('  a62min=%f a62max=%f \n',a62min, a62max);

    else    %  Obstacle 2 outside of sensor range
      obs2=0;
```

```
        fprintf ('Obs 2 outside of sensor range\n');
     end    %  end of if obstacle 2 within sensor range
```



```
     if distance3 <= sensor    %  obstacle 3 within sensor range
        fprintf ('Obstacle 3 within sensor range \n');
        if (vo3xlast ~= vo3x) | (vo3ylast ~= vo3y) | obs3==0 % Obs vel changed or new within
sensor
            fprintf ('          Obstacle   3   changed   velocity   or   new   obstacle
*****************************************\n');
            vo3xlast = vo3x;
            vo3ylast = vo3y;
            xo3last = xo3;
            yo3last = yo3;
            x0last=xr;
            y0last=y;
            T3=t;
            theta0last=theta;
            phi0last=phi;
            z1last = z1;
            z2last = z2;
            z3last = z3;
            z4last = z4;
            fprintf ('     Changing Init Conditions because of Obs 3\n');
            fprintf ('               x0last=%6.2f  y0last=%6.2f  theta0last=%6.2f  phi0last=%6.2f
T3=%f\n',x0last,y0last,theta0last,phi0last,T3);
            fprintf ('               xo3last=%6.2f  yo3last=%6.2f  vo3xlast=%6.2f  vo3ylast=%6.2f
\n',xo3last,yo3last,vo3xlast, vo3ylast);
%          end
        %fprintf ('  xo3last=%f z10=%f l=%f r3=%f \n', xo3last, z10, l, r3);
        %fprintf ('  yo3last=%f vo3ylast=%f \n', yo3last, vo3ylast);
        %fprintf ('    R=%f vo3xlast=%f C=%f T3=%f \n', R, vo3xlast, C, T3);

        obs3=1;
```

```matlab
% Time for object 3 (checking interval)
tm31=(xo3last-z10-0.5*l-r3-R-vo3xlast*T3)/(C-vo3xlast);
tm32=(xo3last-z10+r3+R-vo3xlast*T3)/(C-vo3xlast);
fprintf (' tm31=%6.2f tm32=%6.2f\n',tm31,tm32);

if tm32>tm31
   if tm31<T3
      tmin3=T3;
   else
      tmin3=tm31;
   end
   if tm32>T
      tmax3=T;
   else
      tmax3=tm32;
   end
else
   if tm32<T3
      tmin3=T3;
   else
      tmin3=tm32;
   end
   if tm31>T
      tmax3=T;
   else
      tmax3=tm31;
   end
end
fprintf (' tmin3=%6.2f tmax3=%6.2f\n',tmin3,tmax3);


% the possible a6 for obstacle 3
tau3=tmin3:0.01:tmax3;
for i=1:length(tau3)

   % the possible a6 for obstacle 3
   z=z10+C*(tau3(i)-T3);
   %fprintf (' z=%f C=%f tau3(%d)=%f T3=%f\n',z,C,i,tau3(i),T3);
   g2=(z^6-[1 z z^2 z^3 z^4 z^5]*inv(B)*A)^2;
   g1=2*(z^6-[1 z z^2 z^3 z^4 z^5]*inv(B)*A)*...
      ([1 z z^2 z^3 z^4 z^5]*inv(B)*Y-vo3ylast*(tau3(i)-T3)-yo3last);
   g0=([1 z z^2 z^3 z^4 z^5]*inv(B)*Y-vo3ylast*(tau3(i)-T3)-yo3last)^2+...
      (z-xo3last-vo3xlast*(tau3(i)-T3))^2-(r3+R+l/2)^2;
   b4ac=g1^2-4*g2*g0;
   %fprintf (' z=%6.2f g0=%6.3f g1=%6.2f g2=%6.2f b4ac=%6.2f\n',z,g0,g1,g2,b4ac);
```

```matlab
        if b4ac>=0
           %fprintf ('  Inside if b4ac>=0\n');
           if sign((-g1-sqrt(b4ac))/(2*g2))~=sign((-g1+sqrt(b4ac))/(2*g2))
              %fprintf ('    Inside  if  sign:    (-)=%f  (+)=%f  \n',((-g1-sqrt(b4ac))/(2*g2)),  ((-
g1+sqrt(b4ac))/(2*g2)) );
                 if (-g1-sqrt(b4ac))/(2*g2)<a63min
                    a63min=(-g1-sqrt(b4ac))/(2*g2);
                    %fprintf ('  a63min=%f\n',a63min);
                 end
                 if (-g1+sqrt(b4ac))/(2*g2)>a63max
                    a63max=(-g1+sqrt(b4ac))/(2*g2);
                    %fprintf ('  a63max=%f\n',a63max);
                 end
           end    %  end of if sign
        else
           %fprintf ('  b4ac less than 0\n');
        end    %  end of if b4ac>=0
     end  %  end of for loop
     pause;

end %  Testing to do loop only once per velocity change

        fprintf ('  a63min=%f a63max=%f \n',a63min, a63max);

     else   %  Obstacle 3 outside of sensor range
        obs3=0;
        fprintf ('Obs 3 outside of sensor range\n');
     end    %  end of if obstacle 3 within sensor range




        obs = obs1+obs2+obs3;

     if obs == 3    %  Three obstacles within sensor range
        fprintf ('Three obstacles within sensor range\n');
        %determine a6 based on 3 objects
        if min([a61min, a63min])< a6min
           a6min=min([a61min, a63min]);
        end
```

```
    if max([a61max,  a63max])>a6max
       a6max=max([a61max,  a63max]);
    end
    if sign(a6min)==sign(a6max)
       a6=0;
    else
       if abs(a6min)<=abs(a6max)
          a6=a6min;
       else
          a6=a6max;
       end
    end
    fprintf ('a6=%6.2f\n',a6);
  elseif obs == 2 %  Two obstacles within sensor range
    fprintf ('Two obstacles within sensor range\n');
     %determine a6 based on object 1 and object 2
     if min([a61min, a62min])< a6min
       a6min=min([a61min, a62min]);
     end
     if max([a61max, a62max])>a6max
       a6max=max([a61max, a62max]);
     end
     if sign(a6min)==sign(a6max)
       a6=0;
     else
       if abs(a6min)<=abs(a6max)
          a6=a6min;
       else
          a6=a6max;
       end
     end
     fprintf ('a6=%6.2f\n',a6);
  elseif obs == 1 %  One obstacle within sensor range
    fprintf ('One obstacle within sensor range\n');
    if sign(a61min)==sign(a61max)
       a6=0;
    else
       if abs(a61min)<=abs(a61max)
          a6=a61min; %a61min;
       else
          a6=a61max; %a61max;
       end
    end
    fprintf ('a6=%6.2f\n',a6);
  else       %  No obstacle within sensor range
    fprintf ('No obstacle within sensor range\n');
```

```matlab
        a6=0;
     end


    %fprintf ('Avoid is on, a6=%6.2f\n',a6);
  else    % avoid==0
       a6=0;
       fprintf ('Avoid is off, a6=0\n');

  end    %  end of if avoid==1




  % calculate the remaining coefficients a0 to a5
  a012345=inv(B)*Y-inv(B)*A*a6;
  a0=a012345(1);
  a1=a012345(2);
  a2=a012345(3);
  a3=a012345(4);
  a4=a012345(5);
  a5=a012345(6);

  fprintf ('a0=%f a1=%f a2=%f a3=%f a4=%f a5=%f a6=%f \n', a0,a1,a2,a3,a4,a5,a6);

  % calculate the steering inputs:
  C0=6*(a3+4*a4*z1last+10*a5*z1last^2+20*a6*z1last^3)*C;
  C1=24*(a4+5*a5*z1last+15*a6*z1last^2)*C^2;
  C2=60*(a5+6*a6*z1last)*C^3;
  C3=120*a6*C^4;

  fprintf ('C0=%f C1=%f C2=%f C3=%f \n', C0,C1,C2,C3);



  % the trajectory in z plane
%    z1 = z10 + C*t;
%    z2 = z2  + C0*t + C1*t^2/2 +...
%        C2*t^3/3 + C3*t^4/4;
%    z3 = z3  + C*z2*t + C*C0*t^2/2 +...
%        C*C1*t^3/6 + C*C2*t^4/12 + C*C3*t^5/20;
%    z4 = z4  + C*z3*t + C^2*z2*t^2/2 +...
%        C^2*C0*t^3/6 + C^2*C1*t^4/24 +...
%        C^2*C2*t^5/60 + C^2*C3*t^6/120;


  %fprintf ('NEW: z1=%6.2f z2=%6.2f z3=%6.2f z4=%6.2f \n', z1,z2,z3,z4);
```

```matlab
    v1=C;
    v2=C0+C1*t+C2*t^2+C3*t^4;
    z1dot = v1;
    z2dot = v2;
    z3dot = z2*v1;
    z4dot = z3*v1;

    fprintf ('z1dot=%6.2f z2dot=%6.2f z3dot=%6.2f z4dot=%6.2f\n', z1dot,z2dot,z3dot,z4dot);

sys = [z1dot z2dot z3dot z4dot];

% end mdlDerivatives

%
%==============================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%==============================================================
%
function sys=mdlUpdate(t,x,u)

sys = [ ];



% end mdlUpdate

%
%==============================================================
% mdlOutputs
% Return the block outputs.
%==============================================================
%
function sys=mdlOutputs(t,x,u)
fprintf ('\nIn Outputs @ Time = %6.2f\n',t);

l=u(24);    %  Length between axles of AGV


z1 = x(1);
z2 = x(2);
z3 = x(3);
z4 = x(4);

fprintf ('z1=%6.2f z2=%6.2f z3=%6.2f z4=%6.2f\n', z1,z2,z3,z4);
```

```
if (z1==0) & (z2==0) & (z3==0) & (z4==0)  % Initialize states
    x0init=u(1);    %  Initial X Pos of AGV
    y0init=u(2);    %  Initial Y Pos of AGV
    theta0init=u(3);%  Initial Theta (body orientation) of AGV
    phi0init=u(4);  %  Initial Phi (Steering angle) of AGV
    fprintf ('Initializing Outputs\n');
    theta = theta0init;
    xr = x0init;
    y = y0init;
    phi = phi0init;
else
    theta = atan(z3);
    xr = z1+l/2*cos(theta);
    y = z4+l/2*sin(theta);
    phi = atan(z2*l*cos(theta)^3);
end


fprintf ('xr=%6.2f y=%6.2f theta=%6.2f phi=%6.2f\n', xr,y,theta,phi);


sys = [xr y theta phi];

% end mdlOutputs

%
%=============================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the result is
% absolute time.  Note that this function is only used when you specify a
% variable discrete-time sample time [-2 0] in the sample time array in
% mdlInitializeSizes.
%=========================================================
%
function sys=mdlGetTimeOfNextVarHit(t,x,u)

sampleTime = 1;    %  Example, set the next hit to be one second later.
sys = t + sampleTime;

% end mdlGetTimeOfNextVarHit

%
%=================================================================
% mdlTerminate
% Perform any end of simulation tasks.
```

```
%===============================================================
%
function sys=mdlTerminate(t,x,u)

sys = [];

% end mdlTerminate
```

LIST OF REFERENCES

[1] J. Latombe, Robot Motion Planning. Boston, MA: Kluwer, 1998.

[2] E. Rimon and D. E. Koditschek, "Exact robot navigation using artificial potential functions," IEEE Trans. Robot. Automat., vol. 8, pp. 501–518, Oct. 1992.

[3] J. Borenstein and Y. Koren, "The vector field histogram—Fast obstacle avoidance for mobile robots," IEEE Trans. Robot. Automat., vol. 7, pp. 278–288, June 1991.

[4] J.-P. Laumond, Robot Motion Planning and Control. London, U.K.: Springer-Verlag, 1998.

[5] R. M. Murray, Z. Li, and S. S. Sastry, A Mathematical Introduction to Robotic Manipulation. Boca Raton, FL: CRC Press, 1994.

[6] H. J. Sussmann and W. Liu, "Limits of Highly Oscillatory Controls and the Approximation of General Paths by Admissible Trajectories," Rutgers Ctr. Systems and Control, Piscataway, NJ, Tech. Rep. SYSCON-91-02, Feb. 1991.

[7] M. Fliess, J. Levine, Ph. Martin, and P. Rouchon, "Flatness and defect of nonlinear systems: Introductory theory and examples," Int. J. Contr., vol. 61, pp. 1327–1361, 1995.

[8] R. M. Murray and S. S. Sastry, "Nonholonomic motion planning: Steering using sinusoids," IEEE Trans. Automat. Contr., vol. 38, pp. 700–716, May 1993.

[9] S. Monaco and D. Normand-Cyrot, "An introduction to motion planning under multirate digital control," in Proc. 31st Conf. Decision and Control, Tucson, AZ, Dec. 1992, pp. 1780–1785.

[10] D. Tilbury, R. M. Murray, and S. S. Sastry, "Trajectory generation for the n-trailer problem using goursat normal form," IEEE Trans. Automat. Contr., vol. 40, pp. 802–819, May 1995.

[11] C. Fernandes, L. Gurvits, and Z. Li, "Near-optimal nonholonomic motion planning for a system of coupled rigid bodies," IEEE Trans. Automat. Contr., vol. 39, pp. 450–463, Mar. 1994.

[12] J. A. Reeds and R. A. Shepp, "Optimal paths for a car that goes both forward and backward," Pacific J. Math., vol. 145, pp. 367–393, 1990.

[13] H. J. Sussmann and G. Tang, "Shortest Paths for the Reeds–Shepp Car: A Worked Out Example of the Use of Geometric Techniques in Nonlinear Optimal Control," Rutgers Univ., Piscataway, NJ, Tech. Rep. SYSCON-91-10, 1991.

[14] S. Sundar and Z. Shiller, "Optimal obstacle avoidance based on the Hamilton–Jacobi–Bellman equation," IEEE Trans. Robot. Automat., vol. 13, pp. 305–310, Apr. 1997.

[15] A. E. Bryson and Y.-C. Ho, Applied Optimal Control, 2nd ed. New York: Hemisphere, 1975.

[16] Z. Qu and J. R. Cloutier, "A new suboptimal control design for cascaded nonlinear systems," Optimal Contr.: Applicat. Methods, vol. 23, pp. 303–328, Nov. 2002.

[17] J.-P. Laumond, P. E. Jacobs, M. Taix, and R. M. Murray, "A motion planner for nonholonomic mobile robots," IEEE Trans. Robot. Automat., vol. 10, pp. 577–593, Oct. 1994.

[18] J. Barraquand and J.-C. Latombe, "Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles," in Proc. IEEE Int. Conf. Robotics and Automation, Sacramento, CA, Apr. 1991, pp. 2328–2335.

[19] A.W. Divelbiss and J. T.Wen, "A path space approach to nonholonomic motion planning in the presence of obstacles," IEEE Trans. Robot. Automat., vol. 13, pp. 443–451, June 1997.

[20] B. Donald, P. Xavier, J. Canny, and J. Reif, "Kinodynamic motion planning," in J. Assoc. Comput. Machinery, vol. 40, 1993, pp. 1048–1066.

[21] S. Lavalle and J. Kuffner, "Randomized kinodynamic planning," Int. J. Robot. Res., vol. 20, pp. 378–400, 2001.

[22] M. Erdmann and T. Lozano-Perez, "On multiple moving objects," in Proc. IEEE Int. Conf. Robotics and Automation, San Francisco, CA, Apr. 1986, pp. 1419–1424.

[23] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," Int. J. Robot. Res., vol. 21, pp. 233–255, 2002.

[24] K. Kant and S. W. Zucker, "Planning collision free trajectories in timevarying environments: A two-level hierarchy," in Proc. IEEE Int. Conf. Robotics and Automation, Raleigh, NC, 1988, pp. 1644–1649.

[25] P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles," Int. J. Robot. Res., vol. 17, pp. 760–772, 1998.

[26] Z. Shiller, F. Large, and S. Sekhavat, "Motion planning in dynamic environments: Obstacles moving along arbitrary trajectories," in Proc. IEEE Int. Conf. Robotics and Automation, Seoul, Korea, May 2001, pp. 3716–3721.

[27] Vatana An, "A third order differential steering robot and trajectory generation in the presence of moving obstacles" MS thesis, University of Central Florida, December, 2005

[28] Zhihua Qu, Jing Wang, Clinton E. Plaisted, "A New analytical Solution to Mobile Robot Trajectory Generation in the Presence of Moving Obstacles", IEEE Transactions on Robotics, Vol 20, No. 6, December 2004, pp. 978-993.