Communications, Decision-Making, and Interactions of a Multi-Agent Autonomous Vehicle System

Daniel Davis Smith

Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Bioresource and Agricultural Engineering Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_dissertations/5648

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
COMMUNICATIONS, DECISION-MAKING, AND INTERACTIONS OF A MULTI-AGENT AUTONOMOUS VEHICLE SYSTEM

A Dissertation

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

The Department of Biological & Agricultural Engineering

by
Daniel Davis Smith
B.S, Louisiana State University, 1992
M.S, Louisiana State University, 2011
December 2021
ACKNOWLEDGMENTS

I am thankful for the support, encouragement, and prayers of many. To my parents and family, I thank you for your ongoing encouragement and support. Thanks to Andy Venuto for specific assistance with GIS modelling and mapping. To Steve Hall, thanks for your advice and encouragement over a long period and for both technical and philosophical discussions related to this work and beyond. I am thankful for a supportive committee including Dr. David Constant, Dr. Chandra Theegala and Dr. Steven Hall who each served as major advisor at one point or another; and to Dr. Dana Browne and Dr. Richard Bengtson for their committee work, advice and encouragement I am thankful.

To the faculty, staff and students of LSU BAE who over the years assisted with construction, contributed in senior design or other areas; provided fellowship and accountability I am thankful. I particularly appreciate Ms. Angela Singleton who helped with various graduate school issues; Mr. Tom McClure who provided assistance in the shop and Dr. Kevin Holly who assisted with illustrations of the gradient tracking challenges. First and last, I thank God who makes this all possible and meaningful.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ................................................................................................................................. ii

LIST OF TABLES ................................................................................................................................................ v

LIST OF FIGURES ........................................................................................................................................... vi

ABSTRACT ......................................................................................................................................................... viii

CHAPTER 1. INTRODUCTION ........................................................................................................................... 1

CHAPTER 2. LITERATURE REVIEW ...................................................................................................................... 5
  2.1. Background ............................................................................................................................................... 5
  2.2. Single Vehicle ......................................................................................................................................... 7
  2.3. Multiple Heterogeneous Vehicle Systems ............................................................................................... 9
  2.4. Multiple Homogeneous Vehicle Systems .............................................................................................. 10
  2.5. Conclusions ........................................................................................................................................... 12

CHAPTER 3. PHYSICAL SYSTEMS AND HYBRID SIMULATION ........................................................................ 13
  3.1. Introduction ........................................................................................................................................... 13
  3.2. Physical Hardware ................................................................................................................................. 13
  3.3. Electronics Redesign .......................................................................................................................... 14
  3.4. Hybrid Simulation ............................................................................................................................... 16
  3.5. Python™ Simulation ............................................................................................................................. 20

CHAPTER 4. GUIDANCE AND INTERACTION OF A MULTI-AGENT SYSTEM WITH MINIMAL COMMUNICATIONS ................................................................................................................................. 23
  4.1. Introduction ........................................................................................................................................... 23
  4.2. Materials and Methods ....................................................................................................................... 23
  4.3. Results .................................................................................................................................................. 30
  4.4. Data Analysis ....................................................................................................................................... 33
  4.5. Discussion ........................................................................................................................................... 38
  4.6. Conclusions ......................................................................................................................................... 39

CHAPTER 5. ADDITIONAL ALGORITHM DESIGNS AND FUTURE WORK .......................................................... 41
  5.1. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Sequentially Following a Path ................................................................................................................................................................................................. 41
  5.2. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Parallel Straight-Line Following ......................................................................................................................................................................................... 46
  5.3. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Parallel Straight-Line Path Following With 180 Degree Turns ........................................................................................................................................................................... 50
  5.4. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Gradient Tracking with a Fixed Leader ........................................................................................................................................................................... 54
  5.5. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Gradient Tracking with a Changeable Leader ................................................................................................................................................................................................. 60
LIST OF TABLES

1. Boundary Latitudes and Longitudes .................................................................................................................. 25
2. Starting coordinates for ASVs ............................................................................................................................ 26
3. GPS Log Data .................................................................................................................................................... 30
4. Boundary Check Data ......................................................................................................................................... 30
5. Test Run and Summary Data ............................................................................................................................. 32
6. Summary statistics for Area Patrol tests .......................................................................................................... 38
# LIST OF FIGURES

1. Original LSU ASV ................................................................. 6
2. Physical ASV ........................................................................ 14
3. Block diagram of electronics redesign .............................................. 15
4. Hybrid Simulation Block Diagram .................................................. 17
5. Hybrid Simulation Physical Hardware .............................................. 18
6. NMEA GPRMC Example Sentence ................................................. 18
7. ASV Radio Message Structure ......................................................... 19
8. Python code snippet showing initializations ..................................... 20
9. Python simulation main() program setup ......................................... 21
10. Map showing virtual boundaries and base station ................................ 25
11. Boundary edge calculations .......................................................... 28
12. ASV code to check for simulated collision events ............................. 29
13. Example output showing paths created from GPS log points .................. 31
14. Closeup of out-of-boundary event .................................................. 33
15. Area Patrol Test, Run 1 ................................................................ 35
16. Area Patrol Test, Run 2 ................................................................ 36
17. Area Patrol Test, Run 3 ................................................................ 37
18. Example track with GPS waypoints ................................................ 42
19. Examples of Parallel Navigation ..................................................... 47
20. Example of Error Measurements .................................................... 49
21. Parallel Navigation through 180 degree turns .................................... 51
22. Gradient Tracking Test Setup ........................................................ 55
23. Function to calculate simulated sensor value ..................................... 56
24. Code snippet to determine lead ASV ............................................... 57
25. Arduino® code to select lead ASV ........................................................................................................... 61
ABSTRACT

Autonomous vehicles are becoming ever more common and offer many attractive benefits to society. They can operate for long periods of time unattended, operate in environments that may be dangerous to humans, perform time consuming or repetitive tasks and all with greater efficiency and lower costs than humans. For these vehicles to be able to do these things, algorithms need to be designed and optimized that allow them to interact with the real-world environment in safe, effective, and efficient ways.

We designed and built a set of three homogeneous water-based autonomous surface vehicles equipped with appropriate sensors and communications ability along with algorithms designed to allow these vehicles to perform various cooperative tasks using data obtained from the vehicles’ sensors and data shared between the vehicles. These vehicles were designed to be modular, economical, and, where possible, were constructed using off-the-shelf technology with programming designed to take advantage of these systems. When the COVID-19 pandemic put an end to lab and field work the physical vehicles were stored but the research continued utilizing a hybrid hardware-software simulation of the system. Three microcontrollers identical to the devices controlling the physical boats were attached via a Universal Serial Bus (USB) hub to a desktop computer running a simulated environment written in Python™. The three vehicles (microcontrollers) were given tasks including patrolling adjoining areas of the water body delineated by latitude and longitude boundaries while staying within their own boundary and avoiding collisions with the other vehicles. Initial testing was successful with the algorithm able to maintain the vehicles within their boundary >=95% of the time with no collisions. Additional problem types including parallel travel; wind and current challenges; and gradient tracking and relevant algorithms are discussed.
CHAPTER 1. INTRODUCTION

The goal of this project is to study the design of algorithms for use in a real-world homogeneous multi-agent autonomous vehicle system that operates on the surface of water bodies. Autonomous vehicles have existed for many years but were, for a considerable period, limited to laboratory conditions. When they eventually migrated from the lab to the field, they were mostly land-based wheeled or tracked vehicles. It has only been relatively recently in the development of autonomous vehicles that they have taken to water bodies and to the air. Both environments present unique challenges that are not found in land-based vehicles. In addition, for much of the history of autonomous vehicle experimentation, the research was limited to single vehicles, and most were expensive. With the advent of more powerful small computing, communication, GPS, and other necessary systems, smaller, less costly vehicles began to be developed for both laboratory and field use.

The autonomous surface vehicles (ASVs) created for use in this project consist of a set of three identically constructed water surface vehicles. The design is based on single and multi-vehicle systems used for research in the Department of Biological & Agricultural Engineering at Louisiana State University since 2000. The current vehicle system is the result of the research done during a master’s degree with updated electronics and software. The basic vehicle consists of a dual-pontoon surface vehicle with an aluminum frame that provides the attachment points for the pontoons, the electronics enclosure, support for the solar panels, and support for the radio and GPS antennae. Propulsion is provided by a dual-paddlewheel design that uses two 12-volt DC motors with encoders. Power for the system is provided by two sealed lead acid batteries housed within the electronics enclosure. The batteries are recharged from the two 15-Watt solar
panels mounted atop the vehicle. In addition to the batteries, the electronics enclosure houses three microcomputers – an Arduino® Micro, an Arduino® Mega 2560 embedded controller and a single-board Linux® computer in the form of a BeagleBone® Black. Also contained in the electronics enclosure are the motor controllers, and various other subsystems.

This project seeks to explore algorithm design and the challenges of dealing with the multiple sources of error, incomplete information, and uncontrollable environmental factors that arise in operating a fleet of homogeneous autonomous surface vehicles in a real-world environment. Errors can arise from mechanical and electrical differences in components that should be identical, but they also arise from the inherent limitations of the sensors in use. As an example, the GPS units used in the vehicles constructed at LSU have an accuracy rating of 3-5 meters. Given that the vehicles are approximately 1 meter by 1 meter, the GPS error alone means that a vehicle can be several body widths or lengths from where the control system thinks it is. The control system also has limitations that prevent it from instantaneously knowing everything about the vehicle. Some inputs may be sampled at certain intervals either because of the amount of time it takes for the input to be ready or because of the necessity of the control system to be sampling other inputs. This can result in incomplete information when a control decision needs to be made resulting in the need for the control system to be able to make good estimations based on historical information. The final challenges are the environmental factors over which the vehicle’s control system has no control. These include wind, water flow, sunlight, etc. and the control system can only react to the effect those are having on the vehicle. This necessitates being able to determine what the effect is and attempting to counteract that effect through the use of the vehicle systems.
In order to complete this research after the restrictions of the COVID-19 pandemic were put into place, the physical devices were shelved, and a simulation was developed as a hybrid of the microcontrollers from the physical boats combined with a simulation running on a desktop computer written in the Python™ programming language. This research will attempt to demonstrate that effective cooperative navigation algorithms can be developed for a fleet of autonomous surface vehicles that utilize readily available off-the-shelf components. With such algorithms, these devices become modular, reliable, and significant tools for the researcher.

An algorithm was developed and tested to maintain the fleet of ASVs within a set of virtual boundaries. Each vehicle was assigned an area to patrol designated by virtual boundaries defined by lines of latitude and longitude. To ensure that each vehicle remains in its own area and does not collide with a vehicle from a neighboring area, GPS coordinates are taken at 1 second intervals and are stored. These coordinates are also transmitted to all of the other vehicles. This allows each vehicle to compute the distances to its own boundaries and to any other vehicle. We define safety areas for the boundaries and for individual vehicles. Incursions into those safety areas are defined as collisions for the purposes of our test. Algorithms are tested to ensure that collisions are minimized. The algorithm was successful both in keeping the individual vehicles within their assigned boundaries and in keeping the simulated vehicles from colliding with each other.

Other algorithms were developed and preliminary testing was begun. One of these, an algorithm to have the three ASVs sequentially follow one another through a list of GPS waypoints, was simulated and a sample output map was generated. In the follow-the-leader algorithm, a vehicle is chosen as the lead vehicle and is given a set of GPS waypoints to follow. As the lead vehicle passes each waypoint and turns towards the next waypoint, it transmits the
waypoint to the remaining vehicles. Each of the remaining vehicles builds an in-memory table of waypoints and, when it is allowed to start the course, seeks to follow the same path as the lead vehicle. The algorithm is evaluated based on the amount of error between the ideal path, the path predicted by the algorithm, and the GPS track of the vehicle through the course.

In addition, an algorithm to handle gradient tracking on a water body was written and testing was begun and a second version of the gradient tracking algorithm was discussed. In the first case, the vehicles proceed in a random walk over the surface of a water body seeking a contaminant. The first vehicle to detect a significant amount of the contaminant is elected the ‘leader’ and directs the remaining vehicles as the group attempts to locate the source of the contaminant. The second case is similar to the first with the variation that each time any of the vehicles detects a higher concentration of the contaminant, it then becomes the leader.

Other potential algorithms are discussed with possible applications including two algorithms for parallel navigation. In the first of these, three vehicles are, starting from a line with equal spacing between them, given the task of navigating in a straight line across a body of water while maintain their spacing in both planar dimensions. The task in the second version of the algorithm becomes more difficult as the three vehicles, started in the same fashion as before, are required to navigate through arcs while maintaining their spacing and speed relative to each other.

While significant work has been done in the field in the last few years, these tasks have unique challenges not yet fully addressed in previous studies. We start with a review of current research in various arenas related to autonomous vehicles and discuss how this research project differs and what it brings to this field.
CHAPTER 2. LITERATURE REVIEW

2.1. Background

In order to understand more completely the state of current water-based vehicle research and various studies on coordination of multi-vehicle systems, a review of recent work in the area was completed. Research in autonomous vehicles has experienced significant growth in the last two decades as the size and cost of control system components has decreased while, at the same time, the computational complexity and robustness of those same components has increased.

Land-based autonomous vehicles have been the subject of intense research for a number of years. Some prime examples of the level of research endeavors in that subfield are the Defense Advanced Research Projects Agency’s (DARPA) Grand Challenges in 2004 and 2005 (Seetharam et al., 2006) and the follow-on challenge, the Urban Challenge) in 2007. Military research continues to be a driving force for all areas of autonomous robotics.

Agricultural applications of autonomous vehicles have been explored in research, but as was true of other application areas, were largely land-based vehicle systems (Kim et al., 2000; Lindgren et al., 2002; Noguchi et al., 1999; Tang et al., 2000; Wang et al., 2004; Jeon et al., 2009). Aerial and water-based (whether surface, submarine, or amphibious) autonomous robots have become the focus of more and more research efforts as computational speed has increased and the size and mass of components have decreased. Research into water-based autonomous vehicles, in particular, has become a fast-growing field with application to environmental monitoring, aquaculture, biomass collection, and other areas.
Research at LSU into single vehicle autonomous surface vehicles began in 2001 with the construction (Figure 1) of the first of a number of ASVs (Hall, et al., 2001; Price and Hall, 2002, Hall et al. 2006). The initial research interest was the reduction of bird predation on aquaculture ponds using a non-lethal, environmentally friendly, and low-noise solution. Early vehicles followed a random path across a pond or other water-body and frightened birds away by the movement of the vehicle. The only sensors available on the earliest vehicle were physical contact sensors. Later versions would add additional sensors such as GPS and ambient light to enable more complex behaviors and more actuators to enhance the bird predation reduction mission. Additional research was done in the area of reducing bird predation on aquaculture ponds (Hall et al., 2009; Price and Hall, 2011), vision systems using neural networks (Nadimpalli et al., 2006; Hall et al., 2007b), scaring birds off of water reservoirs to improve drinking water quality (Hall et al., 2007a), and with environmental sensors for taking water quality measurements (Hall et al., 2005). More recent projects using ASVs have included using autonomous surface vehicles for biomass collection (Taylor et al., 2014) and measuring sediment accretion and erosion using an autonomous surface vehicle (Smith et al., 2014). A fleet of ASVs was constructed in 2010 for researching cooperative algorithm designs and applications of a fleet of cooperative ASVs (Smith and Hall, 2013; Hall et al., 2011; Smith 2011). These vehicles are modular in design allowing quick field repairs, sensor package changes, and simple

![Figure 1. Original LSU ASV](image-url)
modifications to the vehicle. The pontoons are made of closed-cell foam that is coated and painted for physical protection. The frames of the vehicles are composed of welded, square aluminum tubing and are secured to the pontoons using sheet metal screws through an aluminum plate which has been glued to the top of the pontoon. The frame supports both the solar panels and the electronics enclosure. The vehicles are propelled by two 12-volt brushed DC motors that are mounted to the aluminum plate that is on the pontoons.

This literature review focuses on the advances made in algorithm design, optimization, and implementation for both single and multiple autonomous surface vehicle (ASV) systems in recent years specifically with regard to guidance and navigation. Multiple vehicle systems are further divided into heterogeneous and homogeneous systems. The research in this project focuses on a multiple homogeneous vehicle system consisting of vehicles operating on the surface of a water body. Evident from the most recent publications, most research is still being done on single vehicle systems although multi-vehicle systems are becoming more common and the possibility of using the research done on single vehicles for multi-vehicle systems is often mentioned in those papers.

2.2. Single Vehicle

There are a number of research efforts involving various aspects of algorithm design applied to single ASVs. Some of the ideas proposed in these single vehicle systems could potentially be applied to multi-vehicle systems. Guidance and navigation are fundamental to the correct functioning of a vehicle in a real-world environment and are necessary for any more advanced operational tasks like sample collecting, object tracking, data collecting, etc.

A significant amount of research into single ASV control systems involves collision avoidance. Because ASV operations in a heterogeneous environment containing some
combination of autonomous vehicles; commercial, civilian, and military water-based vehicles; and various forms of wildlife are becoming more common. ASVs must be able to maneuver in such a way as to minimize the possibility of collisions. Vehicles operating in these environments need to be able to comply with the maritime regulations promulgated by the Convention on the International Regulations for Preventing Collisions at Sea, 1972 (COLREGs). To that end, a number of solutions have been proposed.

One system that is currently in use for manned vehicles is the Automatic Identification System (AIS). AIS equipment typically broadcasts a vessel’s unique identifier, position, course, and speed. That information can then be used to supplement visual, radar and other forms of information to prevent collisions between manned vessels. A method was developed using archived AIS data to model and quantify the potential for collisions of an unmanned vehicle operating in a crowded shipping environment (Filimon and Codiga, 2016). A major shortcoming of using only the AIS for vehicle identification and collision avoidance is that many smaller vessels are not equipped with the system. A second potential problem could result from a hardware failure in which the autonomous vehicle fails to receive the transmitted AIS data. A number of research projects exist that are specifically focused on meeting the COLREGs requirements. One system using a velocity obstacle method as its basis for collision avoidance was created and implemented using the Robot Operating System (ROS) and then tested in simulation for several COLREGs scenarios (Stenersen, 2015). A second system uses monocular vision to estimate tracking data on visible objects and compute safe navigational paths (Park et al., 2015) while a third system uses an adaptive search heuristic on a generated space state to simulate choosing the best trajectory available to it given the initial conditions and the parameters of the simulation (Shah et al., 2015). Guo et al., 2020, modelled a system utilizing a
deep reinforcement learning algorithm to improve autonomous ship path planning that achieves
good results while maintaining COLREG compliance focusing on safety over mission objectives.
An algorithm using a two-layer hierarchical framework to both handle path planning with
collision avoidance while still optimizing the intermediate points along the path was modelled
for a single USV (Yao et al., 2020). Because control systems for USVs can become trapped in
local optimums due to only being able to observe a portion of their environment, Yan et al.,
2021, proposed an algorithm which incorporated a memory-like reward function into their
learning model to counteract this bias.

A control system was developed using the Robot Operating System (ROS) to coordinate
the activities of an autonomous surface vehicle with those of a tethered underwater vehicle
(Conte et al., 2015). The system implements a navigation guidance control (NGC) for the
vehicle system that allows the surface vehicle to navigate to a location and deploy, direct, and
recover the underwater vehicle. Since the system is designed on top of the ROS, interaction with
other ROS-based systems is possible although that scenario was not included in the testing.

2.3. Multiple Heterogeneous Vehicle Systems

Much of the research in the area of multiple heterogeneous vehicle systems focuses on
combinations of surface vehicles cooperating with underwater vehicles. Some systems
incorporate aerial vehicles as well. In most of these, there are not multiples of any particular
vehicle types.

One system that was tested after the Deepwater Horizon spill was developed using one
autonomous surface vessel, one vessel that could function as either a surface vehicle or an
underwater vehicle, an underwater vehicle for bathymetry mapping, and a remotely operated
vehicle (ROV) (Mukhopadhyay et al., 2014). Each vehicle had internal systems, but the coordination was done from a shore-based computer running LabView.

Glotzbach et al., 2015, developed a system for cooperative line-of-sight target tracking for a multi-vehicle heterogeneous system consisting of a combination of multiple surface vehicles and an underwater vehicle. The system is designed to retrieve data from tags attached to fish by maneuvering the underwater vehicle close enough to the fish to download the data. In order to control the underwater vehicle, surface vehicles are used to provide the radio link to an acoustic modem on the underwater vehicle. The algorithms developed allow the surface vehicles to track the underwater vehicle using the pings from the acoustic modem and to follow in a formation that keeps at least one of the surface vehicles in range of the underwater vehicle at all times.

Another unique system involves two autonomous vehicles used to aid in an emergency ship towing maneuver (Bruzzone et al., 2016). Getting lines from a marine vessel in distress to a towing vessel is a very dangerous task. By employing two autonomous vehicles towing lines, one launched from the distressed vehicle and one from the towing vehicle, an algorithm enabling the two vehicles to perform a knotting maneuver results in the recovery of the distressed vessel’s line by the towing vehicle without putting human lives in danger.

2.4. Multiple Homogeneous Vehicle Systems

Some research has focused on algorithms for multiple vehicle systems wherein all the vehicles are identical copies. This is a special case of heterogeneous multiple vehicles systems, but many of the ideas explored for homogeneous systems of multiple vehicles are equally applicable to systems consisting of multiple heterogeneous vehicles.
One area of research in multiple vehicle systems is that of coverage control in which the goal is to create an optimal coverage of a spatial area with multiple vehicles. One such technique that used Gaussian estimation along with an adaptive backstepping technique was developed and modeled in simulation for use with two vehicles in a flowing environment (Zuo et al., 2015). Another system developed by Xiong et al., 2019 used both Veronoi partitioning and ant colony optimization to perform path planning for multiple autonomous marine vehicles for ocean sampling operations.

Mimicking the behavior patterns of living systems as a control method is one area that is receiving a significant focus for research. A 2014 study looked at swarming behavior as applied to path-following for a system of multiple homogeneous vehicles (Bibuli et al., 2014). The algorithm focused on controlling the entire formation rather than controlling individual robots and used swarming to follow the path while using formation control to maintain the overall robot formation. It was designed and tested in simulation to evaluate its performance.

A system consisting of catamaran-type vehicles similar to those in use at LSU has been used for greenhouse gas sampling (specifically methane) on a lake (Dunbabin, 2016). The system consists of two vehicles which perform a semi-random walk over the lake taking samples. The two vehicles share their sample locations with each other to enable a complete map to be created. The system creates a Gaussian potential map of all of the previous sample locations in order to make sure that the next sample point is not ‘too near’ any of the previous points. If a random point chosen is not on land and is below a threshold value in ‘nearness’ to previous sample points, it becomes the next sample point. The vehicle then moves toward that point while doing simple obstacle avoidance using ultrasonic, mechanical, and vision systems.
A 2014 paper (Wang et al., 2014) proposes using a combination of a neural network based dynamic surface control and a distributed estimator to allow marine surface vehicles with input saturation to cooperatively follow a path. Each vehicle in the modeled and simulated system seeks to minimize some geometric error and by sharing path variables, achieves coordination with the other vehicles.

At Louisiana State University, research has been ongoing using a system of homogeneous vehicles since 2010 (Smith, 2011). Algorithm design and the potential difficulties facing a system of multiple homogeneous vehicles in real world environments for a set of three vehicles operating together was discussed in a 2015 paper (Smith and Hall, 2015). Some of the possible algorithms discussed are the basis of experiments found in this project.

2.5. Conclusions

While much research has been and continues to be done on autonomous surface vehicles whether for single or multiple vehicle systems, it is often of a more theoretical nature. The theory of these systems is critical for a complete understanding but applying those theoretical models in a real-world environment can lead to unpredicted results. More research is needed in creating algorithms that take into account the challenges of operating in nature with all of its unpredictability. The experiments documented herein seek to combine the control theory with a real system to produce novel results in the area of autonomous control of a system of homogeneous surface vehicles that allow for their expanded use in research. A fleet of these vehicles can be a useful tool for a number of applications in aquaculture, environmental monitoring, coastal monitoring, and other areas.
CHAPTER 3. PHYSICAL SYSTEMS AND HYBRID SIMULATION

3.1. Introduction

The systems utilized for this research building on the previous work utilizing physical autonomous surface vehicles at Louisiana State University (LSU), a redesign of both the physical hardware and electronic control systems had been completed as part of the M.S. degree and, because of technological advances, another redesign of the electronic control systems of the fleet of ASVs was undertaken at the outset of the Ph.D. work. The design of both the physical hardware and the electronics of the ASVs was focused on creating a set of vehicles with a high value-to-cost ratio using primarily off-the shelf components that are easily repairable through their modular design. With the advent of the COVID-19 pandemic and the limitation on field and lab work, the research shifted towards simulating the systems. In an effort to keep the simulation as close to the original physical hardware as possible, a hybrid simulation was developed.

3.2. Physical Hardware

The physical ASV hardware (Figure 2) is a dual pontoon surface vehicle with pontoons made of shaped closed-cell foam insulation coated with a reinforcing coating that is then attached with adhesive to a rectangular piece of sheet aluminum that is drilled to accept machine screws used to attach the pontoons to the frame. The frame of the ASV consists of two supports made from welded half-inch square aluminum tubing that is attached to the pontoons and to the solar panels. These pieces comprise the structure of the device. Mounted beneath the solar panels is the electronics container holding the batteries, microcontrollers, single-board computer, XBee® radio transceiver, datalogger, and motor controllers. The motors are attached to the top plate of each pontoon. See Appendix A for CAD drawings for the physical components.
3.3. Electronics Redesign

The electronics systems were updated to reflect newer, more powerful microcontrollers and to reflect the additional control options that were desired in the system. This entailed replacing both the existing microcontroller and single-board computer along with adding an additional microcontroller dedicated to the control of the two paddlewheel motors (Figure 3).

The BASIC Atom microcontroller was replaced with an Arduino® Mega 2560 (Mega). There were several reasons for this change. In addition to increased program storage space and temporary working memory, the Mega has four hardware serial ports which enabled better support for the various sensors and actuators needed for the ASVs. It also has more general-purpose input/output pins than the BASIC Atom microcontroller to allow for future expansion.
The original Linux® single-board computer was replaced with a Beaglebone® Black. The Beaglebone® Black is a less costly alternative that also boasts a larger community base with a large number of software libraries including the Robot Operating System (ROS) which may be an avenue for future work utilizing these devices. There are also a number of third-parties who have created plug-in shields (called ‘capes’ in the Beaglebone® hardware ecosystem) to support many different sensors and actuators.

In addition to replacing the two original computers in the control system, an Arduino® Micro was added to control the two motor controllers controlling the paddlewheel motors. In addition to issuing commands to the motors, it also monitors the encoders that are attached to the motors to detect speed differences between the motors. The Arduino® Micro is connected to the Arduino® Mega via the on-board Serial Peripheral Interface (SPI).

Figure 3. Block diagram of electronics redesign
During the electronics redesign, consideration was given to newer GPS modules to replace the existing Garmin® HVS-16 modules utilized in the previous design. A test was conducted using a survey benchmark to review the accuracy of GPS data. Both the existing Garmin® unit and an Adafruit® Ultimate GPS Breakout v.3 were tested. The experimental apparatus consisted of the two GPS units being tested each connected to a different Arduino® Mega 2560 microcontrollers which were each connected to a microSD logging module with the entire apparatus powered by a 12 VDC battery. The apparatus was placed over the known survey benchmark (30.451763, -91.185061) and left to collect data at three second intervals for a period of ten minutes. The points from both GPS units were then mapped in ArcGIS® and compared to the coordinates of the benchmark. The Garmin® GPS unit was found to be more accurate with an average distance of the received coordinate to the actual benchmark for a sample size (n = 99) of 78cm while the Adafruit® GPS yielded an average distance for a sample size (n = 1075) of 14.1 meters. The Garmin® GPS was chosen for the electronics redesign.

3.4. Hybrid Simulation

To continue the research after the COVID pandemic arose, a decision was made to simulate the ASVs in operation. A hybrid simulation environment was developed utilizing three Arduino® Mega 2560 microcontrollers running slightly modified versions of the original ASV algorithms connected via a USB hub to a Windows® PC running a simulated environment written in the Python™ programming language (Figures 4 & 5).
The modifications to the ASV code consist of rerouting all inputs and outputs through the USB port (the hardware Serial 1 port on the Arduino®). The rerouted inputs include the GPS signal, the messages from the XBee® radio, and any sensor data while the outputs are the outgoing XBee® radio messages, data going to the datalogger, and movement commands to the motor subsystem.
The simulated GPS data comes from an algorithm running in the simulation that generates GPS coordinates based on an assigned starting location of each ASV, the movement commands issued by the ASV, and wind effects (if any). The algorithms for the ASVs were designed to utilize the National Marine Electronics Association (NMEA) GPRMC sentence format (Figure 6).

For the purposes of testing, all of the GPS strings contain the same date and all of the tests start at 7:00 am UTC. This makes generating the simulated GPS strings simpler without affecting the results. In addition, the simulation and the ASV algorithms ignore the magnetic variation, its east/west indicator, and the checksum. The simulated area corresponds to an area at
the south end of University Lake on the LSU Campus so that the simulated test runs can be overlayed on real-world satellite imagery.

The incoming and outgoing XBee® messages use the format developed during the master’s degree work with a sentence structure similar to the standard GPS NMEA strings but containing, when necessary, information in addition to the GPS data that is also encoded in the radio messages as a Log Note that is appended to the sentence (Figure 7).

![ASV Radio Message Structure](image)

**Figure 7.** ASV Radio Message Structure

Radio messages are not generated by the simulation but the simulation acts as the radio transmitter/receiver for both ends of the transmission by relaying the messages between the various Arduinos® that are part of the hybrid simulation environment.

The simulation simulates any sensor readings that are required for testing the ASV algorithms and provides them to the ASV algorithms via a special $SENSOR sentence that just consists of the sentence keyword $SENSOR, and the sensor reading for the current simulated location of the ASV. Lastly, the simulation receives the data that would, in the actual ASVs, be sent to a local datalogging device that would write to a microSD card and instead writes the data to a text file labeled with the name of the test run with the ASV number appended and stores the data on the PC’s hard drive for later analysis.
3.5. Python™ Simulation

The simulation consists of a set of commands used to set up the Python™ environment, a main program loop, and a number of other functions that taken together create the simulated environment. Although much of the program remains the same for all of the various experiments, the setup routine and main program loops are customized for each experiment to reflect the needed information exchanges. Complete Python™ code is listed in Appendix C.

The setup does three primary things. It imports necessary Python™ libraries, it initializes the three USB ports via the serial COM port number assigned to them when the USB hub connects them and sets up any needed global variables. The serial initialization causes a reset of the attached Arduino® Megas that forces them to restart and run their own internal setup routines.

```python
import serial
import time
import random
import math

# Define the serial port and baud rate.
# Ensure the 'COM#' corresponds to what was seen in the Windows Device Manager
ser1 = serial.Serial('COM3', 9600, )    # Boat 0 = Serial 1
ser2 = serial.Serial('COM5', 9600, )    # Boat 1 = Serial 2
ser3 = serial.Serial('COM6', 9600, )    # Boat 2 = Serial 3

# Global Variables

# Set Global time variables for the test run. All runs start at 070000.
tHour=7
bMinute=0
tSeconds=0

# Global ASV variables - Three sets for each of three test runs. Uncomment a set
asvLats = [30.408100,30.408100,30.408100]  # Start each ASV at the lower left corner of the box
asvLongs = [-91.168600,-91.167100,-91.165600]
asvBrngs = [0.00,0.00,0.00]
asvSpds = [0.00,0.00,0.00]
asvSense = [0.00,0.00,0.00]

# Point Source Location for Gradient Tracking Test
ptSrcLat = 30.408800
ptSrcLon = -91.165688

# Constants for calculations
EarthRadius = 6371      # Radius of earth in km
TurnSpeed = 30          # Time in seconds for an ASV to rotate through 360 degrees.
```

*Figure 8.* Python code snippet showing initializations
before entering the main experimental algorithm. In the code snippet in Figure 8, the simulation imports the pySerial, time, random, and math libraries, initializes three serial ports (ser1, ser2, and ser3), and sets a number of variables needed for the gradient tracking experiment.

When the main program starts, it prompts the user on the console for a prefix for the four filenames that the simulation creates for each test run (Figure 9). There is a general log file which just gets a ‘.txt’ extension, and a separate file for each of the ASVs named with the prefix and a suffix indicating which ASV the log file is for. The simulation also prompts for a wind speed and a source direction.

```python
def main():
    # Get the log file prefix and create the names for all four logfiles.
    templfName = input("Enter prefix for log files: (Suggested YYYYMMDD_Test##_Run##)"
    lfname = templfName + ".txt"
    lfname0 = templfName + ".Boat0.txt"
    lfname1 = templfName + ".Boat1.txt"
    lfname2 = templfName + ".Boat2.txt"

    # Get the wind speed if any
    wndSpd = float(input("Enter the wind speed in knots (Format x.xx):"))
    wndDir = float(input("Enter the wind source direction in degrees with 0.00 being due North and proceeding clockwise:"))

    # Set up the simulation
    print("Setting up...")

    # Blank the response string
    response = ""

    # Create and blank asvLog strings
    asvLog0 = ""
    asvLog1 = ""
    asvLog2 = ""

    # Set start time
    tHours=7
    tMinutes=0
    tSeconds=0

    # Randomize the starting angles for the ASVs
    for i in range(0,3):
        asvBrngs[i] = random.randrange(360)

    # Build the Initial GPS strings and send them
    for i in range(0,3):
        gpsStr = buildGPS(tHours,tMinutes,tSeconds,asvLats[i],asvLongs[i],asvBrngs[i],asvSpds[i])
        bgpsStr = bytes(gpsStr, 'utf-8')
        print("ASV ",i," starting GPS = ",end='')
        print(bgpsStr.decode('utf-8'))  # Print to console
        writeLogFile(lfname,gpsStr + 
        "n")   #Write it in the overall log file
        # Send the initial GPS string to the ASV
        asvSerW(i,bgpsStr)

    # Loop through to run the simulation for 15 minutes - each iteration is 1 simulated second
    for i in range(0,900):
        # Code snippet...
```

Figure 9. Python simulation main() program setup
speed and direction to utilize in the calculation of the next GPS point. Prior to entering the main experimental simulation loop, the simulation generates the initial GPS location for each ASV and sends the GPRMC string to each ASV. The main program loops for a set number of simulated ‘seconds’.

The simulation uses a number of different functions based on the haversine formula (to calculate bearings given two points and calculate the next point given a starting point, bearing, and speed. The haversine formula models the earth as a perfect sphere and uses spherical trigonometric equations to calculate the length of the arc on the surface of the sphere between two coordinates. This arc length is also known as the great-circle distance between the two coordinates. Other functions calculate the sensor value given an ASV location and a pollution source location (for the gradient tracking) and functions to parse the motor commands from the ASVs, build the various strings to be sent from the simulation to the ASVs, and write the log files.
CHAPTER 4. GUIDANCE AND INTERACTION OF A MULTI-AGENT SYSTEM WITH MINIMAL COMMUNICATIONS

4.1. Introduction

A simple application for a fleet of multiple autonomous surface vehicles (ASVs) with minimal communications is to use them to monitor areas of a water body where each vehicle is given an area defined by virtual boundaries within which to patrol. In this study, we looked at guidance accuracy for such system involving virtual boundaries overlaid on a water body with a minimum ‘safe’ straight line distance constraint between any two vehicles. Three objectives were put forth as indicators of the success of the test. Objective one was that the guidance algorithm kept all vehicles operating within their assigned areas >= 95% of the time. Objective two was that the guidance algorithm eliminated real collisions between vehicles. Objective three was that the guidance algorithm could correct collision events (see definition below) between vehicles within 10 seconds >= 95% of the time.

4.2. Materials and Methods

4.2.1. Definitions

Virtual Boundary: We define a virtual boundary as one or more latitudes and/or longitudes that, taken with other physical features of a water body (shorelines, etc.) completely enclose an area of water within which an autonomous surface vehicle may operate. In this test, the virtual boundaries are completely defined by a line of longitude for both the east and west side of each area taken with a line of latitude that defines the north and south boundaries. The virtual boundaries for the three surface vehicles are adjacent as shown in the figure below.

Home: We define a ‘Home’ point for each vehicle to be the GPS coordinate where the vehicle was launched into the water body.
Safe-zone: We define the safe-zone for a virtual boundary as a 1 meter buffer area on either side of a virtual boundary. The guidance algorithm for a vehicle which has entered the safe-zone should immediately apply a course correction to move away from the virtual boundary.

Straight-line Distance: We define the straight-line distance between any two vehicles as the haversine distance computed using the GPS latitude and longitude of each vehicle.

Collision: We define a collision to be when two vehicles approach within some minimum straight-line distance of each other. This idea derives from the accuracy of the GPS units providing the locational information to the vehicles guidance system. For the current vehicles, the GPS units have a locational accuracy of within 3-5 meters of the actual position 95% of the time. We therefore define a collision as two vehicles being within a straight-line distance of 6 meters of each other. Note in work by Price and Hall (2012), on a pond of approximately 2 hectares, actual physical collision (center points within <2 m) occurred once during a 96 hour run, suggesting that it is possible but unlikely in practice.

4.2.2. Test Setup

The virtual boundaries in the simulated environment were defined by a set of latitude and longitude lines chosen such that they overlay a portion of University Lake at Louisiana State University and expressed as the latitude and longitude of the line in degrees and decimal minutes (Table 1). All three vehicles share their North and South boundaries and have East and West boundaries as indicated in Figure 10 and Table 1 below:
Figure 10. Map showing virtual boundaries and base station

Table 1. Boundary Latitudes and Longitudes

<table>
<thead>
<tr>
<th>Boundary</th>
<th>ASV 0</th>
<th>ASV 1</th>
<th>ASV 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>North (Lat.)</td>
<td>30.408930</td>
<td>30.408930</td>
<td>30.408930</td>
</tr>
<tr>
<td>South (Lat.)</td>
<td>30.408000</td>
<td>30.408000</td>
<td>30.408000</td>
</tr>
<tr>
<td>East (Long.)</td>
<td>-91.167864</td>
<td>-91.166377</td>
<td>-91.164890</td>
</tr>
<tr>
<td>West (Long.)</td>
<td>-91.169350</td>
<td>-91.167864</td>
<td>-91.166377</td>
</tr>
</tbody>
</table>

Notice that the East boundary of Vehicle 1 is equal to the West boundary of Vehicle 2 and the same is true of the East and West boundaries of Vehicles 2 and 3, respectively. The boundaries specified give each vehicle a ~5-second-wide area of longitude within which to operate. At 30.44861 degrees latitude (the latitude of Baton Rouge, Louisiana), a degree of longitude is 96049.323 meters. From this we get a value for a second of longitude as 26.68 meters, so each vehicle has an area with a width of approximately 143 meters (~469 feet) to operate within. At the latitude of Baton Rouge, Louisiana, a second of latitude has a length of 110860.02 meters giving a result of 30.79 meters in one second of latitude. The latitudinal length of each area is 176.1 meters (~578 feet) or 5.72 seconds of latitude of operational area.
Each vehicle received its starting point, a point near center of the bounding box for that vehicle (Table 2) and was given a random initial bearing. The GPS location of the starting point was recorded by the vehicle at the beginning of the test and becomes the “Home” location that the vehicle returns to if it finds itself outside the bounding box that it is assigned. The basic navigation algorithm was a heading-hold algorithm that seeks to stay on the same heading regardless of the impact of wind or water flow. When shorelines, virtual boundaries, or obstacles are encountered, a new randomized direction is chosen for the heading. This should result in a random track covering the area within the virtual boundaries on the lake provided that environmental factors such as wind or water flow or system problems such as electrical or mechanical issues do not intervene.

A series (n=3) of tests were run with the vehicles operating within their specified areas. Each run lasted for 15 minutes. For the duration of each run, the following data was logged:

1. Vehicle GPS data at 1 second intervals consisting of: UTC Time, Statue, Latitude, Longitude, Speed (knots), Track angle (true), Date, Magnetic Variation (degrees), and the checksum

2. Each ASV maintains the current latitude, longitude, speed, and bearing of the other ASVs participating in the test run used to determine collision events and correct them.

3. The following events were logged when detected:

<table>
<thead>
<tr>
<th>ASV</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30.408465</td>
<td>91.168607</td>
</tr>
<tr>
<td>1</td>
<td>30.408465</td>
<td>91.167121</td>
</tr>
<tr>
<td>2</td>
<td>30.408465</td>
<td>91.165634</td>
</tr>
</tbody>
</table>
a. Vehicle outside of its assigned bounding box and corrective action taken;
b. Vehicle violation of any of the four virtual boundary “safe zones” and corrective action taken;
c. Vehicle within straight-line distance constraint (6 meters) adjacent vehicle(s) and corrective action taken.

The basic guidance for each vehicle was the following:

1. The vehicle proceeds in a straight line (attempting to hold the current heading) until an exception condition occurs. Exception conditions are defined as: vehicle GPS location outside of the assigned bounding box, entering a 1 meter “safe zone” on either side of the virtual boundary, computing a straight-line distance to an adjacent vehicle that is less than the given constraint.

2. Upon the occurrence of an exception condition, the vehicle makes a course correction as follows:

   a. Outside of assigned bounding box:
      i. Compute the bearing to its Home location (the starting location for the test)
      ii. Turn to the computed bearing
      iii. Proceed in the new direction

   b. Safe-zone violation:
      i. Determine which virtual boundary’s safe-zone has been violated.
      ii. Choose a random positive angle $\beta$ away from the violated safe-zone (such that $\alpha + 1 < \beta < \alpha + 180$)
      iii. Move the vehicle along the new bearing to resolve the safe-zone violation.
c. Adjacency Constraint:
   
i. Obtain coordinates of other vehicles

   ii. Compute the straight-line distance to every other vehicle

   iii. If the distance to any vehicle is less than the constraint (6 meters) both
       vehicles should select a bearing that takes the vehicle back towards the
       Home location, and then proceed.

The particular function within the generic ASV guidance program that manages the
navigation for this test is the checkBoundary function. This function takes as input the current
location and bearing of the ASV along with the locations of the other ASVs and the coordinates

```c
// First compute the distance to each boundary edge
toFloat = Haversine(currLat[thisASV], currLon[thisASV], currLat[thisASV], pointsLon[0]);
toFloat = Haversine(currLat[thisASV], currLon[thisASV], currLat[thisASV], pointsLon[2]);
toFloat = Haversine(currLat[thisASV], currLon[thisASV], pointsLat[0], currLon[thisASV]);
toFloat = Haversine(currLat[thisASV], currLon[thisASV], pointsLat[1], currLon[thisASV]);
```

**Figure 11.** Boundary edge calculations

of the bounding box for this ASV. The first step in checking is to compute the distance of the
ASV from each of the four boundary edges. It does so using the code in Figure 11.

Once these distances are known, a series of comparison statements (if blocks) are used to
test whether the ASV is within 3 meters of any of them. Corner situations are also tested within
these blocks. If the ASV is determined to be within one or more of the boundary safe-zones, the
violation is flagged and a new bearing is calculated. This block of comparisons also checks for
the condition where the ASV is completely outside of its bounding box. That condition
generates a unique flag and the new bearing is set to navigate the ASV back towards the center
of the bounding box.
After determining the boundary safe-zone violations, if any, the ‘collision’ events are checked. To check these, a loop iterates through the ASVs and calculates the haversine distance to each of the other two ASVs and checks to ensure that the result is greater than or equal to six meters (Figure 12). Since an ASV cannot collide with itself, that condition is excluded from testing. If that test fails for one or both of the ASVs (both should never happen in this test) the violation is flagged and the ASV’s new bearing is set to navigate it back towards the center of its bounding box. Complete Arduino® code developed in this study is listed in Appendix B.

```c
// For loop to check for simulated collision events - getting closer than 6 meters
// to either of the other ASVs
for (asvLoop=0; asvLoop<3; asvLoop++){  
  if (asvLoop != thisASV){ // Cannot collide with yourself
    if (Haversine(currLat[thisASV],currLon[thisASV],currLat[asvLoop],currLon[asvLoop]) < 6){
      // Simulated collision between thisASV and asvLoop has occurred, flag it
      violations[asvLoop+5] = 1;
      // Set the bearing to go back towards the center of our own box
      newBearing = Bearing(currLat[thisASV],currLon[thisASV],pointsLat[4],pointsLon[4]);
    }
  }
}
```

**Figure 12.** ASV code to check for simulated collision events
4.3. Results

The results are a set of log files, one for each vehicle for each test run, which contain all of the data to be analyzed. Each ASV log file contains two types of log entries. One log entry type is the logged GPS location for each second of the test run and contains just the information needed to map the ASV’s location for each second of the test run (Table 3).

### Table 3. GPS Log Data

<table>
<thead>
<tr>
<th>LABEL</th>
<th>SEND_BOAT</th>
<th>REC_BOAT</th>
<th>DATE</th>
<th>TIME</th>
<th>LAT</th>
<th>LON</th>
<th>SPD</th>
<th>BEARING</th>
<th>MESSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ASV</td>
<td>0</td>
<td>0</td>
<td>110421</td>
<td>70000</td>
<td>30.408451</td>
<td>-91.16861</td>
<td>3</td>
<td>189</td>
<td>GPS Logged</td>
</tr>
<tr>
<td>$ASV</td>
<td>0</td>
<td>99</td>
<td>110421</td>
<td>70001</td>
<td>30.408438</td>
<td>-91.16861</td>
<td>3</td>
<td>189</td>
<td>GPS Logged</td>
</tr>
<tr>
<td>$ASV</td>
<td>0</td>
<td>99</td>
<td>110421</td>
<td>70002</td>
<td>30.408424</td>
<td>-91.168617</td>
<td>3</td>
<td>189</td>
<td>GPS Logged</td>
</tr>
<tr>
<td>$ASV</td>
<td>0</td>
<td>99</td>
<td>110421</td>
<td>70003</td>
<td>30.408411</td>
<td>-91.168617</td>
<td>3</td>
<td>189</td>
<td>GPS Logged</td>
</tr>
</tbody>
</table>

The second log entry (Table 4) for each second of the test run contains the flags for the various boundary and collision conditions in addition to much of the same information that is contained in the GPS log entries. In Table 4, columns with duplicate information shown in the GPS log entries have been removed in order to show the portion of each entry with the various flags and rows have been picked from the log file to show various flags.

### Table 4. Boundary Check Data

<table>
<thead>
<tr>
<th>MESSAGE</th>
<th>Quad</th>
<th>Bearing</th>
<th>NBV</th>
<th>SBV</th>
<th>EBV</th>
<th>WBV</th>
<th>OBV</th>
<th>Boat0</th>
<th>Boat1</th>
<th>Boat2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary Check</td>
<td>3</td>
<td>189</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boundary Check</td>
<td>1</td>
<td>320</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boundary Check</td>
<td>4</td>
<td>153</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boundary Check</td>
<td>2</td>
<td>121</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boundary Check</td>
<td>3</td>
<td>189</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boundary Check</td>
<td>3</td>
<td>189</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The GPS log entries for each ASV consists of 900 points for each test run. Those points were imported into ArcGIS and used to create a poly-line showing the path each ASV traveled.
during each test run. An example of the output of this point-to-line feature creation is shown in Figure 13.

![Figure 13. Example output showing paths created from GPS log points](image)

The Boundary Check log messages provide the data on the various possible exceptions that the ASVs could have created during each test run. The output columns in the data correspond to the following violations:

1. **Boundary Violations** – Signifies encroachment of the ASV into the Safe Zone for that boundary:
   - North Boundary Violation (NBV)
   - South Boundary Violation (SBV)
   - East Boundary Violation (EBV)
   - West Boundary Violation (WBV)
2. Out of Box Violation – Signifies that the ASV’s GPS location places it outside of the virtual boundary that it is assigned.

3. Collision Violations – Signifies that the Straight Line Distance between this ASV and the ASV listed in the column is less than 6 meters. This is always 0 for the column that corresponds to the test ASV; meaning ASV0 cannot generate a collision violation with ASV0.

**Table 5. Test Run and Summary Data**

<table>
<thead>
<tr>
<th>Test 1</th>
<th>ASV</th>
<th>NBV</th>
<th>SBV</th>
<th>EBV</th>
<th>WBV</th>
<th>OBV</th>
<th>ASV_0</th>
<th>ASV_1</th>
<th>ASV_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run 1</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Totals</td>
<td>19</td>
<td>11</td>
<td>22</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| Run 2  | 0   | 5   | 2   | 7   | 6   | 0   | 0     | 0     | 0     |
|        | 1   | 17  | 2   | 14  | 0   | 0   | 0     | 0     | 0     |
|        | 2   | 30  | 4   | 21  | 1   | 1   | 0     | 0     | 0     |
| Totals | 52  | 8   | 42  | 7   | 1   | 0   | 0     | 0     | 0     |

| Run 3  | 0   | 13  | 4   | 13  | 3   | 0   | 0     | 0     | 0     |
|        | 1   | 6   | 7   | 6   | 3   | 0   | 0     | 0     | 0     |
|        | 2   | 9   | 5   | 7   | 1   | 0   | 0     | 0     | 0     |
| Totals | 28  | 16  | 26  | 7   | 0   | 0   | 0     | 0     | 0     |

Test 1 Totals | 99  | 35  | 90  | 23  | 1   | 0   | 0     | 0     |

As can be seen from Table 5, the ASVs entered the safe-zones of the various virtual boundaries a total of 247 times and since there is only one Out-of-Boundary violations, the ASVs successfully managed to correct those violations 246 of those times. It is interesting to note that there are no ‘collision’ events recorded across all ASVs over all three test runs. This is perhaps not too surprising considering that each ASV covers an area of one square meter and the total test area is ~59,611 square meters.
4.4. Data Analysis

Of the numerous safe-zone violations registered during the three test runs by the ASVs, each was correctly identified, and a new bearing was chosen such that the violation was corrected with the exception of one such violation. In test run 2, ASV 2 encountered a condition where upon ending up near a corner of the North and East boundaries of its bounding box, the algorithm chose a new bearing which, by the time the next GPS reading was taken, resulted in the ASV’s location being outside of the bounding box. While not obvious in the view of the overall all map, if that area of the map is magnified (Figure 14), there is a definite pattern of oscillation in the bearings chosen and the Out-of-Boundary violation is clearly visible.

![Figure 14. Closeup of out-of-boundary event](image)

Of the 900 GPS points taken per ASV per test run, only one point was outside the appropriate bounding box. This translates into a percentage of points within the bounding box in excess of 99.9% which exceeds the defined success rate for the algorithm of >= 95%. In addition, no collision events were recorded during the three test runs by any of the ASVs.
From the paths generated from the GPS logs of each ASV for each of the three test runs, three maps were generated utilizing a model to determine the percent area covered of the virtual boundary for three different buffer widths (3, 5, and 10 meters). These buffer widths are based on earlier visual observations of the effect of the presence of an ASV on waterfowl present on the water body in terms of frightening the birds away from aquaculture ponds. The ArcGIS® model takes as input a comma separated values file of the GPS log points for one ASV along with the bounding box of that ASV’s virtual boundary. As output, the model creates an ArcGIS® feature that corresponds to the ASV’s path. It then buffers that path at the three widths, dissolves the buffers (so that the 20-meter buffer includes the areas of the 10 and 3 meter buffers), and then clips the buffers to the area of the bounding box since we are only concerned with the area within the bounding box that is covered by the ASV. By running the model for each ASV within a test run, the maps shown in Figures 15, 16, and 17 were generated. The Python™ code generated by the ArcGIS® model builder is included in Appendix D. The data table for each map is created by taking the data from the buffer calculations and editing it in Microsoft® Excel® before adding the table back to the map document.
Figure 15. Area Patrol Test, Run 1
Figure 16. Area Patrol Test, Run 2
Figure 17. Area Patrol Test, Run 3
In Table 6, the summary statistics for the area of the test area covered by the various ASVs both for their bounding boxes and total test area are shown for each buffer width. These coverages were achieved with the ASVs running at a medium speed throughout the tests.

**Table 6. Summary statistics for Area Patrol tests**

<table>
<thead>
<tr>
<th>Test Run</th>
<th>Patrol Area</th>
<th>3 Meter</th>
<th></th>
<th>10 Meter</th>
<th></th>
<th>20 Meter</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Area (m^2)</td>
<td>% Coverage</td>
<td>Area (m^2)</td>
<td>% Coverage</td>
<td>Area (m^2)</td>
<td>% Coverage</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>7817.19</td>
<td>39.34%</td>
<td>15893.41</td>
<td>79.99%</td>
<td>18256.14</td>
<td>91.88%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7756.60</td>
<td>39.04%</td>
<td>16400.43</td>
<td>82.54%</td>
<td>19358.09</td>
<td>97.42%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6722.57</td>
<td>33.83%</td>
<td>13740.58</td>
<td>69.15%</td>
<td>16570.13</td>
<td>83.39%</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7817.19</td>
<td>39.34%</td>
<td>15893.41</td>
<td>79.99%</td>
<td>18256.14</td>
<td>91.88%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7756.60</td>
<td>39.04%</td>
<td>16400.43</td>
<td>82.54%</td>
<td>19358.09</td>
<td>97.42%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6722.57</td>
<td>33.83%</td>
<td>13740.58</td>
<td>69.15%</td>
<td>16570.13</td>
<td>83.39%</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>7817.19</td>
<td>39.34%</td>
<td>15893.41</td>
<td>79.99%</td>
<td>18256.14</td>
<td>91.88%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7756.60</td>
<td>39.04%</td>
<td>16400.43</td>
<td>82.54%</td>
<td>19358.09</td>
<td>97.42%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6722.57</td>
<td>33.83%</td>
<td>13740.58</td>
<td>69.15%</td>
<td>16570.13</td>
<td>83.39%</td>
</tr>
</tbody>
</table>

| Overall | Average ASV boundary Area Covered | 7432.12 | 37.40% | 15344.81 | 77.22% | 18061.45 | 90.90% |
|         | Average Total Boundary Area Covered | 22296.36 | 37.40% | 46034.42 | 77.22% | 54184.35 | 90.90% |

4.5. Discussion

The navigation algorithm developed for the ASVs in this experiment was successful in containing the ASVs within their assigned boundary > 99.9% of the test time thus exceeding the defined success rate of >=95% of the time. Since there were no collision events recorded, it is impossible to determine if the algorithm can successfully deal with this situation although it is expected that this would also be successful.
Visual observation of the paths traced by the ASVs especially in the second test run seem to indicate a possible bias in the chosen bearings towards the northeast corner of the bounding box. The absence or presence of such a bias would require additional geospatial analysis of the data but, if present, could be due to a bias inherent in the algorithm itself or a possible bias in the random number generator used to assist in the selection of the new bearings.

The areas covered by the ASVs within their respective boundaries over the test runs demonstrate a coverage much greater than a single ASV could be expected to cover over the same time frame. This would offer improved predation reduction over a single ASV system since a single ASV system attempting to patrol such a large area would likely just move the waterfowl to the end of the water body away from its current location.

4.6. Conclusions

In this experiment, we created a guidance algorithm for a fleet of homogeneous autonomous surface vehicles that, using minimal communication, can maintain correct guidance and navigation behavior on a water body. Each vehicle successfully operated within its own predefined boundary without incursions into areas assigned to other vehicles and without exiting its assigned boundary greater than 95% of the time, and without any collisions with other vehicles. The three ASV fleet covered more of the overall test area than a single ASV could demonstrating the effectiveness of a multi-vehicle fleet in the predation reduction problem.

Some future options in the simulation arena would be to analyze and correct for any navigational biases through modification of the algorithm to eliminate any inherent navigational bias if present, a more robust random number generator, or potentially modification to the amount of randomness used in choosing bearings. The haversine distance and bearing calculations are computationally expensive so over the short distances in the test scenarios, so a
simpler method that yields adequate results might be preferred. Since the haversine formulas are based on a spherical model, they are less precise than Vincenty’s formulas (Vincenty, 1975) of calculating the distance and bearing that are based on a spheroid model, but these are more computationally expensive since they are a set of iterative equations. Because of the floating-point limitations imposed by only having single-precision floating point numbers (represented by four bytes) in the design of the Arduino® the utilization of a simpler distance method might yield calculations that are faster and do not significantly affect the results obtained. One alternative that could improve the accuracy of the floating point calculations would be to offload those calculations to the BeagleBone® Black if those boards were to be added to the hybrid simulation since they support higher precision floating point representations. Since the basic simulation generates perfect GPS coordinates and provides perfect communication between the ASVs, real world testing is likely to reveal additional areas of potential future work with the patrol algorithm. Simulated variations in GPS signal (e.g. dithering); variations in wind or current could be added to provide further real world aspects to the simulation. In the next chapter, a number of experiments are discussed wherein communication becomes more critical with each successive experiment.
CHAPTER 5. ADDITIONAL ALGORITHM DESIGNS AND FUTURE WORK

In this chapter we explore additional algorithm designs that could have applications across many physical domains (in addition to water surfaces) and many application domains from environmental monitoring to food production and others. For several of the algorithms existing code is shown. This chapter lays the groundwork for future simulation studies and field tests.

5.1. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Sequentially Following a Path

5.1.1. Introduction

For a fleet of multiple autonomous surface vehicles, a guidance algorithm is designed and tested for a ‘follow-the-leader’ scenario in which the lead vehicle transmits a GPS location and heading at the beginning of the test and for each turn it makes to the remaining vehicles. Those vehicles, starting at fixed time intervals, proceed to attempt to follow the original track provided by the lead vehicle.

The goal of the guidance algorithm for the lead vehicle is to traverse the course defined by the list of GPS waypoints given it at the beginning of the test run as accurately as possible. The last waypoint is a point along the shoreline where the test ends. For each of the remaining vehicles, the goal is to follow as accurately as possible the waypoints transmitted by the lead vehicle.

5.1.2. Materials and Methods

A sequential series of points within the test area (See Figure 18) are created for this test as the path that the vehicles should traverse. For each test run, the ‘lead’ vehicle has a preloaded
set of GPS locations that define a route on the surface of the water body. Initiation of the test occurred with the lead vehicle reading the first GPS location (or waypoint) from the array of points, calculating a bearing to that point, and heading towards the point. When the lead vehicle passes that point, it transmits the first waypoint to the remaining vehicles, reads the next point from the array, calculates a bearing to the new waypoint, and proceeds to the point. This process repeats until the last waypoint is processed and the lead vehicle signals that its test run is complete and stops moving. For the remaining vehicles, each waypoint received must be stored in order so that when they are released to start their test run, they know where they are going. To avoid potential collisions and to ensure that no vehicle caught the previous vehicle, the start time for each vehicle after the first was delayed by two waypoints. In other words, if the lead vehicle is denoted as ASV0, ASV1 does not start its test run until ASV0 has reached its second waypoint and has started towards the third waypoint. Similarly, ASV2 waits until ASV1 has started towards its third waypoint before beginning its test run.

Figure 18. Example track with GPS waypoints

Figure 18. Example track with GPS waypoints
A vehicle that is proceeding to a waypoint will calculate the straight-line distance from
the most current GPS reading taken to the waypoint. Once the straight-line distance between the
two points is <= 2 meters, the vehicle would be considered to have reached the point. Once a
point has been reached, any increase in the straight-line distance would be considered passing the
point and the vehicle will then proceed to the next waypoint in its list.

The following data would be recorded for each test run:

1. Every GPS reading taken by the vehicle and the straight-line distance between the GPS
   point and the waypoint currently being sought.
2. Each waypoint and the time that it becomes the current waypoint being sought.
3. The GPS reading and distance to the waypoint when a waypoint is ‘reached’.
4. The GPS reading and distance from the current waypoint when the turn is made towards
   the next waypoint in the list.
5. The time required for each leg of the test as well as the total time for the run.

5.1.3. Results

The results would be a set of log files, one for each vehicle for each test run, which will
contain all of the data to be analyzed.

This test series should highlight the differences in GPS locations from identical sensors
over time and the accuracy limitations of those sensors. Properly designed algorithms will work
in spite of these limitations by taking the limitations into account when making decisions based
on sensor data. Since the vehicles would be traversing the path sequentially, interesting
anomalies could arise from variations in the environmental conditions over the course of the
tests.
5.1.4. Data Analysis

Analysis of the data would be performed as follows:

1. Individual Vehicle Analysis for each Test:
   a. Results for each vehicle would be compared to the ideal track through the course using the RMS error for each GPS point taken. Average RMS error for each leg of the course (one waypoint to the next) as well as overall average RMS error for the vehicle for the whole course would be calculated.
   b. The RMS error between the straight-line track that the vehicle has computed and the GPS track taken would be computed for each GPS point. Average RMS error for each leg of the course (one waypoint to the next) as well as overall average RMS error for the vehicle for the whole course would be calculated.
   c. For each turn taken during the test run, the error between the waypoint given and the GPS point at which the turn was initiated would be calculated. The average (arithmetic mean) error for each vehicle over the entire course will also be computed.

2. Total Individual Test Run Analysis:
   a. Calculated errors from the Individual Vehicle run in 1.a (ideal track versus actual GPS recorded track) above would be averaged for all three vehicles over each leg of the course as well for all three vehicles over the entire course. In addition to calculating the mean, calculations for standard deviations and tests for significance will also be performed.
   b. Calculated errors from the Individual Vehicle run in 1.b (vehicle computed track versus actual GPS recorded track) above would be averaged for all three vehicles
over each leg of the course as well for all three vehicles over the entire course. In addition to calculating the mean, calculations for standard deviations and tests for significance will also be performed.

c. The error for each point calculated in 1.c above would be averaged over all vehicles for the test run. In addition, the overall course error calculated in 1.c for each vehicle would be averaged. In addition to calculating the mean, calculations for standard deviations and tests for significance will also be performed.

3. Total Test Analysis:

   a. For all of the test runs, the individual errors calculated in 1.a would be averaged for the same vehicle across all test runs. In addition, the average errors for all vehicles over all runs would be calculated. Standard deviations would be calculated and tests for significance will also be performed.

   b. For all of the test runs, the individual errors calculated in 1.b would be averaged for the same vehicle across all test runs. In addition, the average errors for all vehicles over all runs would be calculated. Standard deviations would be calculated and tests for significance will also be performed.

   c. For all of the test runs, the individual errors calculated in 1.c would be averaged for the same vehicle across all test runs. In addition, the average errors for all vehicles over all runs would be calculated. Standard deviations would be calculated and tests for significance will also be performed.

5.1.5. Discussion

The results from the test are expected to demonstrate that, within the accuracy obtainable with the sensors, the algorithms perform well. Significantly poorer performance could result
from significant variations in environmental factors; particularly wind. The algorithms will attempt to complete their given task, but if the wind across the lake exceeds the ability of the propulsion system, the vehicles may not be able to compensate. These tests, as did the tests in chapter one, require sparse communications between all of the vehicles of the fleet but more complex navigation and guidance processing. Beginning with the tests in chapter 5, more communication traffic would be generated during the tests and the guidance becomes significantly more complex.

5.2. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Parallel Straight-Line Following

5.2.1. Introduction

For a fleet of multiple autonomous surface vehicles (ASVs) a guidance algorithm is designed and tested for a scenario in which the vehicles, starting from points along a line of longitude or latitude and separated by equal distances in the latitudinal or longitudinal direction respectively, attempt to maintain their positions relative to each other both in terms of latitude and longitude as they navigate across a lake.
5.2.2. Materials and Methods

Two test series (n=5 for both series) will be conducted on Campus Lake at LSU. For each test, the fleet of ASVs would be started at a line of longitude or latitude with equal separation between the individual ASVs in the direction perpendicular to the direction of travel (see Figure 19).

They will then proceed across the lake in the direction of travel while maintaining their spacing perpendicular to the direction of travel and maintaining their orientation along the line perpendicular to the direction of travel at all times. This requires the vehicles to constantly monitor both their own paths and speed as well as those of the other vehicles and make course and speed corrections as needed.

For each test run, the following data would be collected:

1. Every GPS reading taken by the vehicle. This will give not only the position of the vehicle, but the heading and speed as well.
2. Compass readings at each point along the path.
3. The straight-line distance between the ASV and all other ASVs.
4. The angle (if any) between the ASV and all other ASVs.

5. The course corrections implemented (if any).

### 5.2.3. Results

Results for this test series are expected to demonstrate that within the limitations imposed by the sensor accuracy and the vehicles’ ability to compensate for environmental conditions, the algorithm would be successful at maintaining the appropriate spacing for the duration of the test. The challenges for the guidance and navigation algorithm could come from uncertain environmental conditions, the difficulties caused by the demand for increased communication, and, as in every test, the limitations on positional accuracy imposed by the sensors in use. Wind could be a significant factor for this test; especially if the test area is experiencing gusts rather than a slow, steady breeze.

### 5.2.4. Data Analysis

We define the following terms (See Figure 20):

- **Horizontal error** ($e_h$): The error occurring in the spacing between the boats perpendicular to the direction of travel. A horizontal error of zero means that the boats are maintaining the specified spacing.

- **Vertical error** ($e_v$): The error occurring in the spacing between the boats in the direction of travel. A vertical error of zero means that the line drawn between the centers of all three ASVs is a straight line that is perpendicular to the direction of travel.

- **Course error** ($e_c$): The error between the actual course of each ASV and the theoretical course it should be taking based on its starting point and the projected course.
For each test run, the following analyses would be performed:

1. The course error ($e_c$) between the theoretical track for each vehicle and the actual track recorded by the GPS. Error would be calculated as the RMS error for each GPS point taken along the track versus the theoretical track.

2. The horizontal error ($e_h$) for each ASV as the RMS error of the actual distance between the ASV and its neighbor(s) and the specified spacing.

3. The vertical error for each ASV both as the RMS error of the distance between the line perpendicular to the direction of travel drawn through each ASV and the lead ASV and the RMS error of the distance between the line perpendicular to the direction of travel drawn through each ASV and a line representing the modeled behavior of the vehicles operating at their top speed through the course.

   In addition, for all test runs in a particular orientation and again for the total of all test runs, the same statistics would be computed as for the individual runs.
5.2.5. Discussion

An understanding of basic parallel path planning and coordination can be a useful starting point for more complex tasks. It appears that if boats are guided, each to their own predetermined linear route, errors can be introduced due to mechanical variation in the boats; as well as physical issues such as wind or currents. If one boat is attempting to follow a line and others track the first boat, errors may be compounded (e.g. the other boats may continue to run parallel to the first boat, even if that boat gets pushed off course by wind. In extreme cases of wind or currents, it is likely that the ability to follow even a straight line would be compromised, and algorithms may be needed to choose ‘rest periods’ under storm or other situations, conserving energy and intermittently attempting to follow a new parallel path or return to the original path, again resting if unable to do so after a ‘reasonable’ period.

5.3. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Parallel Straight-Line Path Following With 180 Degree Turns

5.3.1. Introduction

For a fleet of multiple autonomous surface vehicles, we design and test a guidance algorithm for a scenario involving parallel straight-line path following with 180 degree turns. Starting from points along a line of longitude and separated by equal distances in the latitudinal direction, the vehicles will maintain their position abreast of each other and equidistant from each other as they navigate across the entire surface of a water body in a north/south pattern. During the turns at the north and south shore of the water body, the innermost vehicle to the turn will execute an arc through 180 degrees with a radius of the width of the vehicle while each of the other vehicles turns through a 180-degree arc with radii such that the initial separation of the vehicles is maintained. Throughout the turns, all of the vehicles will adjust their speeds in such a
way that the line perpendicular to the arc proscribed by the innermost vehicle of the turn passes through all three vehicles at all times during the turn.

5.3.2. Materials and Methods

Two test series (n=5 for both series) would be conducted on Campus Lake at LSU. For each test, the fleet of ASVs would be started at a line of longitude or latitude with equal separation between the individual ASVs in the direction perpendicular to the direction of travel as in the straight-line parallel navigation test. In this test, the vehicles will turn through a 180° arc at the end of each straight section in such a way that the vehicle closest to the center of the arc turns through a turn with a radius such that when the turn is complete, the vehicle has moved one body width over (Figure 21). The middle and outer vehicles will, at all times during the turn maintain their relative positions in such a way that the line through the center of all three vehicles is a straight line that is always perpendicular to the direction of travel of the inmost vehicle. This will require the two inner vehicles to adjust their speed to allow the outermost vehicle to keep up.

Figure 21. Parallel Navigation through 180 degree turns
For each test run, the following data would be collected:

1. Every GPS reading taken by the vehicle. This will give not only the position of the vehicle, but the heading and speed as well.
2. Compass readings at each point along the path.
3. The straight-line distance between the ASV and all other ASVs.
4. The angle (if any) between the ASV and all other ASVs.
5. The course corrections implemented (if any).

5.3.3. Results

The results from this test will demonstrate that the guidance and navigation algorithm was able, with the increased communication between vehicles, to maintain the formation of the vehicles through the 180 degree turns by continuously adjusting the speed and heading of the vehicles. As a result of the increased communications needed, factors such as communication collisions and communication lag could force vehicles to make decisions with incomplete or outdated information resulting in errors in course corrections. This could lead to interesting behaviors as the vehicles attempt to correct for these errors. As with all of the other tests, the accuracy limitations of the sensors will play an important role in the vehicles’ ability to maintain their positions relative to each other. Environmental factors may be even more of a factor in this test than in previous tests in this project. The task of maintaining the formation while turning is significantly harder than that of maintaining the formation while operating in a straight line so the variations that can be introduced by wind or water movement will require more adjustments to the vehicles’ courses.

5.3.4. Data Analysis

We define the following terms:
• Horizontal error: The error occurring in the spacing between the boats perpendicular to the direction of travel. A horizontal error of zero means that the boats are maintaining the specified spacing.

• Vertical error: The error occurring in the spacing between the boats in the direction of travel. A vertical error of zero means that the line drawn between the centers of all three ASVs is a straight line that is perpendicular to the direction of travel.

• Course error: The error between the actual course of each ASV and the theoretical course it should be taking based on its starting point and the projected course.

For each test run, the following analyses would be performed:

1. The course error between the theoretical track for each vehicle and the actual track recorded by the GPS. Error would be calculated as the RMS error for each GPS point taken along the track versus the theoretical track.

2. The horizontal error for each ASV as the RMS error of the actual distance between the ASV and its neighbor(s) and the specified spacing.

3. The vertical error for each ASV both as the RMS error of the distance between the line perpendicular to the direction of travel drawn through each ASV and the lead ASV and the RMS error of the distance between the line perpendicular to the direction of travel drawn through each ASV and a line representing the modeled behavior of the vehicles operating at their top speed through the course.

In addition, for all test runs in a particular orientation and again for the total of all test runs, the same statistics would be computed as for the individual runs.
5.3.5. Discussion

In this test series, similar to the parallel tests, boats attempt to follow their own path but may also be affected by the paths of other boats. Additional algorithm features may be needed (e.g. ‘waiting periods’ when ‘inner’ boats get ahead of outer boats) to maintain reasonably parallel action among the boats. These types of features are likely needed in any parallel turning applications.

5.4. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Gradient Tracking with a Fixed Leader

5.4.1. Introduction

For a fleet of multiple autonomous surface vehicles, we design and test a guidance algorithm for a scenario in which a contaminant is spilling into a water body forming a plume. In the simulation, the gradient is created by choosing a point within the outer bounding box of the test area and using a linear formula to determine the virtual sensor value \(0 < x < 1.0\). The vehicles, starting from random points within the test area will proceed across the area in a randomized straight-line pattern seeking evidence of the gradient. The first vehicle to encounter the gradient becomes the leader and leads all of the remaining vehicles towards the point source of the gradient. In the event that two ASVs encounter the gradient simultaneously, the ASV sensing the larger gradient value would become the leader; in a tie, the lower numbered ASV becomes the leader.
5.4.2. Materials and Methods

For these tests, the gradient source would be generated by the Python™ simulation using a linear function at a point within the boundary of the test area (Figure 22). This gradient would be defined such that the value perceived by the ASV is proportional to the distance the ASV is from the point source of the function. The value of the function would be 1.00 at the point source and will decrease by 0.01 for every 2 meters of distance separating the source and the ASV to a maximum distance of 200 meters from the point source. Beyond 200 meters, the value returned by the ASV’s sensor is 0.00, simulating a gradient that is below the threshold value that the sensor can register. This will create a gradient across the test area such that some portions of the test area may not register a gradient on the ASV sensors depending on the location of the point source of the gradient function (Figure 23).
A test consisting of n=3 samples would be conducted using the Python™ simulation of the test area that is designed to overlay the south end of University Lake at LSU. For each

**Figure 23.** Function to calculate simulated sensor value

def calcSensorVal(currLat, currLon, ptSrcLat, ptSrcLon):

    print("Inside calcSensorVal")
    # Compute Haversine distance between the points
    # First convert coordinates into angles in radians
    radcurrLat = math.radians(currLat)
    radptSrcLat = math.radians(ptSrcLat)
    radDeltaLat = math.radians(ptSrcLat - currLat)
    radDeltaLon = math.radians(ptSrcLon - currLon)

    a = (math.sin(radDeltaLat/2) * math.sin(radDeltaLat/2)) + math.cos(radcurrLat) *
    (math.cos(radptSrcLat) * (math.sin(radDeltaLon/2) * math.sin(radDeltaLon/2)))
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))

    d = 6371000 * c

    if (d > 200):
        sensorRdg = 0.000
    else:
        sensorRdg = (1-(d/200))

    #Print values before returning
    print("Done calcSensorVal")
    print(sensorRdg)

    return sensorRdg

    # End of calcSensorVal
sample, the fleet of ASVs would be started at random points within the boundary of the test area with random starting bearings.

```java
// Check to see if the leader has already been selected (leadASV<99)
if (leadASV == 99){
    if ((currSensor[thisASV]>currSensor[((thisASV+1)%3)]) &&
        (currSensor[thisASV]>currSensor[((thisASV+2)%3)])){
        leadASV = thisASV;
    }
    else if (currSensor[((thisASV+1)%3]) > currSensor[((thisASV+2)%3)])){
        leadASV = ((thisASV+1)%3);
    }
    else {
        leadASV = ((thisASV+2)%3);
    }
}
```

**Figure 24.** Code snippet to determine lead ASV

The gradient tracking algorithm works as follows:

1. At the beginning of the test, all vehicles would be operating independently with no vehicle leading.

2. The first ASV to detect a gradient sensor reading will broadcast its location and sensor reading to the group and would become the leader.

3. After a leader is chosen, the leadership of the group will not change. The remaining vehicles will compute a rendezvous course for the position of the lead ASV each time a new position is received.

4. At all times, all ASVs will log sensor readings and correct boundary events if applicable.

Each time the ASV gets a sensor reading from the simulation, it packages the sensor reading and its current location into a radio message that is then transmitted via the simulation to
the other ASVs. It also receives radio messages of the same type from the other ASVs, parses those messages and uses the sensor value contained in those messages to see if a lead ASV has been determined using the block of code in Figure 24.

For each ASV in each sample run, the following data would be collected:

1. Complete GPS track from sample run start to end.
2. Every sensor reading would be stored with the associated GPS reading at the time the reading was taken.
3. Detection of a change in gradient of more than 0.01.
4. All course and speed corrections produced by the algorithm as the gradient tracking test is run.
5. All course and speed corrections resulting from messages received from other vehicles.
6. Total time of the test. The test would be considered over when all three vehicles have reached the gradient source.

5.4.3. Results

The results from this test are expected to reveal that the vehicles will converge on the source of the gradient. It is expected that the convergence using this guidance algorithm may take significantly longer to occur with a single leader versus the multiple leader scenario tested in chapter 5.5. Since the algorithm is picking a permanent leader to be the first vehicle to detect a gradient change, there could be challenges resulting from a number of possibilities. If the temperature gradient formed for the test is not linear across the surface of the water body, the lead vehicle could find a local minimum rather than the global minimum temperature leading to an incorrect identification of the source. One of the major challenges in this test would be creating and maintaining a detectable gradient using temperature for the duration of the test.
Since the lake will need to equilibrate between runs, this test series will likely have to be run across multiple days which may make environmental factors even more significant than they have been in other tests. Additionally, a sensor failure of the temperature sensor on the lead vehicle could cause a complete failure of the fleet to locate the source. As with all of the tests, environmental conditions and sensor accuracy may be factors.

5.4.4. Data Analysis

The following data analyses would be performed:

1. The tracks of each vehicle would be plotted in ArcGIS for each test run over the gradient map produced by the simulation.

2. The time required to complete the tracking from the start of the test and from the first temperature variation detection event.

3. The straight line distance from each vehicle to the source of the gradient at the time the leader is chosen.

4. The number of times in each test that the leadership would have changed if the algorithm changed leadership each time a new vehicle detected a significant change based on the sensor readings at each point in time.

5.4.5. Discussion

Gradient tracking is both more challenging algorithmically and potentially more useful in practice than some of the previous algorithms. A number of potential path following sub-algorithms might be used to more efficiently track gradients. In practice, the ability to track gradients could be useful for a variety of real-world phenomena such as toxic spills including oceanic or coastal oil spills; toxic spills in rivers or lakes; or natural movement of harmful algal blooms (HABs). Desired components of the water column might also be tracked, for example,
movement of desired algal components potentially useful in culturing of filter feeding shellfish such as oysters or clams; temperature gradients approaching a desired threshold e.g. for predicting spawning or blooming behavior of aquatic wildlife or cultured aquatic organisms. The algorithms themselves could potentially be adapted to other phenomena such as tracking of desirable or undesirable components in foods; blood or other liquids at a variety of scales from cellular to ecosystem level.

5.5. Guidance Algorithm for Multiple Autonomous Surface Vehicles for Gradient Tracking with a Changeable Leader

5.5.1. Introduction

For a fleet of multiple autonomous surface vehicles, we design and test a guidance algorithm for a scenario in which a contaminant is spilling into a water body forming a plume. In the simulation, the gradient is created by choosing a point within the outer bounding box of the test area and using a linear formula to determine the virtual sensor value ($0 < x < 1.0$). The vehicles, starting from random points within the test area will proceed across the area in a randomized straight line pattern seeking evidence of the gradient. As each ASV traverses the test area, it takes sensor readings (provided by the simulation) and shares them with the other ASVs. In the event that two ASVs encounter the gradient simultaneously, the ASV sensing the larger gradient value would become the leader; in a tie, the lower numbered ASV becomes the leader. With each shared sensor reading, the leadership role passes to the vehicle which is currently encountering the highest gradient reading. Upon becoming the leader, all of the other vehicles follow that vehicle towards the source of the gradient. Leadership may change multiple times as sensor readings are shared between the vehicles.
5.5.2. Materials and Methods

For these tests, the gradient source would be generated by the Python™ simulation using a linear function at a point within the boundary of the test area (Figure 22). This gradient would be defined such that the value perceived by the ASV is proportional to the distance the ASV is from the point source of the function. The value of the function would be 1.00 at the point source and will decrease by 0.01 for every 2 meters of distance separating the source and the ASV to a maximum distance of 200 meters from the point source. Beyond 200 meters, the value returned by the ASV’s sensor is 0.00, simulating a gradient that is below the threshold value that the sensor can register. This will create a gradient across the test area such that some portions of the test area may not register a gradient on the ASV sensors depending on the location of the point source of the gradient function (Figure 23).

A test consisting of n=3 samples would be conducted using the Python™ simulation of the test area that is designed to overlay the south end of University Lake at LSU. For each sample, the fleet of ASVs would be started at random points within the boundary of the test area with random starting bearings. The gradient tracking algorithm works as follows:

```c
leadcnt = 0;
for (leadlp=0;leadlp<3;leadlp++){  
    if ((currSensor[leadlp] > 0.0001) && (currSensor[leadlp] > currSensor[leadcnt])){
        leadcnt = leadlp;
    }
    if (currSensor[leadcnt]>0.0001){
        leadASV = leadcnt;
    } else {
        leadASV = 99;
    }
}
```

**Figure 25.** Arduino® code to select lead ASV
For these tests, the gradient source would be a point within the test area of the simulation. The gradient is created with a linear formula such that. This will, over time, produce a temperature gradient across the lake and will allow the testing of the gradient tracking algorithms without harming the flora and fauna present in the environment.

A test consisting of n=3 samples would be conducted on Campus Lake at LSU. For each sample, the fleet of ASVs would be started at random points along the shoreline opposite the gradient source and would be aimed in random directions. No ASVs would be aimed towards the gradient source as a starting direction.

The gradient tracking algorithm will work as follows:

1. At the beginning of the test, all vehicles would be operating independently with no vehicle leading.

2. The first vehicle to detect a gradient difference will broadcast that reading to the group and would become the leader.

3. After a leader is chosen, the remaining vehicles will compute a rendezvous course for the position of the lead vehicle.

4. At all times, all vehicles will log gradient readings and detection events.

5. If, at any time after a leader is chosen, a vehicle which is not the leader detects a gradient value that is greater than the most recent one transmitted by the leader, that vehicle will transmit a message with its GPS position, heading, speed, and sensor reading and would become the new leader.

For each ASV in each sample run, the following data would be collected:

1. Complete GPS track from sample run start to end.
2. Every sensor reading would be stored with the associated GPS reading at the time the reading was taken.

3. The initial leader chosen and any subsequent leadership changes with a date and time stamp.

4. All course and speed corrections produced by the algorithm as the gradient tracking test is run.

5. All course and speed corrections resulting from messages received from other vehicles.

6. Total time of the test. The test would be considered over when all three vehicles have reached the gradient source.

5.5.3. Results

This test series is expected to show that by allowing the guidance and navigation algorithm to change the leadership role during the test based on the latest sensor readings from each vehicle, a significant improvement in the length of time required for the vehicles to converge on the source of the gradient can be achieved. Because the leadership can change, the communications occurring between the vehicles is potentially a larger source of challenges than in chapter 8’s tests since all vehicles need to continually update all of the other vehicles with sensor data. In addition, because the vehicles would be constantly updating which vehicle is the leader and which vehicle they should be vectoring to intercept, a missed communication could result in navigational errors.

5.5.4. Data Analysis

The following data analyses would be performed:

1. The tracks of each vehicle would be plotted in ArcGIS for each test run over the gradient map produced by the linear function of the simulation.
2. The time required to complete the tracking from the start of the test and from the first gradient detection event.

3. The number of leadership changes for each test run and the average number of leadership changes for the entire set of tests.

5.5.5. Discussion

Part of the discussion will compare the results found in the test from Chapter 5.4 with the results in Chapter 5.5 to see if the gradient tracking algorithm performs significantly better when the leadership is allowed to change throughout the run.

5.6. Conclusions

The tests presented in this chapter represent the ongoing research into the types of algorithms that may prove useful in systems of multiple autonomous vehicles. These algorithms represent a broad range of problem types including such applications as: environmental monitoring and remediation; biohazard warning systems; aquaculture monitoring, predation reduction, and harvesting; crop pest identification and eradication; and others. Creating and testing these algorithms on the system of ASVs presented earlier will allow for the exploration of both the capabilities and limitations of the navigational abilities of a cost-effective research system.
CHAPTER 6. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

In this research, we have produced a set of three identical autonomous surface vehicles including the design and construction of the physical hardware along with the design and subsequent redesign and construction of the electronics hardware to create a highly cost effective, modular research platform that is both easily repairable and easily upgradable. The use of off-the-shelf electronics and open-source software enabled the design goals. Using this platform, we have designed multiple navigation algorithms, and performed preliminary testing of the fleet of physical ASVs. Testing continued after the advent of the COVID-19 pandemic restrictions with a hybrid hardware/software simulation to produce successful results for a limited number of the algorithms. The area patrol test demonstrated a navigation algorithm capable of guiding the fleet of ASVs correctly through their patrol tasks while meeting both criteria of maintaining all of the vehicles within their assigned boundaries $\geq 95\%$ while preventing any collision events from occurring between the ASVs. In addition, for the particular issue of area coverage for reducing bird predation, we demonstrated that the ASVs generated adequate coverage of the water surface.

Additional algorithms were written for other navigation tasks that demand communication and coordination including testing the ASVs ability to follow each other through a set of waypoints and the ability of the fleet of ASVs to track gradients in the water body and preliminary testing demonstrated the viability of the follow-the-leader test although no definitive results were obtained. Other navigation tasks that were discussed in detail and for which testing methodologies were created included algorithms to allow the devices to navigate parallel to each other both along straight paths and through turns to allow complete coverage of a water body.
As discussed in previous chapters, much additional work could be done in this developing field. Continued development of understanding of optimal path planning, especially for multiple devices is important. Some of the insights from simple (e.g. virtual boundary problems or parallel straight line travel); as well as from more complex (e.g. curvilinear travel or gradient tracking) could be optimized further. Future research with these devices could focus on incorporating additional capabilities to the system through the inclusion of other sensor systems. Some of these could include vision systems tied to image recognition software in the form of small neural network devices that would give the ASVs better object detection or allow for visual tracking of specific targets, additional water quality sensors for mapping and tracking environmental parameters, the addition of magnetic compass sensors and higher quality GPS sensors to aid in the navigation, and possibly moving the navigational calculations to the larger single board computer on the ASVs to increase computational speed and accuracy.

Opportunities also exist to apply the knowledge gained from the existing system of ASVs and the software designed for them to other types of systems both homogeneous (fleets of flyers or land-based robots) and heterogeneous (see Smith et al., 2021). The potential to build and control larger vehicle systems using this research could allow for addressing some pressing issues in environmental monitoring and mitigation/remediation (oceanic hypoxic zones, open sea aquaculture, etc.) but equally important could be the applicability to much smaller scales as robots of increasingly small size are designed and constructed. Fleets (or swarms) of such small devices could have widespread application in biomedical uses and other areas. In all cases, the ability to correctly communicate and navigate are critical to the success of these groups of devices.
APPENDIX A. CAD DRAWINGS

The drawings in this appendix show the design and configuration of the double-hull autonomous vehicle design created at Louisiana State University for the fleet of autonomous surface vehicles.
APPENDIX B. ARDUINO® SOFTWARE SOURCE CODE

The Arduino source code presented in this appendix is the code utilized in the hybrid simulation environment to test algorithms for the fleet of autonomous surface vehicles. This code was created and updated between 2018 and 2021 to test the physical vehicles and was converted to work in the hybrid simulation environment. The primary change for the simulation was rerouting all of the sensor inputs/outputs through the USB port on the Arduino (known as Serial Port 1). The various functions presented here can be utilized by multiple ASV algorithms but the code within the main program loop is specific to the area patrol test. This code interacts with the Python code listed in Appendix C to perform the testing and recording of data for the hybrid simulation. Comments in the Arduino code are noted with the ‘//’ at the beginning of the comments which continue to the end of the line or ‘/* … */’ for comment blocks and which provide more detail about the functioning of particular portions of the code or provide update information.

// AREA PATROL TEST CODE

// Change Log:

//

// June 27, 2021:DDS: Radio working, made MasterFile with all three boats info in it. Version 12
// June 11, 2021:DDS: Added receiving and parsing radio messages from other ASVs (via Sim)
// June 06, 2021:DDS: Triple Boat program test, ASV = 0
// May 29, 2021:DDS: Remove xBee message send since it duplicates the log message.
// May 25, 2021:DDS: Test program for Sim to test area patrol with single boat.
// May 22, 2021:DDS: Added code to cause the boat program to wait for the simulation to be ready for data
// Apr 23, 2021:DDS: Replace mvASV commands with just bearing and speed.
// Sept 20, 2020: DDS: Rewriting for Simulation

// June 02, 2019: DDS: V11_Straighttest_BoatA - This is a test version of V11 that just runs the motors
in a straight line and logs the GPS data. Mostly testing the Micro control.


// May 11, 2018: DDS: V9 - Wrote routine to navigate to a point.

// May 6, 2018: DDS: V8 - Tested the validation section of parser and home location code in setup.

// May 5, 2018: DDS: V7 - Added the Haversine and Bearing routines to this code. Test compiled.

// Apr. 29, 2018: DDS: V6 - Adding routine to parse GPS data. Tested parsing of lat/lon. Need to test
validation code

// section of parser and bearing section of parser after installing on boat.


// Mar. 3, 2018: DDS: Added XBee code and tested one ASV to the Coordinator. Fixed one problem with
GPS code.


// Jan. 18, 2018: DDS: Created mvASV function to issue commands to the Micro for motor control.

// Jan. 18, 2018: DDS: RENAMED successful _20180117_I2C_Master_Mega to
_20180118_MEGA_ProgMaster_V1 (new naming convention

// _YYYYMMDD_##CPU##_ProgramName_V#

// Jan. 17, 2018: DDS: Modified the code from 20170716 to send multiple bytes from the master to
simulate

// an entire motor command of three bytes. TESTED Successfully.

// Libraries
#include <Wire.h> // Include the required Wire library for I2C
#include<math.h>  // Include the math library for the Haversine function and bearing calc.

// Pin assignments
// XBee Radio
const int xBeeIn = 19;
const int xBeeOut = 18;

// GPS
const int gpsDataIn = 17;
const int gpsDataOut = 16;

// Openlog datalogger
const int openLogIn = 15;
const int openLogOut = 14;

// Constants

const int thisASV = 0; // Store number for this ASV (this must be changed for each ASV (1,2,3,...)
// const int thisASV = 1; // Store number for this ASV (this must be changed for each ASV (1,2,3,...)
// const int thisASV = 2; // Store number for this ASV (this must be changed for each ASV (1,2,3,...)

const int turnTime = 77; // Number of milliseconds delay for the ASV to turn approximately 1 degree

// Global Variables

int destASV; // Stores number for destination ASV in messages (1,2,3,...=ASVs, 99=Broadcast, 100=Coordinator)
int lp,cmd,sp1,sp2;
int mtrCmd,mtrSpd1,mtrSpd2; // Variables to hold the motor command, and the speed (%) for each motor
int gpsbufflen, logbufflen, xbeebufflen; // Variables to hold length of strings in buffers
char gpsBuffer[80]; // Buffer for holding GPS data
char radioBuffer[80]; // Buffer for holding radio data
char logBuffer[120]; // Buffer for holding strings being written to the datalogger
char xBeeBuffer[120]; // Buffer for holding strings to be transmitted/received
// Location Variables for this ASV
float homeLat,homeLon; // Holds the Home location that the ASV returns to after the test run.
float currLat[3], currLon[3], currSpd[3], currBrng[3]; // Array to hold current lat/lon/speed/bearing for all ASVs

// Points Array for Boat 0 - Uncomment the following two lines for Boat 0

// Points Array for Boat 1 - Uncomment the following two lines for Boat 1

// Points Array for Boat 2 - Uncomment the following two lines for Boat 2

float prevLat, prevLon, prevSpd, prevBrng; // Previous lat/lon/speed/bearing for this ASV
char statGPS; // Stores status of most recently read GPS string (A=valid,V=not valid)
String dateGPS;
String timeGPS;
String logString; // Stores strings that need to be logged. Used to build custom log/Xbee messages.
String logNote; // Used to add debug notes to logStrings

// Variables for navigating to a point (or list of points)
int pointCount = 0;
float currPntDistance = 0; // Initialize current distance to point to be 0
float prevPntDistance = 1000000; // Initialize previous distance to be huge
int potErrPts;    // Counter for potentially erroneous distance measurements. Used to account for sudden
                  // erroneous GPS readings for the navToPoint routine.
float currPntBearing;

float width; // Width (E/W) in meters of bounding box
float height; // Height (N/S) in meters of bounding box
float trnBrng; // New Bearing to turn towards
int trnTime; // Time to let the ASV turn

// Added from Haversine function. Double check if these are necessary
float lat1, lat2, lon1, lon2; //Variables to hold two coordinates
float distance; // Variable to hold the distance between the two points
float bearing; // Variable to hold bearing from point 1 to point 2
const float EarthRadius = 6371000; // Mean radius of the earth in m
const float toRadians = 0.01745329251994329576923690768489; // Factor to convert DD to radians
const float toDegrees = 57.295779513082320876798154814105; // Factor to convert radians to DD
float DegInCircle = 360.0;
int currTime; // Time returned from the parseGPS routine. Used to set/compare times.
int currDate;       // Date returned from the parseGPS routine. Only used for logging purposes.

int testStartTime;  // Stores GPS time in minutes since 0000 when ASV acquires valid GPS
                     // string for the first time. Used to end test?

int runStartTime;   // Stores time in minutes since 0000 when the current iteration of
                     // the test started.

int runExecTime;    // Stores the number of minutes that this iteration should last.

int runCount;       // Stores which iteration is the current one.

//****************************************************************************
// setup:   Setup runs once when the program starts (also anytime the arduino
//          is rebooted) to handle one-time setup events that need to occur
//          before the main program begins to run.
//****************************************************************************

void setup() {

    pinMode(LED_BUILTIN, OUTPUT);

    // Initialize the default hardware serial port (the one attached to the USB cable)
    Serial.begin(9600);  // Start up the Serial port attached to the console.
    //Serial.println("MEGA: Inside setup"); //TEST message REMOVE for final program.

    // Initialize the first hardware serial port for the xBee radio
    Serial1.begin(9600);  //Start up the Serial port for the xBee radio

    /*
     COMMENTED OUT FOR SIMULATION SINCE ONLY THE PRIMARY SERIAL PORT IS USED
     (All Debug Serial statements have been commented out prior to this rewrite.)
     */

    // Initialize the first hardware serial port for the xBee radio
    Serial1.begin(9600);  //Start up the Serial port for the xBee radio
// Setup I/O pins for xBee
pinMode(xBeeIn, INPUT);
pinMode(xBeeOut, OUTPUT);

/*@*/

// Initialize the second hardware serial port (connected to the max232 for GPS)
Serial2.begin(19200);

// Setup I/O pins for GPS
pinMode(gpsDataIn, INPUT);
pinMode(gpsDataOut, OUTPUT);

// Initialize the third hardware serial port (connected to the openLog datalogger
Serial3.begin(9600);
pinMode(openLogIn, INPUT);
pinMode(openLogOut, OUTPUT);

// Start the I2C Bus as Master to enable communication with the arduino micro
Wire.begin();
*/

// Get the width and height of the Bounding Box. Used in checkBoundary function
width = Haversine(pointsLat[0],pointsLon[0],pointsLat[3],pointsLon[3]);
height = Haversine(pointsLat[0],pointsLon[0],pointsLat[1],pointsLon[1]);

// Loop in Setup until we get a valid GPS string
statGPS = 'V';
do
{  
    //Serial.println("No valid GPS yet");
    // sxBeeData("ASV1: No Valid GPS Data yet");
    getGPS(gpsBuffer); // Read a GPS string
    parseGPS(gpsBuffer); // Parse it just to check the status
} while (statGPS == 'V');

// Grab the Home coordinates and the testStartTime
homeLat = currLat[thisASV];
homeLon = currLon[thisASV];
testStartTime = currTime;

//delay(1000);
//Serial.print("Home position set: Lat = ");
//Serial.print(homeLat);
//Serial.print(" Lon = ");
//Serial.println(homeLon);
//Serial.print("Start Time = ");
//Serial.println(testStartTime);
//Serial.println();

    // Start the ASV moving
       //mvASV(5,0); // Get us moving fast forward and due North

    // Log that the ASV is starting its run
    // destASV = 99;
    // buildLog(thisASV, destASV, "Exiting setup and starting operation - Straight Line Test");
Serial1.println("ASV Exiting Setup");

} // End of Setup

//******************************************************************************
//******************************************************************************
// loop:  loop is the main program loop for the arduino. It loops as long as
// the arduino Mega 2560 has power.
//******************************************************************************

void loop() {

// Variable to see if there's valid GPS data

int count;

int goodGPS; // 1=good, 0=no data
int goodRadio; // 1=good, 0=no data
float newBrng; // Hold new bearing from checkBoundary routine

while (1) {
    mvASV(3, currBrng[thisASV]); // Keep going forward in case the program blew up somewhere in the middle

    // Get the GPS data from the Simulation
    goodGPS = 0;
gpsbufflen = 0;

    // Loop until we get good GPS data (should be instant with the simulation)
    do{
goodGPS = getGPS(gpsBuffer);
}
// Got the GPS Data, now parse it
Serial1.println(gpsBuffer);
parseGPS(gpsBuffer);
// Log the data (sends to sim for logging and simulated radio broadcast)
buildLog(thisASV, destASV, "GPS Logged");

// Get radio message from one of the other ASVs
goodRadio = 0;
count = 0;
// Tell Sim we're ready
Serial.println("Ready;");
Serial1.println("First Radio Msg");
// Loop until we get good Radio data (should be instant with the simulation)
do{
goodRadio = getRadio(radioBuffer);
count++;
} while (goodRadio == 0);
Serial1.println("End First Radio Msg");
// Got the Radio Data, now parse it
Serial1.print("First Radio Msg: ");
Serial1.println(radioBuffer);
parseRadio(radioBuffer);

// Serial1.print("Back from getRadio -- ");
// Tell Sim we're ready
Serial.println("Ready;");
Serial1.println("Second Radio Msg");

// Get radio message from one of the other ASVs
count = 0;
goodRadio = 0;

// Loop until we get good Radio data (should be instant with the simulation)
do{
goodRadio = getRadio(radioBuffer);
count++;
}while (goodRadio == 0);
// Serial1.print("Back from getRadio -- ");
Serial1.print("Second Radio Msg: ");
Serial1.println(radioBuffer);
parseRadio(radioBuffer);
Serial1.println("Back from parseRadio");

// Check the boundary conditions for this ASV
newBrng = checkBoundary(currLat[thisASV],currLon[thisASV],currBrng[thisASV]);
// buildLog(thisASV, destASV, "Test Log Entry");
// Serial1.println("After Test buildlog");
currBrng[thisASV] = newBrng;
}

} // End main program loop
// wtSim: This routine pauses the ASV program to wait until the Simulation is ready to receive data to prevent data loss. This is necessary since normally the ASV would just send data to external devices that have their own input buffers (xBee, openLog, etc.).

// INPUTS: The input to this routine is a character from the Simulation.

// OUTPUT: The output is the return that allows the ASV program to continue.

void wtSim(){

    // Local Variables

    String wtStr;
    int wtDone;

    wtDone = 0; // We're not done waiting for the Simulation to be ready

    Serial1.println("Beginning wait...");
    do
    {
        // Check to see if the serial port is available to read
if (Serial.available()>0) {
    wtStr = Serial.readString();
    //wtDone = Serial.findUntil("$R","n");
}
if ((wtStr[0]=='$') && (wtStr[1] == 'R')){
    wtDone = 1;
}
Serial1.println("waiting...");
} while(wtDone == 0);  // Loop until we've got the ready signal
Serial1.println("End of waiting...");
} // End of wtSim

//****************************************************************************
// mvASV: This routine takes a speed and bearing and constructs the command
// string to send to the simulation. Speed is from -5 to 5 in integer
// increments with 0 being stop. Bearing is compass bearing in degrees
// and is always a positive integer (0,360).
// INPUTS: The inputs to this routine are the speed and bearing parameters.
// OUTPUT: The output of this routine is the command string sent to the simulation.
//****************************************************************************
void mvASV(int mSpd, int asvBearing){

    // Local Variables
    String commandStr;
commandStr = "$MOV";
commandStr = commandStr + "," + String(thisASV) + "," + String(mSpd) + "," + String(asvBearing) + ",";

/* // Load cmdArray
   cmdArray[0] = mCmd;
   cmdArray[1] = mSpd1;
   cmdArray[2] = mSpd2;

   COMMENTING OUT I2C CODE FOR MOVING THE BOAT SINCE THIS WILL BE SENT OVER SERIAL
   
   TO THE SIMULATION.
   Wire.beginTransmission(9); // transmit to device #9
   Wire.write(cmdArray,3);    // sends the cmdArray
   Wire.endTransmission();    // stop transmitting
*/

   // Wait for the Simulation to be ready
   // wtSim();
   Serial.println(commandStr);  //Send movement command to the simulation.
   //Serial.println("Command sent");  // Test message to the console.

} // End of mvASV

.userdetails

/***********************************************
// getGPS: This routine listens for GPS data coming in from the max232 chip that
// is attached to the RS-232 serial Garmin GPS. It reads an entire GPS

86
// string into a character buffer of 80 characters.

// INPUTS: The inputs to this routine are the data coming in on gpsDataIn.

// OUTPUT: The output of this routine is the GPS string stored in the character
// array called gpsBuffer.

//*********************************************************************************/

int getGPS(char gpsBuffer[])
{
    char tempchar; // temporary character variable used to hunt for the '$' start

    // of the GPS string

    int gpsLoop;

    int gpsReturn=0; // Return value of function. Assume no data.

    String gpsString;

    // Wipe the buffer

    wipeBuffer(gpsBuffer,80);

    //Serial.println("Inside getGPS\n");

    /* COMMENTED OUT FOR SIMULATION

    // Check to see if the serial port is available to read

    if (Serial2.available()>0) {

        //Serial.println("In if statement\n");

        gpsReturn=1; // Have good data

        do {
            tempchar = Serial2.read();
        } while (tempchar != 10);

        // Read characters from the Serial2 port until a newline character or 80 bytes are read.

        gpsbufflen = Serial2.readBytesUntil(0x0A, gpsBuffer, 80);

    }*/

    87
// Check to see if the serial port is available to read
if (Serial.available()>0) {
    // Serial.println("In if statement\n");
    gpsReturn=1; // Have good data

    gpsString = Serial.readString();
    gpsString.toCharArray(gpsBuffer,80);

    // TEST print to see what was read
    // Serial.print("GPS = ");
    // Serial.println(gpsBuffer);
}

______________________________
} // End of getGPS

// getRadio: This routine grabs a simulated radio message from the simulation and
// stores it in a character buffer. It reads an entire Radio
// string into a character buffer of 80 characters.

// INPUTS: The inputs to this routine are the Radio data coming from the sim.

// OUTPUT: The output of this routine is the GPS string stored in the character
// array called radioBuffer.

int getRadio(char radioBuffer[])
{

    char tempchar; // temporary character variable used to hunt for the '$' start

    // of the GPS string

    int radLoop;

    char radchar;

    int chrcnt=0;

    int radReturn=0; // Return value of function. Assume no data.

    String radioString;

    // Wipe the buffer

    wipeBuffer(radioBuffer,80);

    Serial1.println("In getRadio");

    // Check to see if the serial port is available to read

    if (Serial.available()>0) {
        radReturn=1; // Have good data

        Serial1.println(" Reading...");

        radchar = Serial.read();

        while (radchar != \n){
            // Serial1.print(radchar);

            radioBuffer[chrcnt]=radchar;

            chrcnt++;

            radchar=Serial.read();

        }

        radioBuffer[chrcnt] = \n;

        //radioString = Serial.readString();

    }
//if (radioString.indexOf("\n") < 0){
  // radReturn = 0;
  //}
  //else {
Serial1.println("--Got data");
  // radioString.toCharArray(radioBuffer,radioString.indexOf("\n");
  //  }

Serial1.println(radioBuffer);
 }
  // Serial1.println("Exiting getRadio");
  return(radReturn);
} // End of getRadio

//******************************************************************************
//******************************************************************************
// buildLog: This routine builds a String object holding a log entry based on
//           data sent from the calling routine and then calls slogData and
//           sxBeeData to log the data to the datalogger and send it out via
//           the xBee radio link.
// INPUTS: The inputs to this routine are the source and destination ASVs as
//         integers and a String object representing a note to be added to
//         the log entry. Other items in the entry come from global GPS vars.
// OUTPUT: The output of this routine is the complete log entry as a String obj.
//******************************************************************************

void buildLog(int fromASV, int toASV, String logNote)
{ 
logString = "$ASV,";

logString = logString + fromASV + "," + toASV + "," + dateGPS + "," + timeGPS;
logString = logString + "," + String(currLat[thisASV],6) + "," + String(currLon[thisASV],6);
logString = logString + "," + String(currSpd[thisASV],3) + "," + String(currBrng[thisASV],3);
logString = logString + "," + logNote + ";";

//Serial.print("Log Entry = ");
//Serial.println(logString);
slogData(logString);

// sxBeeData(logString);  // Commented out for simulation; duplicates slogData

}  // End of buildLog

));//**********************************************
//**********************************************
//*******************************************************************************
// ologData: This routine takes a character buffer and the length of that buffer
// and writes the character buffer to the microSD card in the openLog
dataLogger.
//
// INPUTS: The inputs to this routine are the character array and the length of
// the message to be written to the log.
// OUTPUT: The output of this routine is the data written in the log file.
//*******************************************************************************

void ologData(char charBuffer[],int bufferSize)
{

// Wait for the Simulation

91
// wtSim();

//Serial.println("In ologData\n");
Serial.println(charBuffer);
// Serial3.println(charBuffer);
//Serial.println("Data written\n");

} // End of ologData

//******************************************************************************
//******************************************************************************

// slogData: This routine takes a String object and writes it to the microSD card in the openLog datalogger.

// INPUTS: The inputs to this routine are the String object

// OUTPUT: The output of this routine is the data written in the log file.

//******************************************************************************

void slogData(String logString)
{

    // Wait for the Simulation
    // wtSim();

    //Serial.println("In slogData\n");
    Serial.println(logString);
    // Serial3.println(logString);
    //Serial.println("Data written\n");

} // End of slogData
void xBeeData(char charBuffer[], int bufferSize){
  // Wait for the Simulation
  // wtSim();
  //Serial.println("In xBeeData.");
  Serial.println(charBuffer);
  // Serial1.println(charBuffer);
  //Serial.println("Data transmitted.");
} // End of xBeeData

// sxBeedata: This routine takes a String object and writes it to the xBea radio
// to be sent to all of the other ASVs and to the coordinator node.
//
// INPUTS: The inputs to this routine are the String object
// OUTPUT: The output of this routine is the data sent to the xBea radio.
void sxBeeData(String xBeeString){
    // Wait for the Simulation
    // wtSim();  //Serial.println("In sxBeeData.
    Serial.println(xBeeString);
    Serial1.println(xBeeString);
    //Serial.println("Data transmitted.
    }  // End of xBeeData

//*******************************************************************************
//*******************************************************************************

// wipeBuffer : This routine writes NULL into every entry of a character buffer
// to wipe it.
// INPUTS: The inputs to this routine are the buffer and its size.
// OUTPUT: The output of this routine is the wiped buffer.
//*******************************************************************************
*******************************************************************************

void wipeBuffer(char charBuffer[], int bufferSize)
{
    int cnt;  // loop control variable
    // Wipe string buffer by filling with the space (ASCII 32) character
    // to ensure it's empty before reading new data
    for (cnt=0; cnt<bufferSize; cnt++){
        charBuffer[cnt]=32;
    }
} // End of wipeBuffer
//*******************************************************************************
//********************************************************************
***********
// parseGPS : This routine parses the most recent GPS string and stores the
// extracted values in the appropriate variables.
// INPUTS: The inputs to this routine are the buffer and its size.
// OUTPUT: The output of this routine is the extracted data stored in the
// appropriate variables.
//*******************************************************************************

void parseGPS(char charBuffer[])
{
    int cnt;    // loop control variable
    int buffcursor;  // index of current character in buffer being examined.
    int commacnt;     // Counts number of commas passed in the buffer
    float tempcoord; // temp variable to hold a coordinate as we're building it from text
    float tempMMmmmm; // temp variable to hold the MM.mmmm portion of a coordinate
    float tempBrng;     // temp variable to hold the bearing
    float tempSpd;      // temp variable to hold the speed
    int   tempTime;     // temp variable to hold the current time HHMM converted into minutes since 0000.

    // temporary character arrays used to extract the various numbers from the GPS character array
    char templatchar[9] = "        ";
    char templonchar[9] = "        ";
    char tempbearing[6] = "     ";
    char tempspeed[6] = "     ";
    char tempTimechar[2] = " ";
///Serial.print("charBuffer = ");
///Serial.print(charBuffer);
///Serial.println("");

// Initialize the variables;

tempcoord = 0.0;

tempMMmmmm = 0.0;

tempBrng = 0.0;

tempSpd = 0.0;

tempTime = 0;

// Start at the beginning of the buffer

buffcursor = 0;

commacnt = 0;

while (commacnt < 1) // Skip to the time
{
    while (charBuffer[buffcursor] != 44)
    {
        buffcursor++;
    }

    commacnt++;
    buffcursor++;
}

// Extract the time. We may throw this away if the data isn't valid, but we have
// to grab it on the way to get the validation code.

timeGPS = "";
tempTimechar[0] = charBuffer[buffcursor]; // First digit of hour
timeGPS.concat(charBuffer[buffcursor]);    // Store first digit of hour in timeGPS string
buffcursor++;

tempTimechar[1] = charBuffer[buffcursor]; // Second digit of hour
timeGPS.concat(charBuffer[buffcursor]);    // Store second digit of hour in timeGPS string
buffcursor++;

tempTime = atoi(tempTimechar); // Convert two digit hour to integer
tempTime = tempTime * 60;       // Convert hours to minutes

tempTimechar[0] = charBuffer[buffcursor]; // First digit of minutes
timeGPS.concat(charBuffer[buffcursor]);    // Store first digit of minutes in timeGPS string
buffcursor++;

tempTimechar[1] = charBuffer[buffcursor]; // Second digit of minutes
timeGPS.concat(charBuffer[buffcursor]);    // Store second digit of minutes in timeGPS string
tempTime = tempTime + atoi(tempTimechar); // Convert minutes to integer and add them
buffcursor++;

timeGPS.concat(charBuffer[buffcursor]);    // Store first digit of seconds in timeGPS string
buffcursor++;

timeGPS.concat(charBuffer[buffcursor]);    // Store second digit of seconds in timeGPS string

// Skip to the comma after the time so we can get to the Validation code

commacnt = 0;

while (commacnt < 1)    // Skip to the validation code
{
    while (charBuffer[buffcursor] != 44)
    {
        buffcursor++;
    
}
}  
commacnt++;  
buffcursor++;  
}

if (charBuffer[buffcursor] == 'A') // Make sure the GPS string is valid data
{
// We have valid data so store the time we computed in currTime  
currTime = tempTime;  
statGPS = charBuffer[buffcursor];  

// Extract the Latitude  
buffcursor= buffcursor + 2; // Start after the comma after the Status  
templatchar[0] = charBuffer[buffcursor]; //First digit of Latitude  
buffcursor++;  
templatchar[1] = charBuffer[buffcursor]; //Second digit of Latitude  
tempcoord = atof(templatchar);  
//Serial.print("Latitude Degrees = ");  
//Serial.println(tempcoord);  
cnt = 0;  
while (charBuffer[buffcursor] != 44)
{
  templatchar[cnt] = charBuffer[buffcursor];  
  cnt++;  
  buffcursor++;  
}
tempMMmmmm = atof(templatchar);  // Convert the MM.mmmm part into a float;

//Serial.print("MM.mmmm = ");
//Serial.println(tempMMmmmm);

tempcoord = tempcoord + (tempMMmmmm/60);  // Convert decimal minutes to DD and add to degrees

prevLat = currLat[thisASV]; // Save the previous current latitude

currLat[thisASV] = tempcoord;  // Store the new current latitude

tempcoord = 0.0;

tempMMmmmm = 0.0;

// Extract the Longitude

buffcursor = buffcursor + 4;  // Start at the beginning of the Longitude (skip first digit which is zero here)

templonchar[0] = charBuffer[buffcursor]; //First digit of Longitude

buffcursor++;

templonchar[1] = charBuffer[buffcursor]; //Second digit of Longitude

buffcursor++;

tempcoord = atof(templonchar);  // Convert the integer portion to a float

//Serial.print("Longitude Degrees = ");
//Serial.println(tempcoord);

cnt = 0;

while (charBuffer[buffcursor] != 44)
{

templonchar[cnt] = charBuffer[buffcursor];

cnt++;
}
buffcursor++;  // Consume the length of the ",", so that we can read the MM.mmmm part.
}

float tempMMmmmm = atof(templonchar); // Convert the MM.mmmm part into a float;

//Serial.print("MM.mmmm = ");
//Serial.println(tempMMmmmm);

tempcoord = tempcoord + (tempMMmmmm/60); // Convert decimal minutes to DD and add to degrees

tempcoord = tempcoord * -1; // Make the longitude negative because in this hemisphere it's West.

prevLon = currLon[thisASV]; // Save the previous current longitude
currLon[thisASV] = tempcoord; // Store the new current longitude

// Extract Speed and Heading and store them. More complex because some of the fields in the field after
// the time and coordinates may not exist so we have to count commas starting with the speed.

prevBrng = currBrng[thisASV]; // Save the current bearing as the previous bearing
currBrng[thisASV] = 0; // Assume we're stopped so there is no bearing
prevSpd = currSpd[thisASV]; // Save the current speed as the previous speed
currSpd[thisASV] = 0; // Assume we're stopped so there is no speed

buffcursor = buffcursor+3; // Jump to first digit of speed.
cnt = 0;

// Check to see if there's a speed present

while (charBuffer[buffcursor] != 44) // Character isn't a comma so there's a speed; grab it.
{
    tempspeed[cnt] = charBuffer[buffcursor];
    cnt++;
}
buffcursor++;

} // while to grab speed

if (cnt > 0) // There was a speed present to convert it and store it.
{
    tempSpd = atof(tempspeed);
    currSpd[thisASV] = tempSpd;
}

// We've either grabbed the speed or there wasn't one. Either way, the next thing is the bearing
// Attempt to extract it.
buffcursor++; // Skip the comma after speed
cnt = 0; // index for temporary character array

// If there's no bearing present, the character under the cursor will be a comma so this while
// will not execute
while (charBuffer[buffcursor] != 44)
{
    tempbearing[cnt] = charBuffer[buffcursor];
    cnt++;
    buffcursor++;  
}

// We need to see if the cnt is still zero. If so, there was no bearing so do nothing.
// If the cnt is > zero, then convert and store the new bearing.
if (cnt > 0) // There was a bearing so convert it and store it.
{  
    tempBrng = atof(tempbearing);
    currBrng[thisASV] = tempBrng;
}

// Grab the date from the GPS string

dateGPS = "";
buffcursor++;// Skip the comma after the bearing

for (cnt=0; cnt<6; cnt++)
{
    dateGPS.concat(charBuffer[buffcursor]);
    buffcursor++;
}

/* COMMENTED OUT FOR SIMULATION */

// TEST PRINT ALL DATA CONVERTED

Serial.print("GPS Time = ");
Serial.println(timeGPS);
Serial.print("Time in minutes = ");
Serial.println(tempTime);
Serial.print("Latitude = ");
Serial.println(currLat[thisASV],6);
Serial.print("Longitude = ");
Serial.println(currLon[thisASV],6);
Serial.print("Speed = ");
Serial.println(currSpd[thisASV],4);
Serial.print("Bearing = ");
Serial.println(currBrng[thisASV],4);
Serial.print("GPS Date = ");
Serial.println(dateGPS);
*/

} // End if for Valid data
else
{
    //Serial.println("No valid GPS data found in gpsBuffer.");
}

} //End of parseGPS

******************************************************************************
// parseRadio : This routine parses a radio communication string from the sim
// and stores the extracted values in the appropriate variables.
// INPUTS: The inputs to this routine are the buffer and its size.
// OUTPUT: The output of this routine is the extracted data stored in the
//         appropriate variables.
******************************************************************************
void parseRadio(char charBuffer[])
{
    int cnt; // loop control variable
    int buffcursor; // index of current character in buffer being examined.
int commacnt;  // Counts number of commas passed in the buffer
float tempcoord; // temp variable to hold a coordinate as we're building it from text
float tempMMmmmm; // temp variable to hold the MM.mmmm portion of a coordinate
float tempBrng;   // temp variable to hold the bearing
float tempSpd;    // temp variable to hold the speed
int    tempASV;   // temp variable to hold the source ASV number

// temporary character arrays used to extract the various numbers from the GPS character array
char templatchar[9] = "      ";
char templonchar[9] = "      ";
char tempbearing[6] = "      ";
char tempspeed[6] = "      ";
char tempASVchar[3] = "   ";

// Initialize the variables;
tempcoord = 0.0;
tempMMmmmm = 0.0;
tempBrng = 0.0;
tempSpd = 0.0;
tempASV = 0;

Serial1.println("Parsing the radio message...");

// Start at the beginning of the buffer
buffcursor = 0;
commacnt = 0;

while (commacnt < 1)   // Skip to the Source ASV
{  
while (charBuffer[bufcursor] != 44)  
{  
  buffcursor++;  
}  
commacnt++;  
buffcursor++;  
}

cnt = 0;  
while (charBuffer[bufcursor] != 44)  
{  
tempASVchar[cnt] = charBuffer[bufcursor];  
cnt++;  
buffcursor++;  
}  
tempASV = atoi(tempASVchar);  // Convert the source ASV digit into an integer  
Serial1.print("Source ASV = ");  
Serial1.print(tempASV);  
buffcursor++;  //Skip the comma after the source ASV

// Skip to the comma after the time so we can get to the Latitude  
commacnt = 0;  
while (commacnt < 3)  // Skip to the latitude  
{  
while (charBuffer[bufcursor] != 44)
{ 
    buffcursor++;
}
commacnt++;
buffcursor++;
}

// Extract the Latitude
cnt = 0;
while (charBuffer[buffcursor] != 44)
{
    templatchar[cnt] = charBuffer[buffcursor];
    cnt++;
    buffcursor++;
}
currLat[tempASV] = atof(templatchar); // Convert the latitude into a float;
Serial1.print(" , Lat = ");
Serial1.println(currLat[tempASV],6);
Serial1.print("Character under buffcursor is: ");
Serial1.println(charBuffer[buffcursor]);

    // Extract the Longitude
    Serial1.println("Skipping Comma");
    buffcursor = buffcursor + 1; // Skip the comma after the latitude
    Serial1.print("Character now under buffcursor is ");
    Serial1.println(charBuffer[buffcursor]);
    cnt = 0;
Serial1.print("nStarting Longitude conversion - ");

while (charBuffer[ buffcursor ] != 44)
{
    // Serial1.print("Inside while for longitude ");
    templonchar[cnt] = charBuffer[ buffcursor ];
    // Serial1.print(cnt);
    // Serial1.print(" th character is: ");
    // Serial1.println(templonchar[ cnt ]); 
    cnt ++;
    buffcursor ++;
}

currLon[tempASV ] = atof(templonchar); // Convert the MM.mmmm part into a float;
Serial1.print(", Long = ");
Serial1.println(currLon[ tempASV ], 6);

// Extract Speed and Heading and store them. More complex because some of the fields in the field after
// the time and coordinates may not exist so we have to count commas starting with the speed.

buffcursor = buffcursor + 1; // Skip comma after Longitude

cnt = 0;

// Check to see if there's a speed present

while (charBuffer[ buffcursor ] != 44) // Character isn't a comma so there's a speed; grab it.
{
    tempspeed[ cnt ] = charBuffer[ buffcursor ];
    cnt ++;
    buffcursor ++;
}
if (cnt > 0)  // There was a speed present to convert it and store it.
{
    tempSpd = atof(tempspeed);
    currSpd[tempASV] = tempSpd;
}
Serial1.print(" , Speed = ");
Serial1.print(currSpd[tempASV],3);

// We've either grabbed the speed or there wasn't one. Either way, the next thing is the bearing
// Attempt to extract it.
buffcursor++;   // Skip the comma after speed
cnt = 0;  // index for temporary character array

// If there's no bearing present, the character under the cursor will be a comma so this while
// will not execute
while (charBuffer[buffcursor] != 44)
{
    tempbearing[cnt] = charBuffer[buffcursor];
    cnt++;
    buffcursor++;
}

// We need to see if the cnt is still zero. If so, there was no bearing so do nothing.
// If the cnt is > zero, then convert and store the new bearing.
if (cnt > 0) // There was a bearing so convert it and store it.
{
    tempBrng = atof(tempbearing);
    currBrng[tempASV] = tempBrng;
}
Serial1.print(" , Bearing = ");
Serial1.println(currBrng[tempASV],3);
Serial1.println("---Done parsing");
} //End of parseRadio

*******************************************************************************
*******************************************************************************
//****************************
*************************************************
//**                           NAVIGATION FUNCTIONS                            **
*******************************************************************************
//****************************************
*******************************************************************************

Haversine: This routine computes the Great Circle distance between two points.
// Note that this is an approximate distance since we assume the earth is a sphere. For distances as small as the ASVs encounter, this is fine. (We could actually just assume the earth is flat over these distances.)
// INPUTS: The inputs to this routine are the latitude/longitude for the two points.
// OUTPUT: The output from this function is the distance in meters returned as the function value.
float Haversine(float lat1, float lon1, float lat2, float lon2) {
    float a, c, dlat, dlon, dist;
    float rlat1, rlat2, rlon1, rlon2; //Variables to hold radian conversions of DD

    // Convert lat/lon to radians
    rlat1 = lat1 * toRadians;
    rlon1 = lon1 * toRadians;
    rlat2 = lat2 * toRadians;
    rlon2 = lon2 * toRadians;

    // Compute deltas
    dlat = (lat2-lat1) * toRadians;
    dlon = (lon2-lon1) * toRadians;

    // Compute intermediate results a and c
    a = square(sinf(dlat/2))+(cosf(rlat1)*cosf(rlat2)*(square(sinf(dlon/2))));
    c = 2 * atan2(sqrt(a),sqrt(1-a));
    dist = EarthRadius * c;

    return(dist);
}

} // End of Haversine

/*******************************************************************************/
//*******************************************************************************
// Bearing: This routine computes the bearing between two points.
// INPUTS: The inputs to this routine are the latitude/longitude for the two points.
// OUTPUT: The output from this function is the bearing in degrees returned as the
//         function value.
//*******************************************************************************

float Bearing(float lat1, float lon1, float lat2, float lon2)
{
    float y, x, dlat, dlon, brng;
    float rlat1, rlat2, rlon1, rlon2; // Variables to hold radian conversions of DD

    // Convert lat/lon to radians
    rlat1 = lat1 * toRadians;
    rlon1 = lon1 * toRadians;
    rlat2 = lat2 * toRadians;
    rlon2 = lon2 * toRadians;

    // Compute delta longitude
    dlon = (lon2 - lon1) * toRadians;

    // Compute intermediate results a and c
    y = sinf(dlon) * cosf(rlat2);
    x = (cosf(rlat1)*sinf(rlat2))-(sinf(rlat1)*cosf(rlat2)*cosf(dlon));
    brng = atan2f(y,x) * toDegrees;

    // Fix bearing to compass bearings
    // brng = fmod((brng+360),360.0);
return(brng);
}

//******************************************************************************
//******************************************************************************
// navToPoint:   T
// This routine computes the bearing between two points.
// INPUTS: The inputs to this routine are the latitude/longitude for the two points.
// OUTPUT: The output from this function is the bearing in degrees returned as the
// function value.
//******************************************************************************

void navToPoint(float lat1, float lon1, float lat2, float lon2)
{
    // Local Variables
    String logNoteTemp;
    // Figure out how far we are to the point we're trying to get to and the bearing to that point
    currPntDistance =
    Haversine(currLat[thisASV],currLon[thisASV],pointsLat[pointCount],pointsLon[pointCount]);
    currPntBearing =
    Bearing(currLat[thisASV],currLon[thisASV],pointsLat[pointCount],pointsLon[pointCount]);

    // Start building the logNote to add to this iteration's log entry
    destASV = 99;
    logNoteTemp = " Point=";
    logNoteTemp = logNoteTemp + pointCount + ",Distance=" + currPntDistance + ",Bearing=";
logNoteTemp = logNoteTemp + currPntBearing;

// Check where we are

if ((currPntDistance <= prevPntDistance) && (currPntDistance > 2)) {
    // If we get here, we're still getting closer to the point and we're more than 2m from the point
    logNoteTemp = logNoteTemp + ", CLOSER but >2m:";

    prevPntDistance = currPntDistance;  // Store the current distance as the previous distance for the next loop

    mvASV(5,currPntBearing);
}

/*
// Check to see if we're pointed in the right direction

if ((currBrng[thisASV] > (currPntBearing+2)) || (currBrng[thisASV] < (currPntBearing - 2))) {
    // We're not heading in the right direction so turn to the new bearing

    if (currBrng[thisASV] > (currPntBearing+2)) {
        logNoteTemp = logNoteTemp + "Turning left";
        mvASV(11,50,100); // Start turning left
    } // End of if for turn

    else if (currBrng[thisASV] < (currPntBearing - 2)) {
        logNoteTemp = logNoteTemp + "Turning right";
        mvASV(11,100,50); // Start turning right
    }
*/
else
{
    logNoteTemp = logNoteTemp + "Going straight";
    mvASV(11,100,100); // We're heading toward the point +/- 2 degrees so just go straight
}
} // End of if for pointed in the right direction
*/
} // End of if for still heading to point and outside 2m radius circle around point
else // For some reason the outer if failed so we're either within the 2m circle or we're moving away from the point
{
    if (currPntDistance <= 2) // Made it to the point
    {
        logNoteTemp = logNoteTemp + ", Within 2m ";
        if (currPntDistance < prevPntDistance) // We're still moving towards the point
        {
            logNoteTemp = logNoteTemp + "Still approaching ";
            prevPntDistance = currPntDistance; // Store the current distance as the previous distance for the next loop
        }
        mvASV(5,currPntBearing);
    } /*
    //Check to see if we're pointed in the right direction
    if ((currBrng[thisASV] > (currPntBearing+2)) || (currBrng[thisASV] < (currPntBearing - 2)))
    { // We're not heading in the right direction so turn to the new bearing
        if (currBrng[thisASV] > (currPntBearing+2))
{ 
    logNoteTemp = logNoteTemp + "Turning left";
    mvASV(11,50,100); // Start turning left
} // End of if for turn
else if (currBrng[thisASV] < (currPntBearing - 2))
{
    logNoteTemp = logNoteTemp + "Turning right";
    mvASV(11,100,50); // Start turning right
}
else
{
    logNoteTemp = logNoteTemp + "Going straight";
    mvASV(11,100,100); // Keep going straight
} //End of else for turn
} // End of if for pointed in the right direction
*/
} // End of If for still approaching the point
else // We got within 2 meters, but we're past the point now; go to the next one
{
    logNoteTemp = logNoteTemp + ",Got within 2m: Now past: Moving to next point";
    pointCount++; // Go to the next point
    prevPntDistance = 1000000; // Reset the previous distance to be huge to restart the navigating
} // End of else for having past the point
} // End of If for being inside the 2m radius circle around the point
else
{
    // We have an increased distance to the point. Is this an erroneous GPS reading? If this is

// the first such increased distance, let's go through again to be sure it wasn't an error.
if (potErrPts < 1) // First time we get a greater distance, just log it.
{
    logNoteTemp = logNoteTemp + ", Possibly missed the point so check one more time";
    potErrPts++; // Count the miss
}
else // We missed the point by more than 2m and we're moving away now. Give up and go to next point
{
    logNoteTemp = logNoteTemp + ", Missed by more than 2m: Moving to next point";
    pointCount++; // Go to the next point
    prevPntDistance = 1000000; // Reset the previous distance to restart navigating
    potErrPts = 0; // Reset counter for potentially erroneous GPS readings
}
} // End of else - we missed the point by more than 2m
} // End of else for outer if (we were either moving away or were within 2m)

// Write the log entry for this iteration of navToPoint
buildLog(thisASV,destASV,logNoteTemp);

} // End of navToPoint

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

// checkBoundary: This routine checks to see if the ASV is within 3 meters of any side of the bounding box within which it should be patrolling.
If so, it computes the angle the ASV should turn to move away from the boundary. It introduces some randomness to the turn so that the ASV doesn't just follow the boundary around.

INPUTS: The inputs to this routine are the current latitude/longitude/bearing of the ASV.

OUTPUT: The output from this function is the bearing in degrees that the ASV needs to turn in degrees. If the ASV doesn't need to turn, the function returns the current bearing that was sent into the function.

float checkBoundary(float lat1, float lon1, float brng1)
{
    // Local Variables

    // Flags for boundary and collision events they are in order:
    // North, South, East, West, Out of box, ASV0, ASV1, ASV2
    int violations[8]={0,0,0,0,0,0,0,0}; // Assume no violations
    int asvLoop; //Looping variable to compare ASV distances
    int quad;
    float newBearing = brng1; // Assume we don't need to turn
    float toWest, toEast, toNorth, toSouth; // Holds distances from ASV to bounding sides
    float rndDegrees; // Random factor for the turns
    String logNoteTemp;

    // Start building the logNote to add to this iteration's log entry
    destASV = 99;
    logNoteTemp = "Boundary Check ";

    return newBearing;
}
// Figure out randomness for turns

rndDegrees = random(1,90);

// See which bearing quadrant we're in: 0-90, 90-180, 180-270, or 270-360
quad = brng1/90; // Since quad is an integer, we'll only get the integer
    // portion of the division (effectively a div operation).

quad++; // May need this.

// Add quad to logNote

logNoteTemp = logNoteTemp + ""," + quad;

// See if the ASV is within all four boundaries

// First compute the distance to each boundary edge

toWest = Haversine(currLat[thisASV],currLon[thisASV],currLat[thisASV],pointsLon[0]);
toEast = Haversine(currLat[thisASV],currLon[thisASV],currLat[thisASV],pointsLon[2]);
toSouth = Haversine(currLat[thisASV],currLon[thisASV],pointsLat[0],currLon[thisASV]);
toNorth = Haversine(currLat[thisASV],currLon[thisASV],pointsLat[1],currLon[thisASV]);

// Log distances to each boundary

    // logNoteTemp = logNoteTemp + "," + toNorth + "," + toSouth;
    // logNoteTemp = logNoteTemp + "," + toEast + "," + toWest;

if ((toSouth < height) && (toNorth < height) && (toWest < width) && (toEast < width))
{ // Bounding box if

// We're inside the box, now see if we are getting too close to an edge
if (toSouth < 3.0) // Within 3 meters of the South edge of the bounding box
{
  // We might be too close to the east/west edges too. Check
  violations[1] = 1; // Flag south boundary violation

  if (quad == 2)
  {
    
    if (toEast < 3.0)
    {
      violations[2] = 1; // Flag east boundary violation

      if (currBrng[thisASV] < 135)
      {
        
        newBearing = 360 - rndDegrees;
      
      }
      else
      {
        
        newBearing = 270 + rndDegrees;
      }
      
    }
    
  }
  else
  {
    
    newBearing = 90 - rndDegrees;
  }
  
}
else
{
  
}
}
if (toWest < 3.0)
{
    violations[3] = 1; // Flag west boundary violation
    if (currBrg[thisASV] < 225)
    {
        newBearing = 90 - rndDegrees;
    }
    else
    {
        newBearing = 0 + rndDegrees;
    }
}
else
{
    newBearing = 270 + rndDegrees;
}
}

if (toNorth < 3.0) // Within 3 meters of the north edge of the bounding box
{
    violations[0] = 1; // Flag north boundary violation
    if (quad = 1)
    {
        if (toEast < 3.0)
        {
            violations[2] = 1; // Flag east boundary violation
        }
if (currBrng[thisASV] < 45)
{
    newBearing = 270 - rndDegrees;
}
else
{
    newBearing = 180 + rndDegrees;
}
}
else
{
    newBearing = 90 + rndDegrees;
}
}
else
{
    if (toWest < 3.0)
    {
        violations[3] = 1; // Flag west boundary violation
        if (currBrng[thisASV] > 315)
        {
            newBearing = 90 + rndDegrees;
        }
    }
    else
    {
        newBearing = 180 - rndDegrees;
    }
else
{
    newBearing = 270 - rndDegrees;
}
}
}

// We're not too close to the north or south edges, now just check west and east
if (toWest < 3.0)  // Within 3 meters of the west edge
{
    violations[3] = 1; // Flag west boundary violation
    if (quad = 3)
    {
        newBearing = 180 - rndDegrees;
    }
    else
    {
        newBearing = 0 + rndDegrees;
    }
}
else
{
    violations[2] = 1; // Flag east boundary violation
}
if (quad = 1)
{
    newBearing = 360 - rndDegrees;
}
else
{
    newBearing = 180 + rndDegrees;
}
}
} // End of if inside Bounding Box
else
{
    // Big problems, we're outside the bounding box
    // Set the bearing to the center point of the box (pointsLat[4],pointsLon[4])
    violations[4] = 1;
    newBearing = Bearing(currLat[thisASV],currLon[thisASV],pointsLat[4],pointsLon[4]);
}

// For loop to check for simulated collision events - getting closer than 6 meters
// to either of the other ASVs
for (asvLoop=0; asvLoop<3; asvLoop++){
    if (asvLoop != thisASV){ // Cannot collide with yourself
        if (Haversine(currLat[thisASV],currLon[thisASV],currLat[asvLoop],currLon[asvLoop]) < 6){
            // Simulated collision between thisASV and asvLoop has occurred, flag it
            violations[asvLoop+5] = 1;
            // Set the bearing to go back towards the center of our own box
newBearing = Bearing(currLat[thisASV], currLon[thisASV], pointsLat[4], pointsLon[4]);

// Add new bearing to the logNote and log it.
logNoteTemp = logNoteTemp + "." + newBearing;
logNoteTemp = logNoteTemp + "," + violations[0] + "," + violations[1] + "," + violations[2];
logNoteTemp = logNoteTemp + "," + violations[6] + "," + violations[7];
buildLog(thisASV, destASV, logNoteTemp);

return newBearing;

} // End of checkBoundary

//******************************************************************************
APPENDIX C. PYTHON™ SIMULATION SOFTWARE SOURCE CODE

The Python™ code presented in this appendix represents the simulated environment within which the three simulated Arduino® ASVs exist in the hybrid simulation. The purpose of this code is to provide all sensor inputs to the three ASVs by simulating GPS readings and other necessary inputs; receive all outputs from the ASVs in the form of commands to various subsystems such as logging data for later analysis and providing simulated movement; and to simulate the radio network between the ASVs by passing messages between the ASVs.

# SimTripleBoat_AreaPatrol_Radio.py
# Simulation for the three boat area patrol problem.
# Still needs wind, better turning, radio messages working.
# Corresponding Arduino program:
#_20210627_MEGA_SimProgMaster_V13_TripleBoat_AreaPatrol_Radio_MF

import serial
import time
import random
import math
# import numpy as np  # Not sure I need this any longer.

# Define the serial port and baud rate.
# Ensure the 'COM#' corresponds to what was seen in the Windows Device Manager
ser1 = serial.Serial('COM3', 9600, ) # Boat 0 = Serial 1
ser2 = serial.Serial('COM5', 9600, ) # Boat 1 = Serial 2
ser3 = serial.Serial('COM6', 9600, ) # Boat 2 = Serial 3
# Global Variables

# --Not needed at the moment--

# with open('MapFiles\maptest1.csv', 'r', encoding='utf-8-sig') as mf:
#    maparray = np.genfromtxt(mf, delimiter=',',

# Set Global time variables for the test run. All runs start at 070000.

tHour=7

# Global ASV variables

asvLats = [30.408465, 30.408465, 30.408465]  # Start each ASV in the center of its box

asvLongs = [-91.168607, -91.167121, -91.165634]

asvBrngs = [0.00, 0.00, 0.00]

asvSpds = [0.00, 0.00, 0.00]

asvSense = [0.00, 0.00, 0.00]

# maparray = np.genfromtxt('MapFiles\maptest1.csv', delimiter=',',

# Constants for calculations

EarthRadius = 6371  # Radius of earth in km

TurnSpeed = 30  # Time in seconds for an ASV to rotate through 360 degrees.

#******************************************************************************
#    MESSAGE ROUTINES - Routines to parse messages and build messages
#******************************************************************************

126
# Function to parse a motor command message received from one of the ASVs in
# the simulation and populate the resulting speed and angle of travel over
# the next second of simulation time.

#***************************************************************************
# Function to parse a motor command message received from one of the ASVs in
# the simulation and populate the resulting speed and angle of travel over
# the next second of simulation time.
#***************************************************************************

def mtrParse(mtrCmdStr):

    print(mtrCmdStr)
    # Parse the motor command. We don't care about the command or the ID
    mtrCmd, strID, strSpeed, strRest = mtrCmdStr.split(b',')
    print(mtrCmd)
    print(strID)
    print(strSpeed)
    strAngle, strJunk = strRest.split(b';')
    print(strAngle)
    asvSpeed = float(strSpeed)
    asvAngle = float(strAngle)
    return asvSpeed, asvAngle

    # End of mtrParse

#***************************************************************************
# Function to parse the radio message received from one of the ASVs in
# the simulation and recreating the message to send to the other ASVs.
#***************************************************************************
def radioParse(radioStr):

    sASVLbl, sSrcID, sDestID, lgDate, lgTime, sASVlat, sASVlong, strSpeed, strAngle, radMsg = radioStr.split(b',')
    asvID = str(int(sSrcID))
    destID = str(int(sDestID))
    asvDate = str(int(lgDate))
    asvTime = str(int(lgTime))
    asvLat = str(float(sASVlat))
    asvLong = str(float(sASVlong))
    asvSpeed = str(float(strSpeed))
    asvAngle = str(float(strAngle))

    sRadMsg = sASVLbl + "," + asvID + "," + destID + "," + asvDate + "," + asvTime + "," + asvLat + ","
    + asvLong + "," + asvSpeed + "," + asvAngle + "," + radMsg + \"n"

    RadMsg = bytes(sRadMsg, 'utf-8')
    return RadMsg

# End of radioParse

#***************************************************************
# Function to build a valid GPS RMC NMEA sentence to be sent to the ASV.
#
#  INPUT: Fields necessary to create a new GPS string. Fields are in the
#    following formats:
#    currHours,currMinutes,currSeconds = integer values

128
# currLat, currLon = Decimal Degrees floating point (with two digit degrees)
# currBearing, currSpeed = floating point
#
# Other info: We know that for this simulation the following constants:
# Latitude will always be 'N' and Longitude will always be 'W'
# Status will always be 'A'
# Date is a constant (it doesn't matter; only simulating a few hours.)
# Magnetic Variation isn't taken into account so is a constant 0.000, W
# Mode is always 'E'
# Checksum is ignored so is a constant string value
#
# OUTPUT: String containing a virtual $GPSRMC string for the vehicle.

# Pad time fields with leading zeros if necessary and build the time
if (currHours < 10):
    hrStr = "0" + str(currHours)
else:
    hrStr = str(currHours)
if (currMinutes < 10):
    mnStr = "0" + str(currMinutes)
else:
    mnStr = str(currMinutes)
if (currSeconds < 10):
    secStr = "0" + str(currSeconds)
else:
    secStr = str(currSeconds)
    timeStr = hrStr + mnStr + secStr

    # Pad Lat / Long values if necessary and convert to strings
    gpsLat, gpsLong = conDD2GPS(currLat, currLon)
    latStr = str(round(gpsLat, 6))  # Needs no padding
    lonStr = "0" + str(round(gpsLong, 6))  # Needs one zero padding

    # Convert Speed and Bearing to strings
    spdStr = str(round(currSpeed, 1))
    brnStr = str(round(currBearing, 1))

    # Build the new gps string
    gpsStr = "$GPRMC," + timeStr + ",.A," + latStr + ",N," + lonStr + ",.W," + spdStr + ",." + brnStr + ",110421,0.00,0.00,E*6F"

    return gpsStr

    # End of buildGPS

#******************************************************************************
# Function to convert GPS formatted lat/long (DDMM.mmmm) to Decimal Degrees
#   DD.dddddddd
#******************************************************************************

def conGPS2DD(gpsLat, gpsLong):

ddLat = int(gpsLat/100) + ((gpsLat%100)/60)

ddLong = -1 *(int(abs(gpsLong)/100) + ((abs(gpsLong)%100)/60))

return ddLat, ddLong

# End of conGPS2DD

#******************************************************************************
# Function to convert Decimal Degree formatted lat/long DD.dddddd to GPS-
#    formatted lat/long format DDMM.mmmm
#******************************************************************************

def conDD2GPS(ddLat, ddLong):

gpsLat = (int(ddLat)*100) + ((ddLat- int(ddLat))*60)

gpsLong = int(abs(ddLong))*100 + ((abs(ddLong)-(int(abs(ddLong))))*60)

return gpsLat, gpsLong

# End of conDD2GPS

#******************************************************************************
# Function to take the current artificial time and increment it to get the
#    new time. Assumes all tests run within the same day so no need for a
#    rollover if for the hours to increment the date.
#******************************************************************************
def incTime(currHours, currMinutes, currSeconds):
    currSeconds += 1  # Increment the seconds
    if(currSeconds > 59):
        currMinutes += 1  # Seconds rolled over so increment the minutes
        currSeconds = 0  # Reset the seconds counter
        print(f"Seconds rollover - time is {str(currHours)}:{str(currMinutes)}:{str(currSeconds)}")
    if (currMinutes > 59):
        currHours += 1  # Minutes rolled over so increment the hours
        currMinutes = 0  # Reset the minutes counter
        print(f"Minutes rollover - time is {str(currHours)}:{str(currMinutes)}:{str(currSeconds)}")
    return currHours, currMinutes, currSeconds  # Send back the new time

# End of incTime

#****************************************************************************
# Function to compute the new GPS coordinates of the ASV given the previous
#    GPS coordinates, the bearing, and the speed of the ASV to be used to
#    construct the next GPS RMC NMEA sentence to be sent to the ASV.
#    Haversine formulas taken from https://www.movable-type.co.uk/scripts/latlong.html
#**************************************************************************

def calcNewCoords(currLat, currLon, currSpeed, currBearing):
    print("Inside calcNewCoords")
    # Compute distance traveled based on speed and convert it to angular distance
    dist = currSpeed * 0.0005144  # Time is always 1 second so this converts to kilometers
angDist = dist/EarthRadius

# Convert Lat and Lon to radians
radLat = math.radians(currLat)
radLon = math.radians(currLon)
radBrng = math.radians(currBearing)

# Compute the new Lat and Lon in Radians
radLat2 = math.asin((math.sin(radLat)*
math.cos(angDist))+(math.cos(radLat)*math.sin(angDist)*math.cos(radBrng)))
radLon2 = radLon +
math.atan2((math.sin(radBrng)*math.sin(angDist)*math.cos(radLat)),(math.cos(angDist)-
(math.sin(radLat)*math.sin(radLat2))))

# Convert the new Lat / Lon back to Degrees
newLat = math.degrees(radLat2)
newLong = math.degrees(radLon2)

# Figure out the new bearing
# First, get the bearing from the endpoint to the origin point
newBearing = calcPtsBearing(radLat2, radLon2, radLat, radLon)
# Then reverse it to get the new bearing
newBearing = (newBearing+180)%360

#Print values before returning
print("Done calcNewCoords")
print(newLat)
print(newLong)
print(newBearing)
print("Returning")

return newLat, newLong, newBearing

# End of calcNewCoords

#******************************************************************************
# Function to take calculate the great circle bearing between two points.
# - All coordinates must be in radians.
#******************************************************************************
def calcPtsBearing(lat1, lon1, lat2, lon2):

    y_coord = math.sin(lon2-lon1)*math.cos(lat2)
    x_coord = (math.cos(lat1)*math.sin(lat2))-(math.sin(lat1)*math.cos(lat2)*math.cos(lon2-lon1))

    # New angle in radians
    radang = math.atan2(y_coord,x_coord)
    # Convert radang to degrees
    tmpbrng = round(((radang*180)/math.pi)+360)%360,1)
    print(f"[calcPtsBearing] - New Bearing = {tmpbrng}"
    return tmpbrng

# End of calcPtsBearing
def writeLogFile( logFileName, logEntry ):

    with open(logFileName, mode='a', encoding='utf-8') as df:

        df.write(logEntry)

        df.close()

# End of writeLogFile

def xonSer(serPortNum):

    bXonStr = bytes("$R\n", 'utf-8')

    if(serPortNum == 0):
        ser1.write(bXonStr)
    elif(serPortNum == 1):
        ser2.write(bXonStr)
else:

    ser3.write(bXonStr)

# End of xonSer

# Function to send a string to the ASV connected to a particular serial port
#******************************************************************************
def asvSerW(serPortNum,strToSend):
    
    if(serPortNum == 0):
        ser1.write(strToSend)
    elif(serPortNum == 1):
        ser2.write(strToSend)
    else:
        ser3.write(strToSend)

# End of asvSerW

# Function to read a string from the ASV connected to a particular serial port
#******************************************************************************
def asvSerR(serPortNum):
    
    response = ""    # Blank the response string
    
    if(serPortNum == 0):
response = ser1.readline()

elif(serPortNum == 1):
    response = ser2.readline()

else:
    response = ser3.readline()

return response

# End of asvSerW

# Main program starts here

def main():

    # Get the log file prefix and create the names for all four logfiles.
    templfName = input("Enter prefix for log files: (Suggested YYYYMMDD_Test##_Run##)")
    lfnName = templfName + ".txt"
    lfnName0 = templfName + "_Boat0.txt"
    lfnName1 = templfName + "_Boat1.txt"
    lfnName2 = templfName + "_Boat2.txt"

    # Set up the simulation
    print("Setting up...")

# Blank the response string
response = ""
# Create and blank asvLog strings

asvLog0 = ""

asvLog1 = ""

asvLog2 = ""

# Set start time

tHours=7

tMinutes=0

tSeconds=0

# Randomize the starting angles for the ASVs

for i in range(0,3):
    asvBrngs[i] = random.randrange(360)

# Build the Initial GPS strings and send them

for i in range(0,3):
    gpsStr = buildGPS(tHours,tMinutes,tSeconds,asvLats[i],asvLongs[i],asvBrngs[i],asvSpds[i])
    bgpsStr = bytes(gpsStr, 'utf-8')
    print("ASV ",i," starting GPS = ",end=""
    print(bgpsStr.decode('utf-8'))  # Print to console
    writeLogFile(lfName,gpsStr + "n")   #Write it in the overall log file
    # Send the initial GPS string to the ASV
    asvSerW(i,bgpsStr)

# Loop through to run the simulation for 15 minutes - each iteration is 1 simulated second
for i in range(0,900):

for j in range (0,3):
    # Get the motor command from boat j and print it out on the screen (for debugging)
    response = ""    # Blank the response string
    response = asvSerR(j)  # Read the response from the j ASV
    print("ASV ",j," Motor response:",end='')
    print(response.decode('utf-8'))
    # Write the Motor response to the log file (for debugging)
    writeLogFile(lfName,response.decode('utf-8'))
    # Parse the motor command response and get new speed and bearing
    asvSpds[j], asvBrngs[j] = mtrParse(response)
    # Calculate the next point
    asvLats[j],asvLongs[j],asvBrngs[j] =
    calcNewCoords(asvLats[j],asvLongs[j],asvSpds[j],asvBrngs[j])
    # Build the next GPS string
    gpsStr = buildGPS(tHours,tMinutes,tSeconds,asvLats[j],asvLongs[j],asvBrngs[j],asvSpds[j])
    # Encode the GPS string
    bgpsStr = bytes(gpsStr, 'utf-8')
    # Print the GPS string to the Console output
    print(bgpsStr.decode('utf-8'))
    # Write the GPS string to the logfile
    writeLogFile(lfName,gpsStr + "\n")
    # Send the new GPS string
    asvSerW(j,bgpsStr)
print("Sent GPS String, reading log message")

# Get the ASV's GPS log message
response = ""  # Blank the response string
response = asvSerR(j)
if j==0:
    writeLogFile(lfName0,response.decode('utf-8'))
asvLog0 = response
print("ASV0 Log = ",end="")
print(asvLog0)
elif j==1:
    writeLogFile(lfName1,response.decode('utf-8'))
asvLog1 = response
print("ASV1 Log = ",end="")
print(asvLog1)
else:
    writeLogFile(lfName2,response.decode('utf-8'))
asvLog2 = response
print("ASV2 Log = ",end="")
print(asvLog2)

print("Out of first for loop")

# Convert log messages to bytes to add end of line
sASVlog0 = asvLog0.decode('utf-8')
sASVlog0 = sASVlog0.rstrip()
sASVlog0 = sASVlog0 + "\n"
print("sASVlog0 = ",end="")
print(sASVlog0)

sASVlog1 = asvLog1.decode('utf-8')
sASVlog1 = sASVlog1.rstrip()
sASVlog1 = sASVlog1 + "\n"
print("sASVlog1 = ",end="")
print(sASVlog1)

sASVlog2 = asvLog2.decode('utf-8')
sASVlog2 = sASVlog2.rstrip()
sASVlog2 = sASVlog2 + "\n"
print("sASVlog2 = ",end="")
print(sASVlog2)

bASVlog0 = sASVlog0.encode('utf-8')
print("bASVlog0 = ",end="")
print(bASVlog0)

bASVlog1 = sASVlog1.encode('utf-8')
print("bASVlog1 = ",end="")
print(bASVlog1)

bASVlog2 = sASVlog2.encode('utf-8')
print("bASVlog2 = ",end="")
print(bASVlog2)

# Send the log messages to the other ASVs
for j in range (0,3):
    print("Inside second for")
    if j==0:
        response = ""    # Blank the response string
        response = asvSerR(j)  # Read the response from the j ASV
        print("ASV "j," Ready to receive:".end="")
        print(response.decode('utf-8'))
        print("Writing ASV1 to ASV0")
        asvSerW(j,bASVlog1)
        response = ""    # Blank the response string
        response = asvSerR(j)  # Read the response from the j ASV
        print("ASV "j," Ready to receive:".end="")
        print(response.decode('utf-8'))
        print("Writing ASV2 to ASV0")
        asvSerW(j,bASVlog2)
        print("Reading from ASV0")
        response = ""    # Blank the response string
        response = asvSerR(j)
        print(response.decode('utf-8'))
        print("Inside j=0")
        writeLogFile(lfName0,response.decode('utf-8'))
    elif j==1:
        response = ""    # Blank the response string
        response = asvSerR(j)  # Read the response from the j ASV
        print("ASV "j," Ready to receive:".end="")
        print(response.decode('utf-8'))
    else:
        pass
print("Writing ASV0 to ASV1")
asvSerW(j,bASVlog0)
response = ""       # Blank the response string
response = asvSerR(j)        # Read the response from the j ASV
print("ASV ",j," Ready to receive:",".end=")
print(response.decode('utf-8'))
print("Writing ASV2 to ASV1")
asvSerW(j,bASVlog2)
print("Reading from ASV1")
response = ""       # Blank the response string
response = asvSerR(j)
print(response.decode('utf-8'))
print("Inside j=1")
writeLogFile(lfName1,response.decode('utf-8'))
else:
    response = ""       # Blank the response string
response = asvSerR(j)        # Read the response from the j ASV
print("ASV ",j," Ready to receive:",".end=")
print(response.decode('utf-8'))
print("Writing ASV0 to ASV2")
asvSerW(j,bASVlog0)
response = ""       # Blank the response string
response = asvSerR(j)        # Read the response from the j ASV
print("ASV ",j," Ready to receive:",".end=")
print(response.decode('utf-8'))
print("Writing ASV1 to ASV2")
asvSerW(j,bASVlog1)

print("Reading from ASV1")

response = "" # Blank the response string
response = asvSerR(j)

print(response.decode('utf-8'))

print("Inside j=2")

writeLogFile(lfName2,response.decode('utf-8'))

print ("Out of second for")

# Get the ASV's log message from checkBoundary routine
# for j in range (0,3):
#     print("Inside checkBoundary for loop")
#     response = "" # Blank the response string
#     response = asvSerR(j)
#     if j==0:
#         print("Inside j=0")
#         writeLogFile(lfName0,response.decode('utf-8'))
#     elif j==1:
#         print("Inside j=1")
#         writeLogFile(lfName1,response.decode('utf-8'))
#     else:
#         print("Inside j=2")
#         writeLogFile(lfName2,response.decode('utf-8'))

print("Ready to increment time.")
# Increment the time by one second

tHours,tMinutes,tSeconds = incTime(tHours,tMinutes,tSeconds)

print('End of Simulation Run')

#End of simulation

if __name__ == '__main__':
    main()
APPENDIX D. ARCGIS™ MODEL SOURCE CODE.

The code presented in this appendix is the Python code generated by the ArcGIS™ model builder and was created to provide visual analysis of the Area Patrol Algorithm. The code takes as its input the list of consecutive GPS points recorded by an ASV, generates a line from those points, buffers that line at various distances, dissolves the buffers, and clips the buffers to the bounding box provided as input to the model.

# -*- coding: utf-8 -*-
#
# AreaPatrol_Model_Python.py
#
# Created on: 2021-08-03 10:25:09.00000
#
# (generated by ArcGIS/ModelBuilder)
#
# Usage: AreaPatrol_Model_Python <Imput_CSV_table> <X_Field> <Y_Field> <Spatial_Reference> <Output_Line_Feature> <Distances> <Buffer_Unit> <Input_Bounding_Box> <Output_Buffer_Feature>
#
# Description:
# Takes CSV output and creates the track for the boat, buffers the track, then calculates the percent of covered area.
#
# Import arcpy module
import arcpy

# Load required toolboxes
arcpy.ImportToolbox("Model Functions")

# Script arguments
Imput_CSV_table = arcpy.GetParameterAsText(0)

X_Field = arcpy.GetParameterAsText(1)

Y_Field = arcpy.GetParameterAsText(2)

Spatial_Reference = arcpy.GetParameterAsText(3)
if Spatial_Reference == '#' or not Spatial_Reference:
    Spatial_Reference = "GEOGCS["GCS_WGS_1984",DATUM["D_WGS_1984",SPHEROID["WGS_1984",6378137.0,298.25722356]],PRIMEM["Greenwich",0.0],UNIT["Degree",0.0174532925199433]],-400 -400 1000000000;-100000 10000;8.98315284109;0.001;0.001;IsHighPrecision" # provide a default value if unspecified

Output_Line_Feature = arcpy.GetParameterAsText(4)

Distances = arcpy.GetParameterAsText(5)
if Distances == '#' or not Distances:
    Distances = "3;10;20" # provide a default value if unspecified

Buffer_Unit = arcpy.GetParameterAsText(6)
if Buffer_Unit == '#' or not Buffer_Unit:
    Buffer_Unit = "Meters" # provide a default value if unspecified

Input_Bounding_Box = arcpy.GetParameterAsText(7)
Output_Buffer_Feature = arcpy.GetParameterAsText(8)

# Local variables:
Layer_Name_or_Table_View = ""
Output_Coordinate_System = 
'PROJCS['WGS_1984_Web_Mercator_Auxiliary_Sphere',GEOGCS['GCS_WGS_1984',DATUM['D_WGS_1984',SPHEROID['WGS_1984',6378137.0,298.257223563]],PRIMEM['Greenwich',0.0],UNIT['Degree',0.0174532925199433]],PROJECTION['Mercator_Auxiliary_Sphere'],PARAMETER['False_Easting',0.0],PARAMETER['False_Northing',0.0],PARAMETER['Central_Meridian',0.0],PARAMETER['Standard_Parallel_1',0.0],PARAMETER['Auxiliary_Sphere_Type',0.0],UNIT['Meter',1.0]]'
Output_Feature_Class = ""
Output_Buffer_Feature_Unclipped = ""
Value = Input_Bounding_Box

Output_Feature_Class__2_ = Output_Buffer_Feature
Output_Feature_Class__3_ = Output_Feature_Class__2_
Output_Feature_Class__4_ = Output_Feature_Class__3_
Output_Feature_Class__5_ = Output_Feature_Class__4_

# Process: Make XY Event Layer
arcpy.MakeXYEventLayer_management(Imput_CSV_table, X_Field, Y_Field, Layer_Name_or_Table_View, Spatial_Reference, "")

# Process: Copy Features
tempEnvironment0 = arcpy.env.outputCoordinateSystem
arcpy.env.outputCoordinateSystem = Output_Coordinate_System
arcpy.CopyFeatures_management(Layer_Name_or_Table_View, Output_Feature_Class, "", "0", "0", "0")
arcpy.env.outputCoordinateSystem = tempEnvironment0

# Process: Points To Line

tempEnvironment0 = arcpy.env.outputCoordinateSystem
arcpy.env.outputCoordinateSystem = ""
tempEnvironment1 = arcpy.env.geographicTransformations
arcpy.env.geographicTransformations = ""
arcpy.PointsToLine_management(Output_Feature_Class, Output_Line_Feature, "", "", "NO_CLOSE")
arcpy.env.outputCoordinateSystem = tempEnvironment0
arcpy.env.geographicTransformations = tempEnvironment1

# Process: Multiple Ring Buffer

arcpy.MultipleRingBuffer_analysis(Output_Line_Feature, Output_Buffer_Feature_Unclipped, Distances, Buffer_Unit, "distance", "NONE", "FULL")

# Process: Clip

arcpy.Clip_analysis(Output_Buffer_Feature_Unclipped, Input_Bounding_Box, Output_Buffer_Feature, ""

# Process: Add Field

arcpy.AddField_management(Output_Buffer_Feature, "Coverage", "DOUBLE", "", "2", "", ", NULLABLE", "NON_REQUIRED", ","

# Process: Calculate Field

arcpy.CalculateField_management(Output_Feature_Class__2_, "Coverage", ",[Shape_Areal]/19870.456732", "VB", """)
# Process: Add Field (2)

arcpy.AddField_management(Output_Feature_Class__3_, "Boat_ID", "SHORT", ",", "2", ",", ",", "NULLABLE", "NON_REQUIRED", ",")

# Process: Get Field Value

arcpy.GetFieldValuemb(Input_Bounding_Box, "BoatID", ",", "0")

# Process: Calculate Field (2)

arcpy.CalculateField_management(Output_Feature_Class__4_, "Boat_ID", Value, "VB", ",")
REFERENCES


Jeon, Hong Y., Lei F. Tian, Direct application end effector for a precise weed control robot, Biosystems Engineering (2009), doi:10.1016/j.biosystemseng.2009.09.005


VITA

Daniel Davis Smith was born on in West Monroe, Louisiana but was raised primarily in Baton Rouge, Louisiana. He graduated from McKinley Senior High School in Baton Rouge, Louisiana, in 1985. In the fall of 1985, he began attending Louisiana State University and pursued an undergraduate degree in Computer engineering with a minor in computer science that he completed in August of 1992. He immediately began a Master of Science program in The Department of Electrical and Computer Engineering that was terminated by illness. In 2005, Daniel returned to Louisiana State University for a master’s in the Department of Biological and Agricultural Engineering which was completed in 2011. Upon completion of the master’s degree, he began to pursue a Doctor of Philosophy degree which will be attained in December 2021.