

# Development of the “Discretized Dynamic Expanding Zones with Memory” Autonomous Mobility Algorithm for the Nemesis Tracked Vehicle Platform

Grant Edward Gothing

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science  
in  
Mechanical Engineering

Dr. Charles F. Reinholtz, Chairman  
Dept. of Engineering Education

Dr. Alfred L. Wicks, Co-Chair  
Dept. of Mechanical Engineering

Dr. Dennis W. Hong  
Dept. of Mechanical Engineering

July 30, 2007  
Blacksburg, Virginia

Keywords: Autonomous Vehicles, Unmanned, JAUS, Dynamic Expanding Zones, Obstacle Memory, Collaboration

# Development of the “Discretized Dynamic Expanding Zones with Memory” Autonomous Mobility Algorithm for the Nemesis Tracked Vehicle Platform

Grant Edward Gothing

## ABSTRACT

The Nemesis tracked vehicle platform is a differentially driven Humanitarian Demining tractor developed by Applied Research Associates, Inc. The vehicle is capable of teleoperational control and is outfitted with a sensor suite used for detecting and neutralizing landmines. Because the detection process requires the vehicle to travel at speeds less than 0.5 km/h, teleoperation is a tedious process. The added autonomous capabilities of waypoint navigation and obstacle avoidance could greatly reduce operator fatigue.

ARA chose to leverage Virginia Tech’s experience in developing an autonomous mobility capability for the Nemesis platform. The resulting algorithms utilize the waypoint navigation techniques of Virginia Tech’s JAUS (Joint Architecture for Unmanned Systems) toolkit, and a modified version of the Dynamic Expanding Zones (DEZ) algorithm developed for the 2005 DARPA Grand Challenge. The modified approach discretizes the perception zones of the DEZ algorithm and provides the added capability of obstacle memory, resulting in the Discretized Dynamic Expanding Zones with Memory (DDEZm) algorithm. These additions are necessary for efficient autonomous control of the differentially driven Nemesis vehicle.

The DDEZm algorithm was coded in LabVIEW and used to autonomously navigate the Nemesis vehicle through a waypoint course while avoiding obstacles. The Joint Architecture for Unmanned Systems (JAUS) was used as the communication standard to facilitate the interoperability between the software developed at Virginia Tech and the existing Nemesis software developed by ARA. In addition to development and deployment, the algorithm has been fully documented for embedded coding by a software engineer. With embedded implementation on the vehicle, this algorithm will help to increase the efficiency of the landmine detection process, ultimately saving lives.

# Acknowledgements

There are several people who I am indebted to for their support and guidance in helping me get to where I am today.

First and foremost, I would like to thank my mother and father, Susan and Gilbert, and my sister, Elise. I couldn't have made it without their support and love. My mother's hard work and devotion is what made attending Virginia Tech possible for me. Her strength, kindness, and selflessness are an inspiration for everything I do. I would like to thank my father for the experience, knowledge, and guidance he has given me, and for always reminding me to use my head. He taught me that being an engineer is so much more than equations and bookwork; and that just because something works on paper, there's no guarantee that it will work in the real world. These are just a few of the lessons that I'll remember for the rest of my life. Lastly, I couldn't ask for a better sister. She is one of my best friends and can always make me smile. She is also one of the hardest working people I know and never ceases to amaze me.

I would also like to thank my Massachusetts friends, Beth, Lizzy, Pam, and Patrick, for listening to all my frustrations and helping me keep my feet on the ground and my head on my shoulders. You guys mean the world to me.

Finally, although she can't talk and will never be able to read this, I would like to thank my dog, Casey. She is the reason I still have my sanity. She cheered me up on bad days, kept me exercised, and constantly reminded me that while working hard is important, sometimes playing is just a little more important.

Next, I would like to thank my advisor Dr. Reinholtz, who first got me excited about autonomous vehicles. He has been a wonderful professor and advisor, and has given me constant motivation and encouragement along the way. I would also like to thank my committee members, Drs. Wicks and Hong, for their added knowledge and support. In addition, thanks to all of my colleagues who have helped me out along the way. Ruel Faruque is the person responsible for getting me into

J AUS and has taught numerous things about LabVIEW, messages, and computing in general. A big thanks also goes to Jon Weekley, Andrew Bacha, Mike Webster, Jesse Hurdus, Dave Van Covern, Steve Cacciola and Patrick Currier for all of the little tidbits of knowledge that saved me hours of frustration.

Last but not least, I would like to thank Applied Research Associates for their funding, support, and the opportunity to work with the Nemesis project. It has been a great experience. A major portion of this research was sponsored by the Humanitarian Demining Research and Development Program of the US Army Night Vision and Electronic Sensors Directorate (NVESD).

# Table of Contents

<b>Abstract</b> .....	ii
<b>Acknowledgements</b> .....	iii
<b>List of Acronyms</b> .....	viii
<b>List of Figures</b> .....	ix
<b>List of Tables</b> .....	xi
<b><u>Chapter 1: Introduction and Background</u></b> .....	1
1.1 Thesis overview .....	1
1.2 Project Motivation .....	1
1.3 The Nemesis Platform .....	2
1.4 Navigation Schemes .....	4
1.4.1 Deliberative .....	5
1.4.2 Reactive .....	5
1.4.3 Hybrid Reactive-Deliberative.....	7
1.5 Virginia Tech’s Autonomous Navigation Experience.....	8
1.5.1 Intelligent Ground Vehicle Competition.....	9
1.5.2 DARPA Grand Challenge .....	11
1.6 LabVIEW Overview .....	13
1.7 Joint Architecture for Unmanned Systems (JAUS).....	14
1.7.1 System Topology and Levels of Interoperability .....	15
1.7.2 JAUS Messages.....	16
1.7.3 Transport Interface .....	17
1.7.4 Virginia Tech JAUS Toolkit for LabVIEW .....	17
1.8 ARA Subcontract to Virginia Tech .....	19
1.8.1 Phases .....	19
1.8.2 Requirements.....	22
1.8.3 Documentation .....	24
<b><u>Chapter 2: The Discretized Dynamic Expanding Zones with Memory Algorithm</u></b> .....	25
2.1 Relationship to Previous Virginia Tech Algorithms.....	25
2.2 Vector Conversions.....	27
2.3 Waypoint Navigation.....	28
2.4 Obstacle Sensing.....	30
2.4.1 Obstacle Remembering .....	31
2.5 Obstacle Avoidance .....	31

2.5.1	Perception Zones .....	32
2.5.2	Obstacle Avoidance.....	34
2.6	Speed Calculation .....	36
2.6.1	Heading-Based Speed .....	37
2.6.2	Distance-Based Speed .....	38
2.7	Rate Limiters.....	40
2.8	Difficult Situation Handling .....	40
2.9	Component States .....	41
<b><u>Chapter 3:</u></b>	<b>Algorithm Implementation .....</b>	<b>44</b>
3.1	JAUS Interfacing .....	44
3.1.1	Issues .....	44
3.1.2	Messages .....	46
3.2	LabVIEW Implementation .....	47
3.2.1	Main Process .....	48
3.2.2	LRF Reader Process .....	48
3.2.3	Visualization Process .....	49
3.3	Testing .....	50
3.3.1	Simulation .....	50
3.3.2	On-Vehicle .....	51
3.3.3	Continued Testing .....	52
3.4	Parameters.....	53
3.4.1	Zone Sizes .....	53
3.4.2	Avoidance Settings.....	54
3.4.3	Waypoint Threshold.....	56
3.4.4	Velocity Parameters .....	57
<b><u>Chapter 4:</u></b>	<b>Flowchart Conventions .....</b>	<b>58</b>
4.1	Data.....	58
4.1.1	Arrays .....	59
4.2	Processes.....	60
4.2.1	Scripts and Variable Assignments.....	61
4.2.2	Arrays .....	62
4.2.3	Time .....	64
4.3	Decisions.....	64
4.4	Wires.....	65
4.4.1	Junctions.....	65
4.5	Terminators.....	67
<b><u>Chapter 5:</u></b>	<b>Algorithm Documentation .....</b>	<b>68</b>
5.1	Function Hierarchy .....	68

5.2	Functions.....	71
5.2.1	Main .....	72
5.2.2	Status .....	75
5.2.3	Waypoint Navigation .....	77
5.2.4	Calculate Zone Length .....	79
5.2.5	Reflexive Driver .....	81
5.2.6	Check Situation .....	84
5.2.7	Rate Limiter.....	86
5.3	Variables.....	88
5.3.1	Velocity Parameters .....	89
5.3.2	Zone Sizes .....	90
5.3.3	Avoidance Settings.....	91
5.3.4	Global Pose .....	92
5.3.5	Waypoint Threshold.....	92
5.3.6	Waypoints.....	93
5.3.7	Travel Speed.....	93
5.3.8	Obstacle Points .....	94
5.3.9	Component Status .....	94
<b><u>Chapter 6:</u></b>	<b>Results .....</b>	<b>95</b>
6.1	Simulation Testing.....	95
6.2	On-Vehicle Testing.....	98
<b><u>Chapter 7:</u></b>	<b>Conclusions and Future Work .....</b>	<b>101</b>
7.1	Conclusions.....	101
7.2	Future Work.....	102
<b><u>References</u></b>	.....	<b>105</b>
<b><u>Vita</u></b>	.....	<b>106</b>

# List of Acronyms

ARA	Applied Research Associates
ASV	All Season Vehicles
AUVSI	Association for Unmanned Vehicle Systems International
AZ	Avoidance Zone
CERDEC	Communications-Electronics Research, Development, and Engineering Center
DARPA	Defense Advanced Research Projects Agency
DDEZm	Discretized Dynamic Expanding Zones with Memory
DEZ	Dynamic Expanding Zones
EMI	Electro Magnetic Induction
ETG	Experimentation Task Group
GPS	Global Positioning System
GPSAR	Ground Penetrating Synthetic Aperature Radar
GUI	Graphical User Interface
HD	Humanitarian Demining
ICD	Interface Control Document
IGVC	Intelligent Ground Vehicle Competition
JAUS	Joint Architecture for Unmanned Systems
JGRE	Joint Ground Robotics Enterprise
LRA	Left Rear A
LRB	Left Rear B
LRF	Laser Range Finder
MRCSS	Modular Robotics Control System
NED	New England Division
NUTS	Non-Uniform Terrain Search
NVESD	Night Vision and Electronic Sensors Directorate
OCS	Operator Control Station
OCU	Operator Control Unit
OS	Operating System
OUSD	Office for the Under Secretary of Defense
RA	Reference Architecture
RDECOM	U.S. Army Research, Development and Engineering Command
RRA	Right Rear A
RRB	Right Rear B
SAE	Society of Automotive Engineers
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UTM	Universal Transverse Mercator
VI	Virtual Instrument
VLF	Vehicle Length in Front of center of gravity

# List of Figures

1.1	ARA Nemesis Platform.....	3
1.2	The OCS used to control Nemesis .....	3
1.3	Nemesis with a mine detection sensor suite.....	4
1.4	Schematic of the Deliberative Navigation Scheme.....	5
1.5	Reactive Navigation Scheme .....	5
1.6	Vector Fields for Waypoint Navigation and Obstacle Avoidance.....	6
1.7	Summation of the previous vector fields.....	6
1.8	Schematic of the Hybrid Navigation Scheme .....	8
1.9	Gemini.....	9
1.10	Rocky .....	9
1.11	IGVC obstacle avoidance paths .....	11
1.12	Zone Layout for the DEZ Algorithm .....	12
1.13	Turning arc used for DEZ obstacle avoidance .....	13
1.14	LabVIEW Block Diagram and Front Panel .....	14
1.15	Levels of JAUS Interoperability .....	16
2.1	Comparison of DDEZm and DEZ.....	26
2.2	Vector conversion reference systems .....	27
2.3	Distance and angle to a waypoint.....	29
2.4	Diagram of the sensor offsets on the vehicle .....	31
2.5	DDEZm layout and dimensions .....	32
2.6	Lateral Summation .....	35
2.7	Avoidance Heading.....	36
2.8	Heading-Based Speed curve.....	38
2.9	Distance-Based Speed curve .....	39
2.10	State Diagram for the DDEZm algorithm.....	43
3.1	LRF Data polar plot.....	49
3.2	Screenshot of the overhead visualization .....	49
3.3	Screenshot of the Simulator Environment .....	51
3.4	Dimensions used in obstacle point conversion .....	55
3.5	Obstacle memory zones .....	56
4.1	Data Representations for inputs, outputs, and previous iterations .....	58
4.2	Elemental array splitting .....	60
4.3	Representations for a basic and complex process .....	61
4.4	Data name size reduction .....	61
4.5	Decision Representations .....	65

4.6	Flowchart representation for process and data wires .....	65
4.7	Wire Merging .....	66
4.8	Terminator Representations .....	67
5.1	Function Hierarchy.....	69
5.2	Main Function Flowchart, Part 1.....	73
5.3	Main Function Flowchart, Part 2.....	74
5.4	Flowchart for the Status function .....	76
5.5	Flowchart for the Waypoint Navigation function .....	78
5.6	Flowchart for the Calculate Zone Length function .....	80
5.7	Flowchart for the Reflexive Driver function.....	82
5.8	Flowchart for the Check Situation function .....	85
5.9	Flowchart for the Rate Limiter function.....	87
6.1	Simulator screenshot with numbered saypoints .....	95
6.2	Simulator Screenshot with overlaid vehicle path .....	97
6.3	Nemesis under autonomous control .....	98
7.1	Obstacle Avoidance with Fuzzy Logic .....	103

# List of Tables

3.1	Behavioral Parameters.....	53
4.1	Data Type Abbreviations .....	59
4.2	Operators Used in Script Blocks .....	62
5.1	Function Hierarchy.....	70
5.2	Status Specifications .....	75
5.3	Waypoint Navigation Specifications.....	77
5.4	Calculate Zone Length Specifications.....	79
5.5	Reflexive Driver Specifications .....	81
5.6	Check Situation Specifications.....	84
5.7	Rate Limiter Specifications .....	86
5.8	Velocity Parameters Specifications.....	89
5.9	Zone Sizes Specifications.....	90
5.10	Avoidance Settings Specifications .....	91
5.11	Global Pose Specifications.....	92
5.12	Waypoint Threshold Specifications .....	92
5.13	Waypoints Specifications .....	93
5.14	Travel Speed Specifications .....	93
5.15	Obstacle Points Specifications .....	94
5.16	Component Status Specifications.....	94
6.1	Final Values for the Behavioral Parameters.....	100

# **Chapter 1: Introduction and Background**

## **1.1 Thesis Overview**

This thesis presents the Discretized Dynamic Expanding Zones with Memory (DDEZm) autonomous mobility algorithm that was developed for the Nemesis platform, a Humanitarian Demining vehicle developed by Applied Research Associates (ARA). The work represents a successful collaboration between industry and academia. The algorithm, which has been coded in National Instruments LabVIEW, provides Nemesis with the autonomous capabilities of waypoint navigation and obstacle avoidance. The thesis starts with by introducing the motivation and background for the research. This includes a description of the Nemesis platform, review of the basic autonomous navigation schemes, an explanation of Virginia Tech's experience with autonomous vehicles, an overview of LabVIEW, a basic description of the Joint Architecture for Unmanned Systems (JAUS), and finally a description of ARA's subcontract to Virginia Tech. After the background is established, Chapter 2 gives provides a description of the algorithm and the theory behind it. Chapter 3 discusses the implementation of the algorithm on the Nemesis platform. Chapter 4 provides a basic documentation of the autonomous mobility algorithm including flowcharts and functional descriptions. Chapter 5 presents the results of both simulation and on-vehicle testing. Finally, Chapter 6 draws conclusions on the project and provides some recommendations for future work.

## **1.2 Project Motivation**

In the past few decades, there has been an increasing interest in autonomous vehicles for a variety of applications. They are employed in situations that are undesirable or unsuitable for a human operator. While in some situations, such as a long car ride, automation may be primarily a convenience, other situations pose a serious threat to the safety of the driver or operator. Often times, these situations involve combat zones, bomb sniffing, or search and rescue missions. One especially

important application for autonomous vehicles is in the detection and removal of land mines. According to a United Nations database “it is estimated that more than 110 million active mines are scattered in 70 countries,” and that “every month over 2,000 people are killed or maimed by mine explosions. Most of the casualties are civilians who are killed or injured after hostilities have ended” [7]. With as many as 24,000 innocent people being killed every year by land mines, an effective means for removing this threat must be established. According to the same United Nations database, while the average land mine costs about \$3 to \$30 and is quick and easy to deploy, neutralization is a slow process that costs between \$300 and \$1000 per mine [7]. By using autonomous vehicles to perform this dangerous, yet repetitive, task the cost can be reduced. Autonomy allows a single operator to control multiple vehicles rather than teleoperating a single manual vehicle. The goal of this project was to provide autonomous navigation capabilities to a current Humanitarian Demining (HD) vehicle in order to reduce operator fatigue and increase efficiency.

### **1.3 The Nemesis Platform**

Applied Research Associates, Inc. has developed the Nemesis platform, a Humanitarian Demining (HD) vehicle, based on the RC-60 model by All-Season Vehicles (ASV), for the purpose of detecting and neutralizing landmines. The work has been supported by the US Army RDECOM, CERDEC, Night Vision and Electronic Sensors Directorate (NVESD), Ft Belvoir, VA. Nemesis is a tracked utility vehicle with a differential steering system, as shown in Figure 1.1. This platform is outfitted with ARA’s Modular Robotic Control System (MRCS) which consists of the platform control components and a Universal Operator Control Station (OCS) that is used to teleoperate the vehicle. The OCS is a suitcase style 45 lb man-portable control station, as shown in Figure 1.2. Data and video is sent between the vehicle and OCS via the MRCS’s wireless data and video link. The OCS allows complete control of general vehicle mobility, loader arms, and pan/tilt/zoom cameras. A touchscreen monitor also allows the operator to view various feedback data such as the Global Positioning System (GPS) position, platform information including fuel level and oil temperature, and landmine

detection data [8]. Closed-loop speed and heading control have also been implemented on the vehicle. The entire system is currently compliant with Reference Architecture 3.2 of the Joint Architecture for Unmanned Systems (JAUS). This allows other JAUS interoperable organizations, such as Virginia Tech, to easily connect to and control the vehicle. A more in-depth explanation of JAUS will be given later.



**Figure 1.1.** ARA Nemesis Platform. An ASV RC-60 base vehicle.



**Figure 1.2.** The OCS used to control Nemesis

During demining operation, a suite of detection sensors consisting of PSI Ground Penetrating Synthetic Aperture Radar (GPSAR) and Minelab Electromagnetic Inductance (EMI) arrays is attached to the front of the vehicle, as

shown in Figure 1.3. The detection process requires the vehicle to drive at speeds not exceeding 0.5 km/h [8]. This makes the task of teleoperating the vehicle tedious and leads to operator fatigue, particularly when driving solely based on the vehicle's camera feedback. By implementing an autonomous mobility package, the computer can handle the basic navigation, allowing the operator to focus on mine detection, possibly from multiple vehicles, simultaneously.



**Figure 1.3.** Nemesis with a mine detection sensor suite.

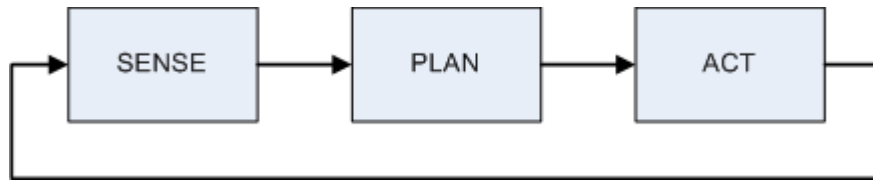
## 1.4 Navigation Schemes

In the domain of robotics there are three commonly accepted primitives used in controlling an intelligent vehicle: sense, plan, and act. Sensing is the task of taking in sensor data and interpreting it. The plan stage involves using the sensor information to develop a strategy to navigate through the world. Finally, the act phase is where the decisions that are made are translated into vehicle control signals. There are three general paradigms for combining these primitives to control a vehicle: deliberative, reactive, and hybrid deliberative-reactive.

### 1.4.1 Deliberative

The deliberative approach to autonomous navigation uses the Sense-Plan-Act scheme shown in Figure 1.4. In this method, all of the sensor data is used by the plan primitive to calculate the vehicle's path. This involves combining all sensor

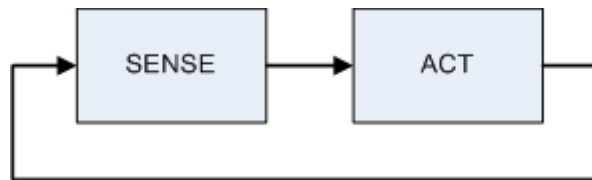
data to create a map of the vehicle's environment, and analyzing all possible solutions to determine the best path. The act primitive then interprets this path and outputs the vehicle control parameters. Therefore, for each iteration, the vehicle senses the world, plans its reaction to the environment, and acts according to this plan. The main advantage of the deliberative approach is that, if a solution to a navigation problem exists, it almost always will be found. However, the creation and analysis of a world map, which is required for this approach, can be computationally expensive [4].



**Figure 1.4.** Schematic of the Deliberative Paradigm.

### 1.4.2 Reactive

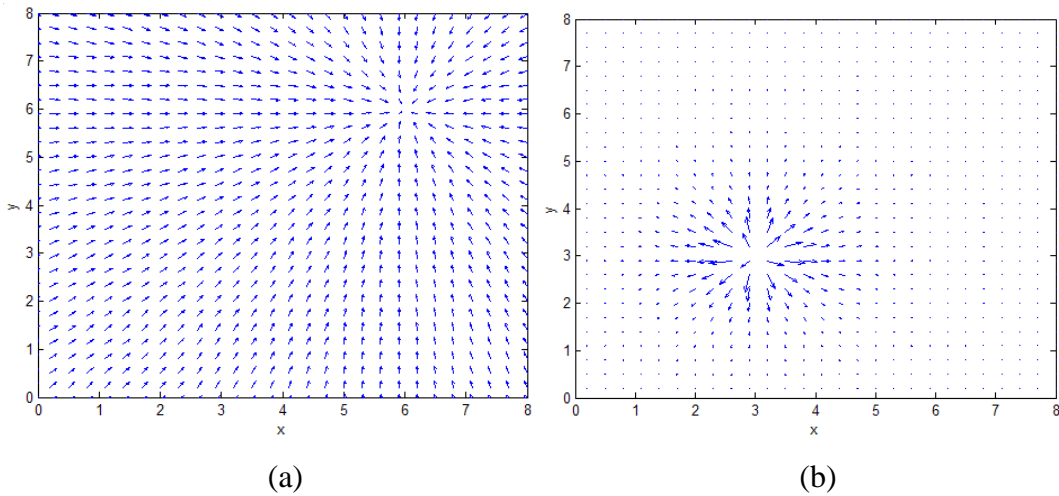
The reactive navigation scheme eliminates the plan section of the deliberative approach, as shown in Figure 1.5. Instead of planning the best path, this approach uses a series of behaviors that are combined to create a single emergent behavior. Each individual behavior represents a single sense-act coupling that operates independent of the other behaviors [4]. There are many methods for combining the outputs these behaviors. Two of the most well known are the potential fields and subsumption architectures.



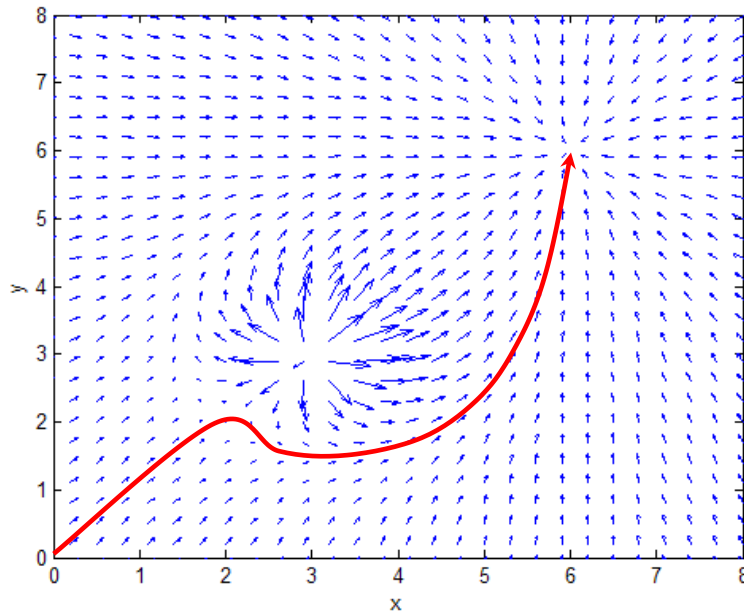
**Figure 1.5.** Reactive Navigation Scheme. Diagram showing the Sense-Act coupling.

The potential fields architecture uses a weighted summation approach to behavior determination. Each stimulus in the environment produces a force field that affects the vehicle's path. The sum of these force fields creates an emergent behavior that guides the vehicle. Figure 1.6 shows an example of waypoint navigation with obstacle avoidance using potential fields. Here, the waypoint exerts

a uniform attractive field, while the obstacle exerts a repulsive field that becomes stronger as the vehicle approaches the obstacle. By summing the effects of these two vector fields, the final behavior, shown in Figure 1.7, is produced. A drawback to this approach is that the obstacle may affect the vehicle's behavior even if it does not actually obstruct its path. The tuning of each stimuli's vector field and the combined weightings is a great challenge in itself.



**Figure 1.6.** Vector fields for with (a) a waypoint and (b) an obstacle.



**Figure 1.7.** Summation of the previous vector fields. The selected path is also shown in red.

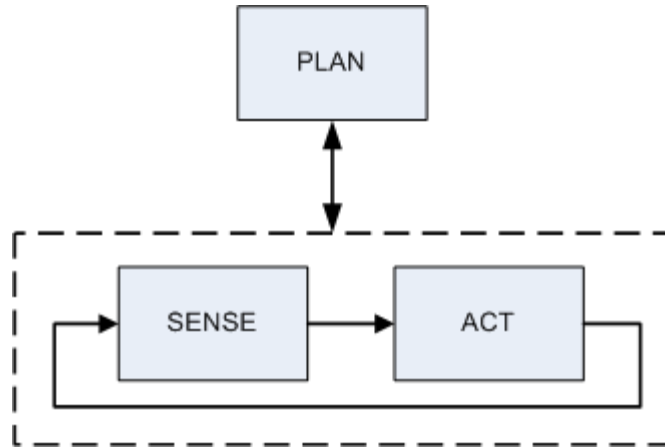
In the subsumption architecture, each behavior is given a priority level. With several behaviors at different priority levels, a hierarchy is formed. The vehicle will

perform the lowest level task until a certain situation prompts the need for another task [4]. For example, in the waypoint navigation/obstacle avoidance problem, the obstacle avoidance behavior takes priority over waypoint navigation. Therefore, the vehicle will perform its waypoint navigation behavior until it encounters an obstacle. This will trigger the obstacle avoidance behavior. Once the vehicle is clear of obstacles, it will return to the waypoint navigation behavior.

Because the reactive scheme does not use the plan primitive that is found in the deliberative approach, the vehicle can sometimes become stuck in a situation it is incapable of handling. However, with the correct tuning of behaviors, many of these situations can be accounted for and solved. The main advantage of the reactive paradigm is that it does not require the creation of a complex world model. The vehicle simply observes the world and draws an immediate conclusion based on this data. Therefore, this approach is much less computationally expensive and can run at much faster update rates [4]. The Nemesis platform is expected to always have an operator in supervisory control. This operator can easily take manual control in complex situations. For these reasons, reactive method was chosen for the autonomous control of the Nemesis vehicle.

### **1.4.3 Hybrid Reactive-Deliberative**

The hybrid approach shown in Figure 1.8 is a way of using the positive aspects of both the deliberative and reactive approaches. The planning stage of the deliberative approach is combined with the close sense-act couplings of the reactive scheme. Here, the planner acts as a higher-level decision making component that breaks a large mission down into subtasks, which are given to the reactive driver as they should be executed [4]. While this is a robust approach, it can be extremely complicated and difficult to implement.



**Figure 1.8.** Schematic of the Hybrid Navigation Scheme. Representation of how the Planning component and Sense-Act coupling interact.

## 1.5 Virginia Tech's Autonomous Navigation Experience

Through the past several years, teams from Virginia Tech have gained prestige in the autonomous vehicle world. Among many notable achievements, they have won the Grand Award at the Intelligent Ground Vehicle Competition (IGVC) in 2004, 2005, 2006 and 2007. One of the Virginia Tech Vehicles, Gemini, is shown in Figure 1.9. In the 2005 DARPA Grand Challenge, Virginia Tech entered two vehicles, Cliff and Rocky, and placed 8<sup>th</sup> and 9<sup>th</sup> out of 23 in the final competition. Figure 1.10 shows Rocky. For a more in-depth discussion of these two vehicles see [5] and [9]. This section presents a closer look at what was involved in the two competitions. More specifically, the obstacle avoidance approaches used for each competition are analyzed to establish the background of the algorithm designed for the Nemesis platform.



**Figure 1.9.** Gemini, one of Virginia Tech's entries to the 2006 IGVC.



**Figure 1.10.** Rocky. This is one of the two Ingersoll-Rand Club Cars that Virginia Tech entered into the 2005 Grand Challenge.

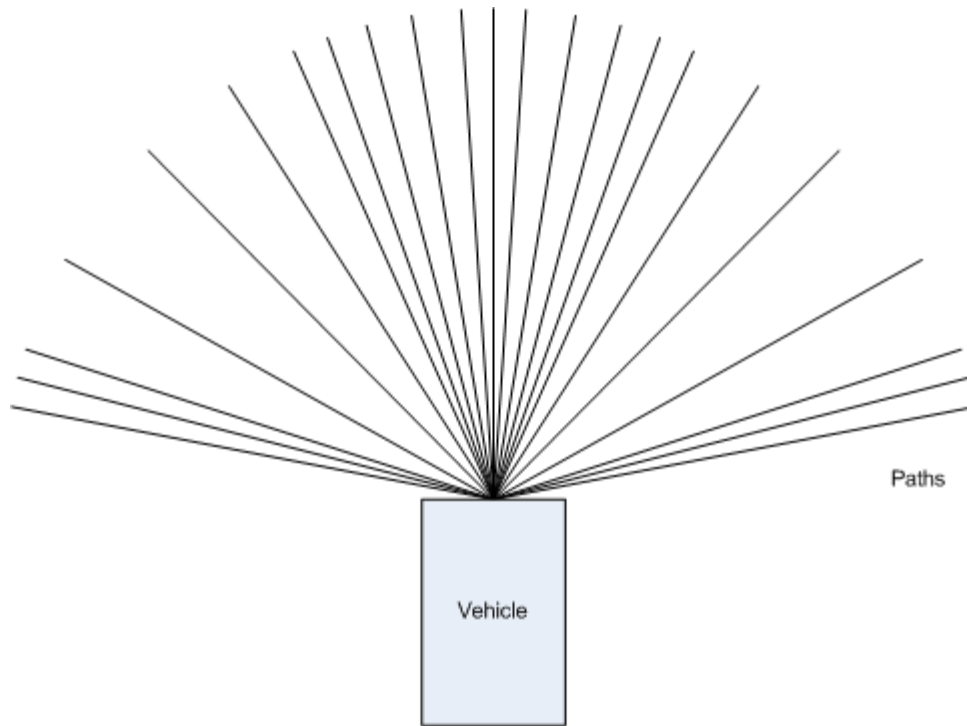
### **1.5.1 Intelligent Ground Vehicle Competition**

The IGVC is an international collegiate competition which requires students to design, build and program a small autonomous ground robot. The Association for Unmanned Vehicle Systems International (AUVSI) has held the event every year since 1993. The competition teams generally consist of students from various engineering backgrounds including Mechanical Engineering, Electrical Engineering, and Computer Science/Engineering. While building a functional robot and outfitting

it with sensors is a project in itself, the main goal of the competition is to promote interest in autonomous navigation. There are three events in the competition: Autonomous Challenge, Navigation Challenge, and Design Competition. While the Design Competition is based on the overall design of the vehicle and its systems, the Autonomous and Navigation Challenges are based on the performance of the vehicle. In the Autonomous Challenge, vehicles must negotiate an obstacle course outlined by painted lines. The Navigation Challenge requires vehicles to follow GPS waypoints while avoiding obstacles [2].

### **Obstacle Avoidance**

Both the Navigation Challenge and the Autonomous Challenge used the same obstacle avoidance algorithm. This was a cost-based approach where a set of 24 discrete paths were evaluated. These paths were represented as desired headings and were initialized at the start of the algorithm. Figure 1.11 shows a representation of the paths. In addition to a straight ahead path, the initialized headings were 3, 7, 10, 15, 20, 27, 35, 45, 55, 60, 61, and 62 degrees on both the right and left side of the vehicle. On each iteration of the algorithm, every path was assigned a cost based on the initial desired heading (to the waypoint) and the distance to the closest obstacle on that path as well as the two paths on either side of it. The cost of a path was increased for closer obstacles and decreased if it was close to the desired heading. Nearby paths with obstacles also slightly increased the cost of the path being analyzed. The path with the lowest cost was chosen as the desired path to avoid the obstacles. Because the cost analysis was so optimized for the IGVC, it was difficult to modify to create different behaviors. Therefore, the 2005 DARPA Grand Challenge obstacle avoidance was considered.



**Figure 1.11.** IGVC obstacle avoidance paths.

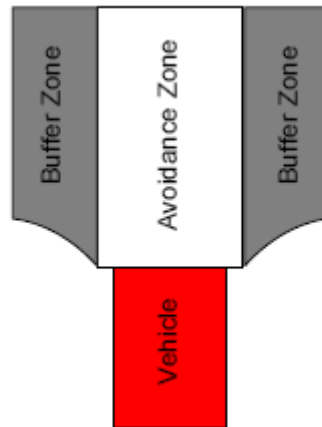
### 1.5.2 DARPA Grand Challenge

The DARPA Grand Challenge was a competition sponsored by the Defense Advanced Research Projects Agency (DARPA) that was first held in March of 2004. Because there were no finishers for this race, a second Grand Challenge competition was held on October 8, 2005. Virginia Tech's first vehicle, Cliff, a modified Ingersoll Rand Club Car, made it to the final event in 2004 but barely left the starting gate before stopping due to a brake software issue. For the 2005 event, Virginia Tech brought back an improved version of Cliff back in addition to a new platform named Rocky. Both vehicles were accepted for the final event and fared better than in the previous event, with Cliff placing 8<sup>th</sup> and Rocky placing 9<sup>th</sup>. The vehicles went 44 and 39 miles, respectively. Although neither vehicle finished the course, both stopped due to mechanical equipment failure. While the two vehicles had similar base platforms, they were designed with completely different software approaches. Cliff followed the reactive paradigm using the Dynamic Expanding Zones (DEZ) algorithm, while Rocky's deliberative algorithm was called the Non-

Uniform Terrain Search (NUTS). Because the DEZ algorithm is reactive, as desired for the Nemesis platform, it is discussed here in more detail.

### Dynamic Expanding Zones

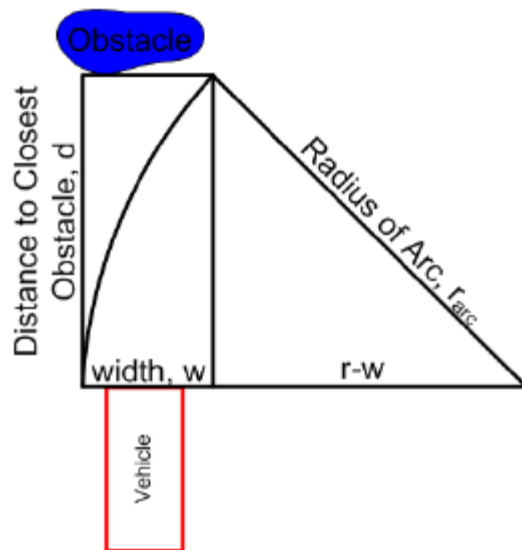
The Dynamic Expanding Zones (DEZ) algorithm is a method of obstacle avoidance that uses a rectangular avoidance zone directly in front of the vehicle to determine which obstacles are in the vehicle's immediate path. The zone is slightly wider than the vehicle to maintain a safe distance from obstacles on the sides, and expands in length as the vehicle drives faster, to account for the greater stopping distance. Because the Avoidance Zone only identifies which obstacles are directly in front of the vehicle, a buffer zone is used on either side to prevent the vehicle from turning into an obstacle. The buffer zone sizes are based on the curvature of the vehicle's path. A tighter turn results in a wider buffer zone. If the buffer zone is occupied during a desired turn, the vehicle must drive straight until the zone is clear. Figure 1.12 shows a top view of the vehicle and zones [5].



**Figure 1.12.** Zone layout for the DEZ algorithm.

All obstacles inside the Avoidance Zone must be avoided by the vehicle. First, the avoidance direction is calculated using the lateral summation of all obstacles within a defined obstacle window. This window is one meter long, starting from the closest obstacle, and twice the width of the Avoidance Zone. The lateral

distance from the centerline of the vehicle to each of the obstacle points within the window are summed. Points to the left of the centerline are given negative values, while points to the right are positive. If the summation is negative there are more obstacles to the left of the vehicle; likewise, a positive summation indicates more obstacles to the right. The vehicle should then avoid to the side that has fewer obstacles. The steering angle is calculated based on an arc with a radius that will safely guide the vehicle away from the obstacles, as shown in Figure 1.13. The DEZ algorithm worked very well for obstacle avoidance in the Grand Challenge. It is easily modifiable to produce a desired vehicle behavior [5].

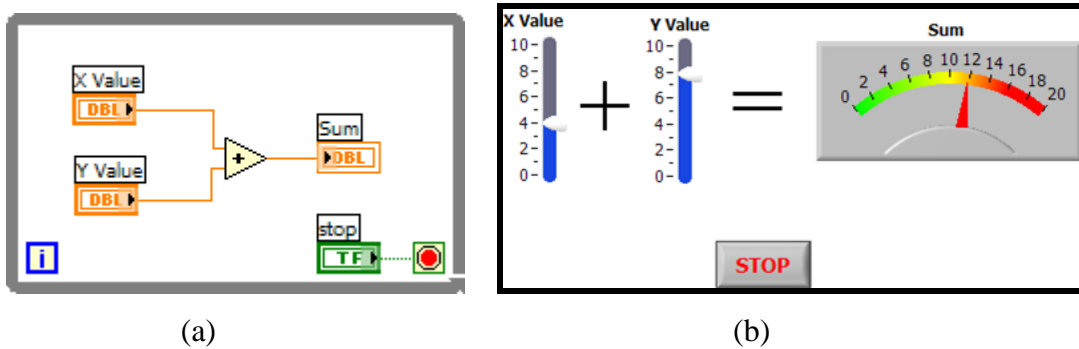


**Figure 1.13.** Turning arc used for DEZ obstacle avoidance.

## 1.6 LabVIEW Overview

LabVIEW is a graphical programming language developed by National Instruments that is easy to use and facilitates rapid development. A program, known as a Virtual Instrument or VI, contains two parts, the Block Diagram and the Front Panel. The processes of an application are programmed by connecting functions blocks, called subVIs, with wires of various data types on the VI's Block Diagram. Execution order is based on the flow of the wires through the program. As soon as a

subVI has all of its data, it will execute. In addition, LabVIEW makes parallel processing and multi-threading easy, because it automatically handles the division of processor time between each subVI that is executing. The Front Panel of the VI is the Graphical User Interface (GUI). An advantage of LabVIEW is that it automatically links the Front Panel objects, called controls (user inputs) and indicators (program feedback), with Block Diagram variables. Therefore, the design of the GUI goes hand-in-hand with the program development. Figure 1.14 shows a simple example of a Block Diagram and Front Panel used for adding two numbers and updating the result.



**Figure 1.14.** LabVIEW (a) block diagram and (b) front panel. The program is used for adding two numbers.

Virginia Tech has been using LabVIEW for programming autonomous vehicles since 2002. The fast development time, ease of debugging, and built-in hardware interfaces make it ideal for the limited time-frame required of the student projects. Because it has a fast learning curve compared to text-based programming languages, students with limited programming backgrounds can quickly become involved in the development of the algorithms and software.

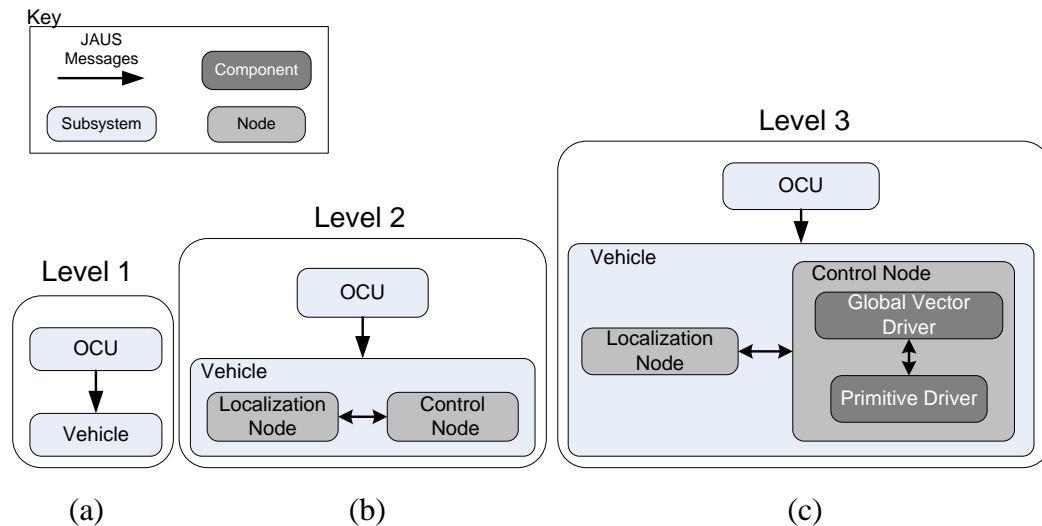
## 1.7 Joint Architecture for Unmanned Systems (JAUS)

The Joint Architecture for Unmanned Systems (JAUS) is a standard message set and transport specification in the field of unmanned and autonomous vehicles. In its simplest terms, JAUS is an upper-level design that defines the messages and interfaces which allow vehicles and computing elements to interoperate, independent

of developing organization, vehicle platform, operating system, and programming language. It also provides the ability to develop modular software components, promoting code reusability and reducing development time. JAUS is sponsored by the Office of the Under Secretary of Defense (OUSD) for Acquisition, Technology and Logistics, and mandated for use by all of the programs in the Joint Ground Robotics Enterprise (JGRE) [3].

### **1.7.1 JAUS System Topology and Levels of Interoperability**

The JAUS System Topology identifies three distinct elements of an unmanned system: Subsystems, Nodes, and Components. These elements correspond to vehicles (or Operator Control Units (OCU)), computing elements, and software applications, respectively. In addition, a System could be defined as a group of Subsystems, including both OCUs and vehicles. A good way to visualize this topology would be to say that a Subsystem (vehicle) contain a set of Nodes (computers), which each run various Components (applications) to perform all the tasks necessary to function. There are three levels of JAUS interoperability based on this hierarchy. In Level 1 interoperability, JAUS messages are only passed between subsystems. This is shown in Figure 1.15a where an OCU from one organization sends JAUS commands to a vehicle made by a different organization. All internal computing occurs independently of the JAUS standard. When JAUS messages are passed between the computing elements of a subsystem, Level 2 interoperability has been achieved. In Figure 1.15b, the vehicle is Level 2 interoperable because there is a localization node and a control node which share information via JAUS messages. Because both computers are JAUS interoperable, it does not matter what operating system they are running or what organization programmed them. Finally, in the case where multiple components on a single node communicate using JAUS messages, they are said to be Level 3 interoperable. Therefore, if Organization A writes a Primitive Driver in C++, Organization B writes a Global Vector Driver in LabVIEW and they are both Level 3 interoperable, they will work together with minimal, or no, effort. This is shown in Figure 1.15c. In addition, Level 3 interoperability assumes Level 2 interoperability, which, then, assumes Level 1 [3].



**Figure 1.15.** Levels of JAUS Interoperability. a) Level 1, b) Level 2, c) Level 3

The collaboration between Virginia Tech and ARA was made smoother due to both organizations' use of JAUS. During the first phase of the project, the Virginia Tech navigation algorithm communicated with the Nemesis platform via Level 2 JAUS interoperability. The navigation algorithm was run in LabVIEW on a Windows laptop while the ARA software ran C++ on Linux machines. Because JAUS messages were used for communication between the two organizations, interoperability was virtually seamless.

## 1.7.2 JAUS Messages

As stated earlier, JAUS defines a large set of standard messages that promote interoperability among products from different organizations. Standard to each message is a 16 byte header which defines the properties of the message, including the message code, source and destination components, sequence number, version number, and various flags (experimental flag, service connection flag). The message code is a two byte number that identifies the message. Based on this code the receiving component knows how to parse the data in the message body. Each message code has its own specifications for how data is represented in the message body. The body of the message follows the header and contains the actual data that

is sent [3]. Messages are divided up into four types: command, core, query, and report. Command messages are used to control a certain aspect of a vehicle or component. This includes messages for teleoperating a vehicle, driving via desired vectors, setting waypoints, and setting a desired travel speed. Core messages must be supported by every component. They include messages to change the component's state (standby, ready, shutdown, emergency) as well as requesting and rejecting control over a component. In essence, core messages are subset of the command class. Query and report messages are generally used together. Any type of general information, such as global pose, vehicle speed, or current status, is sent via the report message class. Because query messages are used to ask for this information, every report has an associated query message [3]. Thus, JAUS provides a common message set that allows software from different organizations using different programming languages and operating systems to easily communicate.

### **1.7.3 Transport Interface**

In addition to specifying the data contained in each message, JAUS provides a set of rules for sending the messages over a particular protocol, such as UDP, TCP, USB, serial, or shared memory. Due to its speed and efficiency most organizations choose to use UDP to send JAUS messages. The typical problem with UDP is that packets can be received out of order or dropped entirely, so it is up to the application to check for errors. However, vehicle control is time critical and packets are sent at a high rate. Therefore, dropped or incorrectly sequenced packets are less of a problem than the delayed packets encountered in TCP. In addition, UDP supports broadcasting of packets to all computers tied to a local network. This is especially appealing during the process of discovering all other subsystems and nodes. Typically, all JAUS UDP traffic travels over Port 3794. The maximum message size is 4096 bytes including the 16 byte application header [3].

### **1.7.4 Virginia Tech JAUS Toolkit for LabVIEW**

The use of JAUS in the collaboration between Virginia Tech and ARA would not have been possible if both organizations did not already support it. At Virginia

Tech, there is a preexisting JAUS Toolkit for LabVIEW. The toolkit provides a framework for creating new components by providing component templates and handling all UDP communications, message routing, message building and parsing, and data structures. It is interoperable with any other system that follows the JAUS Reference Architecture v3.2 and the Interface Control Documents (ICD) for the Experimentation Task Group (ETG) experiments 2.75 and 3.0 [1]. The toolkit provides the flexibility to run components as a stand-alone subsystem or node (Level 1 and Level 2 interoperability, respectively) or use the available Node Manager and Communicator for message routing (Level 3 interoperability). In addition to supporting all of the standard JAUS messages and services, tools are provided to create new experimental messages and capabilities, making the toolkit extremely versatile.

Because of the versatility and convenience, the JAUS Toolkit for LabVIEW has been used to provide JAUS interoperability for a multitude of autonomous vehicles at Virginia Tech. Such vehicles include: a 2004 Cadillac SRX, IGVC vehicles “Johnny 5” and “Gemini”, a Mesa Robotics MATILDA, and DARPA Grand Challenge Vehicle “Rocky” (after competition). In addition to vehicles, a JAUS interoperable vehicle simulator, LRF Payload, and OCU have also been developed using the toolkit [1]. During an impromptu test between ARA and Virginia Tech before this contract was initiated, the lead designer of the toolkit, was able to control the Nemesis vehicle using Virginia Tech’s OCU. After debugging some initial header issues, the OCU was able to perform basic teleoperation control of the tractor. Next, Virginia Tech implemented some of ARA’s experimental messages, which only took two hours. This allowed Virginia Tech’s OCU to power the vehicle on and off, start and stop the engine, teleoperate the tractor, and put it into an emergency stop mode. Because of the versatility and ease of development using the toolkit, complete interoperability took less than three hours. The JAUS toolkit was essential in almost all of Virginia Tech’s autonomous vehicle work by allowing the focus to be on developing capabilities rather than handling communications and messages.

## **1.8 ARA Subcontract to Virginia Tech**

In September of 2006, Virginia Tech was subcontracted by Applied Research Associates, Inc. to develop an autonomous mobility algorithm for the Nemesis project. This algorithm was to be based on the algorithms originally implemented on Virginia Tech's Intelligent Ground Vehicle Competition (IGVC) and 2005 DARPA Grand Challenge vehicles. More specifically, the algorithm was to employ waypoint navigation and obstacle avoidance on the tractor. The algorithm that was developed is presented as the research of this thesis. With the work set to take one year, the contract was executed in two phases. After the phases of the project are outlined, the general requirements will be presented.

### **1.8.1 Phases**

*Phase 1 – Algorithm Development using Virginia Tech specified computer hardware and software*

During the first phase Virginia Tech was responsible for transitioning their autonomous mobility software that was initially deployed in the IGVCs and the 2005 DARPA Grand Challenge to the Nemesis robotic platform. This first phase was broken down into three tasks.

*Task 1 – Familiarization*

Virginia Tech was given access to ARAs Nemesis platform in order to become familiar with its hardware and software. This involved a visit to ARA's New England Division (NED) in South Royalton, Vermont. During this visit, Virginia Tech was introduced to the main vehicle and computing systems, onboard sensors, and vehicle capabilities. There were also discussions on the JAUS interface between the Virginia Tech hardware/software, and the Nemesis platform, required obstacle avoidance sensors, and overall goals of the project.

### *Task 2 – Deployment*

Virginia Tech developed the autonomous mobility algorithm using National Instrument's LabVIEW. During development, the algorithm was tested using a simulator developed at Virginia Tech to help with the IGVC and Grand Challenge testing. Once finished, the software was deployed to the Nemesis platform during a visit by Virginia Tech to ARA's NED. The code was run on Windows laptop and interacted with the Nemesis vehicle using JAUS. The initial hardware conflicts were resolved and the algorithm autonomously controlled the tractor. While the software worked, there was still a lot of room for improvement through modification of the algorithm and further testing.

### *Task 3 – Testing*

Because testing is an iterative process, there was a large amount of communication between Virginia Tech and ARA following the initial deployment of the algorithm onto the Nemesis platform. After the deployment visit, the necessary modifications to the code were made, and a revised copy was sent back to ARA. Because the initial conflicts were worked out during the visit, Virginia Tech did not need to travel to Vermont to further test the algorithm. Testing was also expedited through the use of Virginia Tech's simulator.

### *Deliverable*

As a deliverable for Phase 1, Virginia Tech supplied ARA with a Windows application (.exe) that performed waypoint navigation and obstacle avoidance on the Nemesis vehicle. This .exe was run on the same windows laptop as the initial deployment and testing.

### *Phase 2 – Documentation and revision of autonomous mobility algorithm*

The initial plan for Phase 2 was for Virginia Tech to convert the algorithm written in LabVIEW to C++. However, through discussions

with ARA, it was established that it would be more effective to fully and meticulously document the algorithm so that it could later be coded by ARA's software engineers. In addition, ARA noted some areas of the algorithm that they wanted to be improved. These areas included remembering obstacles that the vehicle passed and determining when a situation required operator assistance. As a result Phase 2 was altered to have the following three tasks:

*Task 1 – Documentation of initial algorithm*

The first part of Phase 2 required Virginia Tech to fully document the initial algorithm that was developed in Phase 1. This documentation included all flowcharts, tables, and descriptions necessary to describe the algorithm in full detail. Because the plan for this documentation was for ARA to later code the algorithm in C++, every detail of it had to be explicitly described. This resulted in a 67 page document that contained 14 flowcharts and 37 variables. Each flowchart and variable was described by its own textual description to minimize misunderstandings.

*Task 2 – Revision*

Because there were additional features that ARA requested from the algorithm, some revisions had to be made. The first addition involved remembering obstacles that passed out of the view of the sensor. This was desirable to address the problem of the vehicle turning into obstacles that it passed and no longer saw. The second addition was implemented to provide the vehicle with the capability of asking the operator for assistance when it got into a situation it had difficulty solving. Additional changes in the algorithm included making the vehicle scan farther ahead for obstacles as it traveled faster, and adding the support for a JAUS obstacle position message. This JAUS obstacle position message is an experimental JAUS message that the two organizations developed to allow the sensor to be decoupled from the avoidance algorithm. Instead

of the algorithm directly reading the sensor, a sensor interface component can send a JAUS message containing the obstacle data that the algorithm requires. Finally, the obstacle avoidance part of the algorithm was separated from the waypoint navigation. This allowed ARA to use the obstacle avoidance as a purely reflexive driver. Once the revisions were finished and tested in the simulator, Virginia Tech supplied ARA with two new windows applications to test on the vehicle. The first application contained the waypoint navigation with obstacle avoidance, while the second was the reflexive driver.

### *Task 3 – Documentation of final algorithm*

Once the new algorithm was verified to be acceptable Virginia Tech moved onto the final task of Phase 2. This task involved updating the documentation to reflect the revisions made in Phase 2, Task 2.

### *Deliverable*

Virginia Tech supplied Applied Research Associates with a complete documentation of the final autonomous mobility algorithm as the deliverable for Phase 2.

## **1.8.2 Requirements**

The requirements of the algorithm were determined during Virginia Tech's initial visit to ARA's NED and through subsequent emails and phone conversations. These requirements defined the direction of the project.

At the initial visit the basics of the algorithm were established. The algorithm would provide ARA with waypoint navigation and obstacle avoidance. The maximum travel speed of the vehicle would be 5 m/s with a speed of 0.2 m/s during the mine detection operation. The sensor used to detect obstacles would be a SICK Laser Range Finder (LRF). This is a laser scanner that determines the distance to an obstacle at every degree for a 180 degree field of view. For the initial testing, it would be mounted on the Nemesis platform horizontally at a low enough position to

detect large obstacles, such as cones and barrels. The algorithm would run a reactive, rather than deliberative navigation scheme (*see* Section 1.5). This would reduce computation to provide a faster running algorithm.

It was also decided that the Virginia Tech LabVIEW software would run on a Windows laptop and communicate with the ARA vehicle via JAUS UDP messages. There were only five messages that would be needed for this communication. These messages were *Query Heartbeat*, *Report Heartbeat*, *Request Global Pose*, *Report Global Pose*, and *Set Global Vector*. The Query and Report Heartbeat pair was used for dynamic registration. When the Virginia Tech code started up, it would broadcast the *Query Heartbeat* message to alert all the other components in the subsystem of its presence. All components that received the query would then report a heartbeat. The autonomous mobility software would query the vehicle for its global pose at 7 Hz. Each time the vehicle received this query it would send back a *Report Global Pose* message. Finally, after the Virginia Tech algorithm determined the desired vehicle vector it would command the vehicle using the *Set Global Vector* message. More information about JAUS is given in Section 1.7.

After the initial deployment, additional requirements were established. These requirements included the addition of three capabilities. The first capability was to make the algorithm look farther ahead of the vehicle as the speed increased. The second was to remember obstacles that passed out of the field of view of the sensor. This would prevent the vehicle from turning into obstacles that it could no longer see. The third added capability was making the vehicle realize when it was in a situation that was too complex for it to handle. In this case, the software should stop the vehicle and ask the operator for assistance.

Thus, these basic requirements were used by Virginia Tech to develop the autonomous mobility algorithm for the Nemesis platform. Chapter 2 discusses the actual algorithm that was developed based on these requirements and how it works. Once the algorithm was developed it was fully documented for ARA.

### **1.8.3 Documentation**

Because the main goal of documenting the algorithm was to allow ARA's software programmers to code the algorithm in their desired programming language, the documentation needed to be meticulous and precise. All nuances of the code needed to be fully explained. Therefore, the method of documentation chosen was flowcharts and textual descriptions. Flowcharts provided a pictorial representation of each function, while the textual description provided the necessary commentary to fully understand the function. In addition to the flowcharts, many variables were used. Because many of these variables had special purposes or were complex arrays, a textual description of each variable was also provided. Together these elements were able to accurately and precisely describe the autonomous mobility algorithm developed for ARA.

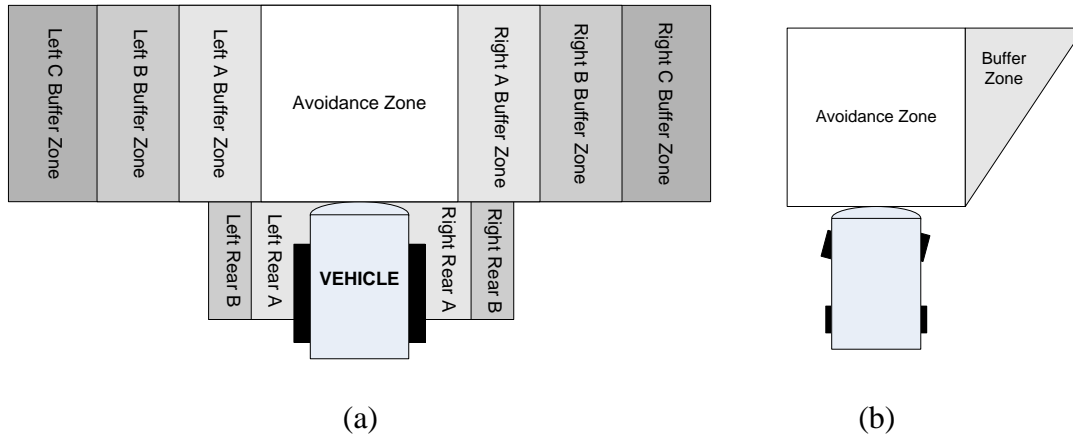
## **Chapter 2: The ‘Discretized Dynamic Expanding Zones with Memory’ Algorithm**

This chapter discusses the conceptual function of the Discretized Dynamic Expanding Zones with Memory autonomous mobility algorithm. While Chapter 4 provides the documentation of each individual function of the algorithm, this chapter explains how the overall process calculates the desired vector for the vehicle to drive. The Algorithm Overview section explains the general background of the algorithm, including its relation to previous Virginia Tech navigation software, namely the DEZ approach. Next, Section 2.2 explains the heading conversions used to relate local and global vectors. In Sections 2.3 to 2.8, the details of the five main sections of the algorithm are explained in sequential order. This starts with Waypoint Navigation to determine the heading to the next GPS waypoint. Next Heading Modification and Obstacle Remembering are used to determine the heading that will avoid obstacles while travelling to the desired waypoint. The Speed Modification section then uses this heading to calculate a safe speed to command to the vehicle. Difficult Situation Handling establishes if the vehicle is “stuck” in a certain situation that it cannot successfully navigate. Finally, the Component States are used to alter the vehicle’s behavior based on algorithm’s status.

### **2.1 Relationship to Previous Virginia Tech Algorithms**

As explained in Section 1.8 the goal of the algorithm is to provide autonomous waypoint navigation and obstacle avoidance capability to the Nemesis platform. The vehicle is differentially steered, allowing zero radius turns, and the maximum travel speed is slow ( $< 4$  m/s). The algorithm developed for ARA is similar to previously developed Virginia Tech software. The waypoint navigation algorithm is identical to the Global Waypoint Driver component of the Virginia Tech JAUS toolkit for LabVIEW (see Section 1.7.4). The obstacle avoidance is a modified version of the DEZ algorithm developed for the 2005 DARPA Grand Challenge, described in Section 1.5. However, the DEZ algorithm was developed for Ackermann steered vehicles moving at relatively high speeds ( $> 10$  m/s).

Therefore, the algorithm needed to be altered to be effective for ARA's Nemesis platform. The first alteration deals with the buffer zones. Because the Nemesis platform is capable of zero radius turns, buffer zones based on path curvature are insufficient. To remedy this, three discrete buffer zones are employed on each side of the main avoidance zone. The final vector is based on the occupancy of all of these zones. Figure 2.1 shows a comparison of the two approaches.



**Figure 2.1.** Comparison of DDEZm and DEZ. Zone layouts for (a) a differentially steered vehicle and (b) an Ackermann steered vehicle (turning right).

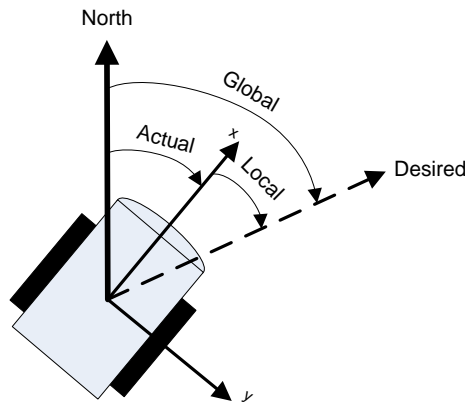
Unlike differential steering, the Ackermann steering properties of the Grand Challenge vehicles prevent them from turning into obstacles that they pass. Therefore, the added ability to remember obstacles that leave the field of view of the LRF sensor is important in the avoidance for the Nemesis platform. This modification also includes the addition of four more buffer zones behind the front plane of the vehicle that help avoid the remembered obstacles. The final modification to the DEZ algorithm involves requesting operator assistance during difficult situations. Because the rules of the DARPA Grand Challenge restricted any human interaction, the vehicle had to be fully capable of solving every situation or else it would not finish the race. However, because the Nemesis vehicle is under supervisory control of a human operator, it can request assistance if it encounters a difficult situation that cannot be solved by the autonomous mobility algorithm.

The final difference between the ARA algorithm and other Virginia Tech software is that, while Virginia Tech deals with local vectors referenced to the

vehicle, the Nemesis platform expects a global vector, which is referenced to north. This requires some conversions based on the actual heading of the vehicle.

## 2.2 Vector Conversions

Although the obstacle avoidance algorithm performs all computations using local vectors, the input and output vectors are globally referenced. The difference is that the heading of a global vector is referenced to north, while a local vector is referenced to the coordinate frame fixed to the vehicle. Both the vehicle fixed frame and global frame assume a right handed coordinate system with the positive z-axis pointing downwards. Accordingly, a positive heading indicates an angle to the right of the reference and a negative heading is to the left. For the vehicle fixed frame the SAE standard coordinate system is used [3]. This defines the origin at the center of the vehicle, the positive x-axis extending out the front of the vehicle, and the y-axis extending out the right side. The local heading is referenced to the positive x-axis. Figure 2.2 shows the conventions used for each system.



**Figure 2.2.** Vector conversion reference systems. Note that both the local and global headings describe the same desired vector. The only difference between the two is their reference axes.

In order to convert between the two systems, a simple calculation is performed using the vehicle's actual heading, known as the yaw. When converting from global to local the actual vehicle heading is subtracted from the desired global heading. To convert from local to global the vehicle yaw is added to the local heading. Equation 2.1 a and b provide these conversions.

$$LocalHeading = GlobalHeading - ActualHeading \quad (2.1 \text{ a})$$

$$GlobalHeading = LocalHeading + ActualHeading \quad (2.1 \text{ b})$$

In both cases, the converted heading must be restricted to the  $(-\pi, \pi)$  space. The formula used to do this is:

$$Heading\_Limited = ((Heading + \pi) \% 2\pi + 2\pi) \% 2\pi - \pi \quad (2.2)$$

where *Heading* is the input heading, *Heading\_Limited* is the heading limited to the  $(-\pi, \pi)$  space, and the % refers to the modulo operator.

## 2.3 Waypoint Navigation

As previously mentioned, the waypoint navigation section of the algorithm is identical to the algorithm in the Global Waypoint Driver component of the Virginia Tech JAUS toolkit for LabVIEW. Once the algorithm receives a list of waypoints it can begin to navigate them. There are three steps to the waypoint navigation algorithm. The algorithm first determines the heading and distance from the vehicle to the waypoint. Next, it checks if the vehicle has achieved the current waypoint. If so, it moves on to the next waypoint in the list. If not, it calculates the heading to the current waypoint and outputs it to the obstacle avoidance section of the algorithm. The algorithm simply drives “point to point,” meaning that it only looks at the waypoint it is currently driving towards, and does not change its behavior based on future waypoints.

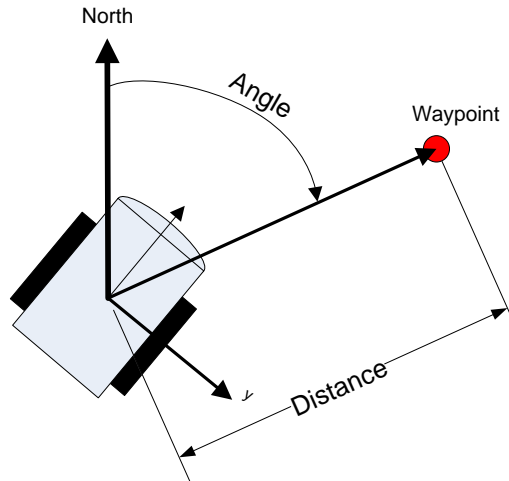
To determine the distance and heading to the waypoint, the algorithm first converts both the waypoint location and vehicle location from latitude/longitude to the Universal Transverse Mercator (UTM) coordinate system. This translates the positions from an angle based measurement to the distance based measurement of meters. Once in UTM coordinates, Equations 2.3 and 2.4 are used to determine the

distance, in meters, and angle, in radians referenced to north, from the vehicle to the waypoint.

$$d = \sqrt{(N_{wpt} - N_{veh})^2 + (E_{wpt} - E_{veh})^2} \quad (2.3)$$

$$\psi = -\text{atan2}(N_{wpt} - N_{veh}, E_{wpt} - E_{veh}) + \frac{\pi}{2} \quad (2.4)$$

where  $d$  is the distance to the waypoint,  $\psi$  is the globally referenced heading to the waypoint,  $\text{atan2}$  is the arc-tangent function that takes the parameters signs into consideration,  $N$  and  $E$  are the Northing and Eastings, where the subscript  $wpt$  and  $veh$  stand for waypoint and vehicle. Figure 2.3 shows how these are measured with respect to the vehicle.



**Figure 2.3.** Distance and angle to a waypoint. Diagram showing the conventions for the distance and angle from the vehicle to a waypoint.

To determine if the current waypoint has been achieved, the algorithm compares the distance,  $d$ , to a specified threshold. The threshold is the radius of an imaginary circle around the waypoint. If the distance is less than the threshold, the waypoint is considered achieved. The process then starts over again with the next waypoint in the list.

If the point has not been achieved, the desired heading is calculated as the local heading between the vehicle and waypoint using the Vector Conversions of

Section 2.2. The desired local heading to the waypoint will be used by the Obstacle Avoidance section of the algorithm as an initial desired direction. However, before this can occur, the obstacle data must be collected.

## 2.4 Obstacle Sensing

The data for obstacle avoidance comes from a single Laser Range Finder (LRF) sensor that is mounted on the front of the vehicle. The scanner has a 180 degree field of view and measures the distance to a point at 1 degree increments. The data from the sensor is in the form of an array of angles and distances to from the sensor to each point. This is converted to obstacle data using simple trigonometry. Each polar point is converted to a Cartesian coordinate in the vehicle-fixed frame using the equations:

$$x = d * \cos(\theta) \quad (2.5)$$

$$y = d * \sin(\theta) \quad (2.6)$$

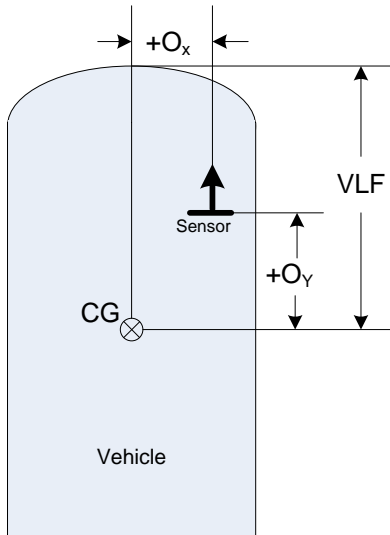
where  $x$  and  $y$  are the coordinates of the point in the vehicle frame, and  $d$  and  $\theta$  correspond to the distance and angle to the point in polar coordinates. In order to remain generic, these calculations occur as part of the sensor interface. Only the array of  $x$  and  $y$  obstacle coordinates are given to the DDEZm algorithm. This allows different sensors to be used in place of the LRF.

To account for the generic sensor offsets shown in Figure 2.4, Equations 2.7 and 2.8 are used.

$$x = x + O_x \quad (2.7)$$

$$y = y + O_y - VLF \quad (2.8)$$

In both the figure and the equations,  $O_x$ ,  $O_y$ , and  $VLF$  correspond to the Sensor X Offset, Sensor Y Offset, and Vehicle Length in Front of CG, respectively.



**Figure 2.4.** Diagram of the sensor offsets on the vehicle.

### 2.4.1 Obstacle Remembering

The method just discussed solves the problem of sensing obstacle in front of the vehicle. When the front of the vehicle passes an obstacle, it is no longer seen, and will be forgotten. The zero-radius-turn vehicle can then turn back into the unseen obstacle. In order to remedy the situation, Obstacle Remembering was implemented. In the Obstacle Remembering process, all obstacles with y-coordinates within 1 meter in front of and 1.5 vehicle lengths behind the front plane of the vehicle are converted from local to UTM coordinates using the vehicle's GPS position. They are then remembered for the next iteration. After all sensed points are analyzed in the next iteration, remembered points are converted back to local coordinates using the vehicle's new GPS position. All duplicate points are ignored to reduce memory usage and processing time.

## 2.5 Obstacle Avoidance

The obstacle avoidance segment of the algorithm acts as a reflexive driver to modify the initial desired heading from waypoint navigation. In this sense, it follows a subsumption-like scheme of reactive navigation, in that the obstacle avoidance behavior takes precedence over the waypoint navigation. It is not pure subsumption because the initial heading is considered when avoiding obstacles. The heading

modification is separated into two parts: perception zones and obstacle avoidance. Perception zones addresses the 11 zones around the vehicle that aid in the avoidance of obstacles. Obstacle avoidance describes the process that is used when the central Avoidance Zone contains obstacles.

### 2.5.1 Perception Zones

As described in the Algorithm Overview, there are eleven perception zones that the vehicle uses to decide how to avoid obstacles. The Avoidance Zone is in the center, directly in front of the vehicle. Three front buffer zones, labeled A, B, and C, extend out laterally from the Avoidance Zone on each side. The length of these zones increases with speed to account for the greater stopping distance. Four more zones of fixed length are included behind the front plane of the vehicle, with two on each side of the vehicle's centerline. These zones which are labeled Left Rear A (LRA), Left Rear B (LRB), Right Rear A (RRA), and Right Rear B (RRB) are outside the field of view of the LRF sensor. Therefore, they are populated using the Obstacle Remembering technique described in Section 2.4.1. Their purpose is to prevent the vehicle from turning into obstacles directly on its side. Figure 2.5 is a diagram of all of the zones with dimensions.

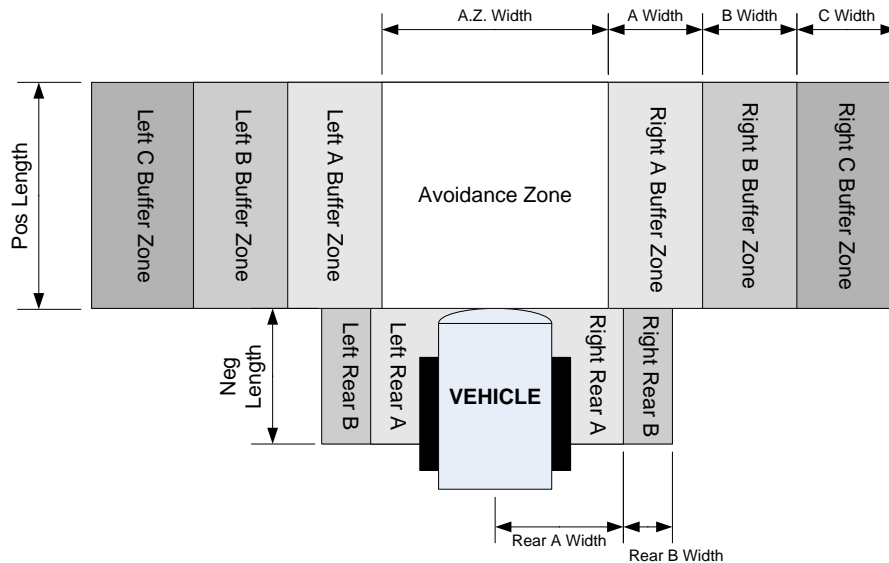


Figure 2.5. DDEZm layout and dimensions.

Of the 11 zones, the Avoidance Zone is the most important. It is slightly wider than the vehicle, and indicates which obstacles directly obstruct the vehicle's path. If the Avoidance Zone is occupied, all obstacles within it must be avoided to prevent a collision. In this case, the front buffer zones are ignored and the obstacle avoidance behavior takes over. When the Avoidance Zone is empty, the vehicle's path straight ahead is clear. However, if the vehicle chooses to turn, the buffer zones must be examined.

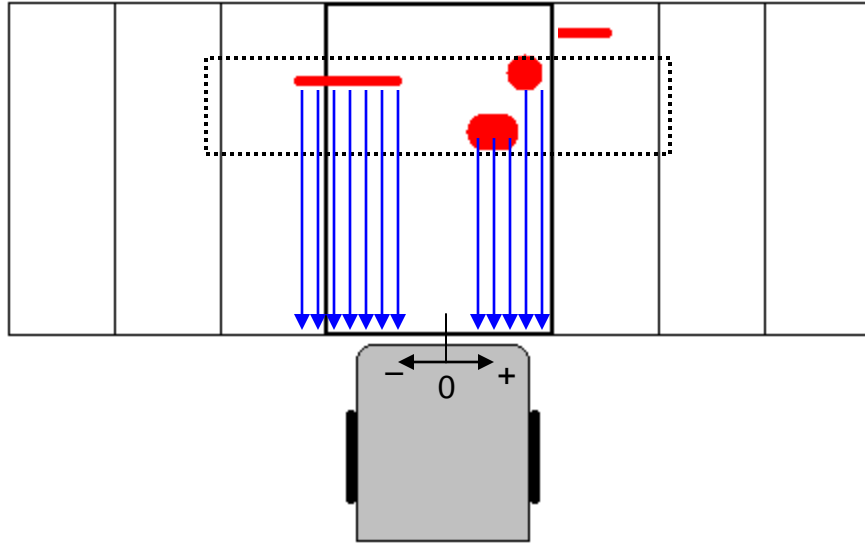
The buffer zones on each side are used to limit the vehicle's local heading when it tries to turn. Before a turn is commanded the occupancies of the buffers on the side the vehicle is turning towards are checked. The A Buffers (both front and rear) are examined first. If these buffers are occupied by an obstacle, the vehicle is not allowed to turn in that direction. The desired local heading set to zero, making the vehicle drive straight. If the A Buffers are clear, the front and rear B buffers are considered next. When a B Buffer is occupied the vehicle's heading is limited to a predefined B Heading Limit. This value is the maximum local heading that may be commanded when the buffer is occupied. If the desired heading is less than the limit, it will pass through unchanged. Assuming that the B Buffers are clear, and the C Buffer contains obstacles, the final commanded heading is limited to the C Heading Limit. This limit has a larger value than the B Heading Limit, allowing sharper turning. When the Avoidance Zone and all of the buffers in the direction of the turn are clear, the initial local heading will not be modified.

The size of the 11 zones as well as the heading limits associated with the B and C Buffers will be different for each vehicle and can be changed by the user to achieve the desired behavior. However, there are a few constraints on some of the values. First, the front length, shown above in Figure 2.5, is the same for every zone and indicates the farthest distance, from the front of the vehicle, at which obstacles are registered. While the length of these front zones changes with speed, maximum and minimum lengths can be specified by the operator to limit the sizes. The rear length defines how far behind the vehicle's front plane that obstacles will affect its behavior. Also, with the stipulation that the zone layout is symmetric about the

longitudinal centerline of the vehicle, the widths of each of the Buffer Zones, as well as the Avoidance Zone may also be adjusted separately. However, the Avoidance Zone must be slightly wider than the vehicle so that obstacles will be completely avoided.

### **2.5.2 Obstacle Avoidance**

When the Avoidance Zone is occupied, the obstacles within it must be avoided. The two steps to this process involve deciding which way to turn and deciding angular rotation required. The first step to deciding which direction would best avoid the obstacle involves forming a search area. The search area is a rectangular box that begins at the closest obstacle. A Search Length and Width are used to specify the size of the search area. This is shown by the dashed box in Figure 2.6. A typical Search Length is about one meter, and a typical width spans from the left A Buffer to the right A Buffer. This area is used to determine the obstacles that merit avoidance. In general, the vehicle should avoid to the side of the search area that is the least heavily weighted with obstacles. To determine this, the lateral distances of all obstacles within the search area are summed, as shown by the blue arrows in Figure 2.6. Each obstacle point has a value equal to its lateral distance. Points to the right of the vehicle centerline have positive values, and those to the left have negative values. If the sum is greater than zero, the right is more heavily weighted with obstacles, so the avoidance direction should be left. Note that obstacle points farther from the centerline have a higher weighting than those that are close.

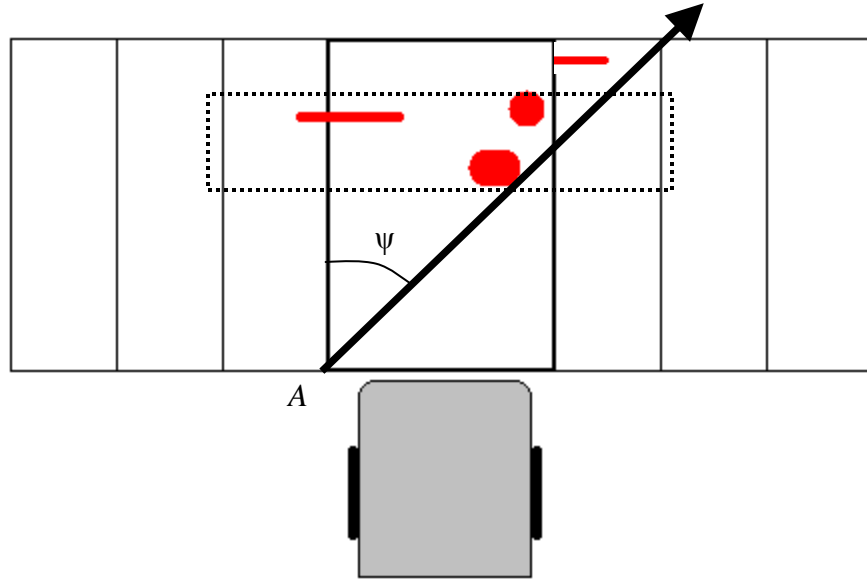


**Figure 2.6.** Lateral Summation. Diagram showing the search area and lateral summation of obstacles used for determining the avoidance direction.

When the summation is close to zero it means that both sides of the vehicle are similarly populated. Therefore, when the magnitude of the summation is less than the user defined Summation Threshold, the direction chosen is the same as the direction of the initial desired heading. This gives the vehicle a better chance of reaching its current waypoint. However, if the rear A Buffer on the chosen side is occupied the vehicle may not turn that direction. In this situation, the distance of the closest obstacle is considered. If this distance is greater than the negative buffer zone length, the vehicle is commanded to drive straight, providing the vehicle the opportunity to clear its rear buffer zone before avoiding the obstacle. However, when the distance is less than the rear buffer zone length, the avoidance direction is set opposite of the previous chosen direction. In the event that both rear A Buffer zones are occupied and the distance to the closest obstacle is less than the negative buffer zone length, the vehicle must ask for operator assistance, as covered in Section 2.8.

Assuming the direction set above is not straight, the heading is calculated based on the obstacles inside the search area. This is determined as the steepest angle which will clear the Avoidance Zone of obstacles. In order to account for the width of the vehicle, the origin of the angle is the bottom corner of the Avoidance Zone, on the side of the vehicle opposite the turning direction. This is done so that

the far edge of the Avoidance Zone clears the obstacle. Therefore, assuming the vehicle in Figure 2.7 below is turning right, the vector starts at the bottom left corner, marked A. The algorithm then calculates the angle,  $\psi$ , to each obstacle point. The angle with the largest magnitude is chosen. This angle is then compared to the initial desired heading. The final local heading is set equal to the heading with the largest magnitude.



**Figure 2.7.** Avoidance Heading. Representation of the avoidance heading,  $\psi$ . Point A shows the origin used to determine the angle to each obstacle point.

## 2.6 Speed Calculation

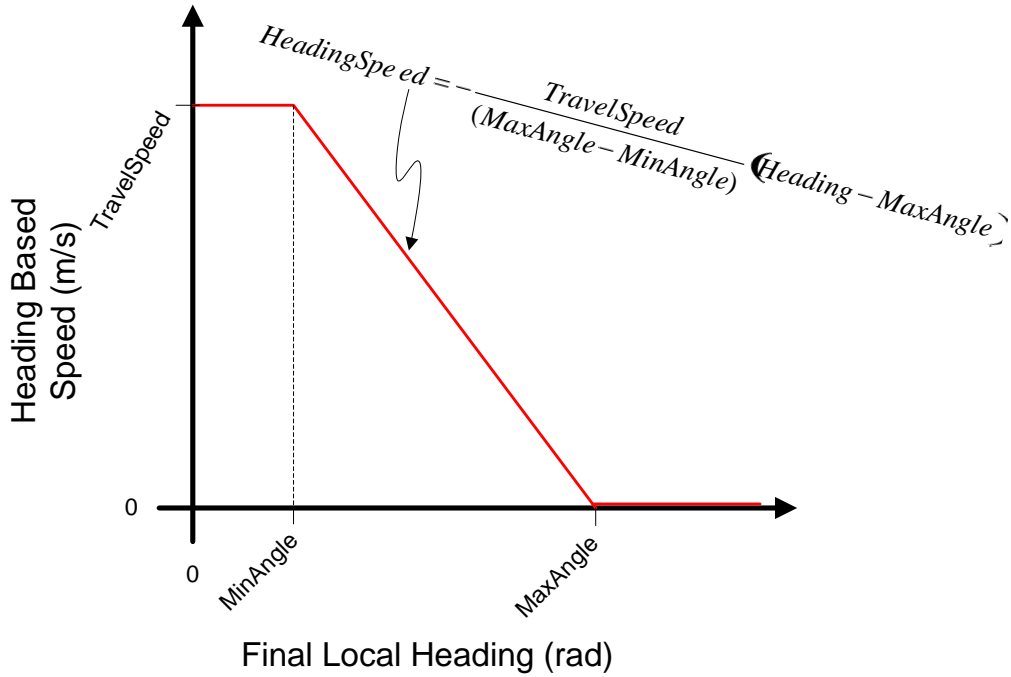
Once the final avoidance heading is calculated, the speed may be determined. If the Avoidance Zone and three other A or B Buffer Zones are occupied, it can be inferred that the vehicle is approaching a large obstacle. Hence, the speed is automatically set to zero. If this condition is false, but the Avoidance Zone is occupied, two possible speeds are calculated: heading-based speed and distance-based speed. The heading-based speed is a function of the desired local heading of the vehicle, while the distance-based speed is related to the distance to the closest obstacle. The slower of the two is chosen as the final speed. If the Avoidance Zone is clear, only the heading-based speed is calculated and used for the final speed.

### 2.6.1 Heading-Based Speed

The Heading-Based Speed is a linear function based on the final local heading output from the Obstacle Avoidance part of the algorithm. There are also three other parameters which affect the function. The first parameter is the maximum allowable speed, known as the Travel Speed. This is the maximum speed that may be commanded to the vehicle. The other two parameters are the Minimum Angle and Maximum Angle, which are local headings, in radians. Thus, the two angles are used as cutoffs to determine when the maximum and minimum speeds should be used as shown in Figure 2.8. All headings below the Minimum Angle result in the output speed being equal to the maximum speed. All headings above the Maximum Angle result an output speed of zero. There is a linear decrease of speed with the increase of heading from the Minimum Angle to the Maximum Angle that defines the Heading-Based Speed. The shape of the output speed versus local heading is defined by the piecewise function:

$$S_H = \left\{ \begin{array}{ll} S_T, & H \leq \alpha_{\min} \\ \frac{S_T}{(\alpha_{\max} - \alpha_{\min})} (H - \alpha_{\max}), & \alpha_{\min} < H < \alpha_{\max} \\ 0, & H \geq \alpha_{\max} \end{array} \right\} \quad (2.9)$$

where  $S_H$  is the Heading Based Speed,  $S_T$  is the Travel Speed,  $H$  is the final local heading,  $\alpha_{\min}$  and  $\alpha_{\max}$  are the Minimum and Maximum Angles, respectively.



**Figure 2.8.** Heading-Based Speed curve. Graph showing the relationship between local heading and Heading-Based Speed.

### 2.6.2 Distance Based Speed

As previously explained, the Distance Based-Speed is only calculated when the Avoidance Zone contains obstacles. This speed is related to the distance to the closest obstacle in the avoidance zone and the maximum vehicle deceleration. If the closest obstacle is less than 0.5 meters from the front of the vehicle, the speed is automatically set to zero to prevent a collision. Otherwise, it is related to the ability of the vehicle to stop before hitting the obstacle. This is better understood if the distance to the closest obstacle is interpreted as a stopping distance. The Distance-Based Speed can then be derived from the linear equation of motion:

$$v_f^2 = v_i^2 + 2ad \quad (2.10)$$

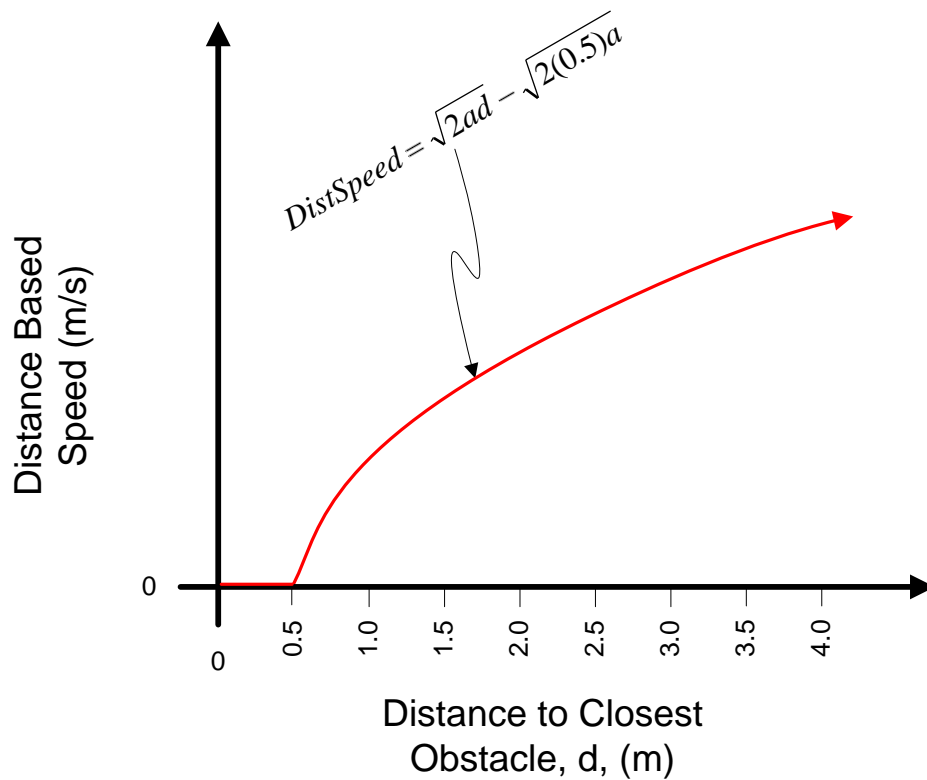
where  $v_f$  and  $v_i$  are the final and initial velocities,  $a$  is the acceleration and  $d$  is the distance travelled from the initial to final states. Assuming that the final velocity is equal to zero, and solving the equation for the initial velocity yields the equation:

$$v_i = \sqrt{2ad} \quad (2.11)$$

where  $v_i$  is the Distance Based Speed,  $a$  is the absolute value of the vehicle's maximum deceleration, and  $d$  is the distance to the closest obstacle. To account for the speed being set to zero for any distance less than 0.5 m a constant is subtracted from the speed as shown in Equation 2.12.

$$v_i = \sqrt{2ad} - \sqrt{2(0.5)a} \quad (2.12)$$

A plot of this relationship is shown in Figure 2.9.



**Figure 2.9.** Distance-Based Speed curve. Graph showing the relationship between distance to the closest obstacle in the Avoidance Zone and the Distance Based Speed.

## 2.7 Rate Limiters

In order to prevent the algorithm from commanding unsafe speed and heading changes, two rate limiters were implemented. These limit acceleration and heading rate. They both employ the principle of discrete derivatives. That is, they multiply the desired rate by the cycle time of the entire algorithm to get the value change, and add this change to the previous value. Equations 2.13 and 2.14 show how this is done for accelerating and decelerating, respectively.

$$v_f = v_i + \frac{t_c}{1000} a_+ \quad (2.13)$$

$$v_f = v_i - \frac{t_c}{1000} a_- \quad (2.14)$$

where  $v_f$  is the calculated speed,  $v_i$  is the previous speed,  $t_c$  is the Cycle Time,  $a_+$  is the maximum acceleration, and  $a_-$  is the maximum deceleration. Likewise, the equation for limiting the heading rate is:

$$H_f = H_i \pm \frac{t_c}{1000} R_H \quad (2.15)$$

where  $H_f$  is the calculated heading,  $H_i$  is the previous heading and  $R_H$  is the heading rate. When the heading is increasing, addition is used, and when decreasing, subtraction is used.

## 2.8 Difficult Situation Handling

Although the autonomous mobility algorithm can navigate most courses with little trouble, there are some situations that can be too difficult for the algorithm to handle alone. This is mainly due to the fact that it runs a reactive, rather than deliberative, navigation scheme. While the reactive scheme requires less computation, an intrinsic problem is that they can get “stuck” in certain situations. However, because the Nemesis platform has continuous contact with an Operator Control Unit (OCU), it can request assistance during these situations. Therefore, a subroutine of the algorithm was implemented to determine when assistance is needed.

Three conditions were identified that could cause the vehicle to become stuck. The first, which was mentioned in the Obstacle Avoidance section above, involves both rear A Buffer zones and the Avoidance Zone being occupied by obstacles while the closest obstacle in the avoidance zone is less than the length of the rear buffer zones. In this case, the vehicle does not have enough room ahead of it to clear its rear buffer zones before avoiding obstacles. This prompts a request for operator assistance. The two other situations are handled separately from the main navigation algorithm. The first requests assistance whenever the commanded speed has been set to zero for more than 30 seconds. Not moving for 30 seconds implies that the vehicle is stuck and needs help. The second situation which requires operator assistance deals with the oscillation of the commanded heading. This is a particularly common problem in many reactive navigation schemes, where the vehicle oscillates from side to side “searching” for a way around an obstacle. Therefore, the Difficult Situation Handler monitors the number of heading changes and the distance travelled. Assistance is requested if the heading changes six or more times within a travel distance of one meter. If the vehicle travels more than a meter before oscillating six times, the heading change count is reset to zero and the process starts over again.

The act of requesting assistance from the operator is not actually an active event. Rather than sending a message to the OCU, the algorithm simply enters the Emergency State (see Section 2.9 below). The OCU has a connection that monitors the state of the algorithm and will see that the component state has changed to Emergency. It will then inform the operator that the vehicle requires assistance.

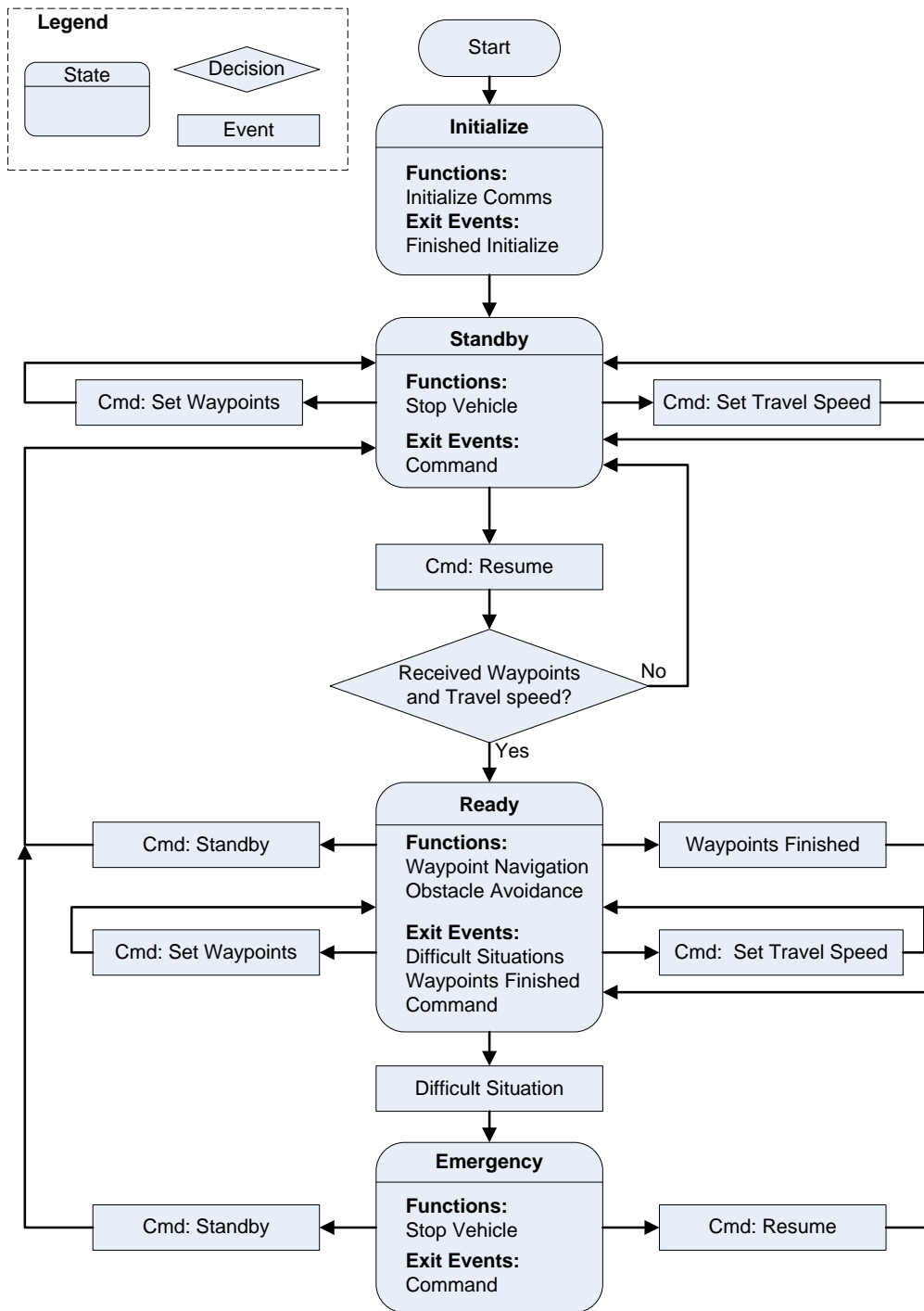
## **2.9 Component States**

In order to separate the behavior of the autonomous mobility algorithm, three component states were used. These states are based on the JAUS core status message and include Standby, Ready, and Emergency. After initialization, the algorithm immediately enters the standby state. In standby, the algorithm is essentially paused, meaning that it commands a zero speed to the vehicle and holds the heading constant. In this state, no computations for waypoint driving or obstacle

avoidance are performed. The algorithm simply waits for a set of waypoints and a travel speed. This can either come from the OCU via the JAUS *040A – Set Global Waypoint* and *040C – Set Travel Speed* commands, or be input directly via the user interface of the component. The algorithm will accept the JAUS *0004x – Resume* command to enter the ready state once the algorithm has received a set of waypoints and a travel speed.

In the ready state, the algorithm performs the waypoint navigation and obstacle avoidance described above. The travel speed and set of waypoints can still be updated in this state. New waypoints can either be appended to the end of the previous points, or replace the points altogether. The algorithm will return to standby if it receives the *0003x – Standby* command or the waypoint navigation part finishes all waypoints. In the event that the algorithm encounters a difficult situation and needs operator assistance, it enters the Emergency state.

The Emergency state is used to convey to the OCU that the vehicle has entered a situation that it cannot handle on its own. This state behaves exactly the same as the Standby state, and does not allow the vehicle to move. However, when the OCU queries the component state and sees that it is in Emergency, it alerts the operator to take control to exit the difficult situation. Once the situation is exited, the Resume command may be sent to the algorithm, which will then continue where it left off. The state diagram is shown below in Figure 2.9.



**Figure 2.10.** State Diagram for the DDEZm algorithm.

## Chapter 3: Algorithm Implementation

As mentioned in Chapter 1, the autonomous mobility algorithm was developed in LabVIEW and was deployed onto the Nemesis platform via a Windows laptop connected to the vehicle's internal network. JAUS was used for communication between the algorithm and vehicle. This chapter explains the details of the deployment, including the JAUS interface, LabVIEW implementation, reiterative testing and determination of behavioral parameters.

### 3.1 JAUS Interfacing

The Joint Architecture for Unmanned Systems provides standard messages and a transport specification to facilitate communication between unmanned systems. This section details the JAUS interface used in the collaboration between Virginia Tech and ARA. First, first the major interoperability issues will be addressed. Next, the JAUS messages that were sent between the two organizations will be described in detail.

#### 3.1.1 Issues

Although JAUS provides a standard framework for interoperability among organizations, it generally only extends as far as defining messages and a routing scheme. JAUS does not explicitly restrict the implementation of the overall system design, requiring the developer to make some basic assumptions. Because developers from different organizations may make different assumptions, a small number of issues must be worked out between organizations before full interoperability is possible. In the collaboration between ARA and Virginia Tech, three major issues emerged. These included the JUDP Header, dynamic discovery, and service connections.

##### Issue 1: JUDP Header

The JAUS UDP (JUDP) Header is an eight byte preamble attached to the front of each UDP message. The eight bytes read "JAUS01.0." It was implemented

by the JAUS Experimentation Task Group (ETG) for their experiments because it is especially useful when scanning network traffic. The network scanner is able to classify all UDP Messages that contain the JUDP Header as JAUS messages. Because Virginia Tech participated in these experiments, all of their sent UDP messages added this header. In addition, they would parse the header off before reading the message. Because ARA does not use this header, Virginia Tech removed the sending and parsing of it from their software.

### Issue 2: Dynamic Discovery

Dynamic discovery, also known as dynamic registration, is the process of determining all other subsystems, nodes, and components in a JAUS system. ARA and Virginia Tech used different methods of discovery. In ARA's method, each component broadcasts a discovery message when it first starts up. All other components respond to the initial message, informing the new component that it exists. This is an efficient way of allowing each component to learn about the system configuration. Virginia Tech, however, follows the ETG protocol, which requires every component to broadcast an unsolicited heartbeat at 1 Hz. The Node Manager receives the heartbeats and maintains a list of all components, nodes, and subsystems on the network. Because the ARA method is more efficient and uses less network bandwidth, Virginia Tech updated their code to work with ARA during the collaboration.

### Issue 3: Service Connections

Service connections provide an efficient way of transmitting periodic messages from one component to another without having to constantly query the providing component. In addition, messages that are routed as a service connection are not queued up with other messages. This guarantees that only the newest data arrives at the requesting component. Because of this, service connections are ideal for messages such as Report Global Pose, which is updated regularly, and old data is unwanted.

However, in order for service connections to work, both organizations must support the setup and routing of them. Because ARA did not support service connections for the deployment of the algorithm, they could not be used. Therefore, Virginia Tech's software periodically queried the vehicle for its global pose.

### 3.1.2 Messages

There were 11 JAUS messages used for the communication between Virginia Tech's algorithm and the Nemesis platform. These messages are described in further detail in this section. They are separated into commands, queries, and reports.

#### Commands:

*0003 – Standby*

Sent by the OCU to set the algorithm state to standby.

*0004 – Resume*

Sent by the OCU to set the algorithm state to ready.

*0407 – Set Global Vector*

Sent by the autonomous mobility algorithm to the Nemesis platform to command the vehicle's speed and globally referenced heading.

*040A – Set Travel Speed*

Sent by the OCU to set the maximum speed the algorithm can command.

*040C – Set Global Waypoint*

Sent by the OCU to set each desired waypoint for the algorithm to drive. Each waypoint is referenced by Latitude, Longitude, and waypoint number. To set a list of waypoints this command is sent once for each point. The order is set by the waypoint number.

#### Queries:

*2002 – Query Component Status*

Periodically sent by the OCU to monitor the component status of the algorithm. The algorithm responds with the *4002 – Report Component Status* message.

*2202 – Query Heartbeat Pulse*

Broadcast by any component or node on the network as a means of dynamic discovery. Any component that receives this must reply with *4202 – Report Heartbeat Pulse*.

*2402 – Query Global Pose*

Periodically sent at 5 – 10 Hz from the algorithm to the Nemesis platform to request the vehicle’s current position and orientation, including Latitude, Longitude, and heading. The vehicle will respond with *4402 – Report Global Pose*.

Reports:

*4002 – Report Component Status*

Sent by the algorithm to the OCU in response to the *2002 – Query Component Status* message. This will report the status as either Standby, Ready, or Emergency. If the reported status of the algorithm is “Emergency,” it has entered a difficult situation which requires operator assistance.

*4202 – Report Heartbeat Pulse*

Sent as a response to the *2202 – Query Heartbeat Pulse* message used in the dynamic discovery process.

*4402 – Report Global Pose*

Sent by the vehicle to the algorithm as a response to the *2402 – Query Global Pose* message. It informs the algorithm of the vehicle’s current Latitude, Longitude and heading.

## **3.2 LabVIEW Implementation**

Because of Virginia Tech’s previous experience in programming with LabVIEW, and its fast development time and ease of debugging, it was employed for

the development of the DDEZm autonomous mobility algorithm for the Nemesis vehicle. Three separate threads, or processes, were used to write the code that was deployed on the vehicle. The main thread was separated into two parts. The first part performed the actual algorithm and the second supported all of the JAUS interfaces required for communication. The second process read and parsed the LRF data and sent it to the main thread. The final thread received information from the main process and visually explained what the algorithm was doing.

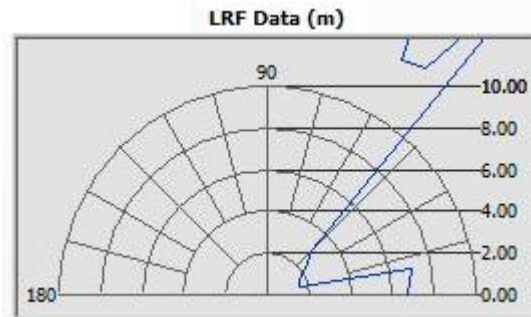
### **3.2.1 Main Process**

Because the autonomous mobility algorithm dealt so closely with the information it received and sent via JAUS messages, the JAUS messaging and algorithm processes were executed together in the same thread. The process would look at the current LRF, Global Pose, travel speed, and waypoint data and calculate the desired vector as explained in Chapter 2. At the same time, the JAUS part of the thread would check for new messages and update the variables for Global Pose, component status, travel speed, and waypoints. This process also handled the component states through the use of a state machine. In the standby and emergency states, the autonomous mobility algorithm would not be run. Once the state was changed to “ready,” using the conditions outlined in Section 2.9, the process would call the algorithm. In all cases the thread would send a JAUS Set Global Vector message at 8 Hz to the vehicle. In the standby and emergency states, this message would stop the vehicle and prevent any further movement by setting the speed to zero and the desired heading equal to the current vehicle heading. In the ready state, the vector that was sent would be calculated by the DDEZm algorithm.

### **3.2.2 LRF Reader Process**

The LRF Reader thread would run outside of the main loop. It was in charge of communicating with the sensor via the serial port, reading and parsing the range data, and sending it to the main process for use by the autonomous mobility algorithm. The data was in the form of a 180 element array of a cluster (a cluster in LabVIEW is similar to a struct in C++). The cluster contained the angle and distance

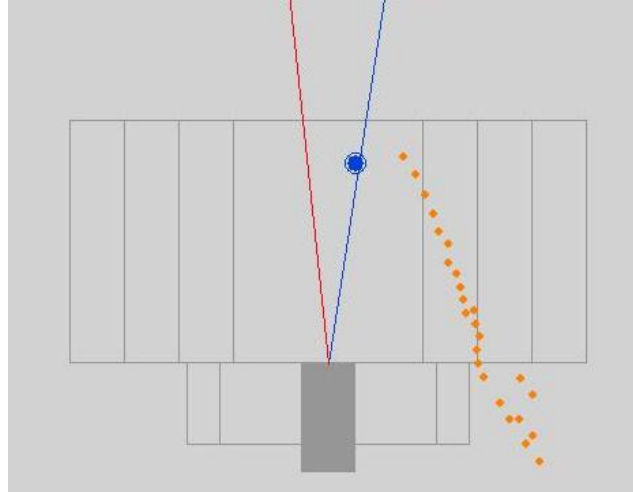
for each point. In addition, the LRF data would be displayed on the LabVIEW front panel as the polar plot shown in Figure 3.1. This plot shows the distance to an obstacle at every degree for the 180 degree field of view of the sensor.



**Figure 3.1.** LRF Data polar plot.

### 3.2.3 Visualization Process

The purpose of the visualization process was to aid in debugging the algorithm and tuning the behavioral parameters. In particular, it is very useful for determining zone sizes and heading limits during testing. It gives an overhead representation of the vehicle with the perception zones around it. It then draws the obstacles within the zones and overlays two desired headings: one for the direction to the waypoint, and one using obstacle avoidance. When a waypoint comes within view of the visualization window, it is also drawn on the screen. Figure 3.2 shows the visualization window with all of these elements. The obstacle points are shown by orange dots, the waypoint heading is blue and the obstacle avoidance heading is red. The waypoint is represented by the large blue circle.



**Figure 3.2.** Screenshot of the overhead visualization. The obstacles are orange dots, waypoint heading is the blue line, obstacle avoidance heading is red, and the waypoint is the blue circle.

While it is closely linked to the main algorithm, the visualization component was placed in a separate process because of the large amount of processing required. The loop was given a lower priority than the main and LRF loops so it would not take processor power away from these time-critical threads. The user interface also allows the visualization to be turned completely off to further reduce processor usage.

### 3.3 Testing

The testing of the autonomous mobility algorithm was performed both in simulation and on the Nemesis vehicle. Simulation testing was used for developing and debugging behaviors, as well as determining useful ranges for parameter values. Once this was in working order, the software was run on the vehicle for full evaluation. New revisions of the algorithm would once again be tested in simulation before being ported to the vehicle.

#### 3.3.1 Simulation

The simulation testing was performed using the simulator that Virginia Tech developed to test their IGVC and Grand Challenge software. A vehicle creator tool provided the environment for a vehicle to be created with the exact dimensions and



However, when commanding a real vehicle there is a slight lag in the actuators, the vehicle performs differently based on the terrain (e.g., gravel versus asphalt), and there are physical limitations on the rates of change of both speed and heading. While this does not drastically change the algorithm, the control and behavioral parameters need to be fine-tuned through testing in order to efficiently control the vehicle.

The initial three-day deployment visit by Virginia Tech to ARA's NED was the first on-vehicle test of the algorithm. The main goal of this first test was to handle all of the hardware and software issues associated with cross-organizational collaboration. Before the visit, Virginia Tech was able to test the UDP JAUS communication over the internet. At the visit, the Virginia Tech LabVIEW software was compiled to Windows application and put on a Windows laptop. This laptop was connected to the vehicle's network through Ethernet. The LRF sensor was then plugged into the laptop. The Virginia Tech code interfaced directly with the sensor and communicated with the vehicle's computers via JAUS over UDP. Once all of the hardware was working together, the code was able to navigate the vehicle through a series of waypoints while avoiding obstacles such as cones and large snowballs. At this test, ARA's engineers requested the addition of the rear zones, remembered points, and difficult situation handler. All of the additional on-vehicle testing new software revisions was performed by ARA's engineers.

### **3.3.3 Continued Testing**

Testing is an iterative process that starts at the initial development and continues through the final product. After the initial visit to ARA, the algorithm was modified to provide the required additions. These additions were once again tested in simulation and then sent to ARA for testing on the vehicle. Because the initial visit resolved all hardware and communications problems, further visits were not necessary. The compiled program could be emailed to ARA and their engineers could run it on the vehicle. This process of testing new features in simulation and on the vehicle continued until the final algorithm was deemed acceptable and the most efficient behavioral and control parameters were chosen.

## 3.4 Parameters

The determination of the behavioral parameters used to control the vehicle was a process of trial and error testing, both in simulation and on the vehicle. Acceptable ranges were determined in simulation, and the values were fine-tuned during on-vehicle testing. Table 3.1 lists the parameters with brief descriptions. This section presents some general guidelines for determining the values of each the parameters, as well as the values used for the Nemesis vehicle.

**Table 3.1.** Behavioral Parameters

Parameter	Description	Elements
Zone Sizes	Lengths and widths of each perception zone around the vehicle	Minimum Length, Maximum Length, Avoidance Zone Width, A Buffer Width, B Buffer Width, C Buffer Width, Rear Length, Rear A Buffer Width, Rear B Buffer Width
Avoidance Settings	List of settings used in the obstacle avoidance part of the algorithm	B Heading Limit, C Heading Limit, Search Length, Search Width, Summation Threshold, Front Remember Distance, Rear Remember Distance, Sensor X Offset, Sensor Y Offset, VLF
Waypoint Threshold	Radius of a circle around each waypoint. If the vehicle is inside the radius, the point is considered achieved.	N/A
Velocity Parameters	Set of parameters used in determining the commanded vehicle speed and length of the perception zones.	Maximum Angle, Minimum Angle, Maximum Acceleration, Maximum Deceleration, Maximum Speed, Heading Rate

### 3.4.1 Zone Sizes

There are 9 parameters to establish the sizes of the perception zones. The parameters for the front zones include minimum length, maximum length, Avoidance Zone width, A Buffer width, B Buffer width, and C Buffer width. The sizes of the rear zones are determined by the rear length, Rear A Buffer width, and Rear B Buffer width. These sizes are shown in Figure 2.4.

While the zone length is a function of vehicle speed, the minimum and maximum lengths are the cutoff limits. The lengths of all of the front perception zones are equal. The Avoidance Zone width has the strictest rule in that it must be wider than the vehicle. However, the wider the zone is, the farther away from obstacles the vehicle will try to avoid. The vehicle will not be able to navigate narrow spaces if this is too wide. The widths of the A, B, and C Buffer zones can be decided by the operator as well. While the B and C Buffers may have a width of zero, essentially eliminating them, the A buffer should be at least 0.5 meters wide. A typical width for each of the three buffers is one meter. The rear buffer zones should be sufficiently long and wide to prevent the vehicle from turning into obstacles on its sides. Therefore, the zone length should extend from the front of the vehicle to at least its center of rotation. To prevent unnecessary avoidance it should not extend past the rear of the vehicle. The rear A Buffer width is measured from the center of the vehicle. This should at least extend out to the edge of the Avoidance Zone, and must be slightly larger than half of the vehicle's width. Like the front B and C Buffers, the Rear B Buffer width is up to the operator, however, a good value is between 0.5 and 1 meter.

### **3.4.2 Avoidance Settings**

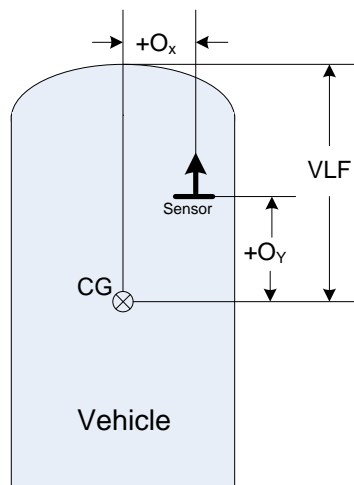
The Avoidance Settings variable is an array containing several parameters which affect the behavior of the obstacle avoidance part of the algorithm. These parameters include the B and C Heading Limits, Search Length and Width, Summation Threshold, Sensor Offsets, and Remember Distances.

The B and C Heading Limits are used to restrict the heading magnitude when the vehicle is turning towards an obstacle in either the B or C Buffers. These are only used when the Avoidance Zone and A Buffer Zone are clear. The B Heading Limit is the maximum local heading allowed if the B Buffer Zone is occupied. The same is true for the C Heading Limit and C Buffer Zone. Because, the C Buffer Zone is farther from the vehicle, it should restrict the heading less than the B Buffer Zone. Limits of 0.3 for B and 0.6 for C were initially used on the Nemesis vehicle.

The Search Length and Width define the size of the search area described in Section 2.5.2. Because this area determines the obstacles that merit immediate avoidance, the obstacle avoidance behavior can be modified by changing the length and width. Making the length and width larger will make the avoidance algorithm look farther and wider when avoiding. This gives a “big picture” view. A shorter, narrower search area is more nimble but could get stuck easier. As a rule, the width of the area should be at least as wide as the avoidance zone. However, a width that spans the two A Buffer Zones is more desirable. The length should be at least one, but no longer than two, meters.

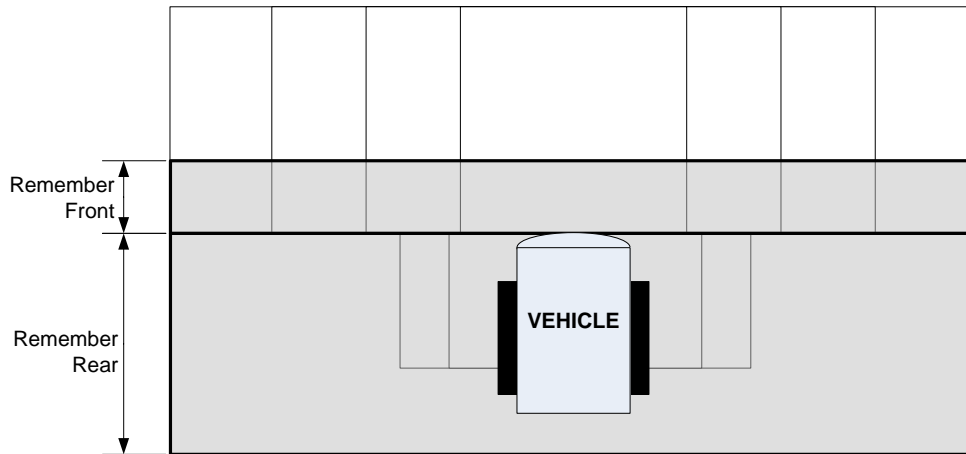
When the obstacle avoidance part of the algorithm is determining the direction it should avoid towards, the Summation Threshold is used. As described in Section 2.5.2, the lateral summation calculates whether there are more obstacles on the left or right side of the vehicle. A higher magnitude for this summation corresponds to a higher weighting of obstacles, while a lower magnitude means that the obstacles are evenly distributed. The Summation Threshold specifies whether the summation is sufficiently close to zero to ignore and thus choose the direction desired by waypoint navigation.

The sensor offsets include the Sensor X Offset, Sensor Y Offset, and Vehicle Length in Front of CG (VLF). These are used when transforming the obstacle point data from the sensor to the vehicle frame. Figure 3.4 shows the representation of each of these variables.



**Figure 3.4.** Dimensions used in obstacle point conversion.

The Front and Rear Remember Distances are used by the function in charge of remembering obstacles that have passed outside the field of view of the sensor. The front distance describes the maximum longitudinal distance in front of the vehicle that the algorithm retains the globally referenced position, in UTM, of an obstacle. A good value for this is one meter. The rear distance is the distance behind the front plane of the vehicle at which the point is forgotten. This should be at least as long as the total length of the vehicle. Figure 3.5 shows the area around the vehicle where points are remembered.



**Figure 3.5.** Obstacle memory zones. The grey section shows the area in which obstacles are remembered.

### 3.4.3 Waypoint Threshold

The Waypoint Threshold is used during waypoint navigation to determine when a waypoint has been achieved. When the distance between the waypoint and the vehicle's center is less than this threshold, the waypoint is said to be achieved. While a smaller threshold means that the vehicle gets closer to the waypoint, it is also more difficult to hit due to GPS error. Therefore, this is generally set between 0.3 and 1.3 meters, depending on GPS accuracy.

#### **3.4.4 Velocity Parameters**

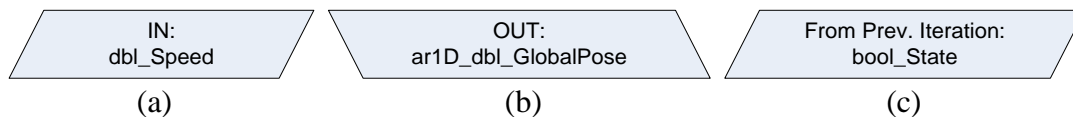
The five components of the Velocity Parameters array are the Minimum and Maximum Angles, Maximum Acceleration, Maximum Deceleration and Maximum Speed. The angles are used to calculate the speed based on the desired heading of the vehicle. For the Nemesis platform the minimum is set to 5 degrees and the maximum is 20 degrees. See Section 2.6 for more information on these settings. The Acceleration and Deceleration are used for the speed limiter and were initially set to  $1 \text{ m/s}^2$ . The Maximum Speed is a safety value that should be set to the highest speed allowed by the vehicle. For Nemesis this is 6 m/s. Finally, the Heading Rate is used by the heading rate limiter to limit how fast the vehicle turns. This was initially set to 1 rad/s on Nemesis.

## Chapter 4: Flowchart Conventions

The main method of documentation for the autonomous mobility algorithm was flowcharts. These flowcharts were used to document every detail of each function and process used in the algorithm. Several conventions were used in the creation of the flowcharts to help explain the algorithm in a clear and precise way. The five elements of the flowcharts are data, processes, decisions, wires, and terminators.

### 4.1 Data

Data is represented as parallelograms in the flowcharts. Any data which is input into a flowchart from an outside process faces right and contains the word “IN:” followed by the variable name. Data output by the flowchart which is used by other algorithms faces left and starts with the word “OUT:” followed by the name. Internal data which is kept within the algorithm also faces right, but does not have a designation in front of the name. This internal data is often used to show the output of a process called within the algorithm that is used in later processes. In some cases, the data for internal variables may have been determined in a previous iteration of a function. In these cases the phrase “From Prev. Iteration:” precedes the variable name. Figure 4.1 a, b, and c show inputs, outputs, and previous iteration data, respectively. Often times, several variables will be declared in a single object to save space.



**Figure 4.1.** Data representations for (a) inputs, (b) outputs, and (c) previous iterations.

Variables are defined using their data types as the first part of the name. This data type is then followed by an underscore and the descriptive name of the variable. For example, the variable for speed would be declared as “dbl\_Speed”. Table 4.1 explains the abbreviations and gives an example for each data type used. An

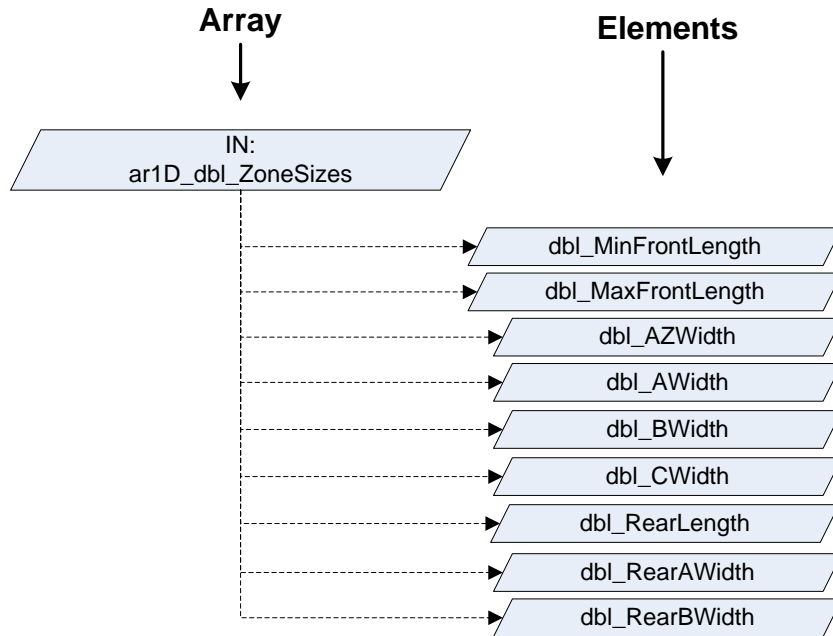
additional suffix may be added at the end to further specify a variable, such as “dbl\_Speed\_Des” for desired speed and “dbl\_Speed\_Fin” for the final speed.

**Table 4.1.** Data Type abbreviations.

Abbreviation	Data Type	Example
dbl	Double (64-bit precision)	dbl_FinalSpeed
u32	Unsigned 32-bit integer	u32_Index
u8	Byte	u8_CompStatus
bool	Boolean	bool_State

#### 4.1.1 Arrays

Arrays are complex data types that hold a large amount of information. They are first defined by the prefix “ar” then their dimension, followed by an underscore, the data type, another underscore, followed by the descriptive name. For example the waypoint data is named “ar2D\_dbl\_Waypoints,” meaning it is a two dimensional array of type double. In the algorithm, there are both 1 and 2 Dimensional arrays. In addition, some arrays have a fixed length while others vary. In some of the fixed-length arrays, such as Zone Sizes, each element has its own variable name. These arrays, called elemental arrays, can be split up into their individual elements in the flowchart, as shown in Figure 4.2. The variable length arrays are used for data, such as the obstacle points array which varies in size depending on the number of obstacles.



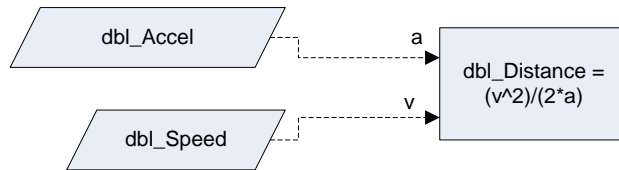
**Figure 4.2.** Elemental array splitting. Diagram showing how an elemental array is split up into its components in the flowcharts.

## 4.2 Processes

In the flowcharts, processes are represented by rectangles. There are both complex and simple process, each with its own flowchart representation. All processes that are complex have their own flowcharts associated with them are described by double line rectangles shown in Figure 4.3a. These blocks are given the same name as the functions they represent. There are four types of simple processes: scripts, variable assignments, arrays, and time. These are all placed in the single line rectangle shown in Figure 4.3b. Often times, data input to a process is shown by a wire from a data block into the process block. To reduce space that would be required for long variable names, an abbreviated name is sometimes given on the arrow going into the process block, as shown in Figure 4.4.



**Figure 4.3.** Representations for (a) a basic and (b) complex process



**Figure 4.4.** Data name size reduction. Diagram showing how variable names can be shortened to reduce space in a process block. Here the Acceleration is reduced to “a” and the Speed is reduced to “v” before the calculations.

#### 4.2.1 Scripts and Variable Assignments

Scripts are process which perform arithmetic or other calculations and assign a value to a particular output variable or set of variables. The simplest script is a variable assignment where the output variable is set equal to the input variable or a constant. Arithmetic scripts can be as simple as adding two numbers, or as intricate as the script for converting Lat/Long to UTM. Often times, the input variables for the script are shown by connection data wires. Simple scripts will only have one equation. The output is the variable on the left side of the equals sign. Some of the more involved scripts have multiple equations and define internal variables that are used in intermediate calculations. If multiple equations are used, variables that are output from the script are defined at the end and preceded by the “Output” command. Table 4.2 gives a list of all operators used in the script blocks.

**Table 4.2.** Operators used in script blocks

Symbol	Name	Description
+	Addition	Add two numbers or variables
-	Subtraction	Subtract two numbers or variables
*	Multiplication	Multiply two numbers or variables
/	Division	Divide two numbers or variables
%	Modulus	Modulus operator between two numbers. Defined as the remainder after division of two numbers.
^	Exponent	Raises the preceding number or variable to the power defined by the following number or variable
min()	Minimum	Takes the minimum value of all parameters inside the parentheses
sin()	Sine	Trigonometric sine of the value inside the parenthesis
cos()	Cosine	Trigonometric cosine of the value inside the parenthesis
tan()	Tangent	Trigonometric tangent of the value inside the parenthesis
atan2(y,x)	Signed Arc Tangent	Trigonometric signed tangent of the value inside the parenthesis. This takes two parameters.
sign()	Sign	Take the sign of value in parenthesis. Outputs -1 if negative and +1 if 0 or positive
length()	Length	Returns the length of an array in the u32 data type
pi	$\pi$ constant	Constant used to define $\pi$
dbl	Define double	Initializes a variable of type double
u8	Define byte	Initializes a variable of type byte
Output	Define outputs	Defines which variables are output from the script

#### 4.2.2 Arrays

This section explains the different processes used to manipulate arrays, including: indexing, adding, removing and replacing elements, and emptying.

The indexing of an array is used to analyze a specific element, row, or column. In the process blocks of the flowcharts, this performed by addressing the array, followed by a set of parentheses with the index value inside. A new variable

is then assigned the value that is retrieved from indexing. Arrays with a single dimension will have one number to identify the desired element, and will be indexed as such:

$$\mathbf{dbl\_X = ar1D\_dbl\_XY(1)}$$

This assigns the variable `dbl_X` the value contained in the first element of the array `ar1D_dbl_XY`. For 2D arrays, two index numbers, first corresponding to the column and second to the row, are required to retrieve a single element. However, often times an entire row is desired. In this case, a colon, “:”, in the column index is used to specify the entire row. An example of this is:

$$\mathbf{ar1D\_dbl\_XY = ar2D\_dbl\_ObstaclePoints( : , u32\_Index)}$$

which retrieves a 1D array, in the form (X,Y), from the 2D Obstacle Points array at the row number corresponding to the value in the variable `u32_Index`.

Adding, removing, and replacing elements of an array are three other common operations performed throughout the algorithm. When adding an element to an array, the syntax in the flowchart is:

$$\mathbf{\text{Add } ar1D\_dbl\_XY \text{ to } ar2D\_dbl\_ObstaclePoints \text{ at } ( : , u32\_Index)}$$

Here, the 1D row vector `ar1D_dbl_XY` is added to the 2D array at the row specified by `u32_Index`. The element that was previously at that index, along with all other elements, are shifted up one index. Therefore, if a vector was added at row 12, the previous row 12 would shift to row 13, 13 would shift to 14, and so on. If no index is given, the element is added to the end of the array. Conversely, when removing an element, all rows higher than that element will shift down one. The syntax for removing an element is:

$$\mathbf{\text{Remove } ( : , u32\_Index) \text{ from } ar2D\_dbl\_Obstacle \text{ Points}}$$

When replacing an element of an array, no shifting occurs. The new element simply overwrites the old element. In the process block, this is represented as:

Replace **ar2D\_dbl\_ObstaclePoints**( : , **u32\_Index**) with **ar1D\_dbl\_XY**

Emptying an array is the simple process by which all elements are removed from the array. The array then has zero elements. The syntax is simply:

Empty **ar2D\_dbl\_ObstaclePoints**

### 4.2.3 Time

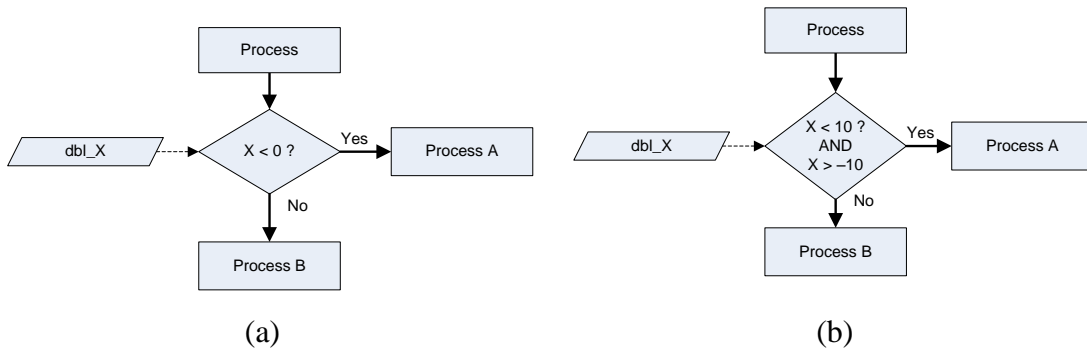
The time process is used for keeping track of how much time has gone by since a specific event occurred. The time is stored as a double. The only special process block used for determining time has the following syntax:

**dbl\_Time\_Curr** = Get Time

where “Get Time” is the language and OS specific method of retrieving the current time. This time is then stored in the variable called `dbl_CurrentTime`.

## 4.3 Decisions

Decision blocks are used to branch the flow of a process based on a certain condition or set of conditions. Each condition is a Boolean comparison. The chosen branch depends on whether the Boolean is True or False. The path for each decision is specified by either a “Yes” (true) or “No” (false) indicator on the outgoing wire. Simple decisions will only contain one condition, such as “ $X < 0$  .” Here, if X is negative, one course of action will be taken, while another course will be taken if X is positive. More complex decisions include Boolean operators, such as AND, OR, XOR, between multiple conditions. An example of this is: “ $X < 10$  AND  $X > -10$ ,” where the decision will result in True if X is between -10 and 10, and False otherwise. Figure 4.5 shows examples of both simple and complex decisions.



**Figure 4.5.** Decision Representations. (a) is a simple decision and (b) is complex.

## 4.4 Wires

Wires are used to connect the various blocks of the flowcharts. There are two categories of wires: process wires and data wires. Process wires are solid and bold while data wires are dashed and thin, as shown in Figure 4.1. Process wires show the logical flow through the processes of each function. Data wires help explain the variables that are used for certain processes or decisions. They are not used in every situation, only where clarification may be necessary. Both types of wires may contain junctions.

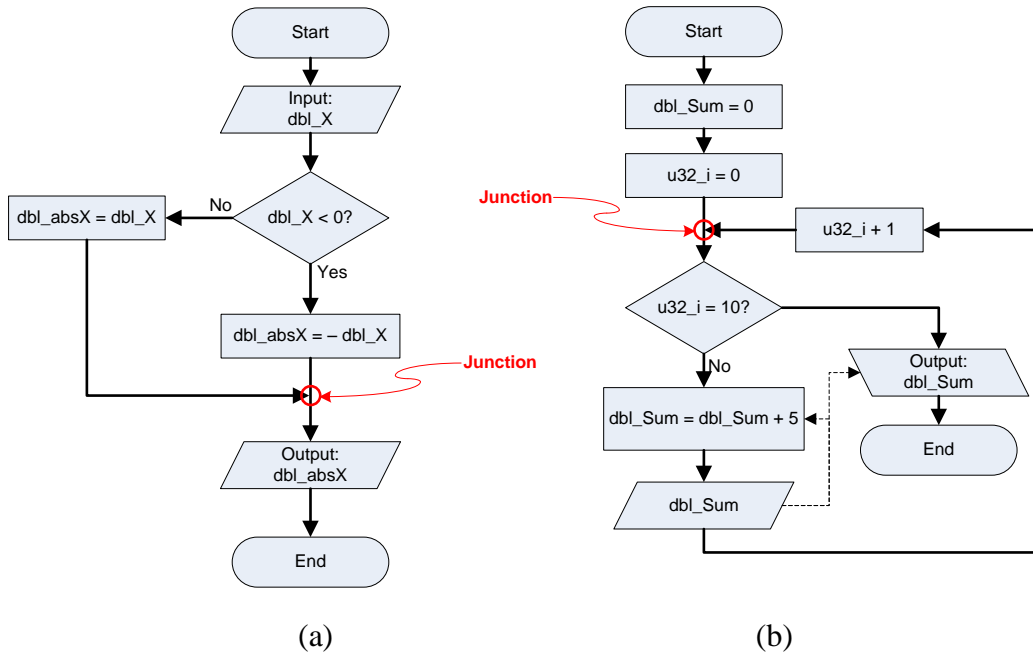


**Figure 4.6.** Flowchart representation for (a) process and (b) data wires.

### 4.4.1 Junctions

Junctions refer to places where wires split or merge. As a rule, only data wires will split and only process wires will merge. Splitting indicates that data from a single variable is used in multiple processes or decisions. Process wires will merge for two reasons: branches and loops. Branches refer to parts of the flowchart that split due to a certain condition, such as “is  $x < 0$  ?.” Each condition will result in a different set of processes. Many times the flow of the function will return to a common point, resulting in a wire merge. This is shown in Figure 4.7a where the absolute value of an input `dbl_X` is determined. Merges also occur in loops where

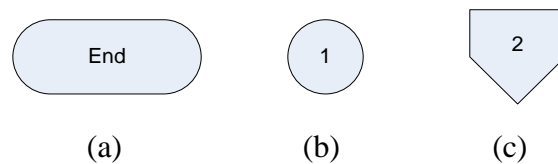
the flow of processes returns to the beginning of the loop, as shown in Figure 4.7b. Here the number 5 is added the number of times specified by  $u32\_i$ .



**Figure 4.7.** Wire merging. Diagram for (a) branches and (b) loops.

## 4.5 Terminators

Every flowchart has a Start and End terminator to indicate where the process flow begins and finishes. These are indicated by rounded rectangles, shown in Figure 4.8a. In addition to these two terminators, several flowcharts have on-page references, which are represented as circles with a number inside them, as shown in Figure 4.8b. These references act as a merge in the process wire, where multiple branches with different operations continue at some common point. Each on-page reference may be called in multiple places but will only have one continuation point. In the case of multiple different references, the number indicates which reference to follow. Off-page references are used in the overall flow diagram because the whole flowchart will not fit on a single page. This is documented by the shape shown in Figure 4.8c. The off-page reference follows the same behavior as an on-page reference.



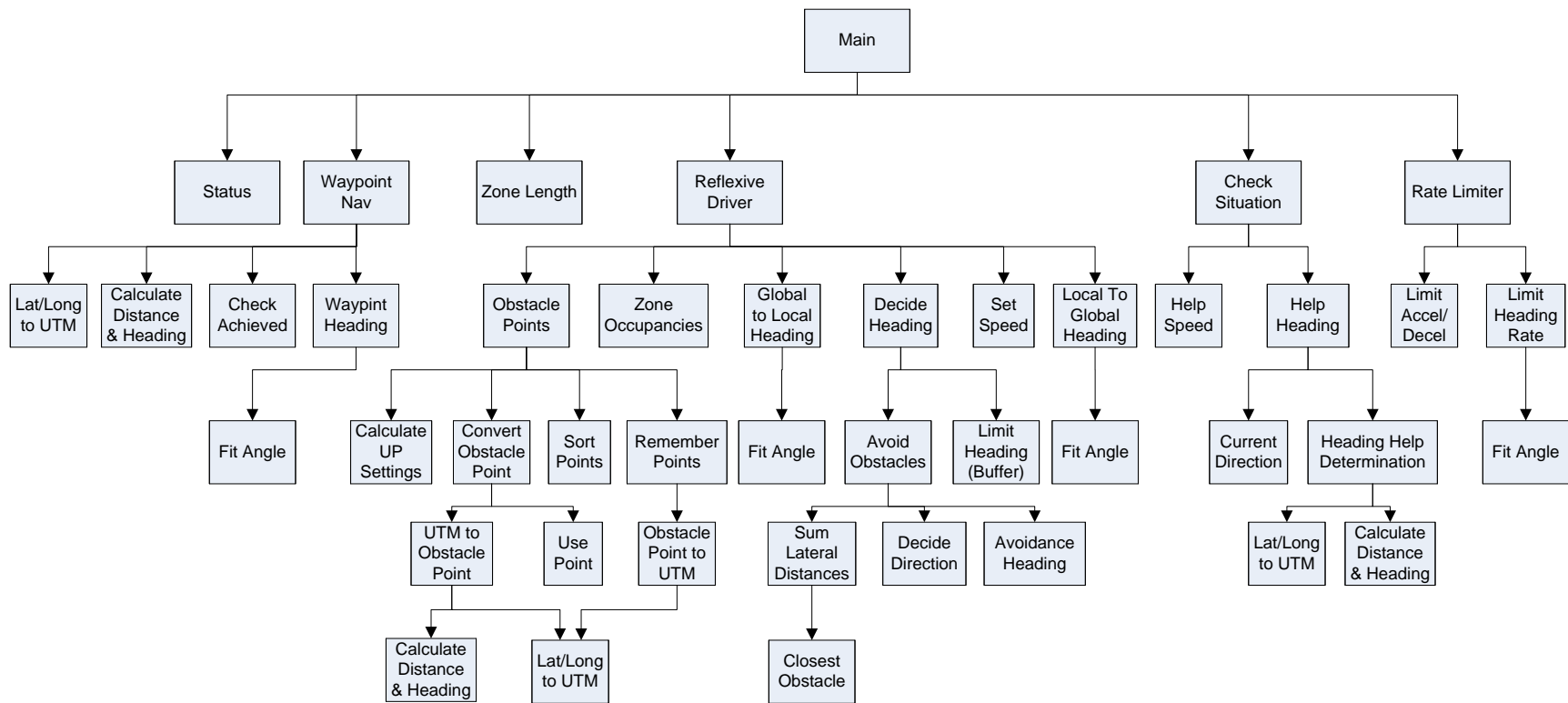
**Figure 4.8.** Flowchart representation of (a) function terminators, (b) on-page references, and (c) off-page references.

## Chapter 5: Algorithm Documentation

The DDEZm algorithm developed for the Nemesis platform was documented for later coding by ARA's software engineers. Therefore, the documentation needed to be meticulous in its description of every detail of the algorithm. It was broken up into function descriptions and variable descriptions. Function descriptions explain the individual processes used in the overall algorithm. The variables are used to represent the information passed between functions. This chapter contains the overall function hierarchy, documentation of the main, top-level function, and the 6 high-level functions of the algorithm. It also includes the descriptions of the most important variables, which control the behavior of the algorithm. Note that because they are not part of the actual algorithm, the user interface and JAUS communications with the vehicle and OCU are left out.

### 5.1 Function Hierarchy

Figure 5.1 shows the hierarchical layout of the functions in the algorithm. This flows from top to bottom and left to right. The Main function is constantly executing. Within it, Status is called first, then Waypoint Navigation, which calls its own four functions, then Zone Length. Reflexive Driver calls six functions starting with Obstacle Points, which calls its own four functions. This flow continues until the end with the Limit Heading function called by the Rate Limiter. After the Rate Limiter, Main outputs the desired Global Vector to control the vehicle. List 5.1 gives the functions in the hierarchical order represented by the Figure 5.1.



**Figure 5.1.** Hierarchical layout of the functions in the algorithm

**Table 5.1.** Function Hierarchy

<b>Function Name</b>
<ul style="list-style-type: none"> <li>➤ <b>Main</b> <ul style="list-style-type: none"> <li>• <b>Status</b></li> <li>• <b>Waypoint navigation</b> <ul style="list-style-type: none"> <li>○ Lat/Long to UTM</li> <li>○ Calculate Distance and Heading</li> <li>○ Check Achieved</li> <li>○ Waypoint Heading                             <ul style="list-style-type: none"> <li>▪ Fit Angle</li> </ul> </li> </ul> </li> <li>• <b>Calculate Zone Length</b></li> <li>• <b>Reflexive Driver</b> <ul style="list-style-type: none"> <li>○ Obstacle Points                             <ul style="list-style-type: none"> <li>▪ Calculate UP Settings</li> <li>▪ Convert Obstacle Point                                     <ul style="list-style-type: none"> <li>• UTM to Obstacle Point   <ul style="list-style-type: none"> <li>○ Lat/Long to UTM</li> <li>○ Calculate Distance and Heading</li> </ul> </li> <li>• Use Point</li> </ul> </li> <li>▪ Sort Points</li> <li>▪ Remember Points                                     <ul style="list-style-type: none"> <li>• Local to UTM   <ul style="list-style-type: none"> <li>○ Lat/Long to UTM</li> </ul> </li> </ul> </li> </ul> </li> <li>○ Zone Occupancies</li> <li>○ Global to Local Heading                             <ul style="list-style-type: none"> <li>▪ Fit Angle</li> </ul> </li> <li>○ Decide Heading                             <ul style="list-style-type: none"> <li>▪ Avoid Obstacles                                     <ul style="list-style-type: none"> <li>• Sum Lateral Distances   <ul style="list-style-type: none"> <li>○ Closest Obstacle</li> </ul> </li> <li>• Decide Direction</li> <li>• Avoidance Heading</li> </ul> </li> <li>▪ Limit Heading</li> </ul> </li> <li>○ Set Speed</li> <li>○ Local to Global Heading                             <ul style="list-style-type: none"> <li>▪ Fit Angle</li> </ul> </li> </ul> </li> <li>• <b>Check Situation</b> <ul style="list-style-type: none"> <li>○ Help Speed</li> <li>○ Help Heading                             <ul style="list-style-type: none"> <li>▪ Current Direction</li> <li>▪ Heading Help Algorithm                                     <ul style="list-style-type: none"> <li>• Lat/Long to UTM</li> <li>• Calculate Distance and Heading</li> </ul> </li> </ul> </li> </ul> </li> <li>• <b>Rate Limiter</b> <ul style="list-style-type: none"> <li>○ Limit Accel/Decel</li> <li>○ Limit Heading Rate                             <ul style="list-style-type: none"> <li>▪ Fit Angle</li> </ul> </li> </ul> </li> </ul> </li> </ul>

## **5.2 Functions**

Each individual function used in the autonomous mobility algorithm is explained by a specifications table, textual description, and flowchart. The specifications table outlines where the function was called, any functions that it calls, and its input and output variables. This section provides the descriptions of the Main function as well as the six functions that it calls: Status, Waypoint Navigation, Calculate Zone Length, Reflexive Driver, Check Situation, and Rate Limiter.

### 5.2.1 Main

This flowchart explains the overall flow from process to process of the entire DDEZm algorithm. All input data for this, and only this, function are user-defined constants. Likewise, all output data is sent to the vehicle. All data that enters the function via an outside JAUS message is prefixed by the words “JAUS Message:” The algorithm starts out by determining the current component status of the algorithm. If the status is in the Standby or Emergency states, the final vector is set to stop the vehicle. However, in the Ready state the full autonomous mobility algorithm is carried out. This part of the algorithm first determines the global heading to the current desired waypoint. Next, the zone length is calculated based on the final vector from the previous iteration of the algorithm. The Zone Sizes array with the new length is then sent to the reflexive driver function, which determines how to avoid the obstacles. The main purpose of the Reflexive Driver is to output the Reflexive Global Vector, which is the desired vehicle vector. In addition, the perception Zone Occupancies and a Boolean Need Help flag requesting operator assistance are output. The Check Situation function then determines if the vehicle is “stuck” in a situation it cannot solve alone. This also takes into account the Need Help flag and Zone Occupancies from the Reflexive Driver. If help is required, the final vector is set to stop the vehicle and the component status is then set to Emergency. Otherwise, both are left alone. Next, the desired vector is sent to the rate limiter so that the commanded vector does not try to instantaneously change the speed and heading of the vehicle, which could be dangerous by causing the vehicle to roll or flip. Finally, the Final Global Vector is remembered for the next iteration and then sent to the vehicle.

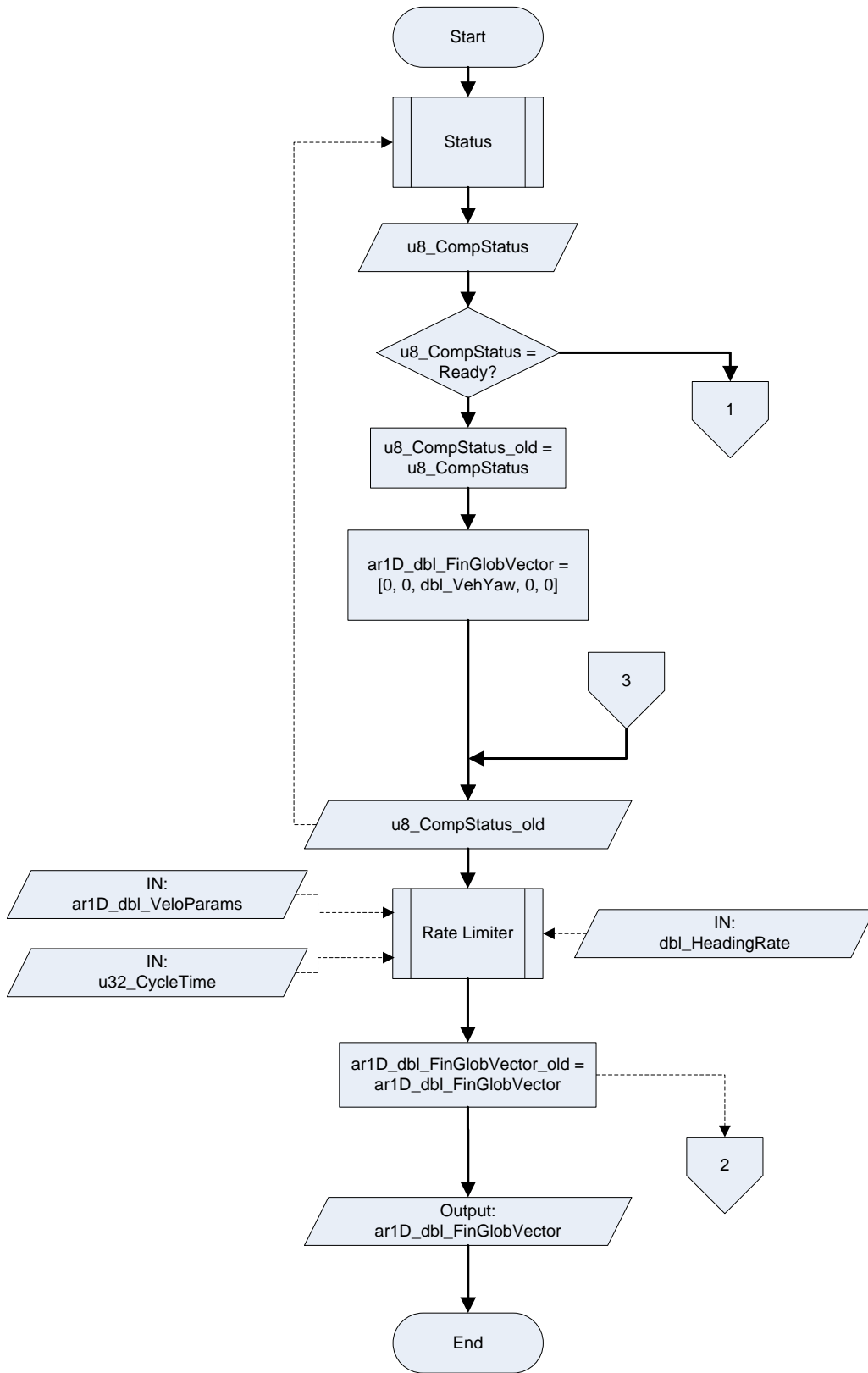
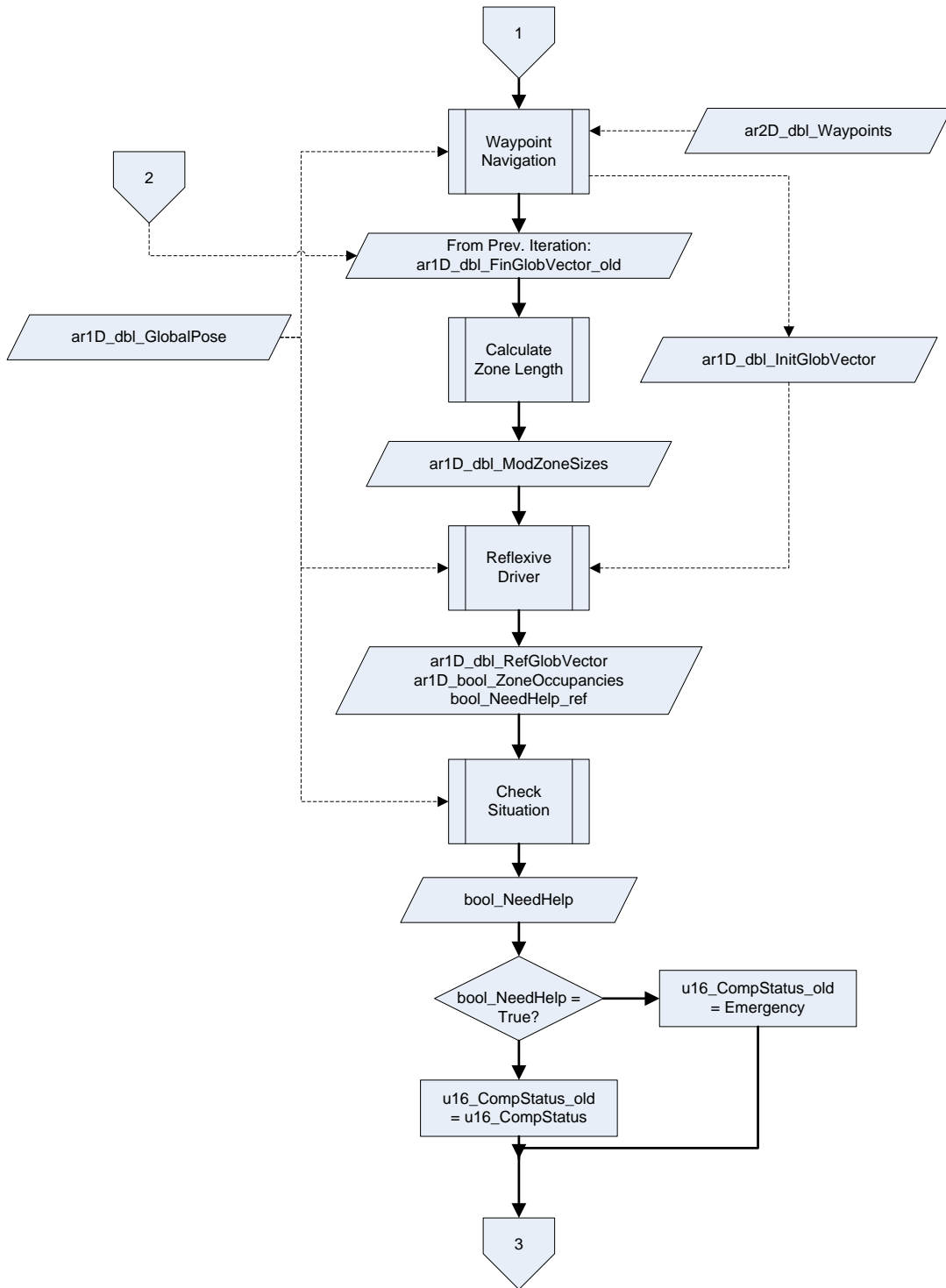


Figure 5.2. Main Function Flowchart, Part 1.



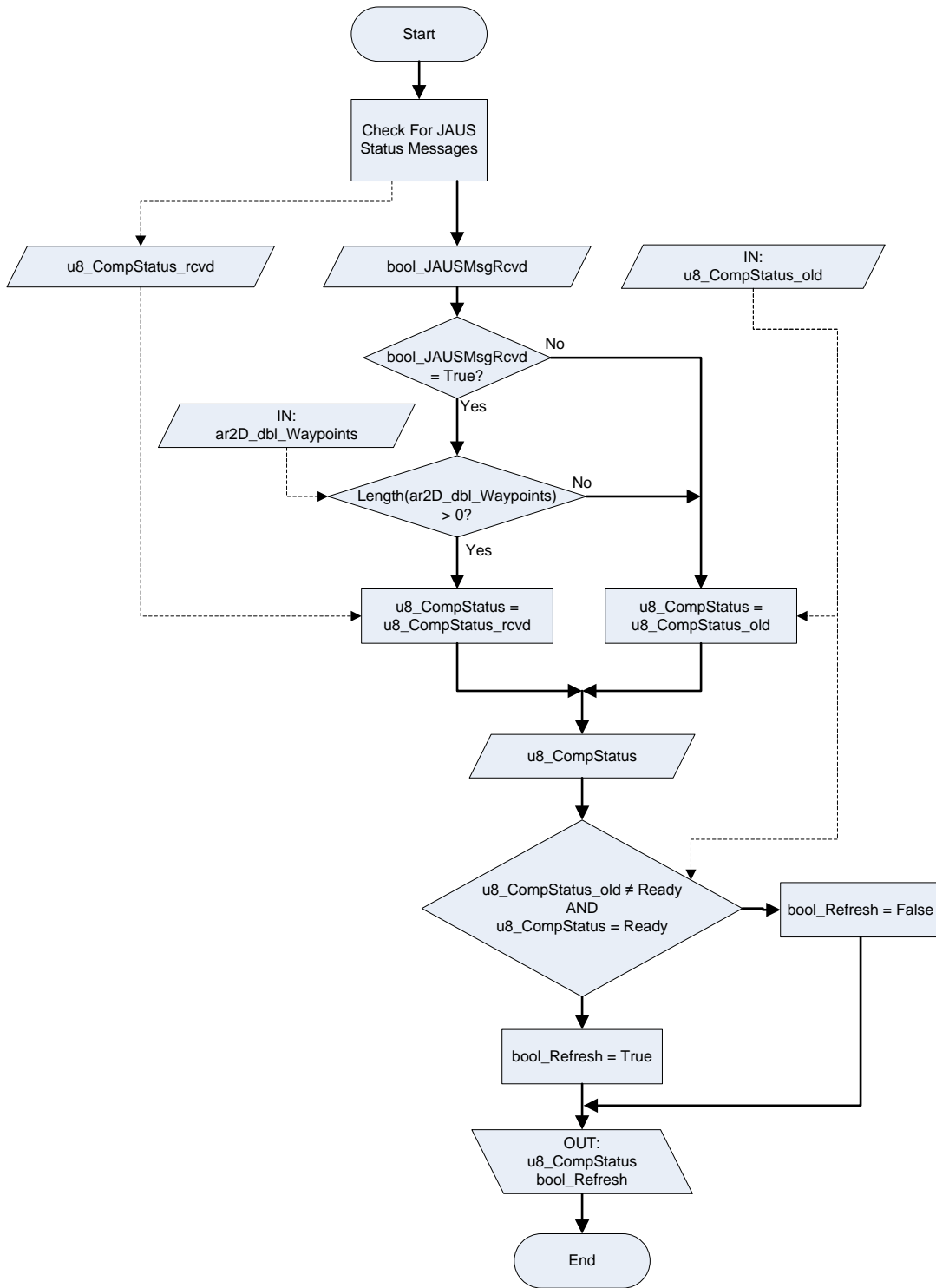
**Figure 5.3.** Main Function Flowchart, Part 2.

## 5.2.2 Status

**Table 5.2.** Status Specifications

Called by	Main Function
Function Calls	None
Inputs	Old Component Status Waypoints
Outputs	Component Status Refresh flag

The Status function is used to set the Component Status (Standby, Ready, Emergency). It first checks for any JAUS Status messages (*0003 – Standby*, or *0004 – Resume*). If there are messages, the status is set based on the last message received. Otherwise the status is set to whatever it was at the end of the previous iteration of the overall loop. This is done because other functions have the ability to change the status based on their own conditions. If the New Component Status is equal to Ready, and the Old Component Status was either Standby or Emergency, the Refresh Boolean is set to True.



**Figure 5.4.** Flowchart for the Status function

### 5.2.3 Waypoint Navigation

**Table 5.3.** Waypoint Navigation Specifications

Called by	Main Function
Function Calls	Lat/Long to UTM Calculate Distance & Heading Check Achieved Waypoint Heading
Inputs	Waypoints Global Pose Component Status Waypoint Threshold
Outputs	Initial Global Heading Component Status

The Waypoint Navigation function is responsible for determining the Initial Local Heading that controls the vehicle's waypoint following behavior. The function uses a list of waypoints that are input to the algorithm via JAUS messages. This waypoint list contains the Latitude and Longitude of each desired point in consecutive order. The Waypoint Number variable is used to keep track of which waypoint the vehicle is currently trying to achieve. At the very first call of this function, the Waypoint Number is set to zero. In addition, if a new set of waypoints replaces the old set, the JAUS messaging must reset the Waypoint Number to zero. The function indexes the Latitude and Longitude of the current waypoint, specified by the Waypoint Number and converts it to UTM. The vehicle's position is also converted to UTM and sent with the waypoint UTM coordinates into the Calculate Distance & Heading function. This function outputs the distance and heading from the vehicle to the waypoint. The distance and Waypoint Threshold are then used by the Check Achieved function to determine if the vehicle has achieved the current waypoint. If so, the function starts over with the new waypoint number. If the point has not been achieved, the Waypoint Heading function determines the desired heading to the waypoint, called the Initial Global Heading, which is output from the function along with the Component Status.

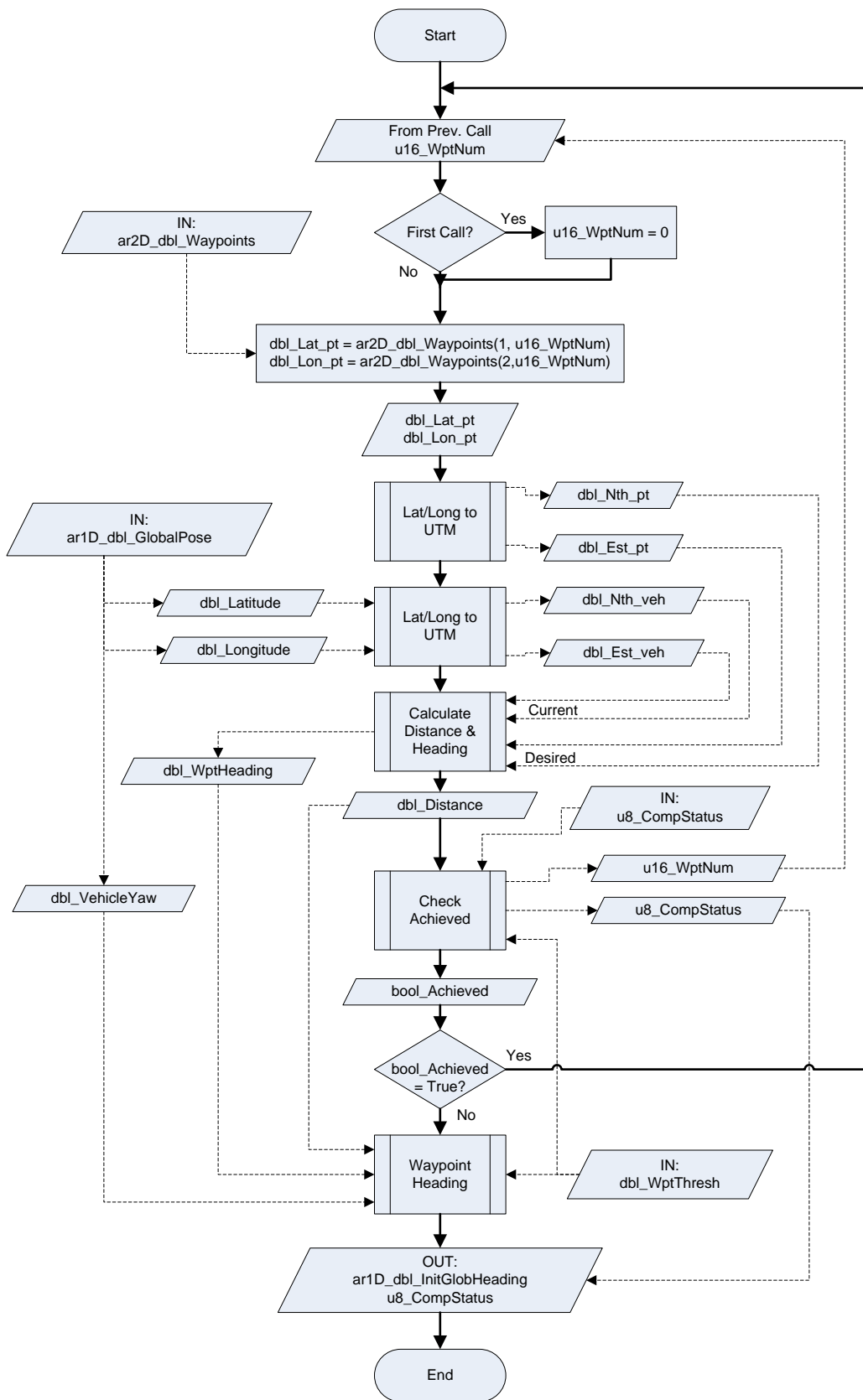


Figure 5.5. Flowchart for the Waypoint Navigation function.

## 5.2.4 Calculate Zone Length

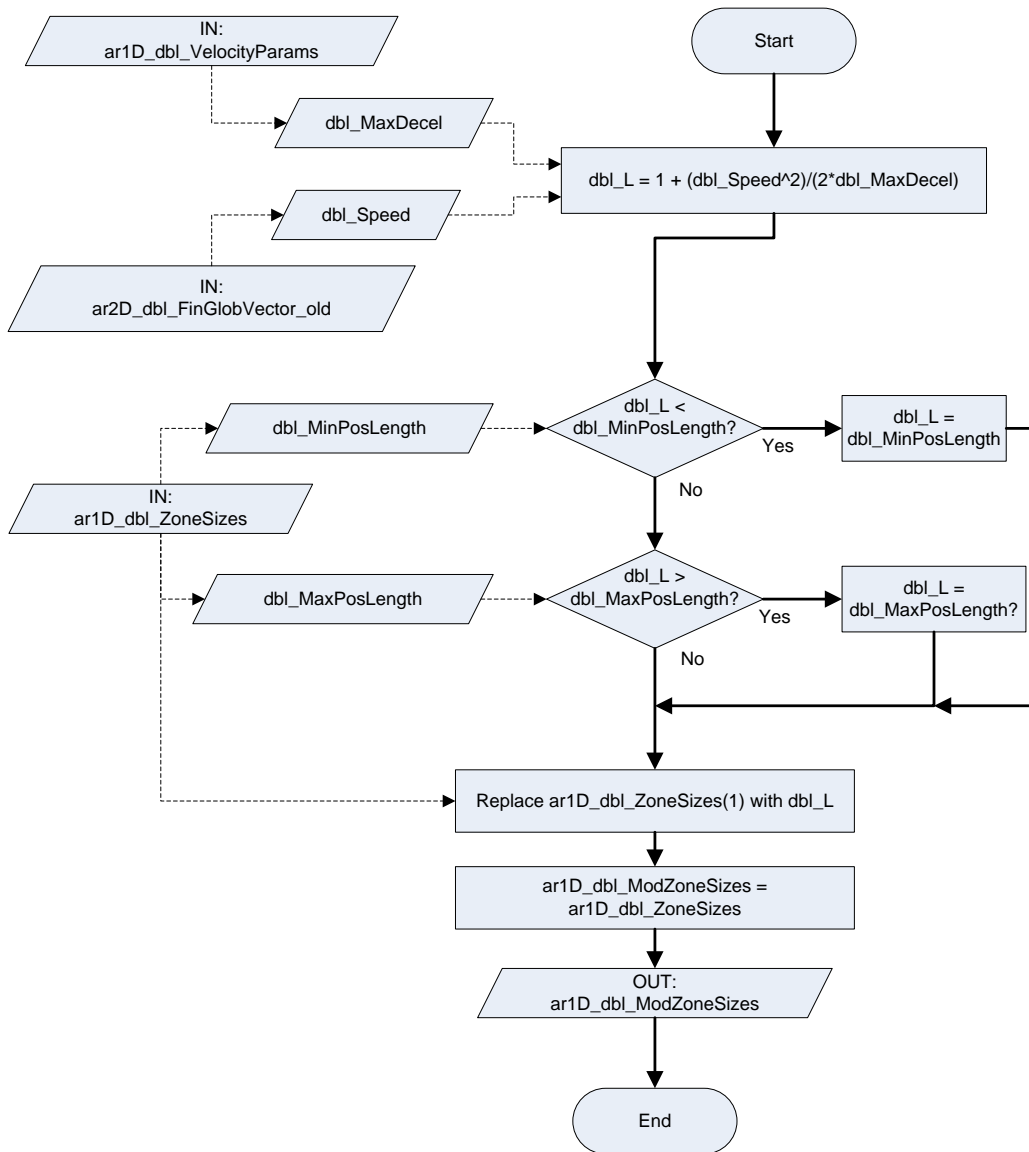
**Table 5.4.** Calculate Zone Length Specifications

Called by	Main Function
Function Calls	None
Inputs	Velocity Parameters Old Final Global Vector Zone Sizes
Outputs	Modified Zone Sizes

The Calculate Zone Length function is used to determine how far the avoidance software looks in front of the vehicle. This distance is based on the vehicle's required stopping distance, which is then based on speed and maximum deceleration. The relationship is:

$$ZoneLength = 1 + \frac{v^2}{2a} \quad (5.1)$$

where  $v$  is the vehicle's velocity, in m/s and  $a$  is the magnitude of the maximum deceleration in  $m/s^2$ . The added one meter is used as a factor of safety added to the stopping distance. The speed used in the calculation is the speed from the Final Global Vector output from the previous iteration of the overall algorithm. The deceleration is an element of the velocity parameters array. After calculating the desired zone length, it is limited to the range of the minimum and maximum lengths specified by the Zone Sizes array. Once the final length is determined, it is added back into the Zone Sizes array in the element corresponding to the minimum length. This is then saved in the variable called the Modified Zone Sizes array. The Modified Zone Sizes array is then output from the function. The minimum length element of the array will be used throughout the rest of the algorithm as the actual zone length.



**Figure 5.6.** Flowchart for the Calculate Zone Length function

### 5.2.5 Reflexive Driver

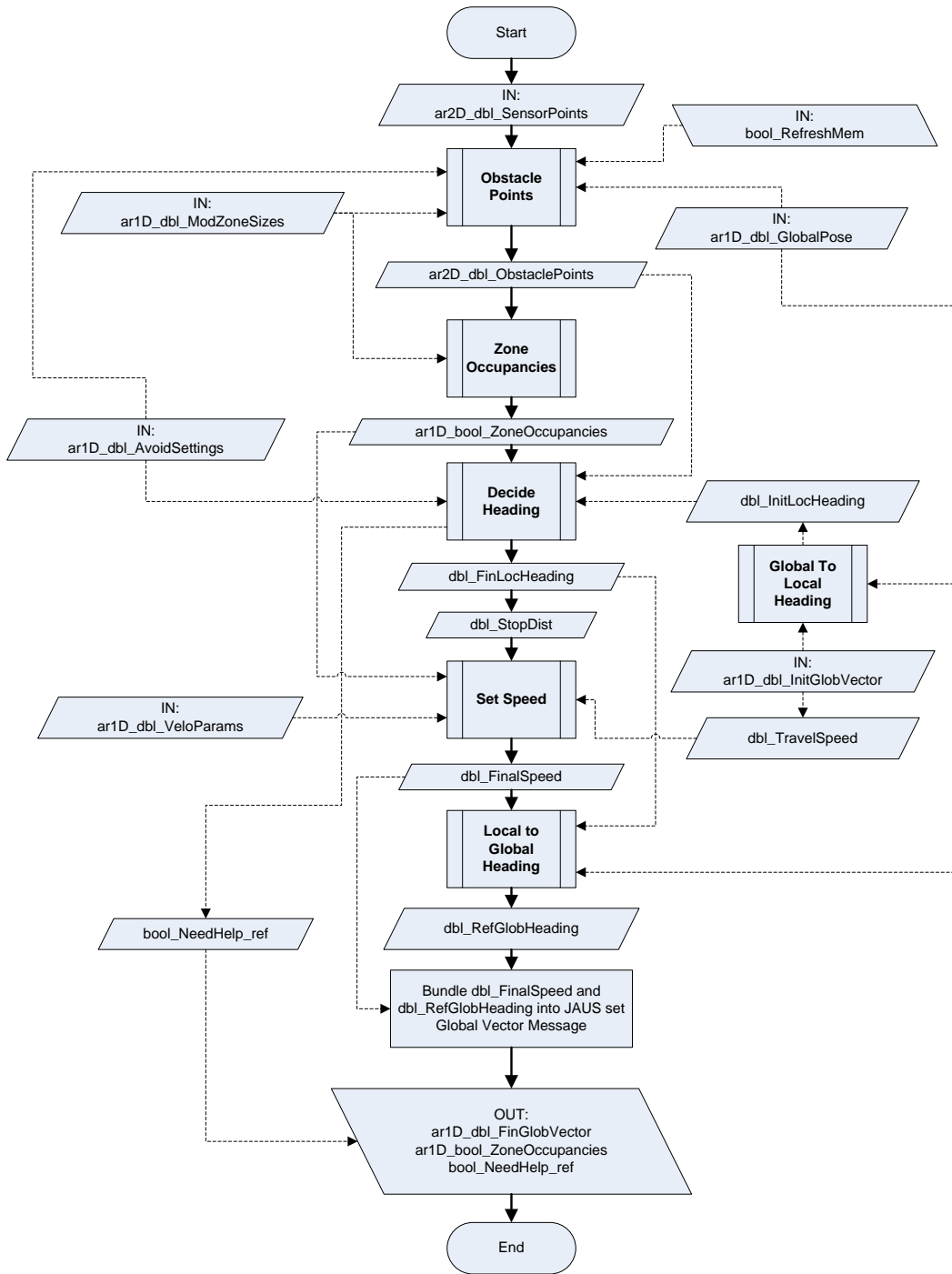
**Table 5.5.** Reflexive Driver Specifications

Called by	Main Function
Function Calls	Obstacle Points Zone Occupancies Global to Local Heading Decide Heading Set Speed Local to Global Heading
Inputs	Sensor Obstacle Points Refresh flag Global Pose Avoidance Settings Initial Global Heading Velocity Parameters
Outputs	Reflexive Global Vector Zone Occupancies Reflexive Need Help flag

The Reflexive Driver function performs the obstacle avoidance behavior of the algorithm. It contains four subroutines that handle the obstacle avoidance, as well as two more that deal with the heading conversions between the local and global reference frames. The two main inputs to the function are the Sensor Points array, which describes the obstacle positions, and the Initial Global Vector, which is the desired vector from the Waypoint Navigation function.

First, the Obstacle Points function determines which of the sensor points are inside the perception zones defined by the Modified Zone Sizes array. These points, called the Obstacle Points are then fed into the Zone Occupancies function, which determines which of the perception zones contain obstacles. This function outputs the array of Zone Occupancies. The heading component of the Initial Global Vector from Waypoint Navigation is then converted to a local heading for the obstacle avoidance function. Decide Heading uses the Avoidance Settings array, initial local heading from the Waypoint Navigation, the Zone Occupancies, and the Obstacle Points array to determine the best local heading that will avoid the obstacles while travelling to the waypoint. This function outputs the Reflexive Local Heading to avoid obstacles, a Stop Distance, which is the distance to the closest obstacle in the

Avoidance Zone, and a Need Help Boolean, flagging if operator assistance is required. The Set Speed function then uses the heading and stop distance to determine the best speed to command to the vehicle. Next the Reflexive Local Heading is converted to a global heading and combined with the speed to create the Reflexive Global Vector. The Reflexive Global Vector, Zone Occupancies and Help flag are then output back to the main function.



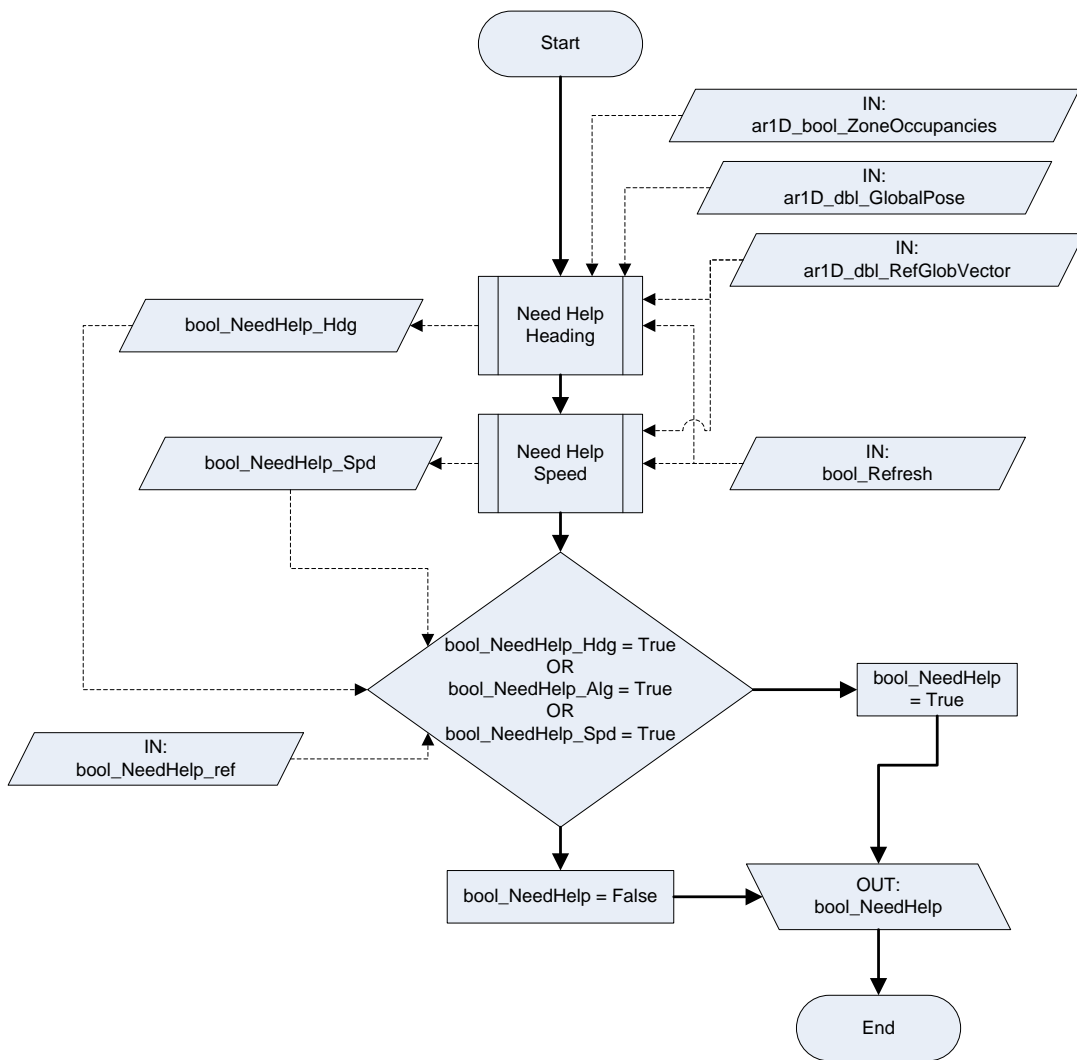
**Figure 5.7.** Flowchart for the Reflexive Driver function.

## 5.2.6 Check Situation

**Table 5.6.** Check Situation Specifications

Called by	Main Function
Function Calls	Need Help Speed Need Help Heading
Inputs	Zone Occupancies Global Pose Reflexive Global Vector Reflexive Need Help Refresh flag
Outputs	Need Help

This function is used to determine if the vehicle has become “stuck” in a situation requiring operator assistance. There are three cases that are monitored: Reflexive Driver, Speed, and Heading. These have corresponding flags called Reflexive Need Help, Speed Need Help, and Heading Need Help. The Reflexive Need Help Boolean is flagged in the Reflexive Driver function. The Speed and Heading Need Help Booleans are flagged in their respective functions, called from this Check Situation Function. These functions are called Need Help Speed and Need Help Heading, respectively. If any one of the three Booleans is flagged true, the final Need Help Boolean is flagged true, signifying that operator assistance is required. This final Boolean is output at the end of the function.



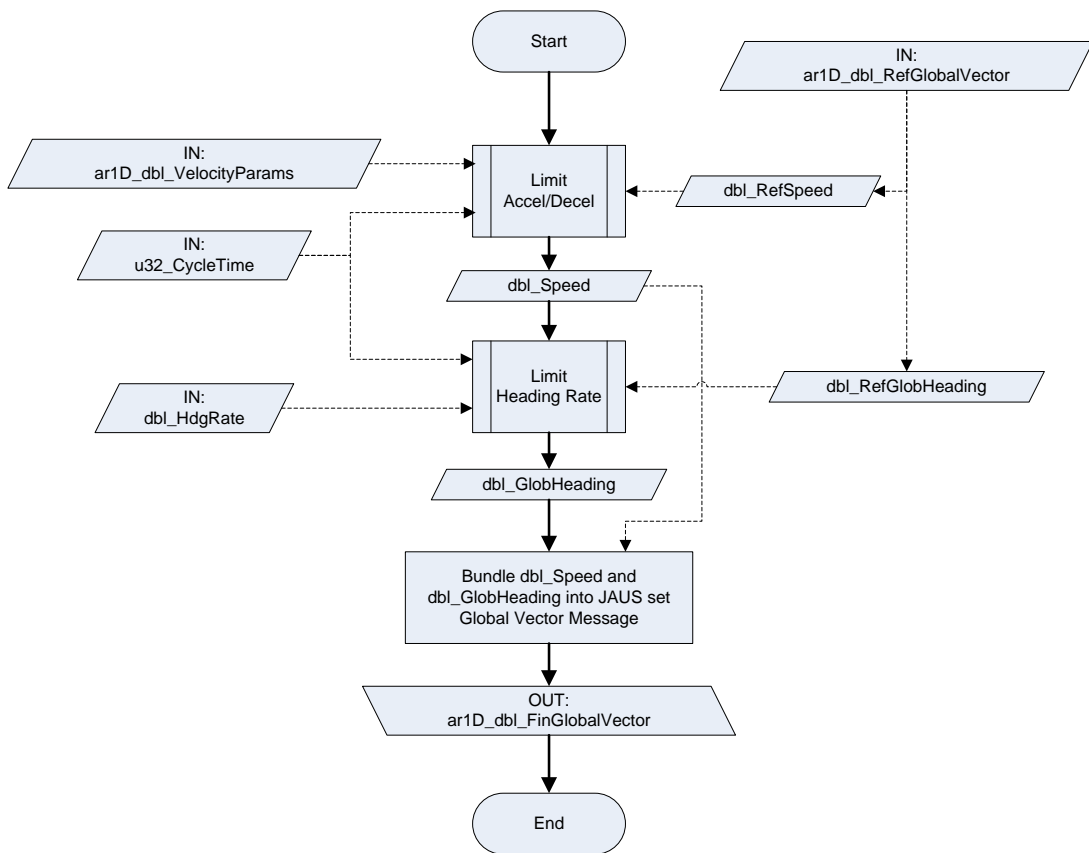
**Figure 5.8.** Flowchart for the Check Situation function.

### 5.2.7 Rate Limiter

**Table 5.7.** Rate Limiter Specifications

Called by	Main Function
Function Calls	Limit Accel/Decel Limit Heading Rate
Inputs	Velocity Parameters Reflexive Global Vector Cycle Time
Outputs	Final Global Vector

The purpose of the rate limiter function is to limit the acceleration and deceleration of the vehicle, as well as the rate of change of the commanded heading. This function calls the Limit Accel/Decel function to limit the acceleration and deceleration, and the Limit Heading Rate function to limit the heading rate. Limit Accel/Decel takes in the commanded speed from the Reflexive Global Vector, the Velocity Parameters, and the Cycle Time, and outputs the final commanded speed. The Cycle Time is an integer which represents the average number of milliseconds required to execute the entire DDEZm autonomous mobility algorithm. The Limit Heading Rate function also takes in the Cycle Time, but requires the global heading from the Reflexive Global Vector, and the Heading Rate variable. It outputs the final commanded global heading. After the two functions execute the output Speed and Global Heading are combined to form the Final Global Vector that will be sent to the vehicle.



**Figure 5.9.** Flowchart for the Rate Limiter function

## 5.3 Variables

This section describes some of the most important variables that control the behavior of the vehicle. These are variables that occur in several functions. Like the function descriptions, the variable descriptions also contain a specifications table and textual description. Here, the specifications table gives the variable's name, type, units, range, the functions it occurs in, and the size and representation if it is an array. While the data for most variables are calculated inside of functions, there are several that are user defined. These include the Avoidance Settings, Velocity Parameters, Zone Sizes, Waypoint Threshold. In addition, the Global Pose, Sensor Obstacle Points, Waypoints, and Travel Speed are set via JAUS messages outside of the main algorithm. This section provides the documentation for each of the aforementioned variables as well as the Obstacle Points array, Temporary Point Set, Zone Occupancies, and Status variables.

### 5.3.1 Velocity Parameters \*User Input\*

**Table 5.8.** Velocity Parameters Specifications

Variable Name	ar1D_dbl_VeloParams
Type	1D array of double
Size	6 x 1
Representation	0. Maximum Angle 1. Minimum Angle 2. Maximum Acceleration 3. Maximum Deceleration 4. Maximum Speed 5. Heading Rate
Units	0 – 1: Radians 2 – 3: m/s <sup>2</sup> 4: m/s 5: rad/s
Range	Multiple (see individual variable descriptions)
Occurs in	Calculate Speed Rate Limiters Limit Accel/Decel

This variable is set by the user to control the speed calculations. The Maximum Speed and Minimum and Maximum Angles help determine the slope of the speed vs. heading curve. The Maximum Acceleration and Deceleration are used in limiting the rate of change of the speed. The Heading Rate is the factor used for limiting the rate of change of heading.

### 5.3.2 Zone Sizes \*User Input\*

**Table 5.9.** Zone Sizes Specifications

Variable Name	ar1D_dbl_ZoneSizes
Type	1D array of double
Size	9 x 1
Representation	<ul style="list-style-type: none"> <li>0. Minimum Length</li> <li>1. Maximum Length</li> <li>2. Avoidance Zone Width</li> <li>3. A Buffer Width</li> <li>4. B Buffer Width</li> <li>5. C Buffer Width</li> <li>6. Rear Length</li> <li>7. Rear A Buffer Width</li> <li>8. Rear B Buffer Width</li> </ul>
Units	Meters
Range	0 to $\infty$
Occurs in	<ul style="list-style-type: none"> <li>Calculate Zone Length</li> <li>Calculate UP Settings</li> <li>Zone Occupancies</li> </ul>

The Zone Sizes array describes the length and width of the perception zones that are used to make avoidance decisions. The Avoidance Zone Width refers to the width of the center zone. The A, B, and C Buffers have the same respective width on the right and left side of the vehicle. While the actual length of the front zones will change with speed, the Maximum and Minimum Lengths define the limits of the length. The Rear Length and zone widths define the sizes of the zones located behind the front plane of the vehicle. See Chapter 2, Section 5 for a more in-depth explanation of these zones.

### 5.3.3 Avoidance Settings \*User Input\*

**Table 5.10.** Avoidance Settings Specifications

Variable Name	ar1D_dbl_AvoidSettings
Type	1D array of double
Size	10 x 1
Representation	0. B Heading Limit 1. C Heading Limit 2. Search Length 3. Search Width 4. Summation Threshold 5. Front Remember Distance 6. Rear Remember Distance 7. Sensor X Offset 8. Sensor Y Offset 9. VLF
Units	0 – 1: Radians 2 – 9: Meters
Range	Multiple (see individual variable descriptions)
Occurs in	Calculate UP Settings Convert Obstacle Point Remember Points Decide Heading Avoid Obstacles

The Avoidance Settings array contains 10 variables that affect the obstacle avoidance part of the algorithm. The heading limits are used for determining the vehicle's heading when the buffer zones contain obstacles. The Search Length and Width, determine the obstacles in the vehicle's path that merit avoidance. The Summation Threshold helps determine the avoidance direction. Elements 5 and 6 control the area around the vehicle at which obstacle points are remembered. Finally, the last three elements are used for translating the obstacle points from the sensor frame to the vehicle frame. The VLF variable stands for Vehicle Length in Front of CG. See Section 3.4.2 for more information on this variable.

### 5.3.4 Global Pose \*JAUS Message\*

**Table 5.11.** Global Pose Specifications

Variable Name	ar1D_dbl_GlobalPose
Type	1D array of double
Size	10 x 1
Representation	0. Latitude 1. Longitude 2. Vehicle Yaw
Units	0 – 1: Degrees 2: Radians
Range	0 : -90 to 90 1: -180 to 180 3: $-\pi$ to $\pi$
Occurs in	Waypoint Navigation Global to Local Heading Local to Global Heading Convert Obstacle Point UTM to Obstacle Point Obstacle Point to UTM

This variable is written by a periodic JAUS message to convey the position and orientation of the vehicle. This is given using Latitude, Longitude and Yaw, or heading. It is periodically written by a JAUS message.

### 5.3.5 Waypoint Threshold \*User Input\*

**Table 5.12.** Waypoint Threshold Specifications

Variable Name	dbl_WptThresh
Type	Double
Units	Meters
Range	0 to $\infty$
Occurs in	Check Achieved Waypoint Heading

The Waypoint Threshold defines the minimum distance from the vehicle to the waypoint at which the waypoint can be considered achieved.

### 5.3.6 Waypoints \*JAUS Message\*

**Table 5.3.** Waypoints Specifications

Variable Name	ar2D_dbl_Waypoints
Type	2D array of double
Size	N x 2, Where N = # of waypoints
Representation	Lat0, Lon0 Lat1, Lon1 ... LatN, LonN
Units	Degrees, Degrees
Range	-90 to 90, -180 to 180
Occurs in	Waypoint Navigation

This is a list of waypoints that is set via a JAUS Message. Each point contains a Latitude and Longitude. The length of the array is equal to the number of (lat, long) waypoints.

### 5.3.7 Travel Speed \*JAUS Message\*

**Table 5.14.** Travel Speed Specifications

Variable Name	dbl_Speed_Travel
Type	Double
Units	m/s
Range	0 to $\infty$
Occurs in	Set Speed

The Travel Speed is set by a JAUS message and portrays the maximum desired speed that the vehicle should drive.

### 5.3.8 Obstacle Points

**Table 5.15.** Obstacle Points Specifications

Variable Name	ar2D_dbl_ObstaclePoints
Type	2D array of double
Size	N x 2, Where N = # of waypoints
Representation	x1, y1 x2,y2 ... xN, yN
Units	Meters, Meters
Range	$-\infty$ to $\infty$
Occurs in	Obstacle Points Sort Points Zone Occupancies Sum Lateral Distances Closest Obstacle

The Obstacle Points array contains a list of (x,y) points that explain the location of obstacles with respect to the front of the vehicle. Each row of the array contains a single obstacle point. Therefore, the length of the array corresponds to the total number of observed points.

### 5.3.9 Component Status

**Table 5.16.** Component Status Specifications

Variable Name	dbl_CompStatus
Type	Byte
Interpretation	0. Standby 1. Ready 2. Emergency
Occurs in	Main Status Check Situation Check Achieved

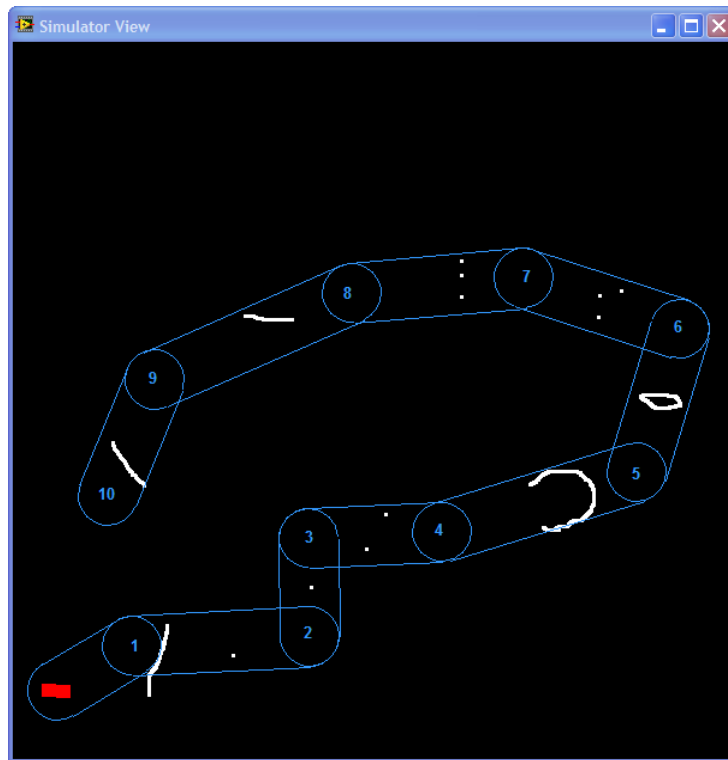
This variable is used to portray the status of the system. It is a single byte enumeration which corresponds to 0 = Standby, 1 = Ready and 2 = Emergency.

## Chapter 6: Results

Throughout the development process, the DDEZm algorithm was tested both in simulation and on the Nemesis vehicle. While the majority of the testing was performed in simulation, it was ported to the vehicle for final assessment. This chapter explains the findings of both simulator and on-vehicle testing. This includes an analysis of each situation the vehicle encountered in simulation and a discussion of how the real vehicle performed.

### 6.1 Simulation Testing

Figure 6.1 shows the course used for simulator testing. The white areas are obstacles and the blue numbers show the waypoints, in order. The blue corridors were not used for anything other than laying out the course. The course was used to test the algorithm's reaction to various situations. This section explains each of those situations and how the vehicle handled them.



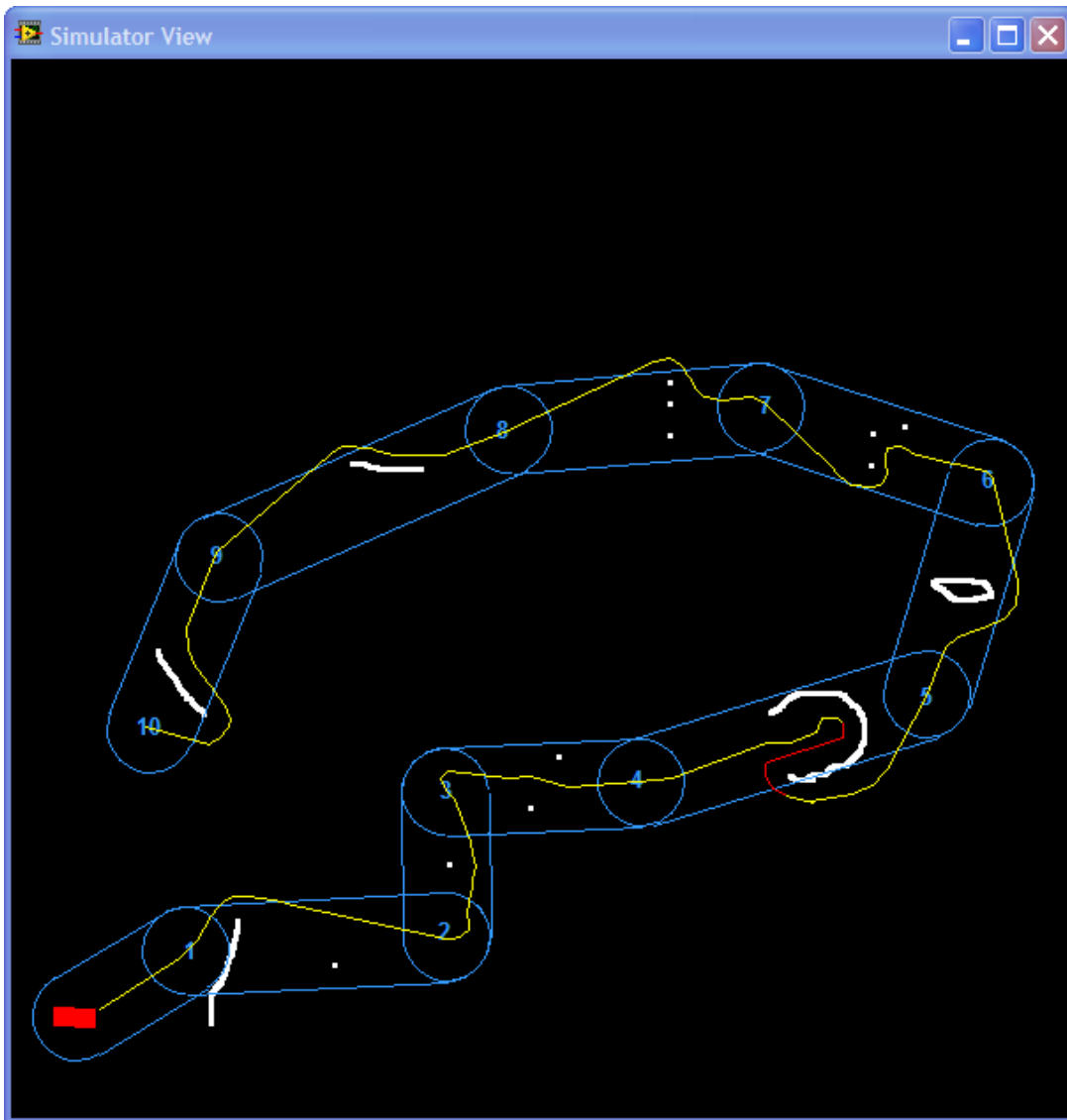
**Figure 6.1.** Simulator screenshot with the waypoints numbered.

The large obstacle between Waypoints 1 and 2 was used to test the capabilities of the rear buffer zones. Without the rear zones, the vehicle would turn into the obstacle once it was out of view of the LRF sensor. With the rear zones and obstacle remembering implemented, the vehicle would successfully drive around the obstacle, even when it left the view of the sensor. This situation was then used to tune the length and width of the rear buffer zones. Another issue that occurred at this section dealt with the waypoint threshold. If the waypoint threshold was less than a meter, the vehicle would be unable to achieve it due to the obstacle avoidance. It would then circle the obstacle and hit the waypoint the second time around.

The section between Waypoints 2 and 3 simulated the algorithm's reaction to a single point obstacle, such as a tree or telephone pole. The algorithm easily navigated this section. The next section proved to be difficult, especially with a wide Avoidance Zone. When the Avoidance zone was slightly narrower than the gap between the obstacles, the vehicle could successfully navigate between them. However, with a wide zone the vehicle would often oscillate back and forth. When it saw the right obstacle, it would try to avoid left, causing it to see the left obstacle and lose the right one. This would prompt it to avoid right, causing it to get stuck. The difficult situation handler would then set the vehicle to Emergency mode, requiring operator assistance. The vehicle was rarely able to navigate through the horseshoe obstacle between Waypoints 4 and 5. This was designed specifically for the purpose of testing the difficult situation handler. Like the previous section, the vehicle would oscillate back and forth until it was put into Emergency mode. Navigating around the obstacle between Waypoints 5 and 6 was trivial and did not pose a problem.

The next section was especially helpful in analyzing the effect of the Search Length. Because the center obstacle is significantly offset from the other two, a short Search Length would not pick it up. The vehicle would generally oscillate while slowly moving forward. When it saw the middle obstacle it could make a better decision and was usually able to navigate to the next waypoint. Because there were three obstacles in a line between Waypoints 7 and 8, this section posed no problem for the algorithm. The vehicle behaved exactly as expected in the final two sections. It would follow down the side of the obstacle. In the first one it would avoid right, and in the second it would avoid left. Figure 6.2 shows the typical path

of the vehicle using a narrow Avoidance Zone. The red portion of the path shows where the vehicle was manually controlled.



**Figure 6.2.** Simulator screenshot with overlaid vehicle path. The yellow line shows the vehicle under autonomous control, while the red line shows manual control.

This course provided a good simulation of the situations the vehicle may encounter in a real-world setting. While the simulated vehicle could not perfectly navigate every situation, it was able to recognize these situations as troublesome, and set the vehicle into the Emergency state, prompting operator assistance. Because in the real-world the Nemesis platform will have supervisory control by an operator, this is not a problem. The key concern is that it always prevents collisions. This is

of utmost importance, first for safety and also for protecting an expensive piece of equipment like the Nemesis vehicle. After hundreds of runs at several different speeds from 0.5 to 6 m/s, the vehicle never contacted an obstacle. However, at speeds above 4 m/s, it often became difficult to achieve waypoints and the vehicle would sometimes miss them for the sake of avoiding obstacles.

## 6.2 On-Vehicle Testing

The on-vehicle testing was performed in many stages throughout the project. The first test was performed at Virginia Tech's visit to ARA's NED. After the initial hardware and communications issues were solved, the algorithm guided Nemesis around a waypoint course containing obstacles such as cones and snowballs. Figure 6.3 shows Nemesis driving under the control of Virginia Tech's autonomous mobility algorithm. At this test the decision to add obstacle remembering, rear zones, and the difficult situation handler was proposed. After the initial visit, all of the on-vehicle testing was performed solely by ARA's engineers.



**Figure 6.3.** Nemesis under autonomous control

The main issues encountered in real-world testing dealt with vehicle dynamics, LRF and GPS sensor data, and communications. While the simulated vehicle performs the exact commands given to it, the actual vehicle cannot instantaneously execute commands. It is restricted to the performance of the actuators and its own vehicle dynamics. In addition, terrain plays a large factor in the vehicle's performance. A command carried out on asphalt behaves differently than the same command on gravel. For these reasons, the accelerations and heading rates used in the simulator needed to be fine tuned for use on the Nemesis platform. The final values used were slightly lower than in the simulator. This made the vehicle react slightly slower, but it was more consistent on a range of terrains. An acceleration of  $0.2 \text{ m/s}^2$  and a deceleration of  $0.5 \text{ m/s}^2$  were used. Also, the heading rate was set to  $0.8 \text{ rad/s}$ .

In the simulator, the LRF data is perfect. The simulated scanner is always horizontal to the ground and never gets confused by spurious data. However, in real-world testing, the vehicle tends to rock back and forth when accelerating, decelerating, or turning. This can cause the laser scanner to temporarily see and try to avoid the ground. Dust and other airborne particles can also give false readings. However, because a reactive navigation scheme was used that did not keep a world model, this was not a serious problem.

Like the LRF simulator data, the simulated GPS sensor represents ideal conditions, where there is no error or lag in the incoming data. However, true GPS data has limited accuracy of about 0.5 meters, and the incoming GPS messages sometimes lag due to high network traffic. This can cause the algorithm to think the vehicle is in one place when it actually may be 0.5 meters away. Again, this is not a serious problem, but it does affect the efficiency of the algorithm.

For these reasons, during real-world testing the maximum driving speed had to be reduced to  $4 \text{ m/s}$ . This allowed the algorithm to control the vehicle much more accurately. Table 6.1 gives the final values of all of the behavioral parameters used for the vehicle.

**Table 6.1.** Final Values for the Behavioral Parameters.

Variable	Parameter	Value	Units
Zone Sizes	Min Front Length	3.0	Meters
	Max Front Length	7.5	Meters
	Avoidance Zone Width	3.4	Meters
	A Buffer Width	1.0	Meter
	B Buffer Width	0.7	Meters
	C Buffer Width	0.5	Meters
	Rear Length	1.7	Meters
	Rear A Buffer Width	1.9	Meters
	Rear B Buffer Width	0.7	Meters
Avoidance Settings	B Heading Limit	0.32	Radians
	C Heading Limit	0.55	Radians
	Search Length	1.3	Meters
	Search Width	5.5	Meters
	Summation Threshold	1.0	Meter
	Front Remember Distance	1.0	Meter
	Rear Remember Distance	3.0	Meters
	Sensor X Offset	0.0	Meters
	Sensor Y Offset	1.0	Meter
	VLF	1.0	Meter
Velocity Parameters	Maximum Angle	.52	Radians
	Minimum Angle	.087	Radians
	Maximum Acceleration	0.2	m/s <sup>2</sup>
	Maximum Deceleration	0.5	m/s <sup>2</sup>
	Maximum Speed	4.0	m/s
	Heading Rate	0.8	Rad/s
Waypoint Threshold	N/A	1.0	Meter

# Chapter 7: Conclusions and Future Work

## 7.1 Conclusions

The autonomous mobility algorithm developed by Virginia Tech for ARA's Nemesis platform was able to successfully navigate the vehicle through a waypoint course while avoiding obstacles. The waypoint navigation algorithm was a simple point-to-point algorithm that had been used on several of Virginia Tech's autonomous vehicles. The Discretized Dynamic Expanding Zones with Memory obstacle avoidance approach was a reactive navigation algorithm modified from Virginia Tech's 2005 DARPA Grand Challenge DEZ obstacle avoidance algorithm. Because the initial Dynamic Expanding Zones algorithm was developed for higher speed (20+ mph), Ackermann steered vehicles it was unsuitable for the slow, differentially driven Nemesis tractor.

Rather than the three perception zones of the initial Dynamic Expanding Zones algorithm, the newly developed approach used 11 zones, with 7 in front of and 4 behind the front plane of the vehicle. Because of the zero radius turning capability of the tractor, a short-term memory of obstacles outside the sensor's view was also implemented. This was used in conjunction with the four rear buffer zones to prevent the vehicle from turning into unseen obstacles. The additional zones, coupled with the retained memory of past obstacles, provided the necessary perception to navigate the tractor through an obstacle field. In addition, the implementation of a difficult situation handler allowed the algorithm to determine when it could not navigate a complex set of obstacles and, thus, required operator assistance.

While the algorithm was developed with the Nemesis platform in mind, it was designed to be generic enough for use on any differentially driven vehicle. This was done through the use of multiple behavioral parameters that could be tuned to a vehicle's size, driving capabilities and sensor placement. It was developed in LabVIEW and tested using the simulator that Virginia Tech developed for their IGVC and DARPA Grand Challenge teams. It was implemented on the vehicle using a Windows laptop connected to the vehicle's network.

Because of this cross-programming language, cross-operating system setup, the Joint Architecture for Unmanned Systems (JAUS) was vital in reducing development time. Virginia Tech used their LabVIEW JAUS toolkit as the communication framework for the code. This toolkit provided all the necessary message sending, receiving, and handling. Since both organizations already supported JAUS, complete interoperability testing was possible over the internet before the first visit by Virginia Tech to ARA in January of 2007. The algorithm was first tested on the vehicle at this visit. Afterwards, all subsequent on-vehicle testing of the revised algorithm was performed by ARA's engineers.

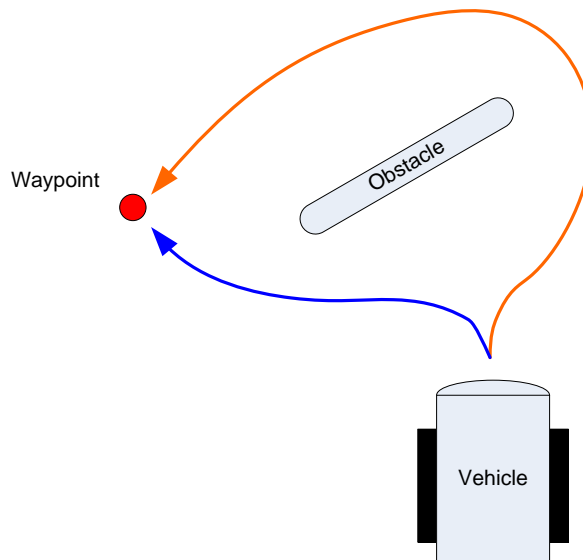
While the LabVIEW development and deployment of the algorithm demonstrated its ability to autonomously control the Nemesis platform. ARA was provided with detailed documentation, giving them the ability to integrate the algorithm directly into their own hardware using the programming language of their choice. This documentation precisely described every function, variable, and process of the algorithm.

The collaboration between Applied Research Associates and Virginia Tech provided a unique partnership between industry and academia. ARA was able to leverage Virginia Tech's award-winning experience with autonomous navigation to increase the capabilities of their Nemesis Humanitarian Demining platform. The added capabilities of waypoint navigation and obstacle avoidance developed by Virginia Tech and implemented on Nemesis will be used to reduce operator fatigue and increase the number of vehicles an operator can control, leading to increased efficiency and productivity in the humanitarian demining effort.

## **7.2 Future Work**

While the algorithm demonstrated the ability to autonomously control the Nemesis vehicle, some added intelligence could improve its performance. For example, there are some cases where the vehicle may avoid an obstacle in one direction (red path), where the desired waypoint is in the other direction (blue path), such as in Figure 7.1. In this case, a fuzzy logic controller could be implemented to provide better decision-making in terms of the avoidance decisions. A similar

approach is described in J. Putney's thesis from Virginia Tech [5]. This process could also be used to help eliminate some of the oscillatory behavior encountered when the algorithm becomes confused.



**Figure 7.1.** Obstacle avoidance with fuzzy logic. Diagram showing the path the vehicle will currently take to avoid an obstacle (shown in red), and a better approach to obstacle avoidance using fuzzy logic (shown in blue).

While ARA has expressed approval of the algorithm documentation, at the time of this thesis, they have not begun coding it in their chosen programming language. This will be the true test of the documentation. In order for this to be done, a software engineer must interpret and produce the software based on the specifications provided by the flowcharts and variables.

In addition to refining and documenting the algorithm, work must still be done to provide the ability for one operator to control multiple vehicles. For example, this capability must be integrated into an OCU. This will require a component that can keep track of multiple vehicle positions, statuses, and sensors. The OCU must also be able to take control of any one vehicle at a given time, either by teleoperation or assigning new waypoints. To increase demining efficiency, the entire landmine detection process must be automated as well. In the ideal situation, the operator could highlight a boundary area or upload a set of waypoints to the vehicle and send it a “go” command. The vehicle would then scan the area or waypoint path and alert the operator each time it found a landmine. The benefits of continuing to work towards this goal are immense. With more than 110 million

landmines in the ground, any added efficiency in the detection and removal process could save lives and reduce cost.

## References

1. Faruque, R., *A JAUS Toolkit for LabVIEW, and a Series of Implementation Case Studies with Recommendations to the SAE AS-4 Standards Committee*, Masters Thesis for Virginia Tech, Department of Mechanical Engineering, 2006.
2. Intelligent Ground Vehicle Competition Website. [www.igvc.org](http://www.igvc.org)
3. JAUS Working Group. JAUS Reference Architecture version 3.2. Available at [www.jauswg.org](http://www.jauswg.org).
4. Murphy, R., *Introduction to AI Robotics*. 2000: MIT Press.
5. Putney, J., *Reactive Navigation of an Autonomous Ground Vehicle using Dynamic Expanding Zones*, Masters Thesis for Virginia Tech, Department of Mechanical Engineering, 2006.
6. Reinholtz, C., et al., *Virginia Tech Grand Challenge Team*, for DARPA Grand Challenge; Primm, NV 2005.
7. United Nations Department of Humanitarian Affairs, *Land Mine Facts*. Available at <http://www.un.org/Depts/dha/mct/facts.htm>.
8. Wetzel, J., *Robots for Humanitarian Demining*. Unmanned Systems Magazine, May 2006.
9. Wicks, A., et al., *Virginia Tech Team Rocky*, for DARPA Grand Challenge; Primm, NV 2005.

## **Vita**

Grant Gothing was born in 1983 to Gilbert and Susan Gothing and raised in Hubbardston, Ma. He received dual high school diplomas in 2001, graduating from both Quabbin Regional High School in Barre, Ma. and the Massachusetts Academy of Math and Science at WPI, in Worcester, Ma. He attended Virginia Tech for his B.S in Mechanical Engineering and graduated in 2005. He remained at Virginia Tech for his M.S. in Mechanical Engineering, where the majority of his research involved unmanned and autonomous vehicles and systems under the advising of Dr. Charles Reinholtz. This research involved work with the Joint Architecture for Unmanned Systems (JAUS), Blind Driver Challenge, Autonomous 2004 Cadillac SRX, and the Nemesis subcontract. He will receive his Masters Degree in 2007.