A MODULAR GUIDANCE, NAVIGATION AND CONTROL SYSTEM FOR UNMANNED SURFACE VEHICLES

by

Thomas C. Furfaro

A Thesis Submitted to the Faculty of

The College of Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Florida Atlantic University

Boca Raton, Florida

May 2012

A MODULAR GUIDANCE, NAVIGATION AND CONTROL SYSTEM

FOR UNMANNED SURFACE VEHICLES

by

Thomas C. Furfaro

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Karl D. von Ellenrieder, Department of Ocean and Mechanical Engineering, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

SUPERVISORY COMMITTEE:

Karl D. von Ellenrieder, Ph.D.
Thesis Advisor

Manhar Dhanak, Ph.D.

Edgar An, Ph.D.

Javad Hashemi, Ph.D.
Chair, Department of Ocean and Mechanical Engineering

Mohammed Ilyas, Ph.D.
Interim Dean, College of Engineering and Computer Science

Barry T. Rosson, Ph.D.
Dean, Graduate College

April 13, 2012
Date

ACKNOWLEDGEMENTS

I would first and foremost like to thank my dearest friend and colleague Janine Mask, who, during her concurrent thesis efforts, would daily make the toils of research enjoyable. I would also like to thank the other students involved in some way, shape or form with this effort, and in particular Phillip Duerr, Jose Alvarez, and Armando Sinisterra. Also, the members of the AUVSI Roboboat team have tangentially contributed massively with their work on *GUSS*, and for that I thank them.

I would also like to recognize my advisor, Karl von Ellenrieder. He has been my advisor in some shape or another for nearly 5 years now. He has given me more freedom and leeway than I really deserve, and under his appropriately-distanced watch I have learned an incredible amount.

I would also like to thank the staff members at SeaTech; without their help, it would be nearly impossible to get anything actually accomplished. Those staff members who contributed greatly to my studies and to this project are Ed Henderson, Tom Pantelakis, John Kielbasa, Beatriz Gomez, and Luis Padilla. I would also like to thank the Department of Ocean Engineering in general, for continuous support throughout my undergraduate and graduate studies.

Lastly, I would like to thank my parents and family for being understanding of my general absence and highly inconsistent communication, and for teaching me what it means to be responsible and work with passion and fervor.

ABSTRACT

Author:	Thomas C. Furfaro
Title:	A Modular Guidance, Navigation and Control System for Unmanned Surface Vehicles
Institution:	Florida Atlantic University
Thesis Advisor:	Dr. Karl D. von Ellenrieder
Degree:	Master of Science
Year:	2012

The design and integration of an unmanned surface vehicle (USV) control system is described. A survey of related work in both USV control, and unmanned vehicle operating software is presented. The hardware subsystem comprising a modular Guidance, Navigation, and Control (GNC) package is explained. A multi-threaded software architecture is presented, utilizing a decentralized, mutex-protected shared memory inter-process communication subsystem to provide interoperability with additional software modules. A generic GNC approach is presented, with particular elaboration on a virtual rudder abstraction of differential thrust platforms. A MATLAB Simulink simulation is presented as a tool for developing an appropriate controller structure, the result of which was implemented on the target platform. Software validation is presented via a series of sea trials. The USV was tested both in open- and closed-loop control configurations, the results of which are presented here. Lastly recommendations for future development of the GNC system are enumerated.

A MODULAR GUIDANCE, NAVIGATION AND CONTROL SYSTEM

FOR UNMANNED SURFACE VEHICLES

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

B	Beam	T	Thrust magnitude
d_i	Distance to the *i*th waypoint	$\mathbf{T}$	Transformation matrix
$d_{\min}$	Minimum closing distance	u	Surge velocity
D	Draft	u_C	Surge command
$\mathbf{D}$	Damping matrix	$\hat{u}$	Measured surge speed
D_x	Thruster longitudinal position	$\mathbf{u}$	Control signal
D_y	Thruster lateral position	v	Sway velocity
e_u	Surge velocity error	$\mathbf{v}$	State vector
e_{CT}	Cross-track error	x	Earth-fixed North oordinate
e_ψ	Heading error	X	Surge force
$\mathbf{f}$	Forcing vector	y	Earth-fixed East oordinate
$\mathbf{k}$	Controller gain vector	$\mathbf{y}$	Output vector
k_P	Heading proportional gain	Y	Sway force
k_I	Heading integral gain	α	Thrust proportionality constant
l	Look-ahead distance	δ_r	Virtual rudder control signal
LOA	Length over All	δ_u	Speed control signal
LWL	Length at Waterline	θ	Thruster attitude
$\mathbf{M}$	Inertia matrix	ψ	Earth-fixed heading
N	Yaw moment	ψ_c	Heading command
P	Earth-fixed waypoint	$\hat{\psi}$	Measured heading
r	Yaw rate	ω	Thruster rpm
R	Earth-fixed position	∇	Displacement
t	Time		

1 INTRODUCTION

Unmanned surface vessel (USV) research is a comparatively new specialization in the realm of marine robotics, [1] [2]. Many of the technical challenges faced by USV developers have analogous solutions from the unmanned underwater vehicle (UUV) realm; however the environment is considerably different. This area of interaction between sea and sky is quite dynamic, on both short and long time scales.

The navigation problems that plague UUV systems is partially mitigated by the availability of Global Position System (GPS) information, yet the control problem is complicated by waves and wind. The guidance problem can be an entirely different (and more difficult) problem altogether, as often the operating zone of many USV concepts lies in coastal and littoral regions, where encounters with non-static obstacles, such as civilian small boat traffic, are a likely occurrence. Thus, much recent discussion has been focused on the regulation of USV operations in the littoral zone [3], however the long-term implications are beyond the scope of the summary presented here.

The basic approach for controlling a surface vehicle, and as described by Fossen in Figure 1, consists of three major tasks: guidance, navigation, and control (GNC) [4]. Firstly, the term *guidance* refers to the highest level of vehicle direction. This segment of the vehicle system generates a *target trajectory* used by the control system based on some mission plan. In the example in the figure, the guidance system takes cues (*waypoints*) from a program that evaluates weather information to determine trajectory. In other cases, these waypoints might come from a pre-determined mission plan, modified in real time by a collision avoidance scheme, or a machine vision guidance system, as in [5]. The second portion of the vehicle automation scheme is a *control system*. This portion takes the synthesized, target trajectory from the guidance system, as well as some estimates of vehicle orientation, position, and velocity (vehicle *poses* or *states*), and performs a control allocation based on a *controller*. This controller is in essence, a mathematical function designed to drive to vehicle states to the specified trajectory based on current state information. The vehicle actuator, along with the physical manifestation of the vehicle itself (primarily the hull and hydrodynamically relevant portions), together constitute the *vehicle plant*. Control signals output from the controller in the control system segment are communicated to the relevant actuators, producing a response that may be represented in the vehicle states. The last segment, the *navigation* portion, is

composed of the sensors and the accompanying data synthesis or fusion, which together produce estimates of the vehicle states. These states are in turn fed back into the controller (forming *closed-loop feedback*), which again performs a control actuation based on the new state estimates and a newly determined trajectory. All of these operations are being performed continuously, at a rate that is commensurate with the vehicle's desired reaction time. This approach is typical of most USV and UUV applications, and is also generally common to most control systems in non-marine fields.

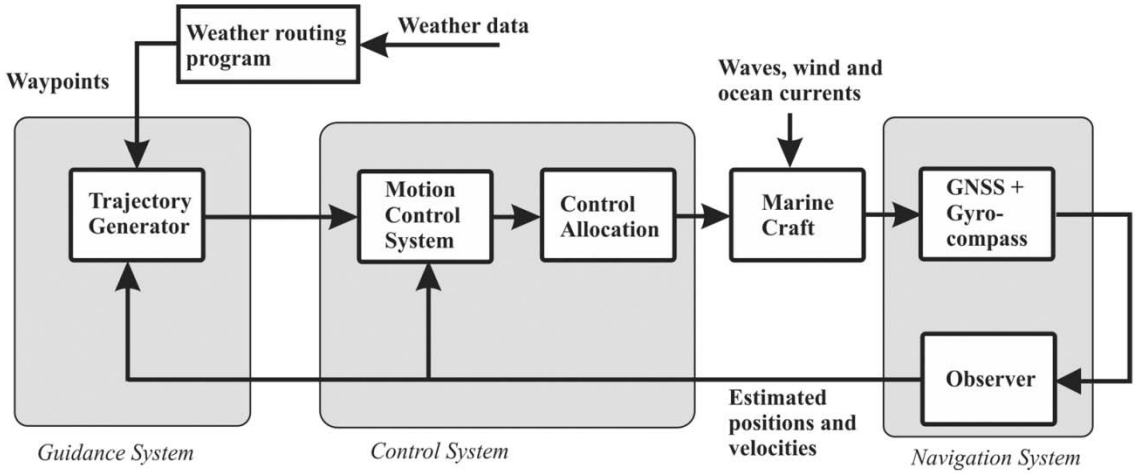

Figure 1: Guidance, Navigation and Control overview as described by Fossen, [4]

Considering these concepts, it is clear that the core of autonomous system development is largely a multidisciplinary systems problem. The task of the developer consists of not only addressing GNC issues, but also considering the design and integration of the appropriate sensory, electrical, and software subsystems that enable higher-level algorithm efficacy. Thus, the thrust of this thesis is not only related to the development of the higher level GNC capabilities, but also the core platform implications revolving around software design and implementation.

This document gives a brief review of some previous work in the area of USV autonomous system design and how it relates to various program objectives at FAU, describes the problem that is being addressed, outlines the experimental approach and methodology, presents an analysis of data collected, and discusses the significant outcomes.

2 BACKGROUND AND PREVIOUS WORK

2.1 PROBLEM DEFINITION

Figure 2 depicts a generic surface vessel reference frame used for GNC. As in the figure, the three degrees of freedom that are considered are: the x and y geographic coordinates, where x is positive Northward and y is positive Eastward, both centered at center of gravity (CG); and the heading, ψ, considered positive counter-clockwise. Vectorially, the vehicle position is written R, and the discrete waypoints that describe a segmented path are written $P_i = \langle x_i, y_i \rangle$. The distance between the *i*th waypoint and the vehicle position is written as d_i. The cross-track error is written e_{CT}, and is defined as the perpendicular distance between the vehicle's current position, R, and the segment formed by the upcoming waypoint P_i and the preceding waypoint P_{i-1}. The heading error, e_ψ, describes the mismatch between the desired or commanded heading, ψ_c, and the actual heading. Also note that the commanded heading is not necessarily constrained to be in the direction of the upcoming waypoint.

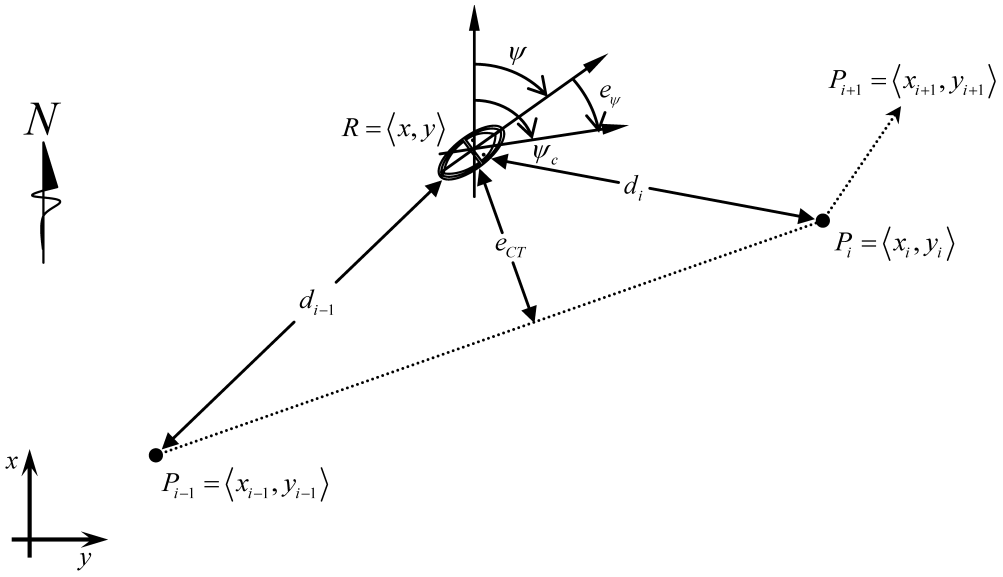

Figure 2: Generic vehicle reference frame.

With respect to surface vessels, it is common practice to reduce the coupled six degree-of-freedom (6DOF) model to two independent third-order systems, the horizontal (sway, roll, yaw) and vertical (surge,

heave, pitch) planes. In the horizontal plane, the controllable states are sway and yaw, while in the vertical plane surge is the controlled parameter. As such, the speed controller (surge) can be decoupled from the heading and cross-track controller (yaw and sway). Furthermore, a *state vector*, which contains not only the degrees of freedom, but also the time derivatives of the system, is assembled to contain information for use in GNC.

The vehicle position and orientation is defined by $\langle x, y, \psi \rangle^{\mathrm{T}}$, containing the earth-fixed geographic frame position and the yaw orientation. The vehicle state vector is $\mathbf{v} = \langle u, v, r \rangle^{\mathrm{T}}$, comprised of the body fixed planar velocities and the yaw-rate. Note that the yaw rate, r, is in fact the time derivative of the global heading, ψ, whereas the global translational velocities are non-linear functions of the body-fixed velocities. That is,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ r \end{bmatrix} = \mathbf{T}_{\mathrm{B}}^{\mathrm{G}}(\mathbf{v})\mathbf{v}, \tag{2.1}$$

where $\mathbf{T}_{\mathrm{B}}^{\mathrm{G}}(\mathbf{v})$ is a non-constant transformation matrix from the body-fixed to geographic frame.

The general form of the USV equations of motion appear as,

$$\mathbf{M}\dot{\mathbf{v}} + \mathbf{D}(\mathbf{v})\mathbf{v} = \mathbf{f}(\mathbf{v},\mathbf{u}) = \mathbf{f}_{\text{environment}}(\mathbf{v},\mathbf{u}) + \mathbf{f}_{\text{control}}(\mathbf{v},\mathbf{u}) \tag{2.2}$$

where $\mathbf{M}$ is the inertia matrix, containing inertial terms for both vehicle mass and added fluid mass, and $\mathbf{D}$ is the damping matrix, which is a non-linear function of the state $\mathbf{v}$. Typically, the hydrodynamic parameters in these matrices are estimated to be a result of a Taylor-series expansion around the operating point. In the case of a linear model, the expansion is of the first order. System identification is used to determine the parameters in the inertia and damping matrices, either via entirely theoretical means (white box modeling), purely experimental methods (black box), or some hybrid of the two (grey box).

The vector function $\mathbf{f}(\mathbf{v},\mathbf{u})$ includes all control ($\mathbf{f}_{\text{control}}$) and external forces not encapsulated in the deterministic hydrodynamic and inertial forcing ($\mathbf{f}_{\text{environment}}$). The external forces include perturbations to the system such as wind and current fluid forces. The control component consists of the forces arising from application of the thrusters and actuators. Thus, the role of any GNC system is to measure and drive the vehicle along some trajectory $\mathbf{v}(t)$ via the control influence exerted via $\mathbf{u}(t)$.

Frequently throughout this document an output, $\mathbf{y}(t)$, will be used. For simulation purposes, it is sometimes easier to contain all potential information of interest in a single vector. This expanded state vector is defined as

$$\mathbf{y} = \langle u, v, r, \psi, x, y, \omega_{\mathrm{P}}, \omega_{\mathrm{S}}, \theta_{\mathrm{P}}, \theta_{\mathrm{S}} \rangle \tag{2.3}$$

There are many ways in which to arrive at a suitable controller, ranging from highly theoretical methods to purely heuristic ones, or from intensely plant-dependant control to more adaptive methods. The choice in approach for any given problem has many driving factors, but knowledge of system dynamics is a key aspect. Given a system model, one may produce simulations and predict performance for a designed controller. Conversely, without a mathematical system characterization, there are also approaches one may take to design a stable system. In a similar light, implementation of a control system on the target platform may in some cases be a simple task, so heuristic approaches may be preferred. The primary objective of this thesis work is to design, implement, and validate an autonomous GNC system capable of reliably driving a USV. The system will integrate navigation sensors, actuator-interface electronics, and control hardware. Software has been developed to synthesize state information from various sensor measurements, and to perform control actions based on predefined mission parameters and feedback through state estimates. There are multiple platforms upon which the system has been deployed; however, the particulars of the target vehicle are relatively unspecified in order to promote cross-vehicle modularity and portability, a key characteristic of the designed GNC hardware and software package. The electrical and software interface are generic; the control signals themselves are delivered in a common signal format, and important vehicle-specific parameters, such as control gains, are quickly reconfigurable via the use of user-friendly mission configuration files. Two potential platforms are presented here, but by extension, many of the design features and interfaces of the GNC system are compatible with other vehicles as well.

2.1.1 **WAM-V *USV12***

One such candidate platform is the Wave Adaptive Modular Vessel (WAM-V) class *USV12*. It is a novel class of surface craft conceived to improve sea-keeping performance, endurance, and fuel efficiency [6] . Currently, there are four WAM-V form factors: a 33.5 m LOA manned vessel and three unmanned vehicles of 10 m, 3.7 m and 4.3 m LOA; the 3.7 m (12 feet) vehicle has been named the *USV12*. The WAM-V class is characterized by inflatable, non-rigid hulls in a catamaran configuration, connected to a central superstructure via a passive suspension system, as seen in Figure 3. The principal goal of the design

is to attenuate wave-induced vibrations felt in the payload compartment. The design is unprecedented and little work has been done to quantify the vessels dynamics. Virginia Tech is currently investigating the structural/mechanical aspects of the vehicle design. The focus of work is largely on the dynamic response in various operating conditions, with particular emphasis being placed on the analysis of the payload-stabilizing aspects of the vehicle. The maneuvering performance, and particularly the *USV12*'s eligibility as a fully unmanned system, is currently being investigated within the FAU Department of Ocean & Mechanical Engineering. Furthermore, there are long-term plans to use the *USV12* and its derivative prototypes as an agent in a multi-vehicle, cooperative autonomous fleet.

Figure 3: Wave Adaptive Modular Vehicle (WAM-V) 12' Prototype

The passive suspension system on the WAMV *USV12* is a distinct design feature that defines the class. The two forward mounting points allow rotation in the vehicle x, y and z axes, along with some vertical translation mitigated by a spring-shock assembly. The aft mounts allow rotation in y, and z, with no translation. The total effect of such a linkage system is that the pitch responses of the two hulls are largely uncoupled from each other, with the payload compartment only experiencing a fraction of the excitation. Physical characteristics of the model are listed in .

The vehicle is powered by two gas-driven waterjets, each with independent RPM, nozzle direction, and bucket (reverse) control. There is an existing remote control (RC) subsystem on board, used to wirelessly control vehicle movement. This consists largely of a RC transmitter, or radio, with a paired set of receivers. The RC system, as delivered from the manufacturer, gives a user the ability to start and choke the engines, adjust engine throttle, change nozzle direction, and trim bucket level for each engine module,

as depicted in Figure 4. The objective of the control design is to interface and manipulate these actuators for vessel maneuvering.

Table 1: WAM-V 12' Model Attributes

Parameter	Value
Length Overall (*LOA*)	3.7 [m]
Beam (*B*)	2.0 [m]
Demihull Beam	0.22 [m]
Displacement (∇)	73 [kg]
Maximum Payload	18 kg
Power Plant	Two (2), 2.1 kW, 2 stroke engines
Propulsor Type	Waterjet
Control Actions	Nozzle Yaw, Reverse bucket pitch

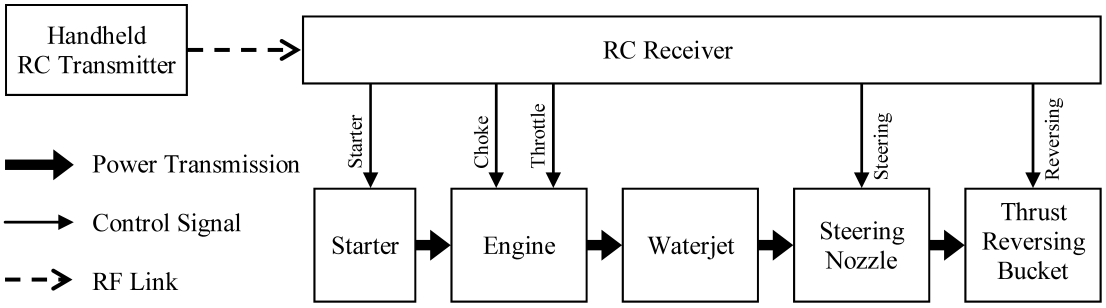

Figure 4: Existing *USV12* propulsion and control system. (Single hull shown)

2.1.2 *GUSS*

A 1.4 m tugboat model was utilized as a vehicle of opportunity (Figure 5). This vehicle, named *GUSS*, has the basic characteristics listed in Table 2. Earlier work regarding the development of *GUSS* can be found in [7] and [5]. While the vehicle is considerably smaller than the *USV12*, and significantly different in hull form (monohull vs. catamaran), the vehicle is similar from a GNC perspective. Particularly, *GUSS* has four control actions, two correlating to thrust and two correlating to thruster direction. Figure 6 shows a schematic and image of a single thruster. Furthermore, the control actions on *GUSS* are serviced by control signals that are electrically identical to those used on the *USV12*, both utilizing remote-control (RC) style pulse-width-modulated (PWM) signals.

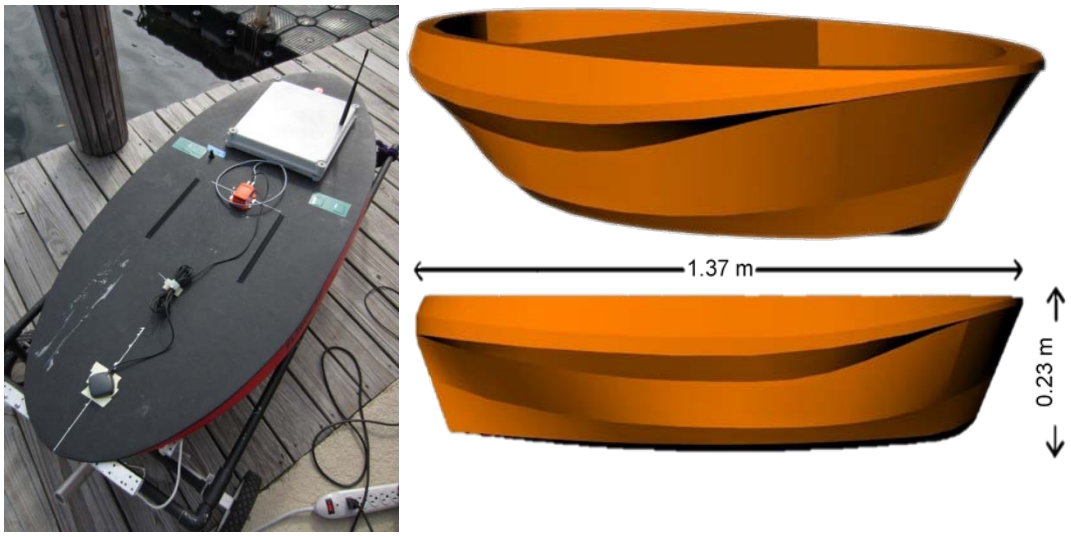

Figure 5: *GUSS* USV with GNC package installed and MTi-G motion sensor mounted on deck (left). GUSS principal dimensions (right).

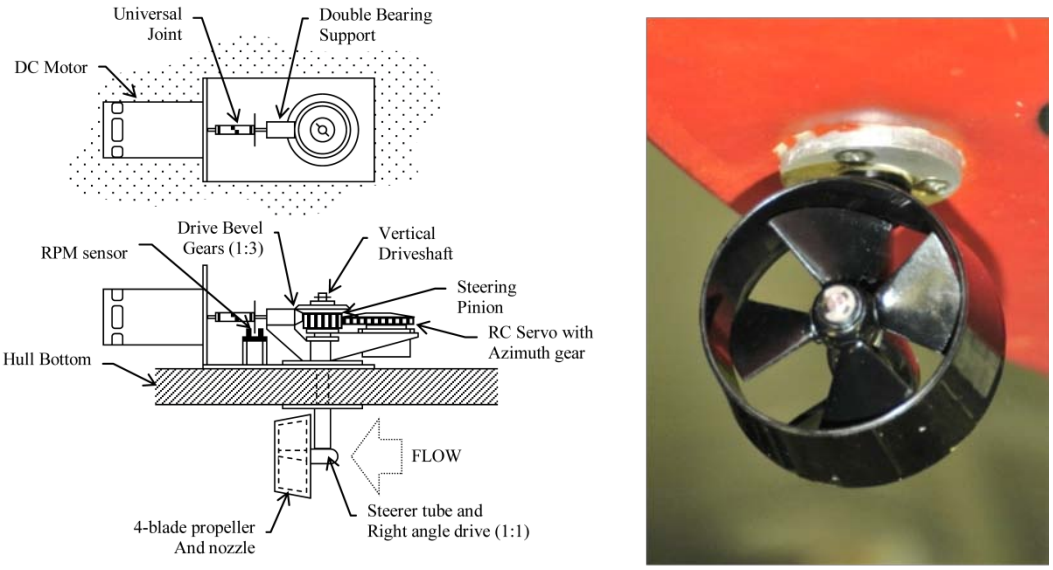

Figure 6: *GUSS* azimuthing thruster schematic (left); thruster picture (right).

The size and weight of *GUSS* make it a particularly attractive platform for hardware and software testing, as all aspects of testing operations are easily manageable by a single person. The ease of use allows rapid deployment and thus rapid debugging cycles when performing low-level systems integration.

Table 2: *GUSS* USV Geometry

Parameter	Value
Length Overall (LOA)	1.37 [m]
Length at waterline (^{LWL})	1.26 [m]
Beam (B)	0.58 [m]
Draft (T)	0.23 [m]
Molded Depth	0.10 [m]
Displacement (∇)	24 [kg]
Length-to-beam ratio (L/B)	2.7

2.2 LITERATURE REVIEW

Over the last decade, there has been a significant increase in the rate of USV research, from mission-optimized system design, dynamic modeling, advanced control, and high-level navigation [2]. The most pertinent of these advancements are in the areas of control software design, and the application of path following control.

2.2.1 **Robotics Control Software**

There have been various approaches to standardizing robotics software into some form of an operating system. The Mission Oriented Operating Suite (MOOS) [8] provides mid-level libraries for common aspects of autonomy software, in particular the structure and format for inter-process communication, as well as guidelines for expansion. MOOS has since been combined with the interval-programming helm (IvP Helm) package to provide a more complete autonomy software suite [9]; IvP Helm fills the space of the highest level, most intelligent behavior segments of the software. MOOS-IvP operates on the "backseat driver" paradigm; the operating suite is not designed to address the hard timing requirements of vehicle control itself, but is designed to enable easy integration of all of the other components that deliver controller setpoints. The core of MOOS is a database-like server (MOOSdb) connected to a network of client applications via a star-shaped topology using the TCP/IP protocol. The strength of the approach is that peer-to-peer (client to client) connections are not permitted, and because the MOOSdb manages the communication allocation tasks (except for initiation, a client-side responsibility),

client applications that are slow, hang-up, crash, or are otherwise poorly written, do not bog down the other applications running on the network. The TCP/IP protocol is strictly one-to-one, meaning that every connection has strictly one sender and one recipient. However, this centralized architecture places significant demand on the server-side code to be robust to applications that have serious faults, such as inappropriately timed loops or memory leaks, that have potential to adversely affect the run-time dynamics of the entire architecture.

Similar to MOOS, Matczynski in [10] presents his concept of an Onboard Planning Module (OPM). This conceptually operates on a similar principle, as a backseat driver to a lower-level control and navigation system. In particular, the OPM is distinct from a MOOS implementation in that it was conceived with inter-vehicle communications as a major feature. In fact, the work Matczynski presents is the integration of OPM into a swarm of UAV agents.

Quigley, at Stanford University, began to develop a robot operating system in the STanford Artificial Intelligence Robot (STAIR) project, [11], which after some further refinement in collaboration with Willow Garage, has been renamed the Robot Operating System (ROS) [12]. The ROS is similar to MOOS in that several distributed clients perform more specific tasks, such that the most computationally-intensive operations do not occur on a single node. They are also similar in that ROS has a “Master” node. However, the similarity ends here. At ROS startup, the master node manages startup and configuration of a user-defined cluster topology. At shutdown, the master node manages node shutdown flags. The master node does not manage actual messages between nodes, just the connections themselves. Communication occurs between different nodes on different segments of the network based on user pre-configuration. ROS uses TCP/IP like MOOS, yet the connections are always peer-to-peer, instead of peer-to-server. This decentralized control obviously counters the largest weakness of MOOS, which is the inherent speed limitations arising from having a single central server.

Most recently, Huang has developed the Lightweight Communication and Marshalling (LCM) protocol at MIT’s Computer Science and Artificial Intelligence Laboratory (CSAIL). LCM is quite similar to ROS, however node management is completely distributed; there is no inherent specification in the protocol for a master node to manage startup and shutdown. In this case, connections are based on UDP multicast, which unlike TCP/IP, has multiple senders and receivers on a given channel. The major point to LCM is that because the messages are being sent via the UDP channels, read/write access is not an issue because when a message is published, the data are available to any node that is interested in using them.

Both Huang [13], [14], and Bingham [15] provide some performance metrics of LCM. Bingham provides comparison between LCM (implemented in both C and Python), and MOOS, whereas Huang compares LCM to other commercially available operating systems. The data make apparent the fact that MOOS is highly inflexible to changing bandwidth requirements, whereas LCM is scalable.

2.2.2 **Surface Vehicle Control**

The distinction between various set-point assignment methods should be made: 1) The simplest case is a heading controller, which simply modulates the vehicle heading to be some constant value. This obviously leads to unbounded cross-track error and a final position error if trying to navigate between waypoints using a static heading assignment. 2) The next level of complexity is a line-of-sight (LOS) controller, where the heading set-point is constantly reevaluated based on the newest vehicle position fix, as in the author's previous work [16]. This scheme also has unbounded cross-track error, but is capable of driving the vehicle to the desired position given that the problem is possible, insofar as the vehicle has enough command authority (power) to overcome external forcing. 3) A path-following controller modulates control outputs in order to minimize cross-track error, usually with a velocity controller running independently to maintain a forward or along-track speed. The goal of path-following controllers is to drive the cross-track error to zero while still progressing towards the final destination. This is the approach that will be used here. 4) The next level is a tracking controller, where there is not only a specific position that the vehicle is trying to be driven to, but also an associated timeline. Set-point determination in this case becomes easier because the desired position is purely at the directed position at the given time (from higher-level guidance mechanisms), but difficulty arises when external disturbances drive the vehicle off-schedule, and thus increased control effort must be exerted. In a given environment, a specified trajectory may be unfeasible in terms of controller effort and potential plant power output. These set-point modulation schemes are a hybrid of the guidance and control problems. The assumption is that one has a very basic but working controller operating at the lowest level, and the stability and response of the low level control (such as heading or velocity) can be leveraged by creatively manipulating the setpoint inputs.

For example, Greytak, [17], develops a PID controller with a spatial integrator term using various set-point control schemes, including basic LOS and feed forward set-point control. A linear controller with a time-integrated error term is presented for contrast, showing the considerable advantage of using spatial-integration in the case of path following. The controller architecture is dual-layered. A low-level heading controller is used in coordination with a high-level path following controller. The overall result is a

practical approach to the path-following problem, using set-point control methods to simplify the control design to the sum of several distinct, easily understandable sub-controllers.

Approaches using simple linear controllers coupled with clever set-point assignments schemes are most effective in cases where the vehicle response is linear (or the design operating regime exists in a mostly linear range of the response) and environmental disturbances are largely time-invariant. If operating conditions deviate largely from this domain, then more advanced methods are required.

There has been much work performed in non-linear control. The most basic approach to dealing with nonlinear systems is linearization, as described in [18], where the governing equations are simplified by linearizing around some specific operating point, and control is performed using a linear controller designed for the simplified system model. Computationally this is the easiest route, but stability is not guaranteed because the typical stability analysis of the linearized system does not reveal the true nature of the full nonlinear system. A version of system reduction via order-of-magnitude, or dominance, arguments is presented in [19], and while both simulation and experimental validation revealed stable controller performance, the importance of non-linear behavior is made apparent by differences in predicted versus measured performance. There have been implementations of linear optimal controllers, such as in [20], but these do not take into account external disturbances. Sliding mode controllers are a non-linear form of optimal control, stabilizing the system with a high-frequency gain.

Methods such as those in [21] and [22] use an online unscented Kalman filter (UKF) to estimate the unknown model parameters while using a backstepping technique to maintain stability of the previously-unknown system. The backstepping approach uses a known, stable subsystem, called the originator, as a starting point for each controller subsystem. Every consecutive layer, or step, in the total controller is of a form that guarantees stability of the controllers below it. Thus, the final control structure is guaranteed to be stable. This basic characteristic of backstepping is used in parallel with the real-time model estimation to guarantee a controller that is stable at the highest level. This method allows for extremely robust controller performance, and very few assumptions regarding the system model are made. The only major decision made beforehand by the designer is the selection of the model structure and order; the parameters themselves are evaluated in real-time.

3 METHODOLOGY AND APPROACH

A modular, easily adaptable hardware and software guidance, navigation and control platform is needed to aide autonomous behavior developments on various USV platforms. This section describes the specific contributions constituting the thesis effort, and describes the methodology and approach used to acheive said contributions.

3.1 CONTRIBUTIONS

The overarching goal of this thesis is to design, construct, implement and evaluate a modular GNC system for USVs. The effort encompasses the electrical design of the control hardware, integration with a suite of navigation sensors, and the software realization of the controller on the target platform. The major goals of the project are:

- USV System Identification through open-loop testing
- Controller synthesis and implementation with verification
- Autonomous behavior development

Much of the work revolves around the second contribution. In more detail, this consists of the following:

- Electrical subsystem design, fabrication, and installation
- Software design and implementation for data-acquisition
- Software design for navigation and control
- Controller simulation in MATLAB's Simulink based on vehicle model
- Controller realization and verification on-board, in the embedded system

The remainder of this section describes the method by which these goals were achieved.

3.2 APPROACH

The first task is to design and integrate an electronic GNC package for deployment on the target platforms (*GUSS* initially and the WAM-V *USV12* in the future). The hardware system incorporates appropriate sensors and interface hardware to enable both data logging and closed-loop feedback control. Secondly, a vehicle operating system (VOS) was written and implemented on the GNC package. The development of operating software that integrates all onboard sensors and output peripherals to make appropriate control allocations based on received state measurements and estimates was the largest task associated with this thesis.

Concurrently, high-level controller architectures were developed using MATLAB's Simulink. A virtual vehicle was used as the basis for controller structure development. Once the controllers were configured to provide acceptable performance with the digital model, the control laws were realized on the USV control hardware. The controller will be further tuned when implemented in hardware. Finally, the path following controller and VOS are validated by performing closed loop trials. Cross-track error is minimized, with forward speed maintained at a predefined setpoint, and actual trajectories determined by a basic path planning algorithm fused with a human-defined mission plan. Which controller is chosen (e.g. sliding mode, Adaptive Kalman Filter, etc.) depends heavily on the specified system model, but the first approach taken was to implement a PID controller.

3.3 HARDWARE DESIGN

A hardware subsystem was needed to capture the open-loop dynamic response measurements, and then eventually to implement a hardware controller. As such, the first step in developing the autonomous subsystem for the *USV12* was designing and integrating a comprehensive hardware package that manages data acquisition, sensing, control, communications and remote override.

The hardware design is driven by several requirements. For the USV12, the weight of any added subsystems should be less than 18 kg (40 lbs), as recommended by the manufacturer [23]. In order to preserve the integrity of the existing system, the GNC package was designed to require as few modifications to the current configuration as was practical. A block diagram of the modifications made to the original system (Figure 4) is shown in Figure 7. The specific hardware schematic, including the printed circuit board (PCB) design, sensor datasheets and other hardware specifications, can be found in Section 6.1: Electrical Subsystem.

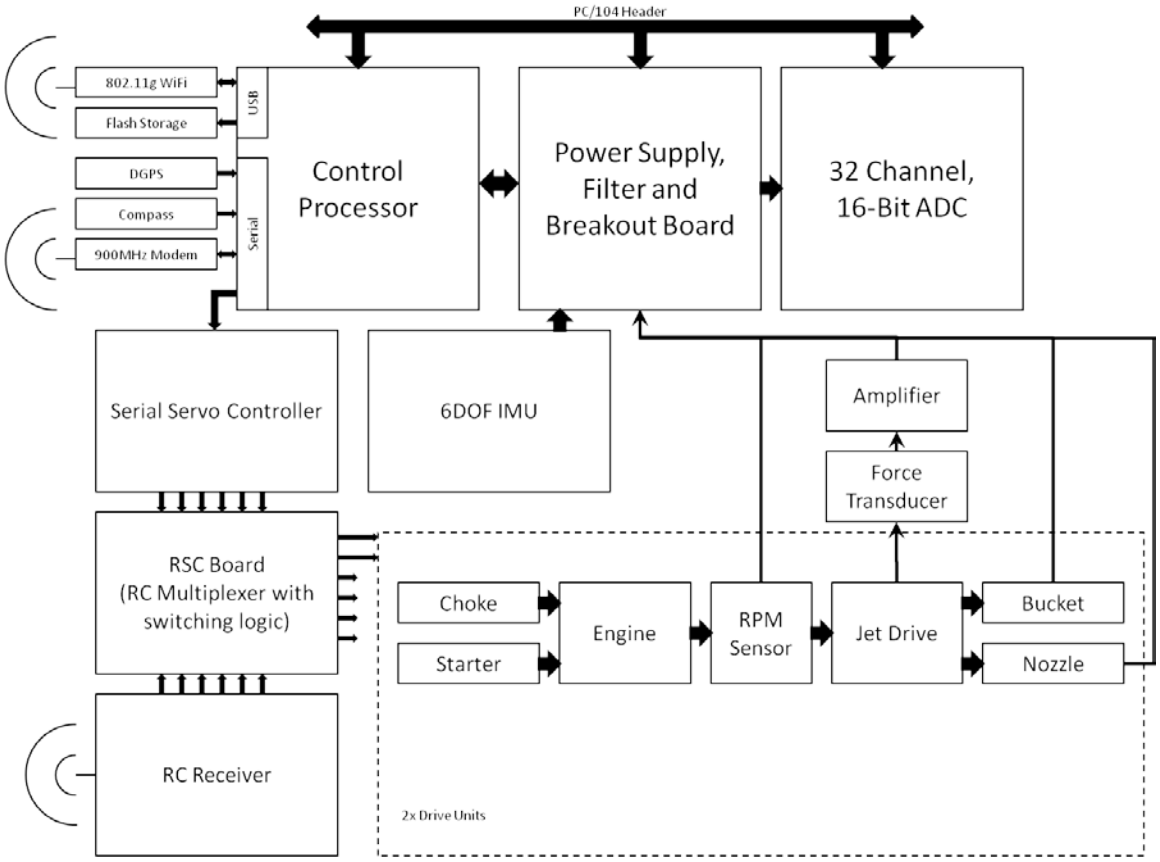

Figure 7: *USV12* Hardware functional block diagram. The portion in the dashed box is the only distinction between the *USV12* system and that of *GUSS*.

3.3.1 **Sensor suite**

Beyond practical constraints such as size, weight, and power consumption, any system designed must be capable of recording actuator input commands and the resulting USV response. For example, on the *USV12* inputs include engine throttle position, impeller RPM and nozzle angle for each propulsion module, whereas the outputs consist of all motions of interest, namely the position, orientation, velocity, and acceleration of the USV in the horizontal plane. The sensors used are enumerated in Figure 3.

The hardware platform was designed with integration of these sensors in mind. Specifications such as available power supplies and communication types are dictated by these sensors. The overall layout of the sensor suite can be seen in Figure 8. It should be noted that there are redundant measurements onboard; specifically, heading and roll-angle are measured by both the differential GPS and the compass. Overlapping measurements by different sensors are beneficial for both redundancy and sensor fusion purposes (a Kalman filter, for example).

Table 3: USV Sensor Suite Comparison

USV12	*GUSS*	Measurements
PNI TCM5 Tilt-compensated compass		Euler angles (Pitch, Roll, Yaw)
CSI Wireless Vector Sensor PRO Differential GPS	San Francisco Wireless FV-M8 GPS	Latitude, Longitude, UTC Time, Heading (*USV12* only)
Systron Donner MotionPak II IMU		Linear accelerations, rotational velocities
-	Xsens MTi-G IMU and GPS	Euler angles, Position (lla), Velocity (linear locally earth-fixed)
AMTI UDW3.0 Force Transducer and amplifier		Propulsion unit forces

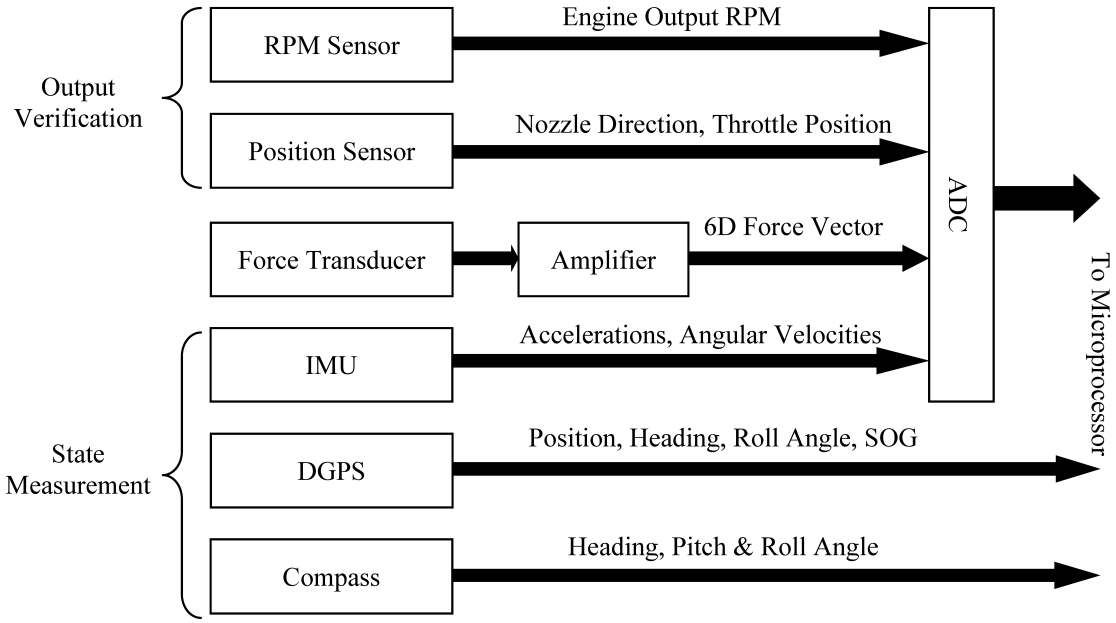

Figure 8: Sensor suite layout

3.3.1.1 Input Verification

Each subsystem between the software control algorithm and the physical actuator that drives the vehicle may introduce adverse dynamics to the total plant model. In order to establish as accurate a vehicle model as possible, the plant inputs will be measured as close to the actuators as is reasonable. Therefore, a method has been devised to measure the inputs with minimal system modification.

Furthermore, basic plant monitoring is important for solid control system design. The benefits are two-fold. Firstly, for controller operation, it is important to be able to verify, in real-time that the desired

actuator responses are achieved. This applies for both discrete actions, like waterjet bucket deployment, and continuous actions, like throttle adjustment for RPM modulation. In cases where there is a large actuator dynamic, it is important to be able to modulate control signals to achieve desired setpoints. An example would again be that the engine output of the waterjet impeller RPM on the *USV12* is directly related to the engine dynamics, which is itself highly dependent on uncontrolled variables such as fuel mixture, humidity, and engine temperature. Thus open-loop control of the system is quite difficult, and indicates the need for plant-monitoring sensors. Secondly, fault detection is an important capability for complex systems such as the *USV12*. The ability to detect in real-time actuator responses to control signals can give the user (and in this case, the control system) an immediate indication of a possible fault condition. These fault conditions can include an over-heated engine, low battery or fuel, water intrusion sensor faults, or propulsor unresponsiveness. For unmanned systems, it is particularly important that fault countermeasures be implemented to not only protect materiel assets, but also to protect users and the general public from errant behaviors.

For both the *USV12* and *GUSS*, each propulsion plant is driven by RC-type pulse-width-modulated (PWM) inputs. Inside each waterjet module (*USV12*) and each azimuthing thruster (*GUSS*), individual actuations are controlled by RC position servos. Each servo consists of a small permanent-magnet DC motor attached to a potentiometer with a control circuit that modulates shaft position. Each servomotor that controls an action of interest (e.g. throttle position or nozzle direction) is paired with a Hall-Effect potentiometer. This sensor, external to the internal servo dynamics, is used to monitor actuator response throughout operation.

A tachometer is mounted on the motor output shafts to measure shaft rotation speed. In open-loop testing, this directly measures the actuator output. For the *USV12*, this is a magnetic tooth pick-up, triggered by rotation of the starter gear on the input side of the engine. For *GUSS*, this consists of an optocoupler circuit measuring the rotation of the PMDC motor output shaft on each propulsion unit. In either case, the tachometers are connected to a Schmidt trigger and single-shot filter whose output characteristics can be calibrated specifically to the application depending on the selection of passive component values.

The total result from these modifications and additions is the capability to measure plant inputs during open-loop maneuvering trials, and provide actuator feedback during closed-loop trials. Furthermore, this approach requires minimal modifications and additions to the existing prototypes.

3.3.1.2 State Measurement

In order to measure the vehicle states, a suite of sensors is required. The primary navigation suite contains a differential GPS (DGPS), a tilt-compensated digital compass, and a 6DOF inertial measurement unit (IMU). These sensors have been specified in Section 3.3.1.

The differential GPS currently used on the *USV12* is the Vector Sensor from Hemisphere GPS, Inc. This unit utilizes a pair of GPS antennas spaced 2 m apart to provide sub-meter positional accuracy at 5 Hz and 1 degree RMS pitch or roll accuracy and 0.1 degree RMS heading accuracy at 10 Hz (depending on antenna mounting with respect to body frame). The unit also interfaces with an antenna that receives DGPS updates.

The tilt-compensated compass installed inside the GNC package itself (and thus implemented on both USV platforms) is a TCM5 from PNI Inc. This compass provides an accuracy of 0.3 degree RMS heading and 0.2 degree RMS pitch and roll at 25 Hz.

The IMU contained inside the GNC is a MotionPak from the Systron Donner Corporation. It is capable of measuring rotation rates up to 100 degrees/second and accelerations up to 10 g's. The IMU has analog outputs, so hardware pre-sampling filters must be used to reduce image distortion in high frequency bands. The filter corner frequencies are chosen to satisfy Nyquist sampling limit with some margin. The IMU specifications can be found in Table 4.

Table 4: Systron Donner MotionPak IMU Specifications

Rate Range	**+/- 100 deg/sec**
Rate Interface	Analog, +/- 2.50 VDC
Rate Bandwidth	> 60 Hz
Acceleration Range	x,y: +/- 2 g, z: +/- 3 g
Acceleration Interface	Analog, +/- 7.50 VDC
Acceleration Bandwidth	> 300 Hz
Voltage	+/- 15 VDC
Current	< 270 mA

The last major sensor utilized on the *USV12* is a 6DOF force transducer from Advanced Mechanical Technology, Inc. The UDW3 is a waterproof, 6-axis force element. The x and y axes of this device have dynamic ranges of 220 N force and 11 Nm moment, whereas the z axis has ranges of 440 N and 5.6 Nm.

The force transducer will be mounted between one of the propulsor modules and the main hull. An amplifier, the AMTI Miniature Amplifier Board Set (MABS), will be used to excite the strain bridges, and to amplify the output from the transducer itself to a full-scale analog signal. The MABS also has the capability of transmitting measurements via RS-232 serial, although this option will be reserved. The UDW3 will be mounted between one of the propulsion units and the forward hull section. A force measurement module was fabricated to integrate the UDW3 into a mounting bracket that connects the propulsion units to the existing pinned joints in the forward pontoons. These additions are modular, allowing the vehicle to be returned to its original state.

3.3.2 **Control Hardware Design**

The control subsystem itself consists of a single-board-computer (SBC) connected to the sensors on the input side and to a serial servo controller (SSC) on the output side. The total effect is the ability to perform closed loop control.

The SBC controlling all vehicle operations is the TS-7800 by Technologic Systems. The PC/104 form-factor board runs on a 500 MHz ARM9 processor with 128 MB of RAM and 512 MB of integrated flash. It also has card readers for both SD and micro-SD flash devices. The board has 2 USB ports, and a 10/100/1000 Ethernet port. The processor also has up to 10 serial ports, of both RS-232 and TTL variety, as well as several digital input/output (DIO) and analog-to-digital (A/D) lines.

The SBC is connected via the PC/104 header to two 16-bit analog-to-digital converter (ADC) cards, which allow sampling of all of the analog signal lines, from the IMU, force transducer output, and servo potentiometers. The SBC is connected via full-duplex serial (TTL level) to a 12-channel serial servo controller (SSC). The SSC converts serial commands into RC-style PWM signals. This allows the SBC to interface with all of the actuators onboard both the USV prototypes.

3.3.3 **Communications**

Several communication paths are integrated into the hardware design. This consists of a serial modem, an 802.11g wireless modem, and a remote control receiver and radio pair (see Figure 9).

The serial modem is a 900 MHz Xtend from Digi International. The modem has a 60 km line of sight (LOS) range with a properly matched antenna pair, and is used to relay information back to a base station and provide some rudimentary override capabilities. This modem is connected to the primary serial telnet login port on the main control processor, providing an end-all connection if the faster 802.11g link fails.

The USB 2.0 802.11g wireless modem used operates on the 2.4 GHz band, and allows communication to any WiFi enabled device. In particular, this wireless link provides quick exchange (up to 6.75 MB/s transmission) of data and vehicle software. It is the preferred link because of its speed, but is limited to a range of 60-100 meters.

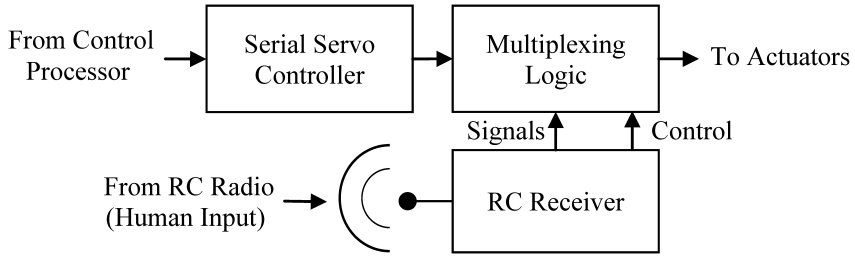

Figure 9: Remote Supervisory Control Functional Schematic

Lastly, the RC receiver provides a link for instantaneous override capabilities. Basically, the RC controller gives a human supervisor immediate control via a hardware-level override, at ranges up to 300 m. The RC receiver's role in the remote supervisory control (RSC) subsystem is described in more detail in Section 3.3.4: Remote Supervisory Control System Design.

3.3.4 **Remote Supervisory Control System Design**

One of the driving requirements for control system integration is to maintain vehicle controllability throughout testing. As the safety of people and property is of the utmost importance, a method has been devised to allow the control hardware to interface directly with the RC system during autonomous operations, but allows instantaneous override by human operators. Serial sentences from the control processor are received by a serial servo controller (SSC) – the Pololu MiniMaestro 12. The serial nomenclature is converted from the binary representation into a pulse-width modulated (PWM) format. Simultaneously, switching circuitry is monitoring the status of the RC receiver. Depending on the state of one of the channels on the receiver, as well as the signal quality, the switch outputs either PWM from the SSC or the RC receiver itself. This hardware enables remote supervisory control and fault sensitive operation even in the case of main processor failure.

3.4 VEHICLE OPERATING SOFTWARE DESIGN

The implementation of any data acquisition or autonomous control system requires a software backbone, the development of which encompassed a substantial portion of this thesis effort. The goal from the outset was to create a software architecture capable of satisfying the both open and closed loop testing requirements. The basic requirements dictated things such as timing and the amount of computer memory available to the CPU. Moreover, from its conception, the vehicle operating system (VOS) was to act as a jumping-off point for future autonomous vehicle efforts at FAU. As such, broad requirements pointed to an architecture that was modular in nature. Lastly, while it would be desirable to implement new user tasks easily in the existing control system, it is also conceivable that different hardware would be used in the more distant future. Thus, software portability was also an important consideration. In order to satisfy all of these requirements, a multi-threaded OS, running on a real time Linux kernel, written in the C language was selected.

3.4.1.1 GNU/Linux

Linux specifically refers to a kernel, originally conceptualized to replace proprietary UNIX kernels, written by Linus Torvalds [24]. The kernel provides the backbone, the interface with peripherals, memory, and the processor for all other utilities and software. The kernel manages virtual memory, paging and other important low-level functionality. GNU, which stands for GNU's Not Unix, is an ever-growing set of tools and utilities that interface directly with the kernel to provide the full functionality of an operating system. GNU was written largely by Richard Stallman and the Free Software Foundation. Linux and GNU together form the operating system that many people simply refer to as "Linux". The primary advantages of a GNU/Linux system are that it is open-source, it is free, and it is well documented.

GNU/Linux is often packaged in different forms, with different sets of software, to satisfy different user requirements. The varying shapes are referred to as distributions, or ‘distros’. Each distro targets a different user application. Some distributions, such as Ubuntu, are full-blown PC operating systems, providing the GUI elements that consumers today associate with a typical computing environment. Conversely there are stripped down versions, for example μLinux or Slackware, that are much more lightweight, aiming for integration on embedded or resource constrained projects. The distribution we are using is Debian, which is what is provided by our SBC manufacturer. This particular Debian distribution

lacks the graphical components of many of the utilities, as the SBC is designed to run “headless”; that is, without a keyboard or monitor attached.

The software implementation also makes great use of the GNU element of the GNU/Linux couplet. Used most frequently is the GNU C Compiler (gcc), along with the GNU C Library (glibc) and the GNU debugger (gdb). These tools form the basis of the code-compile-test cycle.

3.4.1.2 Real-time Operating System

Typical computing systems, such as a personal computer, are usually not considered real-time systems. That is, it is difficult to guarantee that a piece of user-run code will run as fast and as consistently as might be specified. On PC's, this is primarily due to the fact that there are so many processes running at a single time, and each one is generating what are called "asynchronous interrupts".

An interrupt, in the most abstract sense, is a flag generated by some software event, indicating that the state of some subsystem has changed which requires management by some piece of software (interrupt handler) that was previously not queued to run. Interrupts, for the most part, are random events, i.e. asynchronous (they do not conform to a schedule). This nature causes any piece of software that uses them to be undeterministic. Said differently, this means that given the initial starting conditions, and knowledge about the running software, one cannot predict the state of the machine at any given time. In complex systems, such as PCs, where there are many processes and subsystems all running together at once, asynchronous interrupts happen frequently and from completely random sources. This wholly non-deterministic nature of the system means that achieving any kind of strict timing requirement is impossible, if not difficult.

The term *real-time* is used to indicate software that runs with particular timing requirements. There are variations on the term *real-time*, such as *hard* and *soft*, indicating how strictly a particular software construct adheres to timing requirements. Real-time operating system satisfy their namesake through various means, including pre-emption, kernel demotion, or other methods. The discussion of these methods is outside the scope of this thesis, but more information, specifically about GNU/Linux based RTOS, can be found in [25], [26] and [24]. An important point is that there are classes of software that allow user software to run with specified timing characteristics. Furthermore, the options vary in that some of these solutions are open-source, while others are part of commercial packages.

For use in this project, a real-time Linux kernel was used. The variant being used is called the *rt-preempt* patch, originally maintained by Ingo Molnar. Originally, this patch was a branch of the mainline

Linux kernel, but it is now being fully integrated into the full kernel as a compile option. This patch, along with a high-resolution timer (hr-timer kernel option) transform the basic Linux kernel into an RTOS, capable of running tasks with average jitter (deviation in actual run time) of less than 100 µs, depending on hardware [27] [28] . For our purposes, this latency falls well below the threshold of significance. Significant synchronous tasks of the USV software are running at 5 Hz, so a jitter of less than 100 µs corresponds to 0.0005% deviation, an unnoticeable time span when compared to the USV's system dynamics.

3.4.1.3 POSIX Multi-Threading

Multi-threading is a term used to describe a software architecture that uses parallel processing. This parallel processing may or may not occur on multiple processors. In our case, our system uses a single processor (single core). In the most basic terms, multiple pieces of software run at the "same time" in order to achieve the goals of the programmer. Basically, parallel programming allows separate pieces of code to run concurrently. In multi-core or multi-processor systems, distinct portions of code are run on each core. This not only allows greater throughput, but more importantly for control system software, it allows different tasks to be operating at different speeds. This is the alternative paradigm to serial processing, in which all code runs in some programmatically specified order. For interface with asynchronous devices, this is a more difficult architecture to handle, because the time delay between each task iteration is very difficult to modulate. Multi-threaded software allows much simpler and precise control. Even on single-core systems, multi-threading is achieved by *time-slicing*, in which each thread is apportioned a very small amount of time (called a *quanta*) on the processor by the kernel. The result is software that appears to run in parallel, even though there is only a single processor performing the calculations. This approach is highly advantageous even with single core/processor systems where it is necessary to run different segments of code at different speeds, as is the case with a USV control system.

POSIX is a specification for the C programming language, which declares various functions and capabilities. Included in this superset are threading functions, called *pthreads*. The pthread library provides the utilities for creating, initializing, running, exiting, cleaning up, and destroying threads in user software. The control system described here relies heavily on threading capabilities.

The benefits of multi-threading come at a cost. Due to the fact that different segments of code are running in parallel as opposed to in serial, multi-threading raises issues with regards to reentrancy and resource sharing. The major sticking point for threads is that each thread within a process has unrestricted

access to a shared memory space, as depicted in Figure 10. Concurrent access to global variables potentially allows software bugs to arise when read/write cycles from uncoordinated threads overlap [29] [30]. This is handled by *locking* areas of memory when they are being accessed. The tool used to protect important segments of code is called a *mutex*, which is short for mutual-exclusion lock. Whenever a thread must access a shared area of memory (to read or write data), it requests to lock the mutex associated with that data from the kernel. If the mutex is unlocked, this indicates the memory is not being accessed by another thread, so the requesting thread is free to perform its critical function of reading or writing to that area of memory. If the mutex is already locked, the programmer either elect for the program to wait until it is unlocked, or continues along without accessing the shared memory in question.

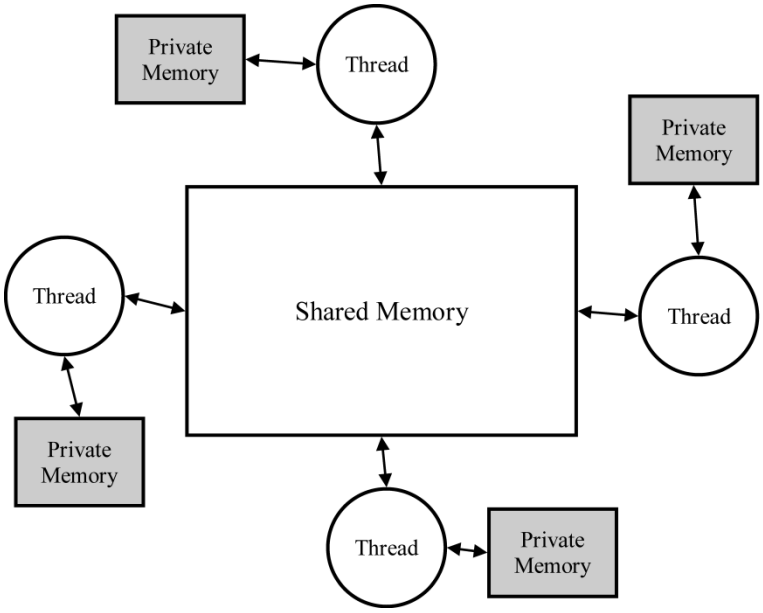

Figure 10: Memory allocation scheme in multithreaded software,

The advantage of passing information through a centralized shared memory space is that it greatly decreases the *coupling* between individual threads, and increases portability and modularity of the system as a whole. By placing the data in a shared space, other threads that may be integrated later in the development process will have easy access to the information. Similarly, because the operations of each thread are largely decoupled from each other, threads may be phased out, merged, split, or replaced altogether.

3.4.2 **Software Design and Implementation Methodology**

As previously mentioned, the C language is utilized to implement the entire VOS. The GNU C Library is used, with the GNU C Compiler. More specifically, the compiler used is a cross-compiler allowing software to be compiled on a host machine and then moved to the TS-7800 SBC. The actual creation of software and the associated debugging requires an enormous number of "code-compile-test" cycles, and so an efficient procedure, in terms of hardware and software configuration, is required.

3.4.2.1 Hardware Setup

In order to accommodate the continuous "code-compile-test-debug" cycle, an appropriate test stand must be set up. The hardware connection scheme used for software design is shown in Figure 11. This setup accommodates multiple access routes to the target hardware throughout the debug process, accommodating range or speed, as is appropriate. Furthermore, the serial connection provides a final hard-wired connection in the event of network lockout, or total operating-system failure. The TS-7800 has a boot loader that may be interfaced with to repair the Linux operating environment in the case the file system becomes corrupted.

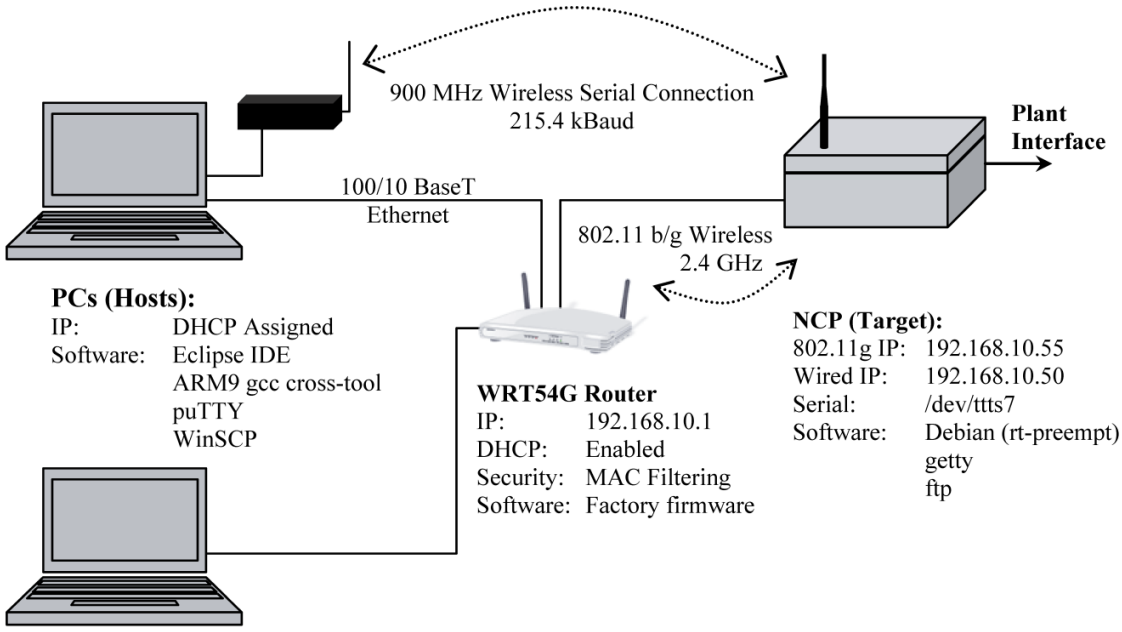

Figure 11: Software coding-debugging setup.

The hardware setup also accommodates multiple access points from separate debug computers. This allows several individuals to access the target hardware simultaneously. This method is safe assuming that

each programmer makes his resource requirements known to his collaborators. Simultaneous, unprotected access of any given resource produces undefined behavior, and thus such interactions should be accounted for. For instance, two users should not be running software simultaneously that accesses a specific file without some type of resource access protection.

3.4.2.2 Software Development Setup

In order to facilitate all of the code-test-debug processes that continuously happen during software development, many different software packages and utilities are used. The actual process of writing and compiling code occurs in an integrated development environment, the Eclipse IDE. This software is a modular platform that contains many of the functions that a programmer uses during the development cycle. Specifically, Eclipse has a text editor, interface for a compiler, a file transfer protocol client (FTP), a debugger, and an error parser. The most important component utilized for this project is a cross-compiler (also called a cross-tool) that is configured to generate binary executables that are compatible with the ARM9 architecture. This portable utility allows software to be compiled on non-ARM9 systems, which would be typically faster than compiling on the target itself. It should be noted here that because the development team was small (one or two people), there was little issue with code integration between team members. However, in cases of larger code bases or larger team sizes, it is common practice to use software versioning tools (such as Apache subversion, SVN, or Bazaar, BZR). These act as managed databases that maintain code updates from different developers to allow synchronous development and integration of total system software.

3.4.3 **Software Architecture**

The requirements of the software are manifold, but said summarily, the VOS architecture must be structured such that all asynchronous signals from sensors and peripherals are caught, and all scheduled deadlines are met. The core architecture of the VOS revolves around the parallel programming, or multi-threaded concept. A flow chart of the method of thread creation and rejoining is depicted in Figure 12 and specific thread descriptions are given in Section 3.4.3.1.

The structure of the software architecture developed in this thesis lies in between the approaches used in ROS and LCM. Inter-thread communication is conducted via a shared memory area, where access is controlled via a mutex-protection scheme; at any given time, only a single thread may have permission to access a particular data structure. However this permission guarantee occurs at the thread level, and is not

managed, so additional modules must appropriately adhere to access rules. Data is technically available to be read or written by all threads, like the LCM approach; there is no sense of hierarchy or priority. Access rules are a logical extension of the mutex schema to the single-writer multiple-reader case; if a thread at any point needs to write or read to the data in the shared space, this access must be protected by an lock/unlock procedure. A master thread exists, but doesn't manage the configuration of the inter-thread communications, just the start-up and shutdown behaviors, as well as high-level fault monitoring of the system., similar to the ROS approach.

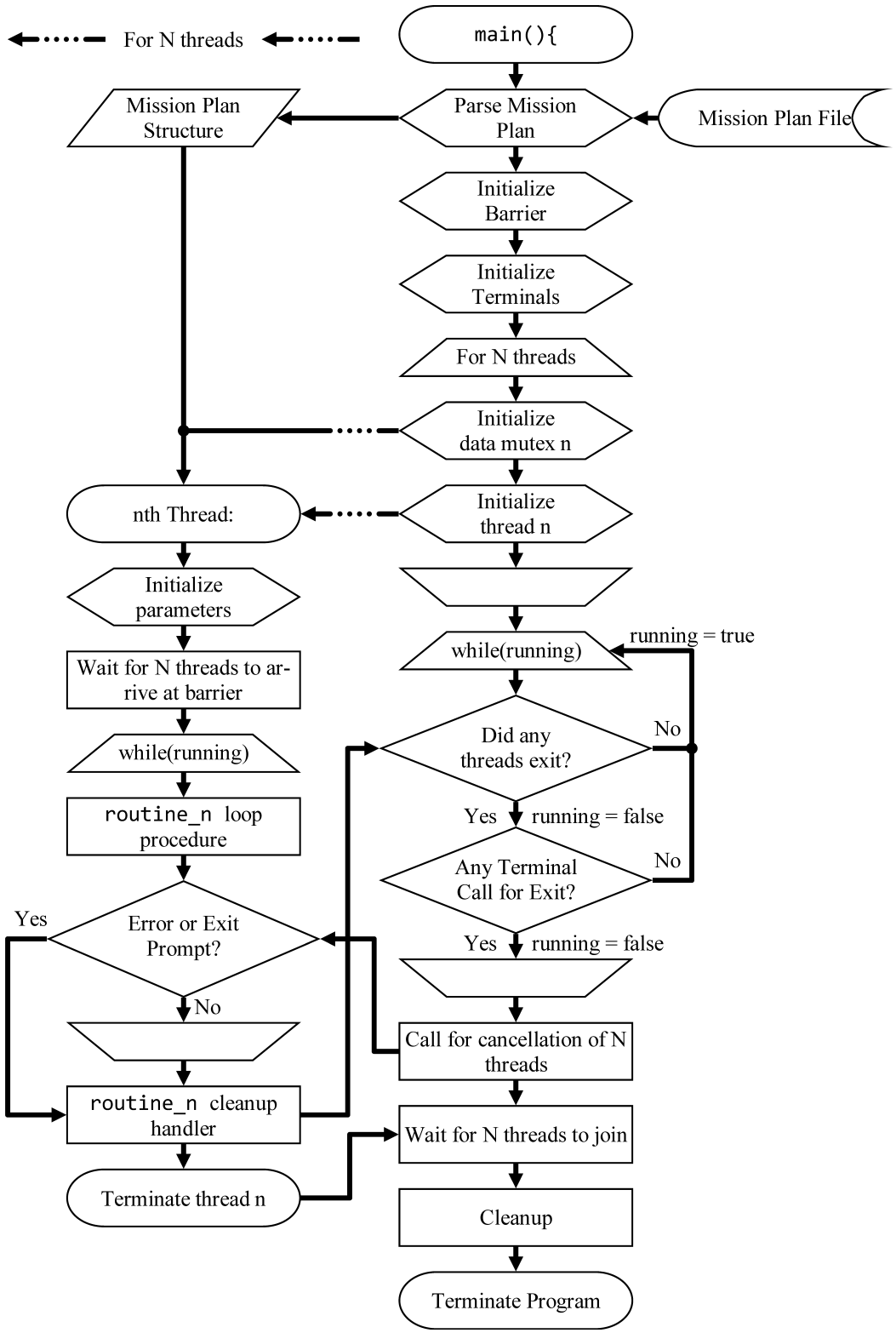

Figure 12: Thread creation and joining mechanism.

The procedure presented here is common to all threads, regardless of their specific function. This template allows for safe thread creation, cancellation and joining. In case of an unexpected error in a thread during run-time, that particular thread will run an associated cancellation handler (such as closing files and releasing mutexes), and then signal the master thread (`main`) that it has exited. Upon this signal, main will then send the cancellation signal to all threads, who will then each independently initiate safe shutdown procedures.

3.4.3.1 Thread Descriptions

Each major software task is written in a separate thread. More specifically, the software is split into logical units based on functionality and timing requirements. The thread interactions are shown in Figure 13. Items with a rounded rectangular box are a physical hardware peripheral. Items labels with the suffix `_thr` are spawned threads. Signals labeled with the suffix `_t` are data structures being passed *via the shared memory space*. As mentioned previously, passing these data to the shared memory space decreases the coupling that would occur if the information would be passed directly between threads.

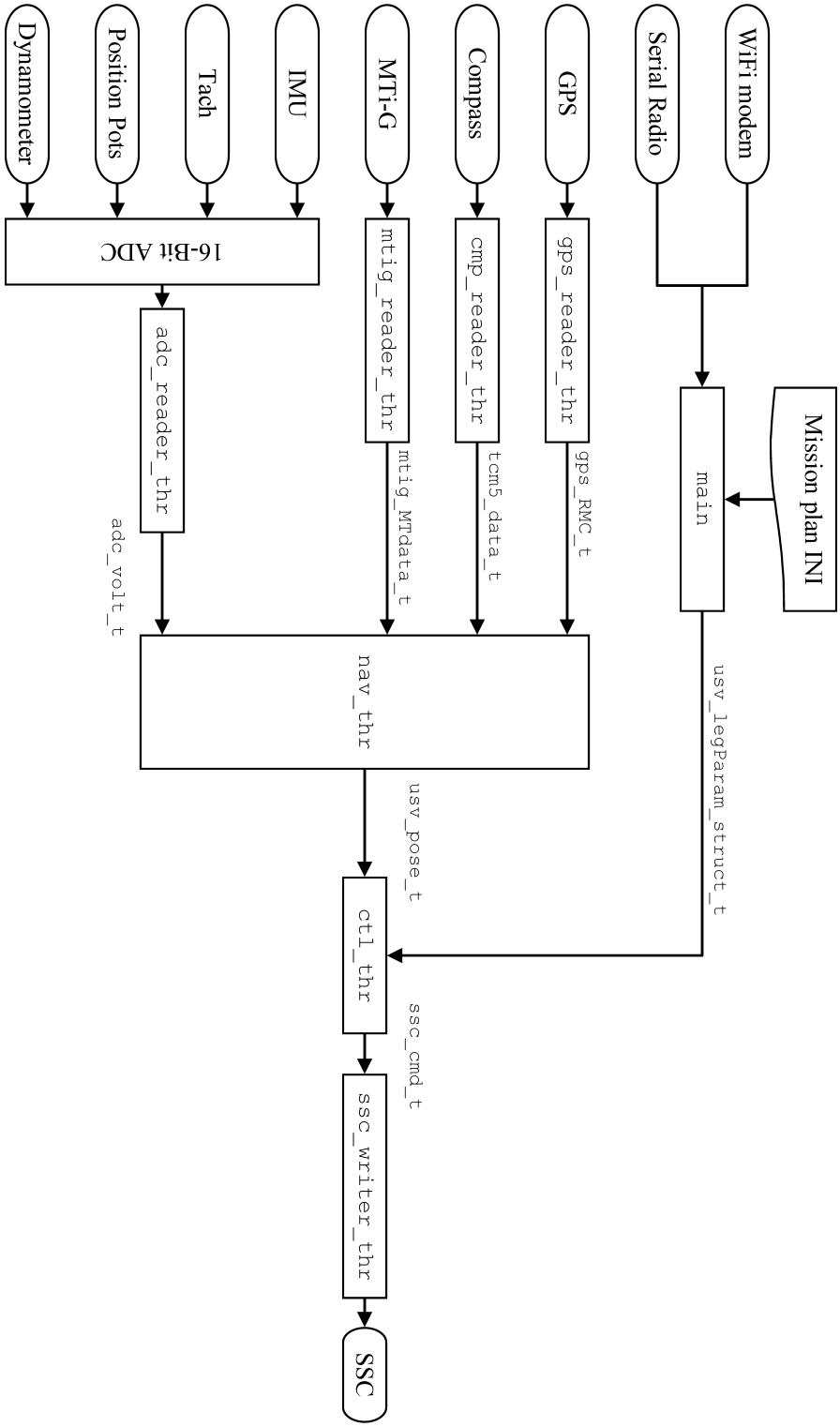

Figure 13: Thread Interactions and Data Types

- **Thread:** `main()`
  - **Timing:** Asynchronous or timed, variable rate
  - **Associated mutex:** none
  - **Cleanup handler:** None, is sub-thread creator/joiner
  - **Description:** This is the original parent thread. When the VOS software first initializes, this is the only thread in existence (it is the entire process). This thread is responsible for the spawning of all other threads, as well as handling their cancellation and closing. This thread also handles the mutex creation and destruction for each important data structure. This thread does not modify any global data structures in any way, it is a read-only thread, therefore it does not itself lock or unlock any mutexes. Furthermore, this thread monitors the connected terminals to determine if there is a command from the human operator, usually a termination request. This thread can also used to output global data to the remote terminal.
- **Thread:** `cmp_reader_thr()`
  - **Timing:** Paced, 20 Hz
  - **Associated mutex:** `cmp_raw_mutex`
  - **Cleanup handler:** `cmp_reader_cleanup()`
  - **Description:** This thread is the device interface thread for the connected compass. The compass being used is operated in a polling mode, where a request is sent from the SBC to the device to initiate a transmission of the latest data sample. Operating in polling mode only increases the SBC's overhead infinitesimally, and allows data rates to be controlled from the master device without having to reconfigure the slave, or sensor in this case. In particular, this method is used for the TCM5 compass because the message format is not particularly conducive to live byte-by-byte parsing. This thread constructs the binary datagrams from the message stream, and parses the data accordingly. The data, now converted to floating point (single), are stored in the global memory space and protected by the associated mutex.
- **Thread:** `gps_reader_thr()`
  - **Timing:** Asynchronous
  - **Associated mutex:** `gps_raw_mutex`
  - **Cleanup Handler:** `gps_reader_cleanup()`
  - **Description:** This thread is the entrance point for GPS data. It operates asynchronously, simply waiting for activity on the associated serial port. It constructs NMEA0183 messages

from the incoming stream of data, and parses the RMC and GGA strings. The GPS outputs these messages at 5 Hz, regulated internally and by reference to the satellite signal. The parsed data structures are stored globally and protected by the single associated mutex.

- **Thread:** `mtig_reader_thr()`
  - **Timing:** Asynchronous
  - **Associated mutex:** `mtig_raw_mutex`
  - **Cleanup Handler:** `mtig_reader_cleanup()`
  - **Description:** This thread is the interface driver for the MTi-G motion sensor. The thread first configures the device, and then reads the asynchronous measurements that are being broadcast by the device. The measurements are transmitted via a MTi-G-specific binary messaging scheme through a serial port. The timing from the perspective of the VOS is asynchronous, but the update frequency is set to 20 Hz (configurable at compile time).
- **Thread:** `adc_reader_thr()`
  - **Timing:** Paced, roughly 5 Hz.
  - **Associated mutex:** `gps_raw_mutex`
  - **Cleanup Handler:** `gps_reader_cleanup()`
  - **Description:** This thread manages the interface with the ADC board. It is paced at roughly 5 Hz, but is able to fluctuate in order to prevent FIFO overflow on the ADC itself. The 16-channel ADC samples at 100 Hz, and fills its buffer internally, waiting for user software to read off of it. This thread reads the data, converts the ADC count values to voltages and then finally to the appropriate units. The resulting structure is stored globally and appropriately protected.
- **Thread:** `ssc_writer_thr()`
  - **Timing:** Paced, 5 Hz.
  - **Associated mutex:** `ssc_setpts_mutex`
  - **Cleanup Handler:** `ssc_thr_cleanup`
  - **Description:** This thread performs low-level controller calculations. It also deals with the low-level timing and specifics of writing to the serial servo controller (SSC). The output speed is set at 5 Hz. At this rate, the thread sends out complete command strings in the SSC protocol, commanding all of the configured actuators simultaneously. The thread receives setpoints from the navigation thread.

- **Thread:** `nav_thr()`
  - **Timing:** Asynchronous or timed, variable
  - **Associated mutex:** `nav_out_mutex`
  - **Cleanup handler:** `nav_cleanup()`
  - **Description:** This thread accesses GPS, compass, and IMU sensor data, the mission plan structure, as well as the onboard real-time clock to make pose estimations. For the current version of the VOS, this thread simply accesses the raw data from several sensors, performs some basic conversions and coordinate transformations, and then makes the results available to the controller thread. In the future, this is where a navigation Kalman Filter would be implemented to fuse all of the various sensor inputs into a single estimate of vehicle pose.
- **Thread:** `ctl_thr()`
  - **Timing: Timed, currently 5 Hz.**
  - **Associated mutex:** `ctl_out_mutex`
  - **Cleanup handler:** `ctl_cleanup()`
  - **Description:** This thread acts as both the guidance and control thread. Currently, joining these two functions into a single thread is advantageous for cases where closed-loop control is effectively bypassed for open-loop testing. By combining both guidance and control roles into a single thread, this implementation is programmatically much simpler. Depending on the pre-defined mission plan, this thread reads navigator outputs to make both setpoint declarations and control calculations, the results of which are posted for use by the SSC writer thread.

These threads together form the core framework for the entire control system program. Each thread calls other subsets of functions. The framework relies heavily on standard C functions, as well as many custom user-written functions that are designed specifically for each major sensor or task.

3.4.3.2 Mission INI File

In order to promote usability by end-users and ease the iterative process of reprogramming the platform for a given mission profile, a method of quickly reconfiguring the GNC package for new missions was implemented. An INI file parsing library was integrated into the mission planning component of the run-time initialization to read a user-written configuration file that contains mission directives such as controller style, gains, waypoint and heading setpoints, timeout values, closing distances, and other

pertinent values. Using a INI parser library released by Nicolas Devillard under the MIT License [31], the run-time software will read in and parse a mission configuration file, assuming it follows certain formatting and keyword guidelines. A small piece of a larger INI file is listed in Listing 1, but a full template can be found in Section 6.2.3.

Listing 1: Mission INI file example

```
[SEG1]
type            = OPEN_START;
waitForGps      = 1; Wait for GPS before starting?
gpsPeriod       = 20; Time between GPS retries at init.
waitTime        = 5; How long to wait before starting

[SEG2]
type            = CLOSED_WAYPOINT;
wait            = 0; How long to wait before we move.
latitude        = 26.055595; Target latitude in decimal degrees.
longitude       = -80.113209; Target longitude in decimal degrees.
speedCmd        = 1; Command velocity [m/s]
dt              = 0.2;  controller discrete dt
kp_heading      = 120;  Proportional speed gain.
ki_heading      = 0;    Integral speed gain.
kd_heading      = 0;    Derivative speed gain.
kp_speed        = 80;   Proportional speed gain.
ki_speed        = 20;   Integral speed gain.
kd_speed        = 0;    Derivative speed gain.
duration        = 80;   Maximum run time.
maxDistance     = 3;    Maximum acceptable distance to wpt, [m]
```

Every mission is divided into a series of segments, labeled as `[SEGN]` for segment `N`. The parser assembles a mission sequentially from these segments. Segments may literally be segments in space, i.e. a leg of a track between two points, or they may represent another event driven period in the mission. For example, there are segment types whose labels begin with `OPEN_`; the subtypes of this group include open-loop type mission segments, for straight line, circle, and zig-zag testing. The full template in Section 6.2.3 includes all of the possible types and all of the required parameters for each type, with descriptions.

There are several benefits to using this as a user-interface. First, it prevents having to compile specific mission details into the program executable. This speeds up mission reprogramming and reduces the possibility of the run-time code being accidentally modified. Second, the mission INI is simple human-readable file that may be modified by any text editor. Lastly, mission INI files server as a good log, along with the output data files, of what mission was being run at a particular time.

3.5 MOTION CONTROL STRATEGY

The USV12 and GUSS may both be maneuvered with up to four control actions; the USV12 has two water-jet RPMs that may be varied, as well as two nozzle directions, while GUSS similarly has two thruster RPMs and two azimuth angles that may be varied. In the most unrestricted cases, both the USVs are considered fully-actuated vehicles. That is, they have at least as many control actions as degrees of freedom. While this is may be preferred to an under-actuated vehicle, it presents a challenge from a control perspective, by giving the controller multiple solutions for any particular motion solution. The strategy taken in this thesis is to restrict the possible degrees of actuation to simplified cases where control laws become more self-evident. The approach taken here will be to first use the speed controller to modulate the USV's forward speed, then vary each thruster around the nominal speed using the heading error to induce the required turning moment. The commanded heading is determined by a guidance routine that calculates a look-ahead distance on the target path. Each individual controller contributes to the total control actuation using independent PID control schemes, with separately tuned gains. The setpoint for each state is determined by a set-point control scheme, using techniques presented above, such as an LOS or feed forward algorithm. The navigator runs concurrently making the most current state estimate available to both the controller and the guidance subsystem.

3.5.1 **Vehicle Model**

As discussed previously, there are many possible ways to design a controller, ranging from purely theoretical approaches to entirely heuristic ones. The initial approach here was a combination of heuristic and simulation methods. A system model was desired, but not necessarily required for controller synthesis, as the target platform was readily available. A dynamic model for the *USV12* was the goal of Mask (2011) through the use of various system identification tools to estimate model parameters based on measured USV response. A model of the real system was never realized, so a stand-in model was used with particular modifications for the specifics of anticipated USV platforms. The expansion and development of the control vector, $\mathbf{u}$, is of primary interest.

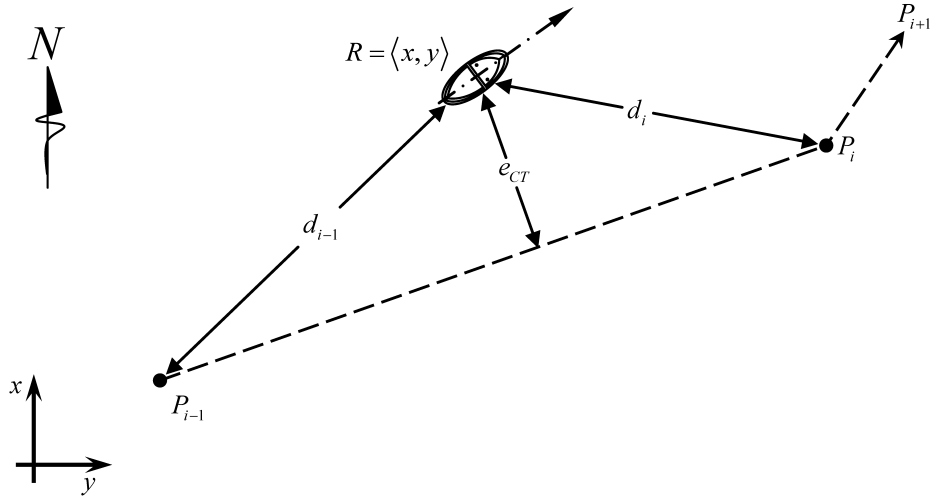

Figure 14: Path following coordinate definitions.

The control force $\mathbf{f}_{\text{control}}$ in Equation (2.2) is a function of both the state, $\mathbf{v}$, and of the control vector,

$$\mathbf{u} = \langle \omega_{\text{P}} \quad \omega_{\text{S}} \quad \theta_{\text{P}} \quad \theta_{\text{S}} \rangle^{\text{T}}. \tag{3.1}$$

The control actions consist of the impeller RPM, ω, and thruster direction, θ, for each the port and starboard thruster. The forcing function that arises from control input, $\mathbf{f}_{\text{control}}$ assumed to be a nonlinear function of both the vehicle state and the control input. That is,

$$\mathbf{f}_{\text{control}} = \mathbf{f}_{\text{control}}(\mathbf{v}, \mathbf{u}). \tag{3.2}$$

Results for vectored thrusters, as in [32] and [33], show that the apparent thrust angle is not the same as the nozzle direction. Further development of a model for this relationship is the main motivation for including of the force transducer system within the designed data acquisition system; the thrust measurements and subsequent thruster model development are outside the scope of this thesis and will be conducted in planned follow-on work. The role of the controller design is to formulate a strategy for manipulating the control vector, $\mathbf{u}$, to produce the desired response in $\mathbf{v}$.

The propulsor modeling is the area of greatest disparity between the *USV12* and *GUSS* implementations of the control design. In the case of *GUSS*, thruster modeling can be kept quite simple; the thrusters are considered to have a thrust proportional to the square of the propeller RPM, the line of action follows the thruster azimuth angle, and the response can be considered symmetric for both forward and reverse propeller directions (indeed, the props themselves are face-back symmetric).

The task in modeling the thruster force is to determine some relationship between the propulsion plant input and output. The azimuthing thrusters installed on *GUSS* are the single form of propulsion and maneuvering. The port and starboard thrusters, located at $\langle D_x, -D_y \rangle$ and $\langle D_x, D_y \rangle$, produce thrusts T_P and T_S in directions θ_P and θ_S, respectively. It is assumed that the line-of-action of each thrust component is in-line with each propeller center axis. It was shown in [34] that in cases where closely-spaced (less than approximately two propeller diameters) thruster vertical axes are co-linear with respect to the freestream flow, downstream thruster dynamics are significantly influenced by the upstream thruster presence. However, as is apparent in the diagram here, such a situation would only arise in instances of sway-dominant maneuvering. Furthermore, this operating mode is outside of the scope of the following controller design, so such secondary effects will be assumed negligible. Thus, because the angle of the thruster is directly controlled, the thrust angle is also directly controlled.

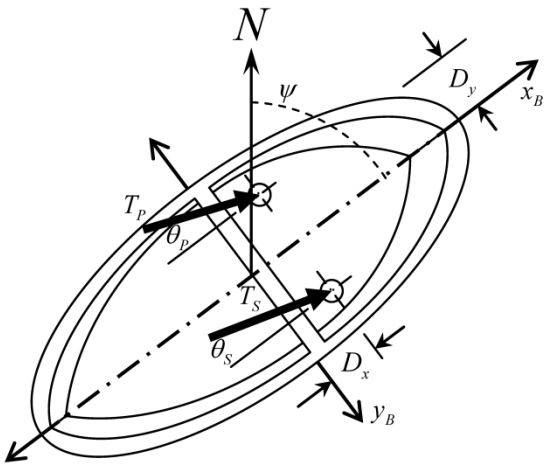

Figure 15: *GUSS* thruster configuration and term definition.

First, we consider a simple thrust model, similar to that presented in [35], is based on a quadratic relation between thruster rotational velocity and thrust,

$$T_i = \alpha_i |\omega_i| \omega_i . \tag{3.3}$$

Here, the thrust of the i^{th} thruster, T_i, is related to the corresponding propeller rotational velocity, ω, by a proportionality constant, α_i. Based on the mechanical and geometric similarities of each drive unit, the constant will be assumed to be equal among thrusters, $\alpha_P = \alpha_S = \alpha$. The assumption here is that there is little thruster dynamic – if the rotational rate is specified, that is sufficient to predict thrust produced. This approach doesn't consider electrical dynamics in this electro-mechanical device, nor does it take into

account any hydrodynamic effects, such as the advance velocity of the flow into the propeller (it is assumed that the propeller is always operating in a bollard-pull condition). The motor controller, an off-the-shelf component, is considered to be a black box. The motor controller commands produce an RPM output at the plant, and the internal dynamics are not considered. However, from experience, the thrusters (and accompanying power plant) are sufficiently over-powered that overall vehicle states, such as forward velocity, tend to have little effect on thruster response in normal conditions.

If we now consider the geometry as presented in Figure 15 along with Equation (3.3), an expression for the control force, $\mathbf{f}_{\text{control}}$, may now be developed:

$$\mathbf{f}_{\text{control}} = \begin{bmatrix} X_{\text{control}} \\ Y_{\text{control}} \\ N_{\text{control}} \end{bmatrix},$$

$$\begin{aligned}
X_{\text{control}} &= T_P \cos(\theta_P) + T_S \cos(\theta_S) = \alpha\left[\omega_P |\omega_P| \cos(\theta_P) + \omega_S |\omega_S| \cos(\theta_S)\right], \\
Y_{\text{control}} &= T_P \sin(\theta_P) + T_S \sin(\theta_S) = \alpha\left[\omega_P |\omega_P| \sin(\theta_P) + \omega_S |\omega_S| \sin(\theta_S)\right], \\
N_{\text{control}} &= T_P D_x \sin(\theta_P) + T_S D_x \sin(\theta_S) + T_P D_y \cos(\theta_P) - T_S D_y \cos(\theta_S) \\
&= \alpha D_x\left[\omega_P |\omega_P| \sin(\theta_P) + \omega_S |\omega_S| \sin(\theta_S)\right] + \alpha D_y\left[\omega_P |\omega_P| \cos(\theta_P) - \omega_S |\omega_S| \cos(\theta_S)\right].
\end{aligned}$$

These expressions are merely results of expanding the force-moment equilibrium conditions for each thruster unit. Commenting on these expressions, in their full form they are non-linear. However, for cases such as zero-azimuth the equations reduce significantly. Thus, when considering operation using purely differential rudder (variation of rpm with zero azimuth angle, the equations reduce significantly,

$$\begin{aligned}
X_{\text{control}} &= T_P + T_S = \alpha\left[\omega_P |\omega_P| + \omega_S |\omega_S|\right], \\
Y_{\text{control}} &= 0, \\
N_{\text{control}} &= T_P D_y - T_S D_y = \alpha D_y\left[\omega_P |\omega_P| - \omega_S |\omega_S|\right].
\end{aligned}$$

Thus, in the differential thrust case, even though the vehicle has three degrees of freedom (surge, sway, and yaw), only two are being directly controlled: surge and yaw. Moreover, the previously-fully-actuated system is now underactuated, as we only have two remaining degrees of control. This approach is taken largely due to simplicity. By "fixing" the azimuth angles to be constant, the control forces reduce to forms that have immediate intuitive relationships. Primarily, given a system where the surge and sway-yaw modes are decoupled (which we are assuming), the surge force is proportional to the sum of the squares of

propeller speeds, and yaw moment is proportional to the difference of the same values. By abstracting these differential thrusts using a *virtual rudder* term $\delta_r = \left\| \omega_p \right| \omega_p - \left| \omega_s \right| \omega_s \right|$, and a thrust term, $\delta_u = \left| \omega_p \right| \omega_p + \left| \omega_s \right| \omega_s$ we can rewrite the control actions,

$$\begin{aligned} X_{control} &= \alpha \delta_u, \\ N_{control} &= \alpha \delta_r. \end{aligned} \tag{3.4}$$

Within the limits of the linearity assumption, the two control modes are entirely decoupled. In other words, a controller can be devised to independently modulate surge or yaw control forces, without considering the consequence to the opposite channel. The control signal, $\omega_{\mathrm{P,S}}$ for each thruster can be rewritten as,

$$\omega_{\mathrm{P,S}} = \delta_u \pm \frac{\delta_r}{2}. \tag{3.5}$$

3.5.2 **Guidance Mechanism**

The guidance subsystem is the routine that makes setpoint declarations that are fed in to the controller. In order to produce a track-following controller using this underactuated control-system,, a *look-ahead distance*, l, is employed to calculate the heading setpoint at any given instant, as in Figure 16.

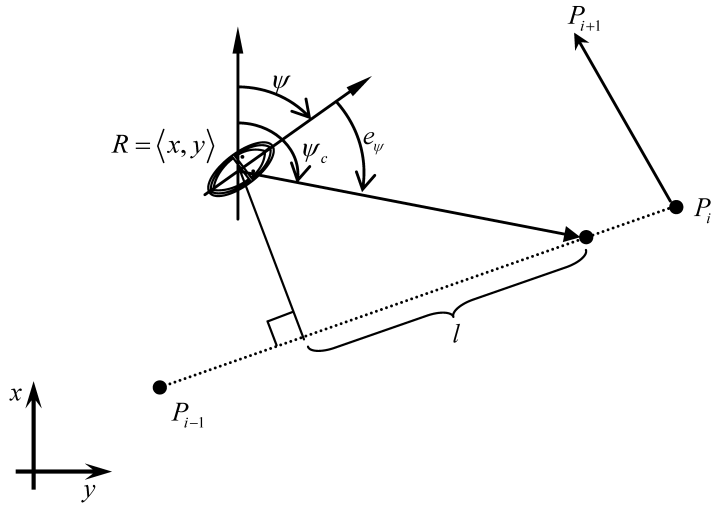

Figure 16: Guidance Schematic

The guidance algorithm accomplishes a form of track following control for an underactuated system. Instead of the heading setpoint being purely a function of the leg's static course, the heading setpoint is

constantly modulated to point at a point on the current track leg a specified distance along the same track. Upon approach of a track vertex (a waypoint), a *minimum closing distance*, $(d_i)_{\min}$, is used to manage the transition between track legs, as in Figure 17. When the USV reaches the minimum closing distance, the guidance system increments the segment counter, initiating a transition to the next operating mode. If the succeeding segment is another waypoint-following segment, the target waypoint is simply switched to the next waypoint in the list.

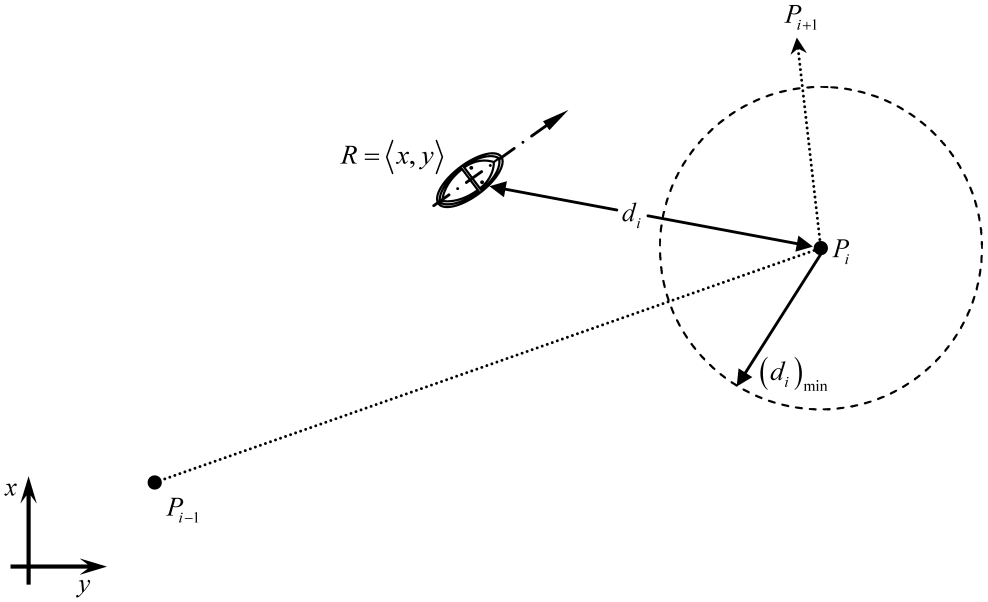

Figure 17: Definition of minimum closing distance.

Selection of the value for $(d_i)_{\min}$ is based on several factors. Firstly, the typically vessel turning radius and reaction time should be considered. Thus, the closing distance is usually proportional to vehicle length; the larger the vehicle, the larger the minimum closing distance. Similarly, a vehicle moving faster (which again usually corresponds to a longer vehicle) will require a larger minimum closing distance to minimize overshoot on the next segment. Another consideration in this selection is sensor accuracy. The specification of a closing distance that is smaller than the typical positional accuracy of any measurement does not lend itself to tighter maneuvering or better path-keeping abilities. Given sufficient data, in particular the turning rate for any given speed, one could potentially calculate a closing distance for any leg vertex angle which eliminates overshoot.

Thus, for the implementation realized here, where vessel length (1.5 m) was considerably smaller than the positional accuracy of GPS measurements (5 m), a closing distance of 5 meters was chosen. Moreover,

the validation platform has relatively sluggish turning behavior, so $(d_i)_{\min} \approx 3 \cdot LOA$ was deemed to be an appropriate parameter.

3.5.3 **Control Algorithms**

The control algorithm is a multi-input, multi-output (MIMO) system. Estimates of position and orientation (pose), and velocity are supplied by a basic navigation routine, along with set points based on a pre-defined mission plan. Together these parameters are used by the MIMO controller to drive the vehicle to the set points. This controller is composed of a series of single-input, single-output (SISO) proportional-integral (PI) controllers, modulating speed, and heading independently.

Conceptually, the PID control scheme is simple. The error, which is the difference between the desired and actual trajectory, of a particular variable is calculated. A proportional gain, $k_P^{(u)}$, is multiplied directly by the error. The result of this operation is Proportional control; the larger the error, the more control influence will be exerted. A second term is created by integrating the error over time and multiplying by an integral gain, $k_I^{(u)}$. The contribution from this integral term reduces the steady-state error. Together these form a PI controller. Such a controller is applied at various levels in the overall motion control algorithm; at the lowest level, a speed and heading controller in parallel contribute to the control output.

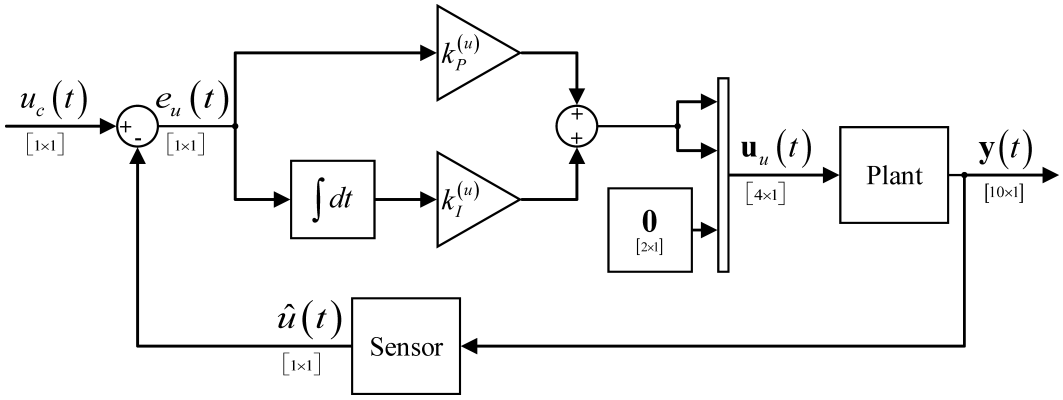

Figure 18: Speed PI Controller

3.5.3.1 Speed Controller

The lowest level controller in the entire control strategy is the speed controller. As shown in , the control algorithm follows the typical PI control methodology. The speed controller uses the error term created by comparing the commanded forward speed over ground u_C, with the measured speed over

ground, $\hat{u}$. Also notice here that the vector $\mathbf{u}_u$, which is the contribution of the speed controller to the total control vector $\mathbf{u}$, has a length of 4, with the last two elements being padded as zeroes. This is because our control vector includes the thruster angles, even though this controller in particular doesn't use those terms.

3.5.3.2 Heading Controller

Similarly to the speed controller, the heading controller is a PI controller. A commanded heading, ψ_C, is compared to a current heading, $\hat{\psi}$, to produce an error, e_ψ. This error is used in a PI controller with gain vector $\mathbf{k}^{(\psi)} = \left\langle k_P^{(\psi)} \quad k_I^{(\psi)} \right\rangle$, to produce a virtual rudder δ_r. This term is then bifurcated and added and subtracted to the port and starboard thrust commands, as in Equation (3.5) Similar to the speed component, the azimuth angle components of the control vector are padded as zeroes, and then these commands are passed to the plant.

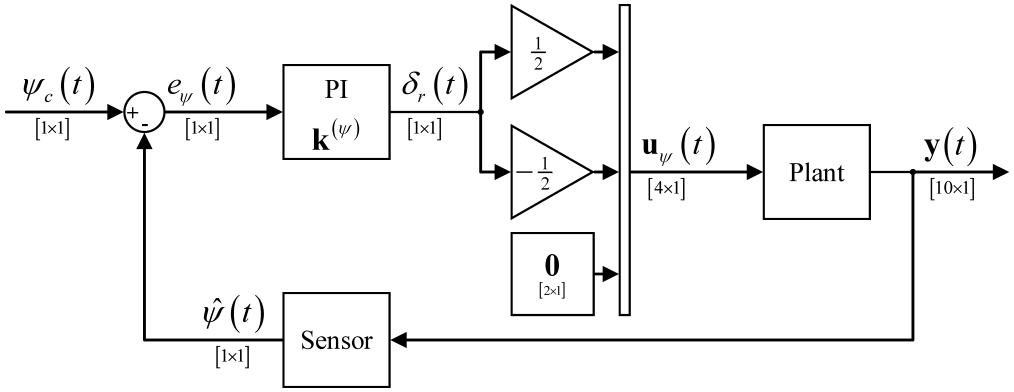

Figure 19: Heading Controller with virtual rudder implementation

3.5.3.3 Speed and Heading Controller Superposition

As stated in Section 3.5, the heading and speed controllers run in parallel, and their outputs are summed to contribute to the total control action. Thus, the two independent SISO controllers together form a MIMO controller. As presented in Section 3.5.1, this is possible due to the decoupling of the surge and yaw equations. Because the states are decoupled, we can separate the controllers for each state, tune and analyze them independently, and disregard influence from other controllers running in parallel. This parallel architecture will again be obvious in the discussion regarding the Simulink simulation construction.

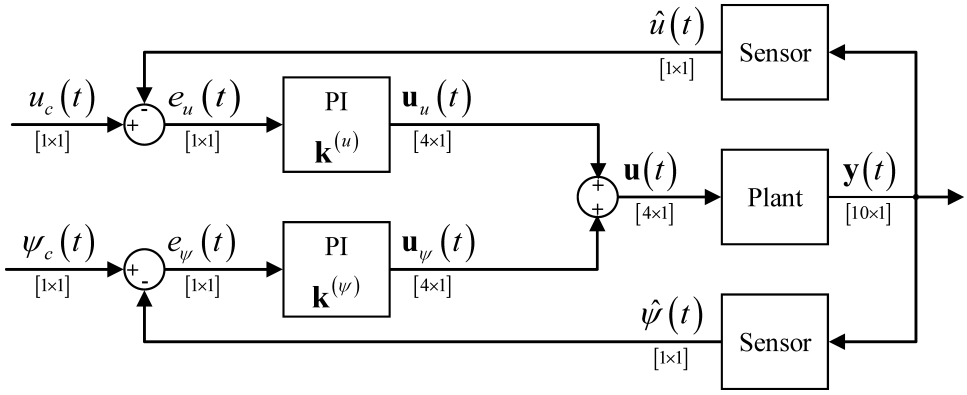

Figure 20: Speed and Heading Controller Superposition

3.6 SIMULINK SIMULATIONS

Using an existing surface vessel parameterization, a model was assembled in MATLAB Simulink. Then, a controller was constructed based on the theoretical model. This allowed some preliminary tuning to be done, but primarily served to validate controller behavior, rather than quantitatively estimate real controller gains. The implementation is presented here, with simulation results being presented in Section 4.2.

3.6.1.1 Simulink Block Diagram

The block diagram that forms the core of the simulation is given in Figure 21. The simulation uses a heading and speed schedule defined in the workspace to determine setpoints during run-time. These commands are fed into the heading and speed controller, which in parallel make control allocation calculations, which are then added together to form the total control output. These outputs are fed into a plant model, which is encapsulated in a Simulink S-function M-file Level 2 block. This block contains the discrete vehicle model and associated plant dynamics. The full m-file describing the S-function can be found in Section 6.2.1. The S-function contains relations between the discrete derivatives and the states themselves.

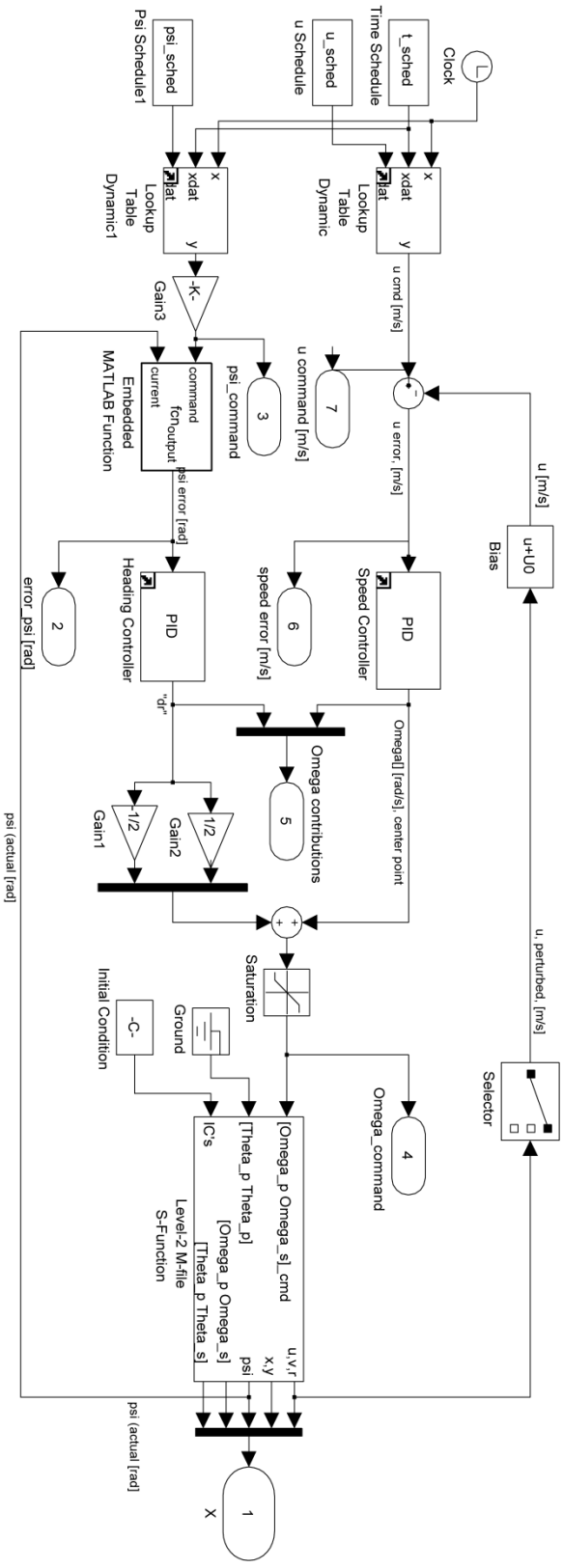

Figure 21: Simulink Controller Realization

4 RESULTS AND DISCUSSION

4.1 OPEN LOOP TRIALS

In order to better quantify the vehicle dynamics with respect to control implementation, the initial approach included using open-loop data sets to drive a system-identification analysis. Characterizing the vehicle dynamics with respect to plant input is known as system identification. The task of taking open-loop data sets and arriving at a valid performance parameterization is the work done in parallel by Ms. Janine Mask [36]. The results from system identification work are helpful both for dynamical simulation and controller development. Specific tests were chosen to excite certain modes of the vehicle dynamics, in order to obtain estimates of specific model parameters. The open-loop tests consist of:

- Straight line test
- Circle test
- Zigzag maneuver

These specific tests are popular in maneuvering testing because they reveal low-order response characteristics with relatively simple input functions. For instance, the straight-line test is valuable since it very directly shows the relationship between propulsor commands and forward velocity, invaluable information for speed-controller design. The results of the testing are surveyed here. However, due to some malfunctions of the GPS unit and the RPM sensors, the data was not of appropriate resolution and overall quality to provide a basis for model development.

4.1.1 **Straight Line Test**

Straight line tests were performed with the goal of identifying two important dynamical characteristics. The first is tracking bias or control asymmetry. Typically open-loop tracking response is an unstable phenomenon, [4], due to a destabilizing hydrodynamic moment (the Munk moment) and a unstable propulsor arrangement (most surface craft have the propulsor located aft of the lateral center of

lift). As such the open-loop straight line response isn't typically expected to be stable, however, it is useful to characterize if the system has a physical preference for clock-wise or counter-clockwise yaw drift.

The USV platform was subjected to various open-loop symmetric thruster commands, $\delta_u = C,\ \delta_r = 0, \left(t_i < t < t_f\right)$, and the GPS location and heading were recorded. For one such run, the vehicle measured vehicle track is given in Figure 22. As mentioned, with open-loop speed commands the vessel is track-unstable (it does not naturally follow a track). Over the 45 second run, the vehicle traveled roughly 25 meters southward (the starting point is $\langle 0,0 \rangle$).

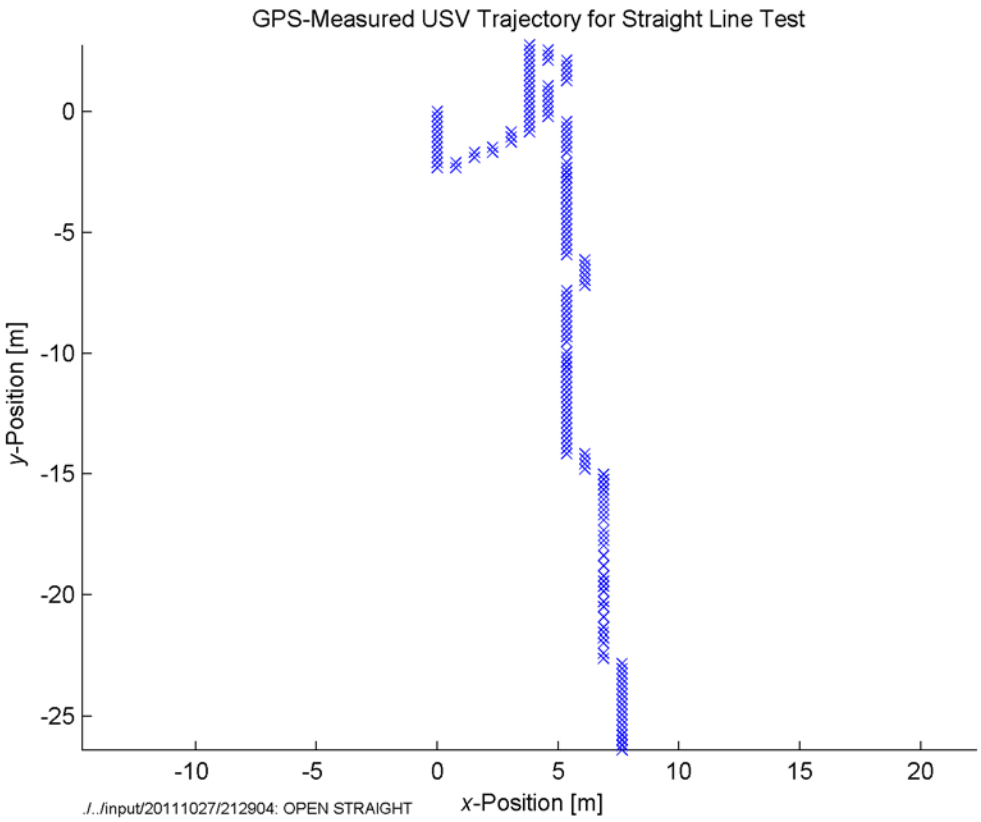

Figure 22: USV trajectory demarcated by GPS fixes during a straight-line test.

Also the GPS quantization is apparent; in both the x and y direction, there are clearly a discrete set of points that the measurements fall on. Moreover, the distance associated with a single quanta is time-variant. Depending on reception quality and number of satellites in contact with the receiver, the size of the quanta will change. This is referred to as the horizontal dilation of precision (HDOP). For the GPS unit initially used on the USV (the FV-M8), this run represents the best performance of the unit, with resolutions of 0.8 and 0.2 m, in the x and y directions, respectively. This is a problem for slow-speed and

small-scale (on the order of meters) measurements, as can be seen in the velocity calculations associated with this track (Figure 23).

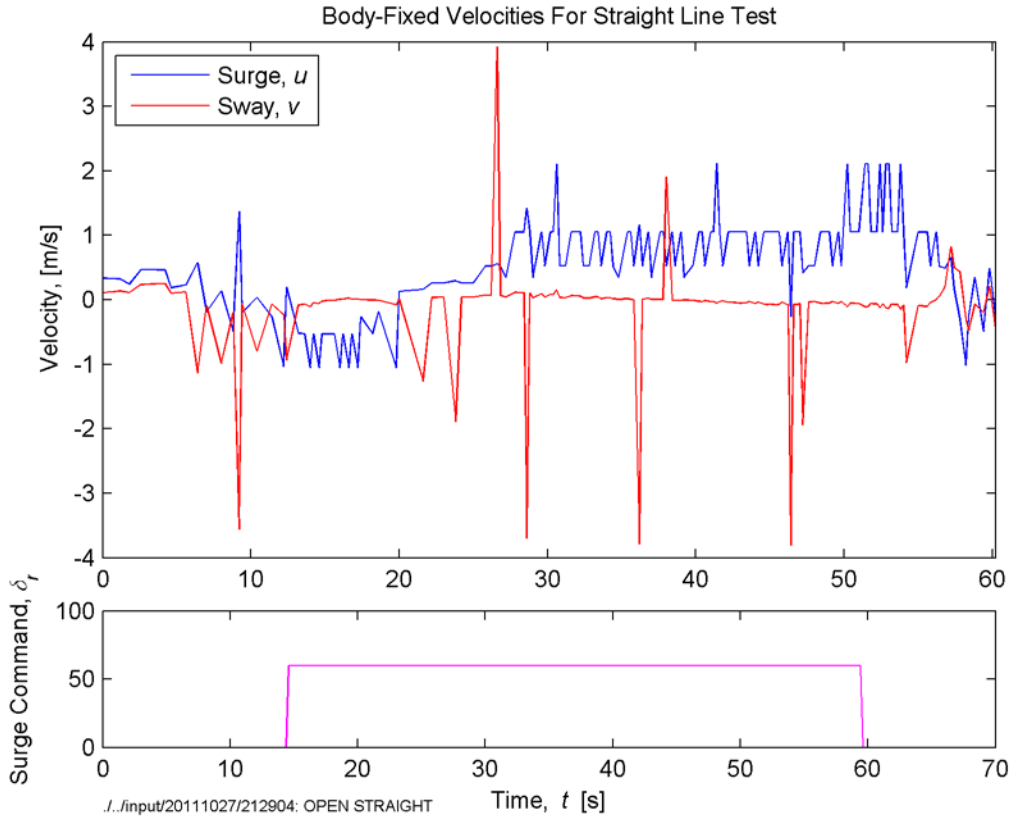

Figure 23: Straight line ($\delta_u = 60$) test velocities calculated from GPS and compass data.

Figure 23 shows the surge and sway velocity calculated from the track shown in the previous figure. Using the relation presented in Equation (2.1), along with the GPS positions and the heading information also recorded, the surge and sway velocities were calculated. The data set used for this was preprocessed to remove successive repeated measurements from the GPS, to give a better idea of the true velocity. Both the surge and sway quantities hover around zero before the command is input, and 15 seconds after the vehicle was commands to move forward, the surge measurement begins to respond. After this transient period the derived value for the surge motion is 1.0 m/s.

Given the surge command also shown, $\delta_r = 60,\ (15\,\text{s} < t < 60\,\text{s})$, it is apparent that there is a significant sensor dynamic involved. The quantization noise is significant at this measurement scale (on the order of 0.5 m/s). Furthermore there is a significant time lag before the internal receiver filter adapts to the moving platform. From observation during this trial, the USV was moving at roughly 1 m/s at steady state,

with the acceleration period lasting less than 2 seconds. In the figure the receiver takes approximately 15 seconds to respond to the movement. Due to the problems with the GPS sensor, and the tachometer used to measure thruster RPM, a formal analysis of the surge response could not be performed with the goal of constructing a reliable mathematical model.

4.1.2 **Circle Test**

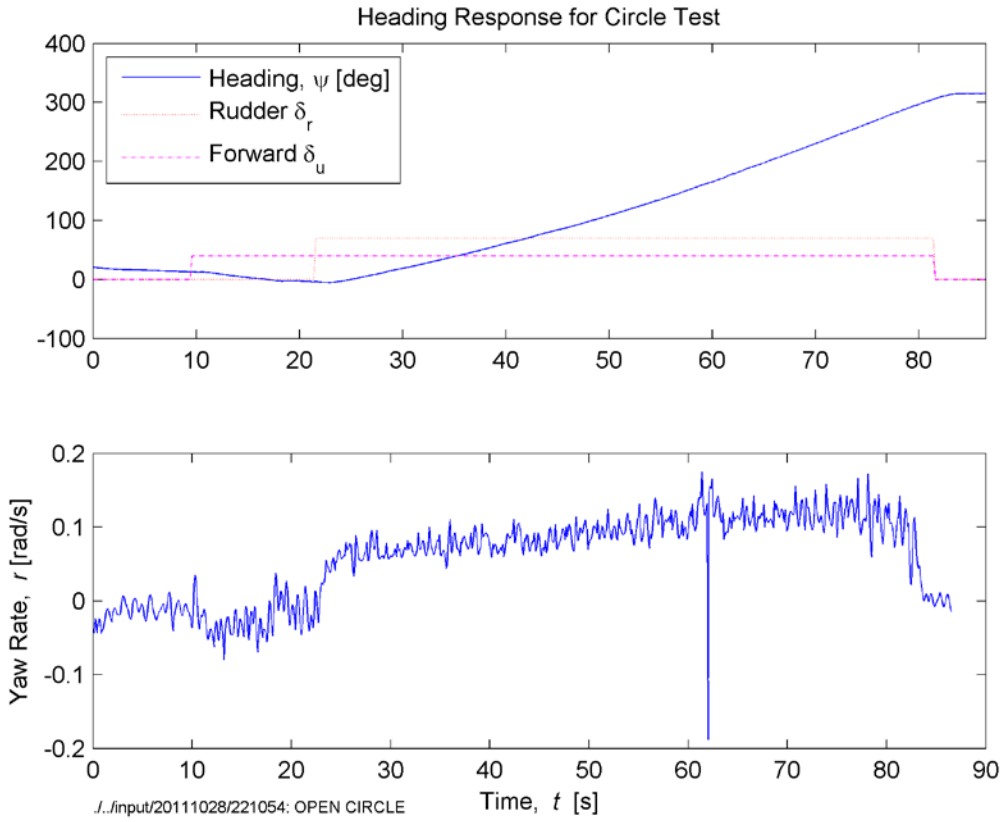

Figure 24: Individual circle test result sample, depicting heading and yaw rate response to given input.

Circle tests were also performed to help characterize the turn rate and tactical diameter of the USV. Even for most basic surface vehicle steering models, the circle test can be valuable in identifying useful steering dynamics. In this test, the vehicle is subjected first to a symmetric thrust command, $\delta_u = C_1,\ \delta_r = 0, (t < t_i)$, for long enough to reach a steady speed. Then, a step input is applied to the rudder, $\delta_u = C_1,\ \delta_r = C_2 (t_i < t < t_f)$ for a period long enough to allow the vehicle to reach steady surge, sway, and yaw velocities.

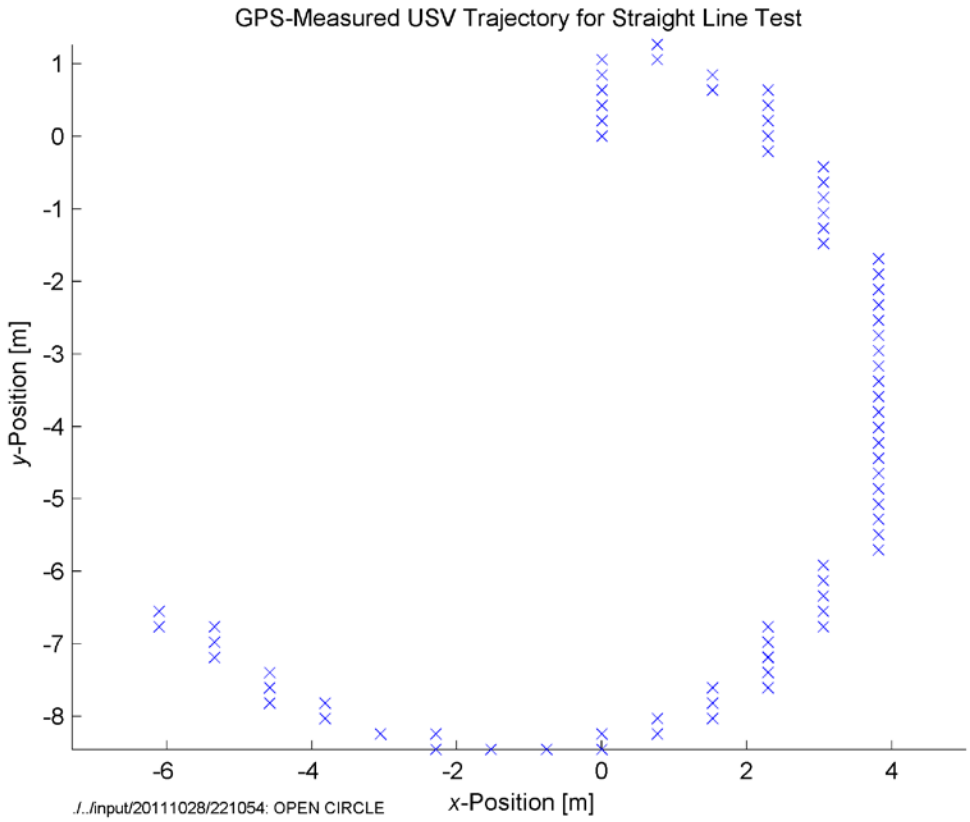

Figure 25: USV trajectory for circle test.

Figure 24 shows the heading dynamics of one such turning test. The USV was subjected to a 60-second virtual rudder input of 75 percent forward. The heading response (unwound, in degrees with respect to north), are shown, along with the calculated rate of turn. The heading response shows that after an initial time lag of 2 seconds, the vehicle quickly reaches a steady rate of turn, covering nearly 300 degrees in 60 seconds. This yaw rate was calculated directly from the compass measurements. One can see that this value has a considerable amount of noise. This sample is quite representative of the other calculated yaw rates. The noise here and in other samples is highly sinusoidal, indicating the sampling rate of the compass (5 Hz) was sufficient to measure the encounter frequency of the waves the vehicle was subjected to during testing.

Figure 25 shows a sample of the vessel trajectory during the trial. Starting at $\langle 0,0 \rangle$, the vehicle travels due North when the forward command is received but before the rudder step response is applied. After several meters, and 10 seconds, the rudder is applied and the vehicle turns due East, quickly reaching a steady rate of turn and thus a constant circular trajectory. The vehicle does not complete an entire circle, but the tactical radius is seen to be around 12 m.

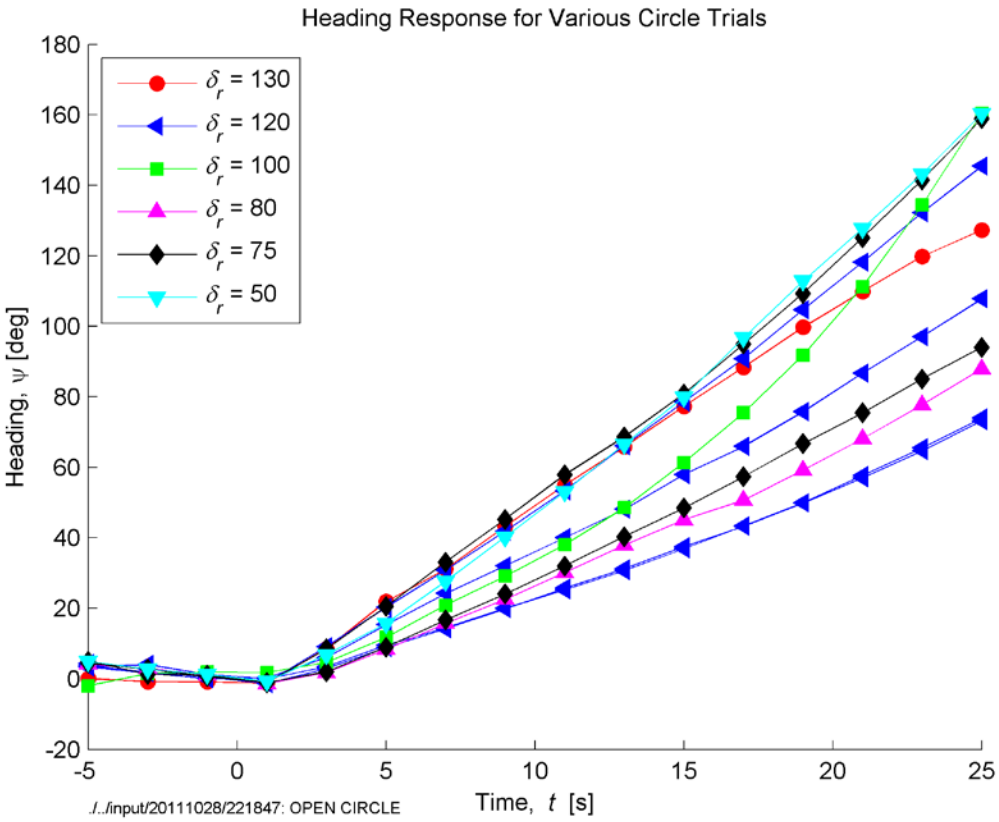

Figure 26: Heading response during turning trials at various virtual rudders.

Figure 26 shows a comparison of the circle tests performed. The heading responses are shown relative to the starting heading. That is, all headings are relative to the point at which the virtual rudder step was applied. It is apparent that while the heading response for any individual run behaves as expected, there is a great amount of variation between runs, with little respect to the actual rudder applied. This is largely attributed to plant time variance. Without a RPM sensor to actively monitor and control thruster RPM, factors such as mechanical friction and battery voltages have a large influence on vehicle performance. Furthermore, while testing was done at times chosen for the favorable environmental factors (tide and wind speed), they may also have an influence not captured here.

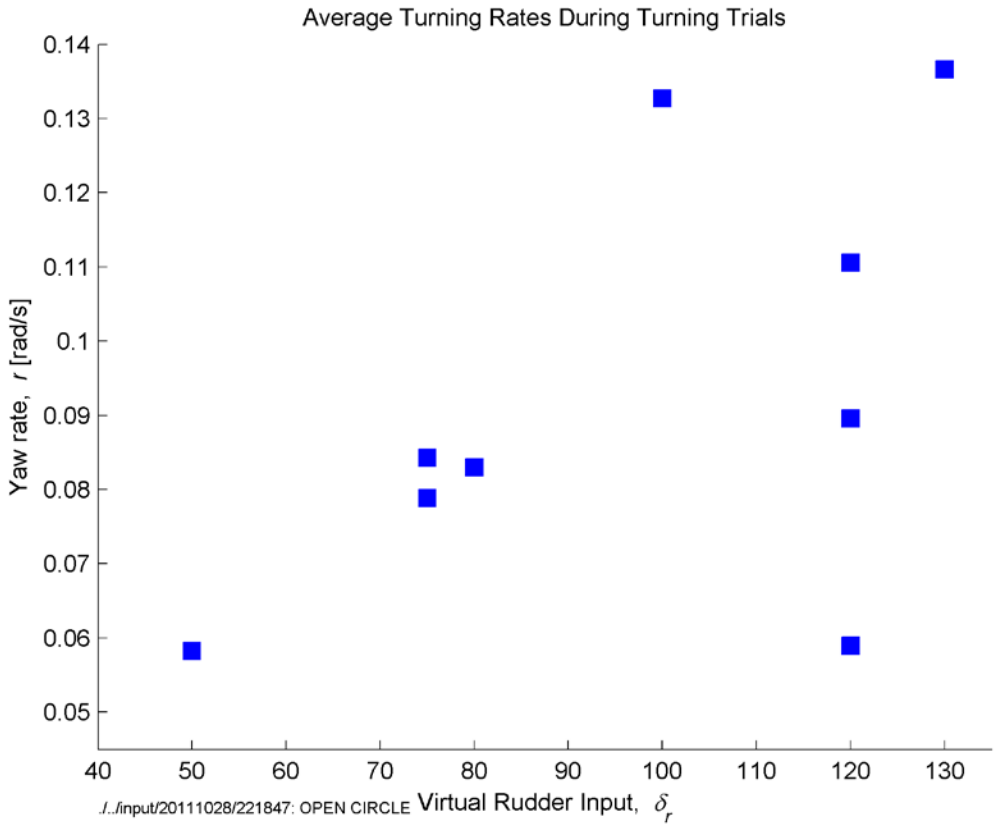

Figure 27: Steady-state yaw rate responses over various rudder inputs.

Figure 27 shows a similar relationship. In this figure the runs have been reduced to single points, by averaging the yaw rate during the steady-state turning. In this condensed view, the instances of lower rudder action do tend to result in slower yaw rates. The high end of the spectrum however there is a great deal of disparity. Qualitatively, ignoring the outliers, the relationship makes sense; an increase in rudder command should correlate to an increase in yaw rate, to a point. More exhaustive testing may more completely reveal the relationship.

4.1.3 **Zig Zag Test**

The last open loop test performed was a zig-zag maneuver. In this test, initially a steady-forward speed is established. Then, a cyclical step input is applied, oscillating the rudder between port and starboard turns. This is a useful indicator of the lag between the application of control action and the resulting response.

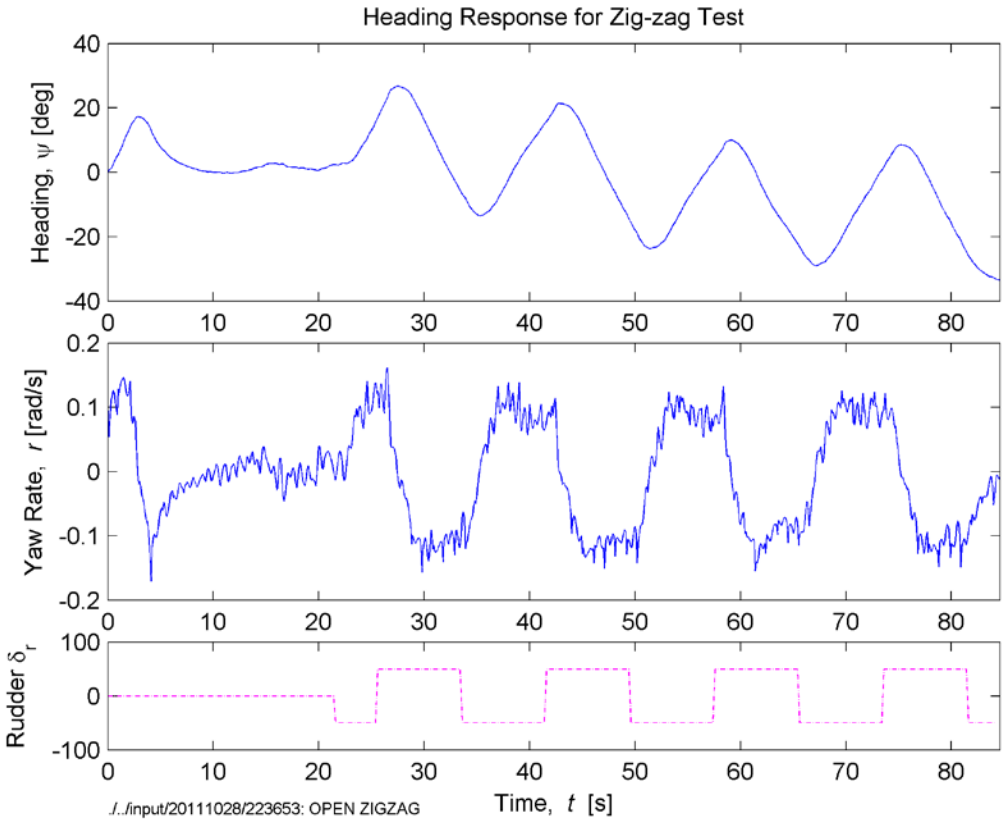

Figure 28: Heading response of USV during zig-zag test.

Figure 28 shows the heading response of the USV during a zig-zag test. The vehicle was subjected to several 50 percent virtual rudder deflections in either directions, each lasting for approximately 16 seconds. The measured heading is depicted, along with the calculated yaw rate. In particular, the yaw rate indicates how quickly the vehicle reaches the steady-state yaw rate. Consistently, the lag time is 2 s. The asymmetry is apparent in the heading measurement, as the vehicle consistently turns more aggressively to port than to starboard.

4.2 SIMULINK RESULTS

As outlined in Section 3.6: Simulink Simulations, MATLAB Simulink was used to construct a simulation of a virtual vehicle running on a controller based on the same principles as that which was implemented on the target platform. This allowed for rapid validation of controller structure design and guidance algorithm. The gains used in the simulation served as a starting point for deployment on the real

system, but as the simulation model is not matched to the real plant in any quantitative sense, the tuned simulation gains differ from the tuned parameters onboard.

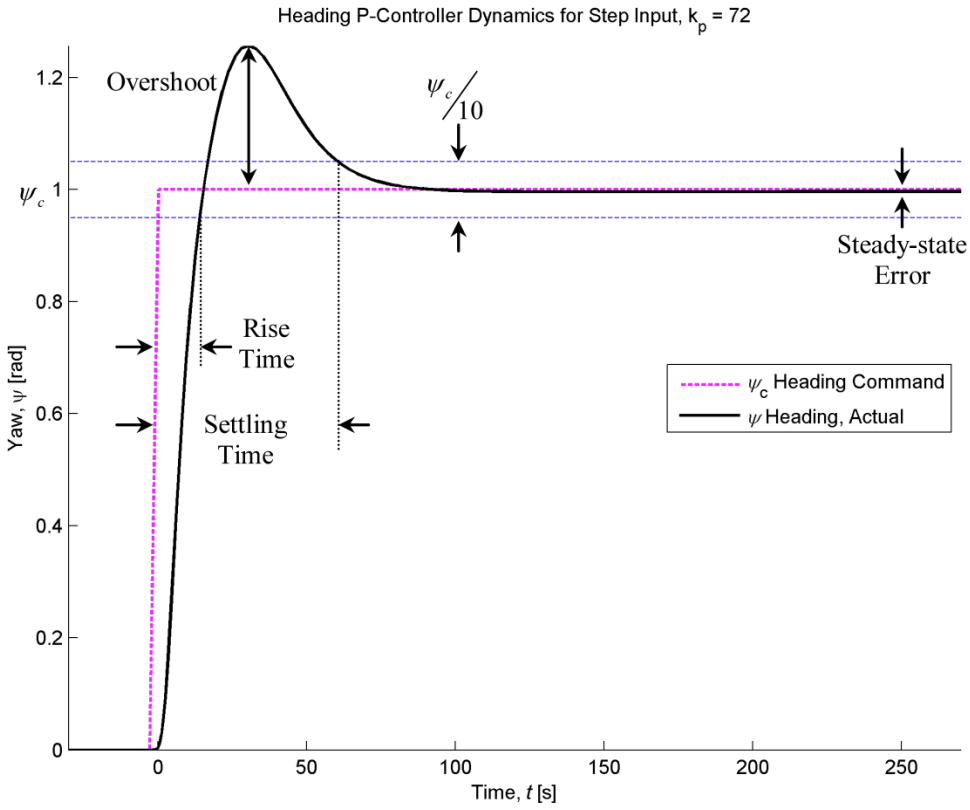

Figure 29: Example Heading Controller Step Response

There are many well-developed methods to perform PID controller tuning [37]. Some of these, such as pole-placement, rely on a system model in order to analytically estimate closed-loop stability. Gains can be calculated such that closed-loop poles are positioned exactly at points that produce a predictable system response. Conversely, there are many heuristic methods that assume black-box plant models – that is, the system model is unknown or not easily identified. In these cases, there are various methods that allow the designer to tune the controller *in situ*, producing acceptable, if not optimized, response. These methods, such as the Ziegler-Nichols PID method, procedurally adjust the controller gains to produce a practical control law that works in most cases. Most experimental procedures such as this are basically proven approaches to arriving at a controller that is stable, is moderately fast, and has little steady-state error. That is essentially the approach taken here: adjust controller gains (whether in Simulink or on the platform itself) to achieve stability, minimize steady-state error, minimize overshoot and generate a fast response.

Figure 29 shows a typical example of the step response of a USV heading proportional-controller. In fact, the response shown here is quite typical of any second-order system. For all $t < 0$, the actual heading and commanded heading are equal; at $t = 0$, a step command is applied to the system. After a very small acceleration period, the heading reaches approximately steady velocity (this is the turn rate). After a rise time, the heading reaches for the first time the commanded heading. However, due to system inertia, the heading is carried beyond the commanded heading. In general, the goal of the designer is to minimize steady-state error, decrease both rise and settling time, minimize overshoot, and depending on the scenario, limit the integral error (also called error integral wind-up).

4.2.1 **Heading Controller**

The step response of the heading controller for various proportional gains is shown in Figure 30. It is apparent how increasing the proportional gain, $k_P^{(\psi)}$ effectively decreases damping in the closed loop system. The system exhibits critical damping behavior at $k_P^{(\psi)} = 36$. Upon initial inspection, the critically damped case may be the design point for the controller. However, this choice produces significantly longer rise time, with a comparable settling time, and a significant amount of error wind up. A better design point in this case is a less-damping gain, $k_P^{(\psi)} = 72$. Overshoot actually occurs at this point, however this can actually be beneficial because it can lessen error in other states (see Figure 31). , and produces significantly better rise times.

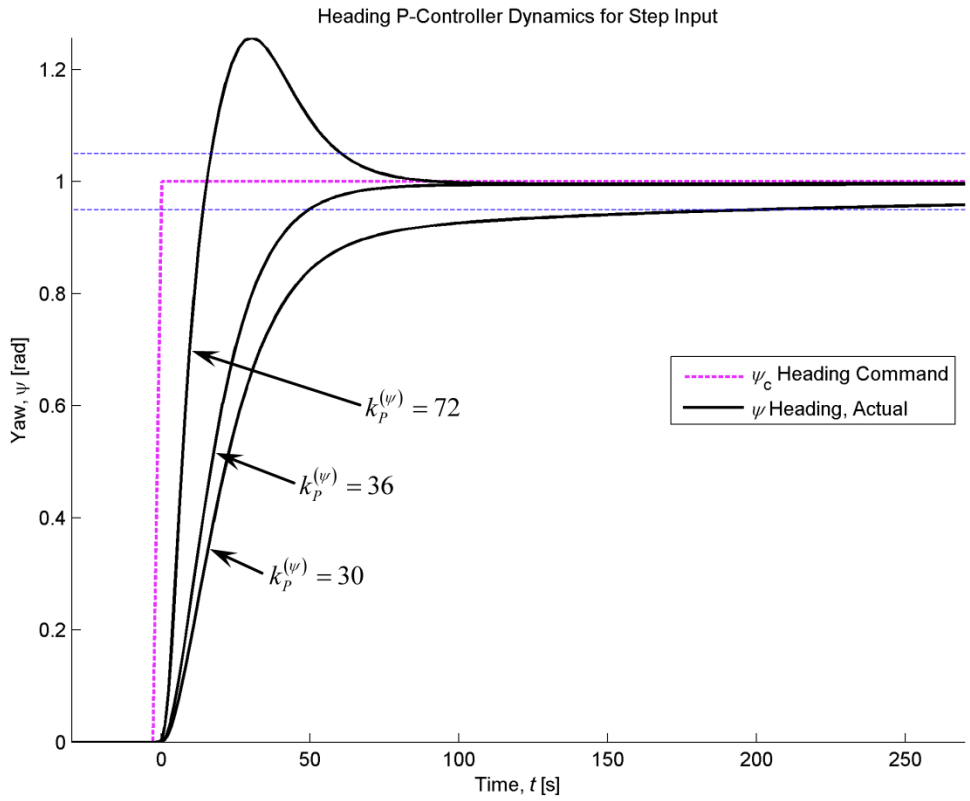

Figure 30: Heading P-Controller Step Response with Varying Proportional Gains

This approach makes physical sense. As seen in Figure 31, when comparing the actual vehicle response to the "desired" track that is represented by the step input, overshoot actually works to correct the response lag that is the rise time. The critically-damped controller ($k_P^{(\psi)} = 36$), trajectory reveals that while the vehicle achieves the target heading quickly, there is significant steady-state cross-track error. Qualitatively one can think of the heading controller error integral as relating roughly to the time spent off-course, which, depending on external disturbances, will usually correspond to some steady-state cross-track error when not coupled with a mechanism to drive the cross-track to zero. Also recall, in this simulation there are no environmental disturbances to provide external excitation to the system. Thus, a pure heading controller with no track feedback appears capable of maintaining course. The scenario is idealized; however it provides insight into the practical relevance of controller gain tuning.

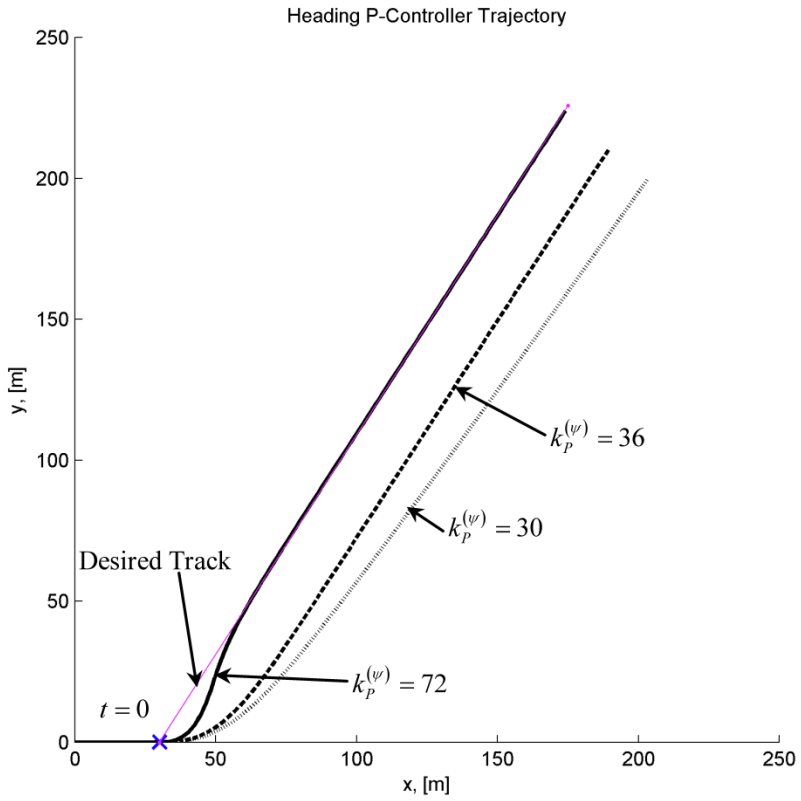

Figure 31: Simulated Heading P-Controller Trajectory for Various Gains, with no environmental inputs

4.2.2 Speed Controller

Figure 32 shows the simulated surge response for a speed P-controller operating at various gains. The fundamental difference between a first order and a second order system is apparent when comparing Figure 32 and Figure 30. The surge response lacks the typical overshoot and oscillation of the heading response, instead exhibiting a logarithmic response, rising at various rates to the final setpoint. Qualitatively however, the effect of varying proportional gains is quite similar to the heading control case; increasing $k_{\mathrm{P}}^{(u)}$ decreases rise time.

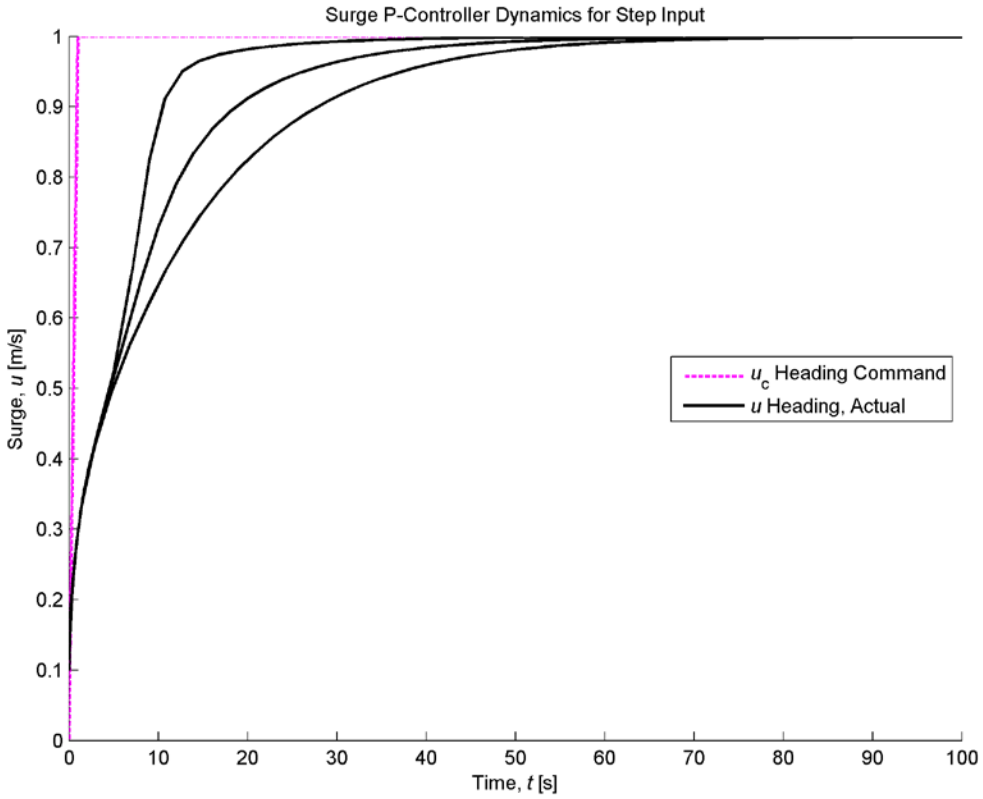

Figure 32: Surge P-Controller Step Response

4.2.3 **Speed and Heading Controller Combined**

When the speed and heading controller are combined as described in Section 3.5.3, and are subjected to a series of both heading and speed step inputs, the controller responds as in Figure 33. The relationship between the abstracted virtual rudder term and the actual thruster responses is depicted; for a virtual rudder setpoint, thruster RPM is modulated in equal and opposite directions, while a zero-rudder command modulates each thruster equivalently, according to the speed command.

The associated trajectory is shown Figure 34. The time delay between each step input is equal, so the change in forward speed is seen clearly as a progressive elongation of each leg of the box. The vehicle starts at $\langle 0,0 \rangle$ pointed Northward, and then moves first to the North-East, before turning and simultaneously accelerating to the South-West. This is repeated two more times, each turn being clockwise 90 degrees. During the 500 second run, the vehicle covers nearly 900 meters.

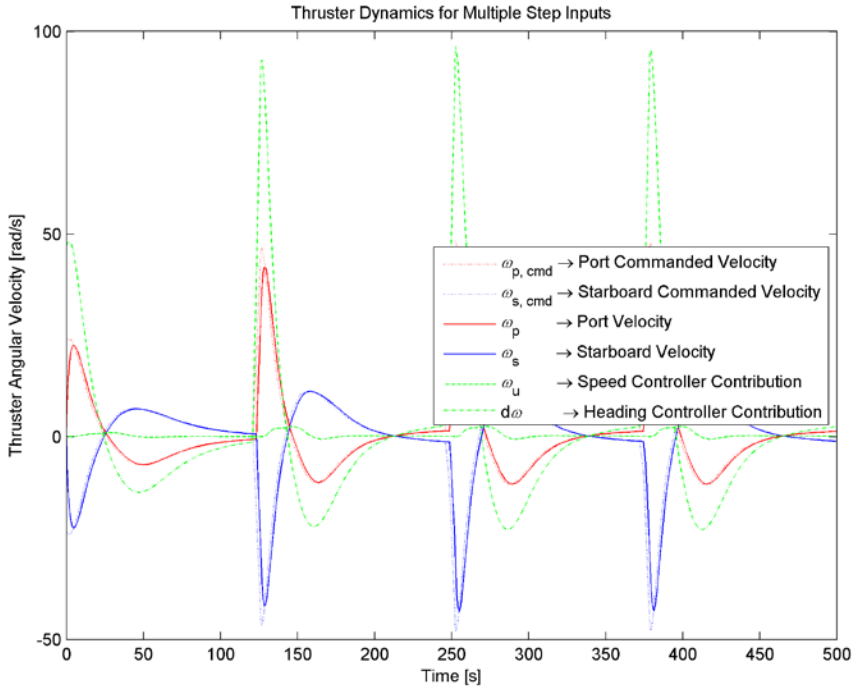

Figure 33: Plant Outputs for Multiple Step Inputs

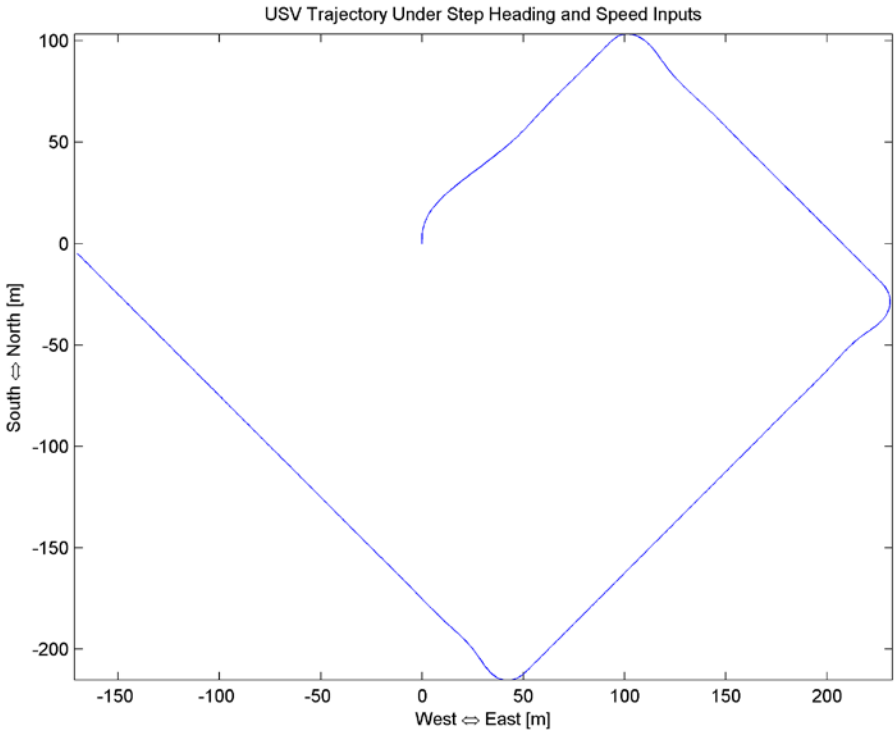

Figure 34: Simulated Trajectory for Heading and Speed Control Solution

4.3 CLOSED LOOP TRIALS

Following the implementation of the vehicle operating system on hardware, a serial of closed loop trials were performed. The purpose was two-fold; first and foremost this would validate the GNC package operation under closed-loop conditions; secondly the trials would serve to show implementation of the basic guidance and control algorithms discussed previously. These consisted of both heading and line-of-sight waypoint tracking. Speed control trials were removed from the trial plan due to inconsistent performance in the sensor subsystems that would be needed to adequately perform surge control. The results of the trials are presented here.

4.3.1 **Heading Controller Tests**

The first type of closed-loop control to be implemented on the USV was heading control. Under this scheme, the speed command is fixed to a pre-selected value. An error between current heading and commanded heading is calculated in real-time using sensor feedback and a predetermined heading setpoint schedule. The error term is simply multiplied by a proportional gain, to produce a virtual rudder command, which then applied to each of the current thruster setpoints.

Figure 35 shows the response of the USV during a run in which the proportional gain has been set to 80, and the speed command has been set to 40. After a short wait period, the controller receives a heading command of 70 degrees, and then after 40 seconds, the vehicle receives another step command of 13 degrees (equivalent to 1 radian). During both the initial turn (from the transient period) and the fully-controlled turn, the proportional behavior of the controller can be observed in behavior of the virtual rudder command with respect to the calculated error; except for the magnitude of each response, the dynamics are identical. This is a direct consequence of the controller being purely proportional. Rise-time in this run as well as others averages 10 seconds, with a steady-state error being roughly 3 degrees. These values are typical of the few runs performed. With an integral component of the controller, the vehicle is expected to have steady-state error when subjected to environmental disturbances such as wind or current. There is little overshoot depicted in the heading response, an indicator that the gain could perhaps be increased. This gain, which was used across all 6 runs performed, was chosen to maximize controller capability throughout the entire error range. That is, the gain was chosen such that saturation would only be achieved in the case of the largest error (180 degrees).

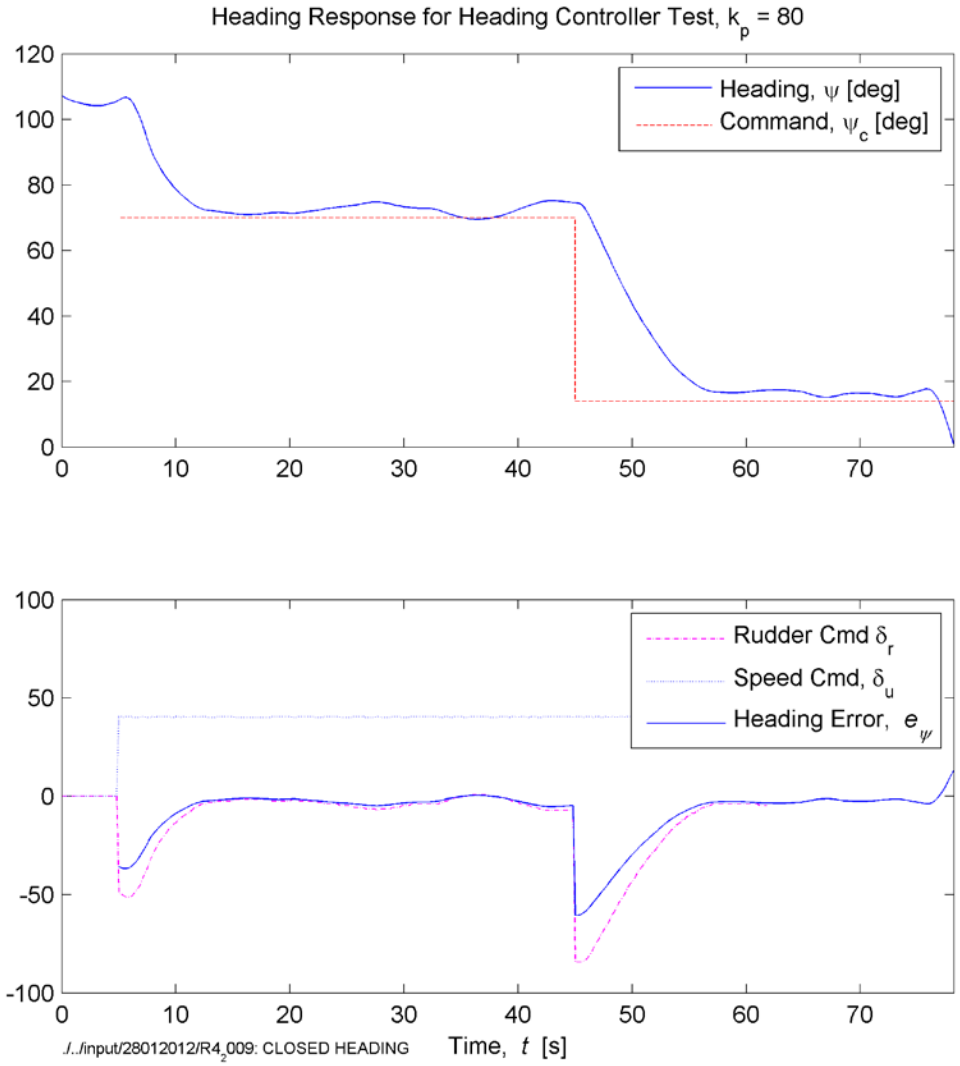

Figure 35: Heading and controller response for 1-radian change in setpoint.

Figure 36 shows the position measurements for the same heading trial. The track indicates that the vehicle was subject to external forcing which the controller was unable to overcome. In the first leg of the mission the average course-over-ground was approximately 80 degrees, 10 degrees different from the target heading. During the second leg, the effective course-over-ground was approximately 30 degrees, more than 15 degrees from the target. This is an indication that the USV was being blown off track, either by wind or waves. This is a good example of why heading control in itself is not always sufficient for effective guidance; using pure heading control with static heading assignments, the final vehicle position may be extremely different from the anticipated position.

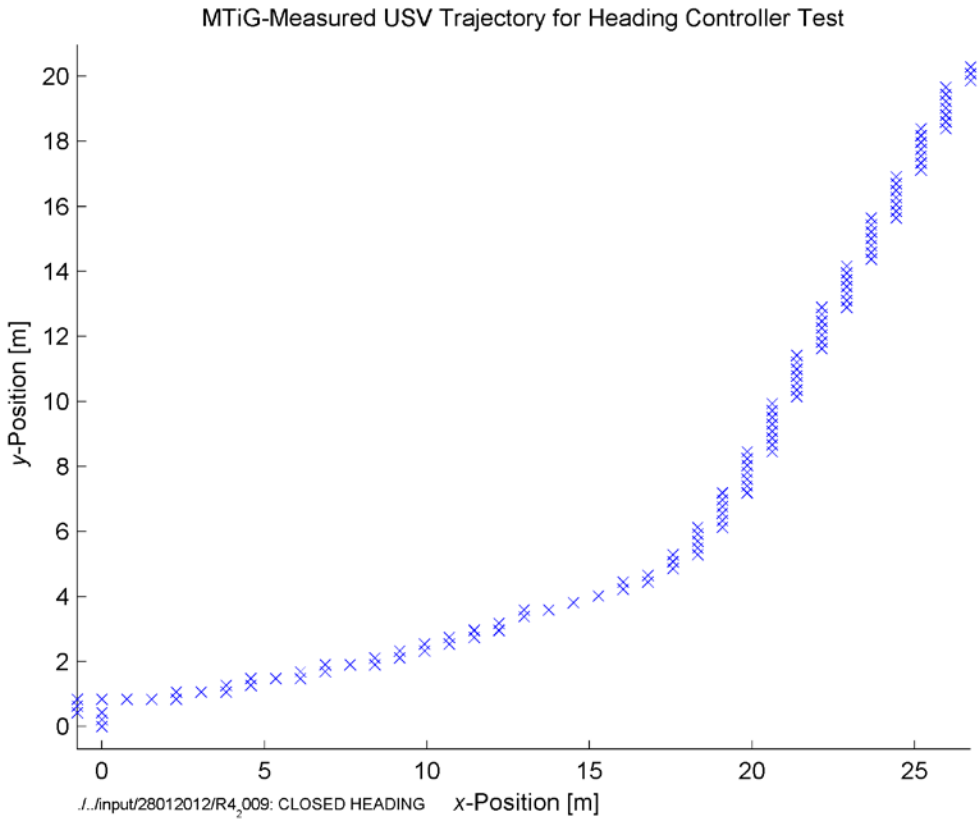

Figure 36: MTi-G-measured trajectory during closed loop heading trial.

4.3.2 **Waypoint Following Tests**

Following the heading controller trials, several tests were performed using a closed-loop waypoint-tracking algorithm using the LOS mechanism. The vessel was preprogrammed with a series of target waypoints. For every time step throughout the mission, the bearing and distance to the target waypoint from the current position was calculated. If the distance to the target waypoint was below a certain threshold (the value of 3m was used for all trials), the waypoint index was incremented. If the distance was above the threshold, then the bearing to the target was used as the heading controller setpoint. The results from the trials are presented here.

Figure 37 shows the USV trajectory for one such trial. The USV started at position $\langle 0,0 \rangle$. The first waypoint was at approximately $\langle -28\text{ m}, 9\text{ m} \rangle$, represented by the red diamond. The vehicle progresses in the direction of this waypoint until reaching a certain point, 10 meters from the target. Here the guidance system transitions to the next waypoint in the schedule, and a new heading is calculated to the second

waypoint shown, at $\langle -36\text{ m}, 26\text{ m}\rangle$. The USV follows a similar routine, closing the distance directly until reaching a sufficiently close distance before transition to the next segment of the mission.

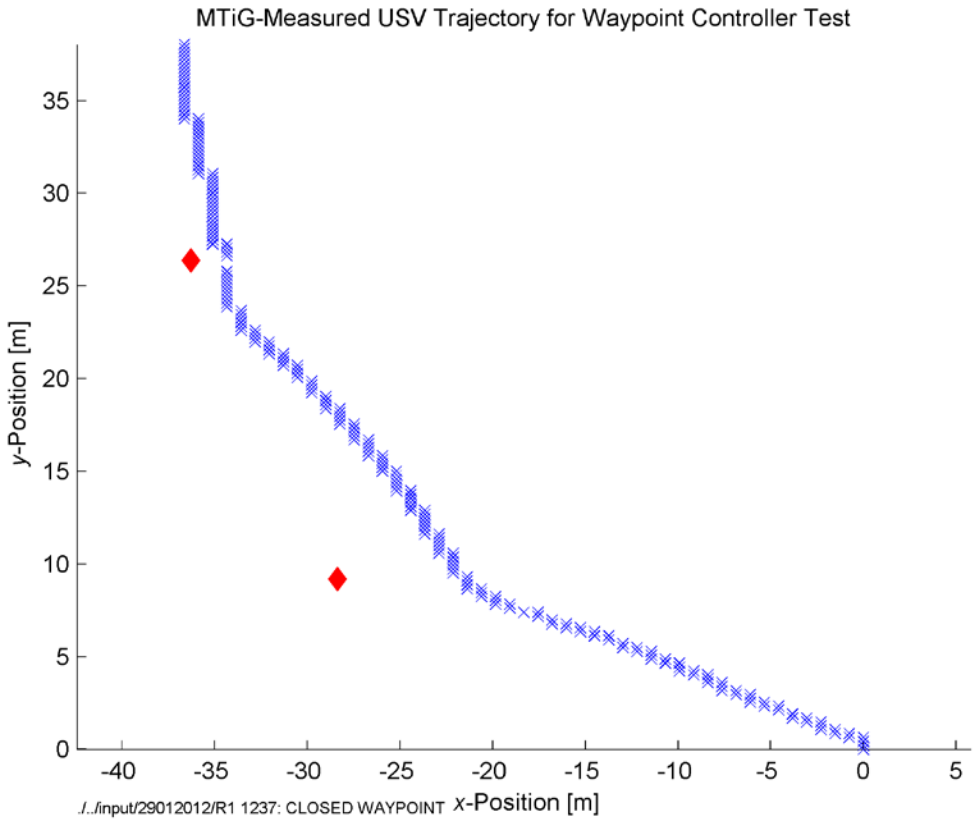

Figure 37: LOS-waypoint controller trial. Target waypoints are represented by red diamonds.

Figure 37 shows the heading measurement and the controller output for the same run. The heading dynamic discussed previously with respect to the heading trials can be seen in the heading response for this test. During the steady-state period between waypoint transitions, the USV behaves similarly to the constant-heading-command case; the vehicle follows a consistent heading until reaching a transition. This discontinuity is effectively a step-input, and thus the second-order dynamic response is observed; after a small lag period the heading accelerates shortly to reach a steady turn rate before arriving at the target heading, with a small amount of overshoot. At a sufficiently far distance from the target waypoint, the bearing-to-target is relatively constant, explaining the flat response during steady-state periods. Note here that the gain had been increased to $k_p = 120$ from the heading-controller trials; the slow response noted earlier indicated that an increased gain would be beneficial.

Also shown in Figure 38 is the controller output. As seen in the heading-controller trials, large discontinuous jumps are experienced during discrete events, in particular at command input transition points. Unfortunately for these trials the actual controller dynamics and guidance calculations were not recorded, so the distance-to-waypoint and heading error cannot be shown.

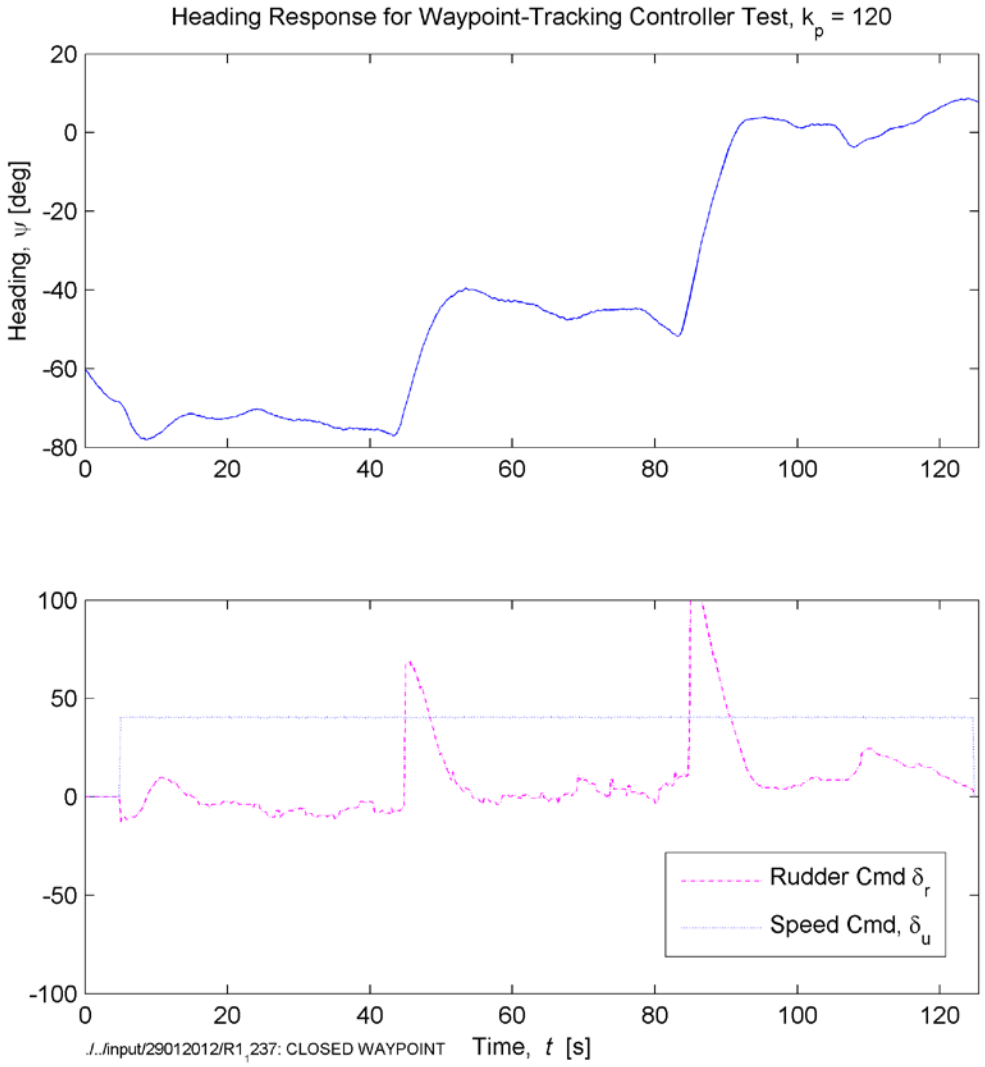

Figure 38: Heading response and plant commands for waypoint-tracking trial.

Figure 39 shows another such waypoint-tracking trial. Again, the vehicle started at the coordinate origin, and was directed to maneuver to the waypoints in the sequence indicated. Here the closing distance constraint does not appear to be followed correctly; in some instances the vehicle never reaches appropriate

proximity to the target waypoint. It was discovered after the trials had been concluded that a watch-dog timeout had been set inappropriately low, causing the guidance system to transition to the next waypoint before the destination had been achieved. This occurred for the first and second turns, explaining the distance between the track and the target waypoints during those legs. The last turn occurs early because again, a premature timeout increased the mission waypoint index.

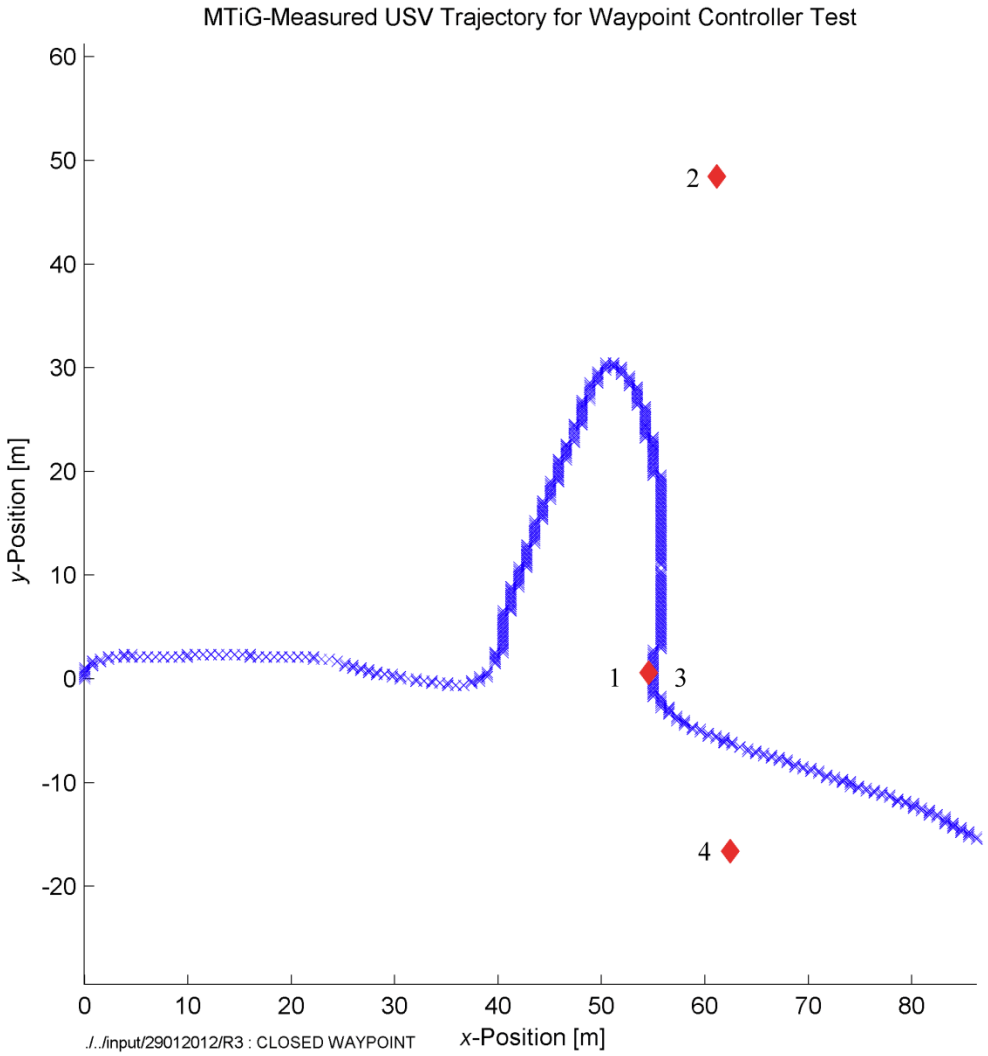

Figure 39: USV trajectory for waypoint-tracking trial.

Figure 40 shows the heading dynamic as well as the thruster commands for the same run. Again, the gain here has been adjusted. Increasing heading oscillation is experience throughout the steady-state periods, which is likely due to the increased gain.

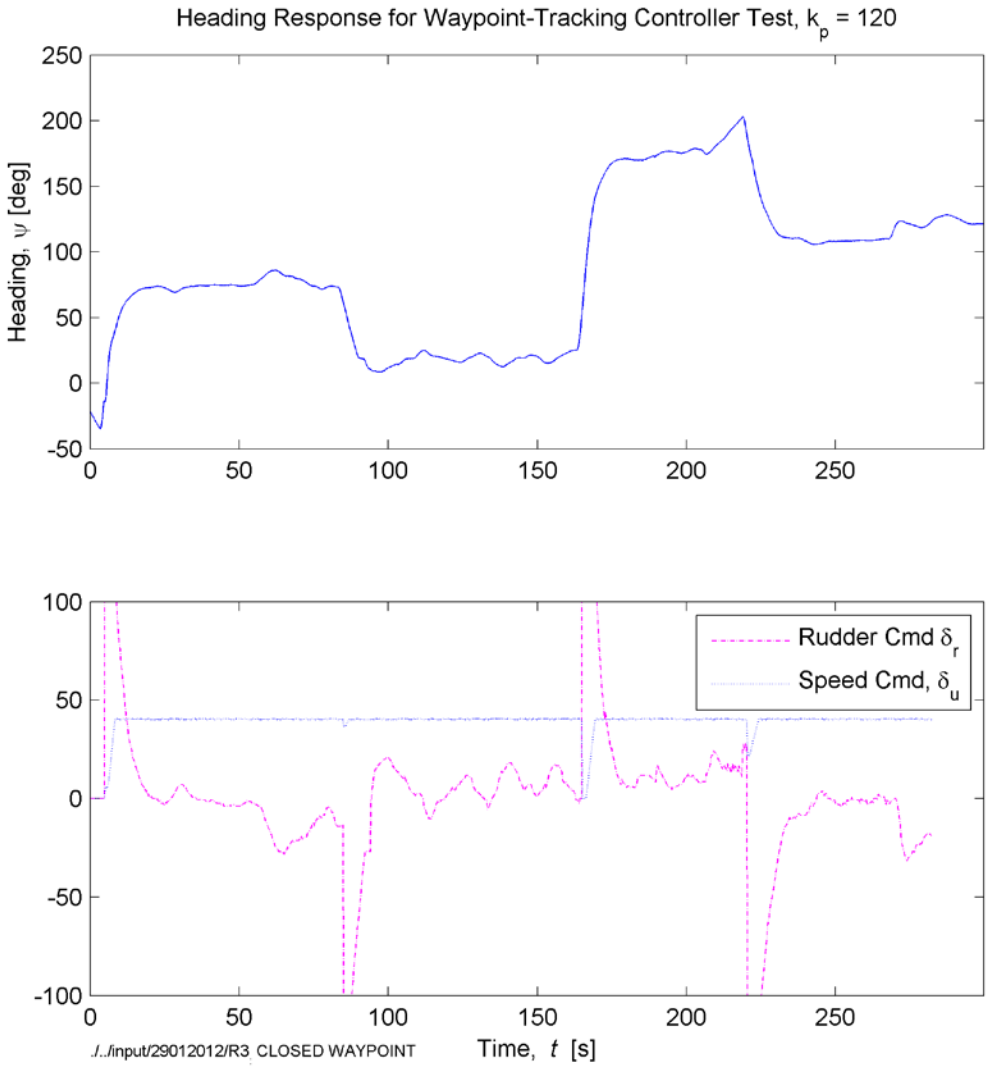

Figure 40: Heading response and plant commands during waypoint-tracking trial.

The controller output shows how the vehicle transitions exactly 80 seconds after the previous transition for the first two turns, and then exactly at 40 seconds for the last turn, which is exactly the internal timeout that had been preprogrammed into the vehicle. The controller input here also shows that in extreme cases, such as at 5, 85, 165 and 220 seconds, the differential rudder output from the controller reaches such magnitude that saturation occurs. This reduces the average thruster command, as indicated by the inconsistencies in the speed command.

5 CONCLUSIONS

The GNC package described functions as required to perform basic guidance, navigation, and control functions. The modular hardware package is easily ported between platforms of many shapes and sizes, with sufficiently generic hardware to allow easy adaptability to platforms not considered during its design. More importantly, a modular framework was implemented to allow easy expansion of the existing software base into a more mature GNC platform. As is, the vehicle operating system can be quickly reconfigured for pure data acquisition, open- or closed-loop control. Basic heading and waypoint-tracking controllers were implemented.

5.1 FUTURE WORK

Firstly, there is much room for improvement to the current state of the vehicle operating system. Currently the system uses a shared-memory architecture, but this is not ideal for systems containing multiple computers. Ideally a transition to a system like LCM that utilized network broadcast messages to exchange information would be far more scalable. Furthermore, there are many improvements to be made with respect to human interactions with the operating system during autonomous operations. In particular, a more robust system for delivering information back to a base station would be ideal, along with an improved method of displaying information. Currently, all system messages brought back for the human operator are presented simply as scrolling text items. Additionally, the current version of the vehicle operating system has no provisions for real-time human input other than end-of-mission overrides. Ideally the human operator should be able to intervene and adjust the current mission without disrupting autonomous operations.

Secondly, the navigation calculations that are done currently on the vehicle are quite simple, providing little more than raw sensor information to the user. The shortcomings were exposed in several cases in this work. An improved navigation routine, using all of the sensor information in an appropriate adaptive filter to better estimate the vehicle states would improve performance greatly.

Lastly, the control implementation presented here is very basic, largely a proof-of-concept approach used to validate the hardware and software components of the GNC package. With a platform completed, there is much room for more advanced motion control algorithm development and implementation. In particular, several degrees of actuation were suppressed in this work, using a simplified control approach to allow ease of implementation. Allowing the target platforms to take full advantage of all possible control actions is required to improve mobility beyond the basic yaw and surge control.

6 APPENDICES

6.1 ELECTRICAL SUBSYSTEM

6.1.1 **PCB Schematics**

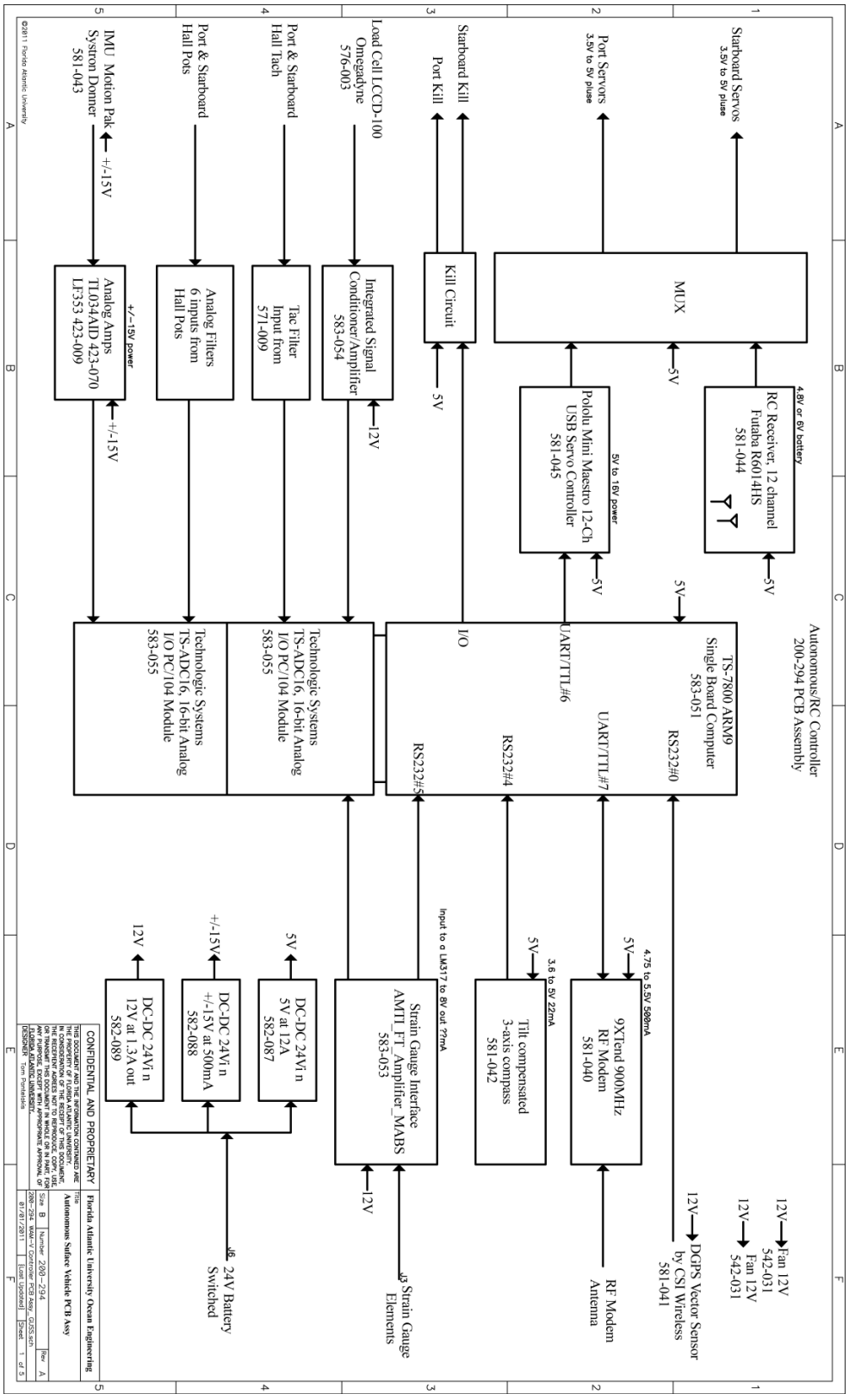

Figure 41: FAU Assembly 200-294: WAM-V Controller Board Schematic, Page 1

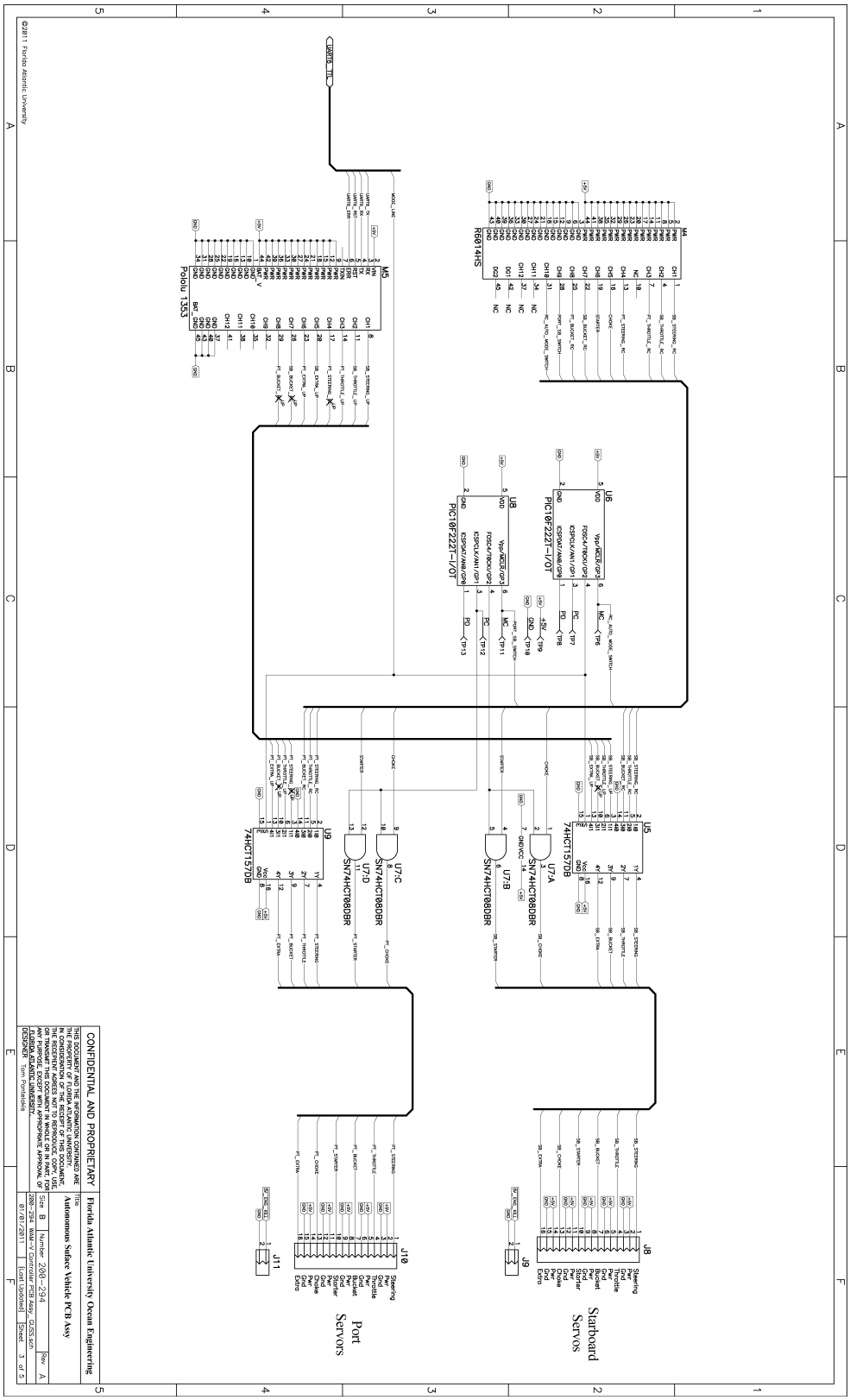

Figure 42: FAU Assembly 200-294: WAM-V Controller Board Schematic, Page 2

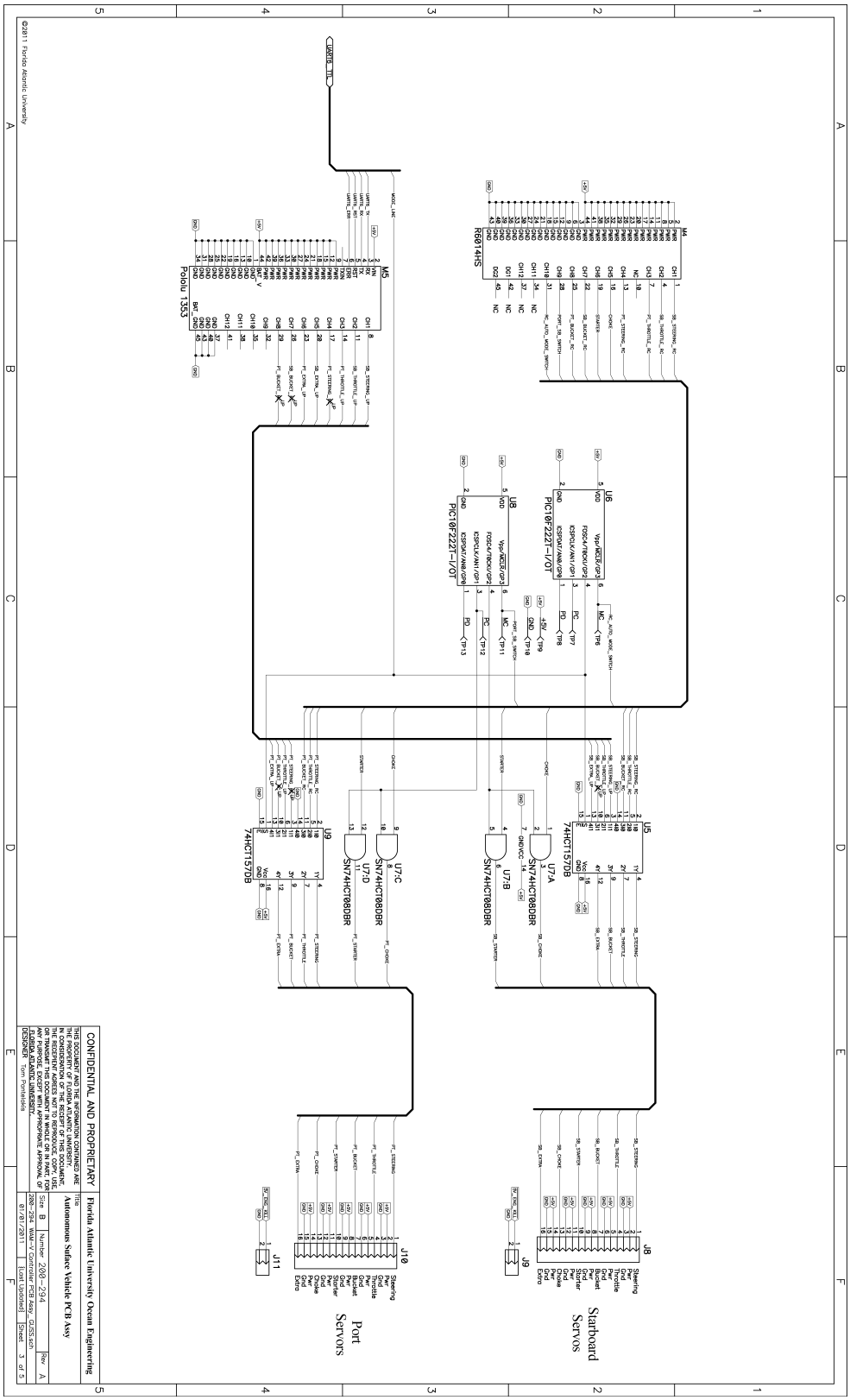

Figure 43: FAU Assembly 200-294: WAM-V Controller Board Schematic, Page 3

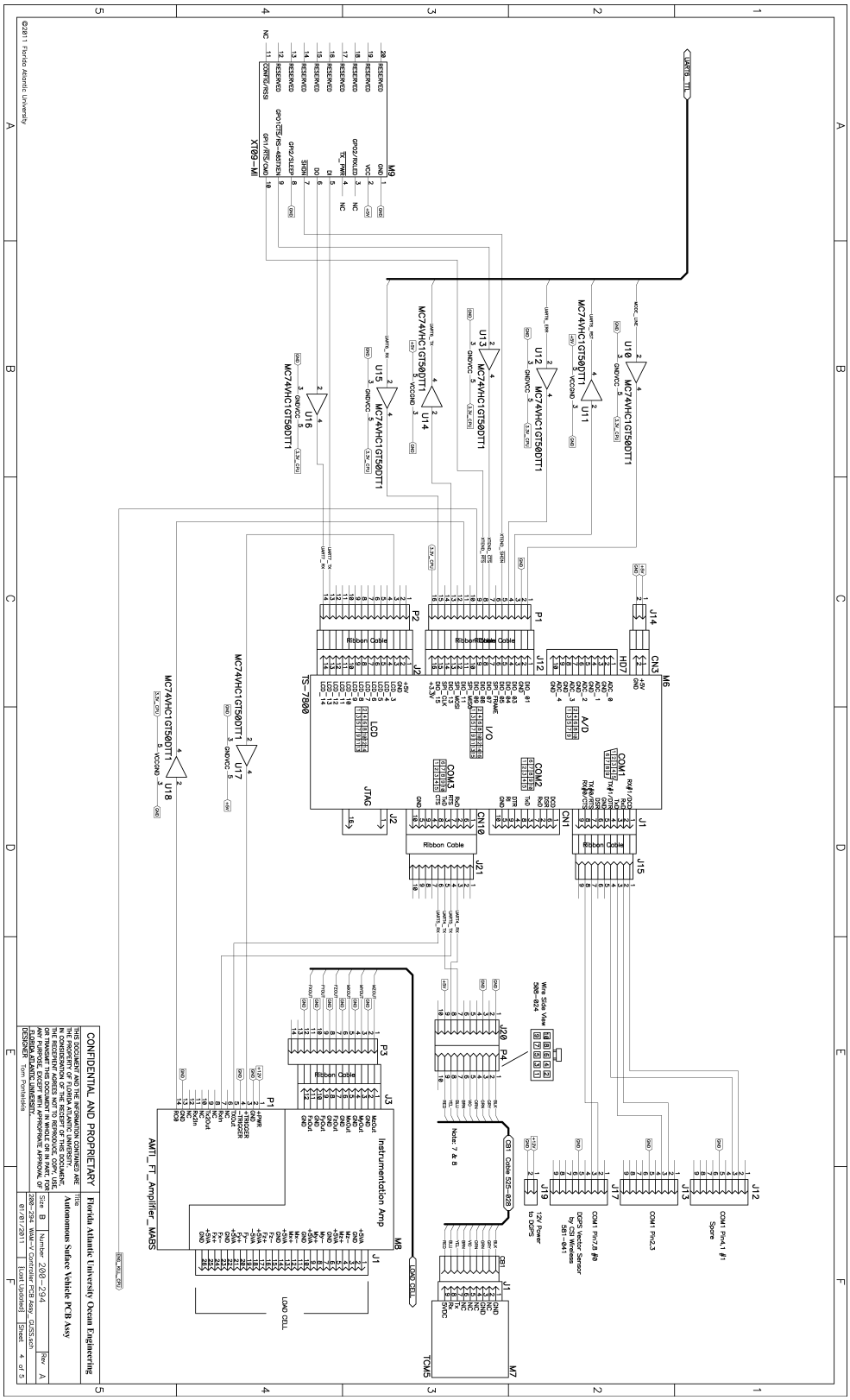

Figure 44: FAU Assembly 200-294: WAM-V Controller Board Schematic, Page 4

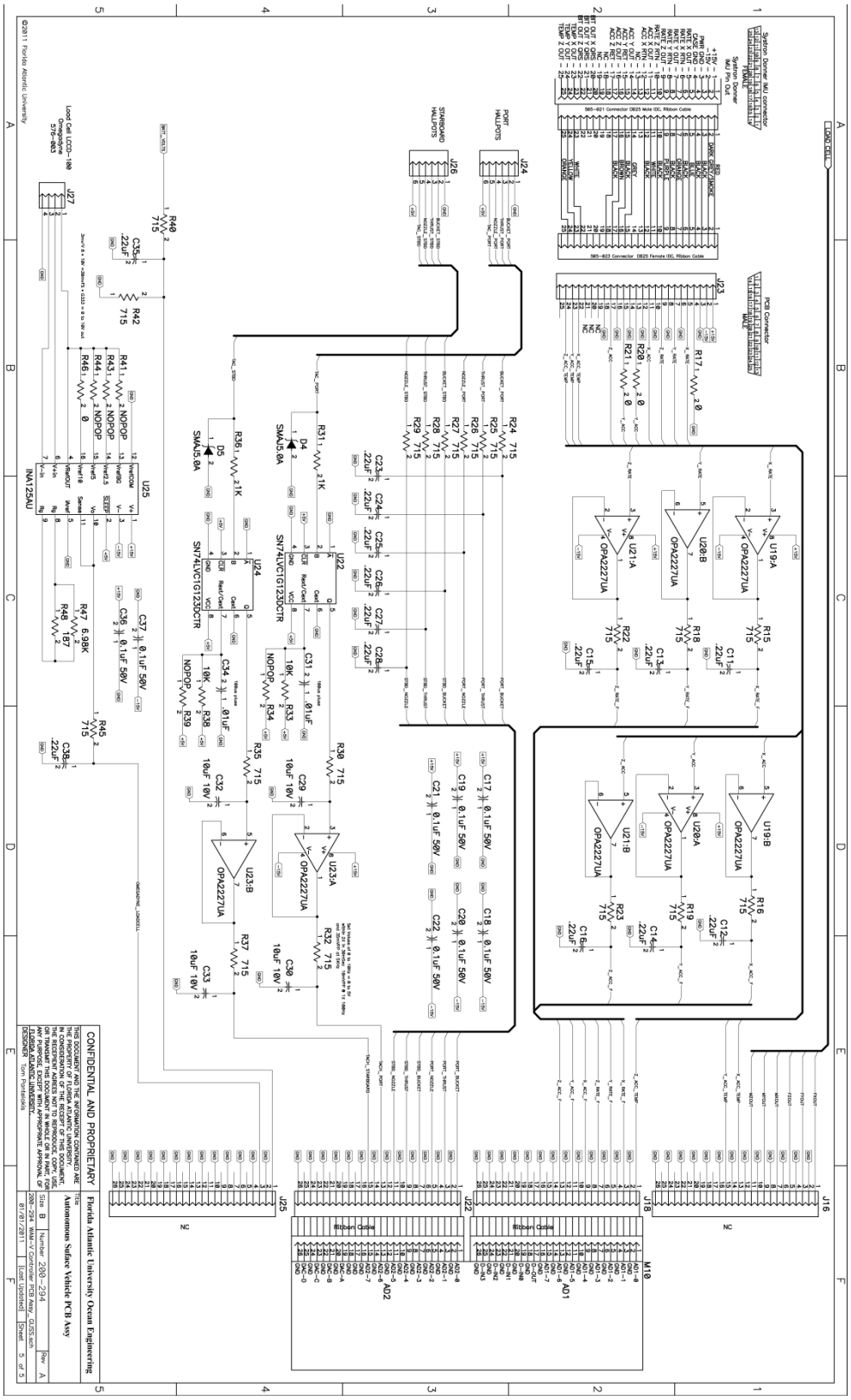

Figure 45: FAU Assembly 200-294: WAM-V Controller Board Schematic, Page 5

6.1.2 **Vehicle Wiring Diagrams**

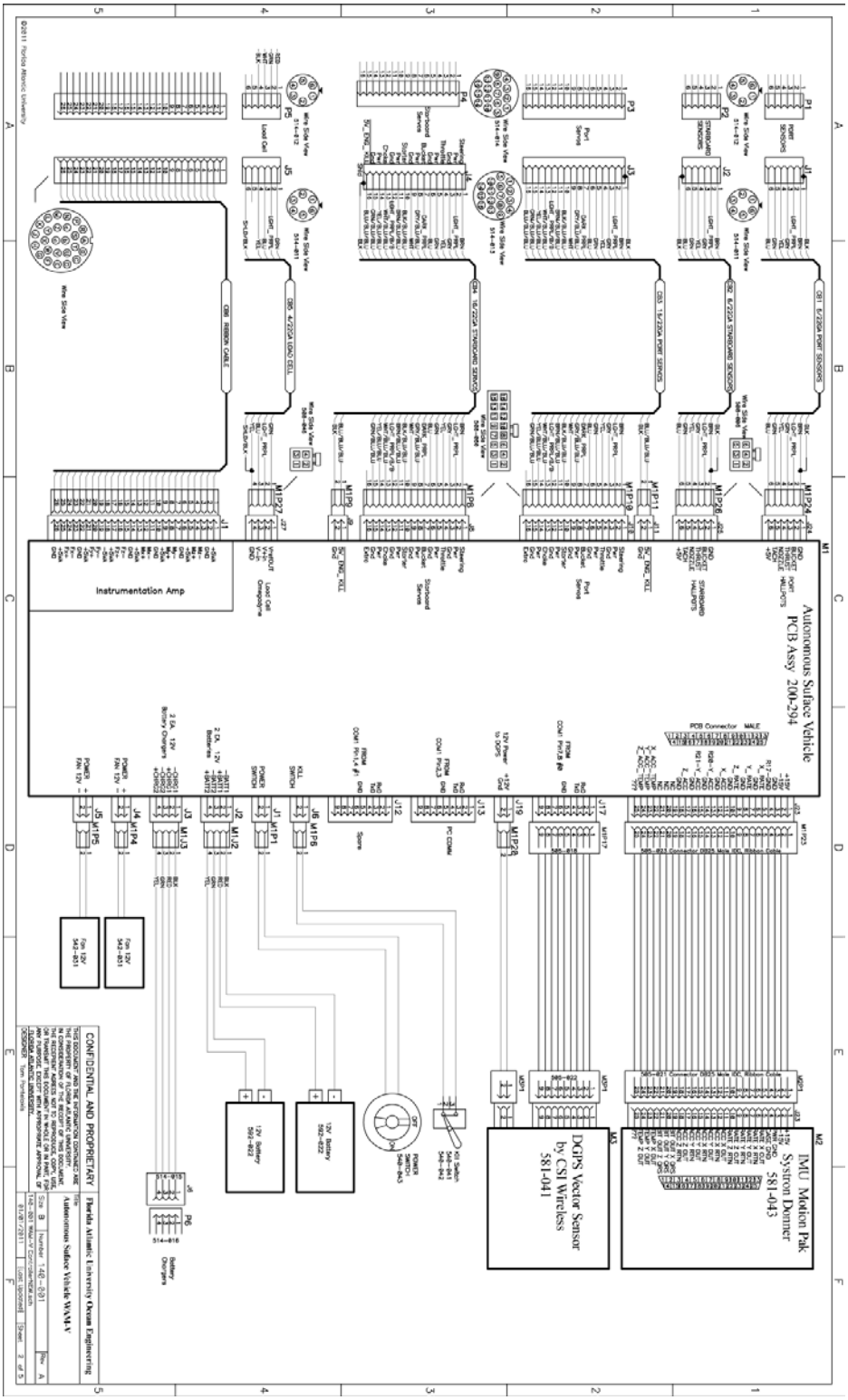

Figure 46: 140-001 WAM-V Wiring Diagram

6.2 SOFTWARE

6.2.1 **MATLAB Simulation Code**

mariner_sfcn.m:

```
function mariner_sfcn(block)
%
% The setup method is used to setup the basic attributes of the
% S-function such as ports, parameters, etc. Do not add any other
% calls to the main body of the function.
%

setup(block);

%endfunction

% Function: setup ===================================================
% Abstract:
%   Set up the S-function block's basic characteristics such as:
%   - Input ports
%   - Output ports
%   - Dialog parameters
%   - Options
%

function setup(block)

% Register number of ports
block.NumInputPorts  = 3;
block.NumOutputPorts = 5;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
%Port 1 -- thrust input.
block.InputPort(1).Dimensions          = 2;
block.InputPort(1).DatatypeID          = 0;  % double
block.InputPort(1).Complexity          = 'Real';
block.InputPort(1).DirectFeedthrough   = false;

%Port 2 -- azimuth angle.
block.InputPort(2).Dimensions          = 2;
block.InputPort(2).DatatypeID          = 0;  % double
block.InputPort(2).Complexity          = 'Real';
block.InputPort(2).DirectFeedthrough   = false;

%Port 3 -- IC's.
block.InputPort(3).Dimensions          = 10;
block.InputPort(3).DatatypeID          = 0;  % double
block.InputPort(3).Complexity          = 'Real';
block.InputPort(3).DirectFeedthrough   = false;

% Override output port properties
block.OutputPort(1).Dimensions  = 3;
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';

block.OutputPort(2).Dimensions  = 2;
block.OutputPort(2).DatatypeID  = 0; % double
```

```
block.OutputPort(2).Complexity  = 'Real';

block.OutputPort(3).Dimensions  = 1;
block.OutputPort(3).DatatypeID  = 0; % double
block.OutputPort(3).Complexity  = 'Real';

block.OutputPort(4).Dimensions  = 2;
block.OutputPort(4).DatatypeID  = 0; % double
block.OutputPort(4).Complexity  = 'Real';

block.OutputPort(5).Dimensions  = 2;
block.OutputPort(5).DatatypeID  = 0; % double
block.OutputPort(5).Complexity  = 'Real';

% Register parameters
block.NumDialogPrms     = 1;

% Register sample times
%  [0 offset]            : Continuous sample time
%  [positive_num offset] : Discrete sample time
%
%  [-1, 0]               : Inherited sample time
%  [-2, 0]               : Variable sample time
block.SampleTimes = [0 0];

% Setup Dwork
block.NumContStates = 10;

% Specify the block simStateComliance. The allowed values are:
%    'UnknownSimState', < The default setting; warn and assume DefaultSimState
%    'DefaultSimState', < Same sim state as a built-in block
%    'HasNoSimState',   < No sim state
%    'CustomSimState',  < Has GetSimState and SetSimState methods
%    'DisallowSimState' < Error out when saving or restoring the model sim state
block.SimStateCompliance = 'DefaultSimState';

% -----------------------------------------------------------------
% The M-file S-function uses an internal registry for all
% block methods. You should register all relevant methods
% (optional and required) as illustrated below. You may choose
% any suitable name for the methods and implement these methods
% as local functions within the same file. See comments
% provided for each function for more information.
% -----------------------------------------------------------------

block.RegBlockMethod('InitializeConditions',    @InitializeConditions);
block.RegBlockMethod('Outputs',                 @Outputs);     % Required
block.RegBlockMethod('Derivatives',             @Derivatives);
block.RegBlockMethod('Terminate',               @Terminate); % Required
block.RegBlockMethod('SetInputPortSamplingMode',@SetInputPortSamplingMode);
%end setup



function SetInputPortSamplingMode(block, idx, fd)

    block.InputPort(idx).SamplingMode = fd;

    block.OutputPort(1).SamplingMode = fd;
    block.OutputPort(2).SamplingMode = fd;
    block.OutputPort(3).SamplingMode = fd;
    block.OutputPort(4).SamplingMode = fd;
    block.OutputPort(5).SamplingMode = fd;
```

```
%
% InitializeConditions:
%   Functionality    : Called at the start of simulation and if it is
%                      present in an enabled subsystem configured to reset
%                      states, it will be called when the enabled subsystem
%                      restarts execution to reset the states.
%   Required         : No
%   C-MEX counterpart: mdlInitializeConditions
%
function InitializeConditions(block)
    %IC's for states.
    block.ContStates.Data = [-0.9 0 0 0 0 0 0 0 0 0];

%end InitializeConditions
%
% Outputs:
%   Functionality    : Called to generate block outputs in
%                      simulation step
%   Required         : Yes
%   C-MEX counterpart: mdlOutputs
%
function Outputs(block)
    %u,v,r
    block.OutputPort(1).Data = block.ContStates.Data(1:3);
    %x,y
    block.OutputPort(2).Data = block.ContStates.Data(5:6);
    %psi
    block.OutputPort(3).Data = block.ContStates.Data(4);
    %Omegap, Omegas
    block.OutputPort(4).Data = block.ContStates.Data(7:8);
    %Thetap, Thetas
    block.OutputPort(5).Data = block.ContStates.Data(9:10);

%end Outputs
%
% Derivatives:
%   Functionality    : Called to update derivatives of
%                      continuous states during simulation step
%   Required         : No
%   C-MEX counterpart: mdlDerivatives
%

function Derivatives(block)
State = block.ContStates.Data;
%The continuous states are stored in block.ContStates.Data.
%
%(1)    u       [m/s] perturbed surge velocity.
%(2)    v       [m/s] perturbed sway velocity.
%(3)    r       [rad/s] perturbed yaw velocity.
%(4)    psi     [rad] Heading (not bounded to -pi < psi < pi).
%(5)    x       [m] X position.
%(6)    y       [m] Y position.
%(7)    Omegap  [rad/s] Port thruster rotation.
%(8)    Omegac  [rad/s] Stbd thruster rotation.
Params = block.DialogPrm(1).Data;

rho             = Params(2);
L               = Params(3);
U0              = Params(4);

U   = sqrt((U0 + State(1))^2 + State(2)^2);
m               = Params(1)/rho/L^3;            %Mass.
Izz             = Params(5)/rho/L^5;
xG              = Params(6)/L;
Dx              = Params(7)/L;
Dy              = Params(8)/L;
```

```
Ct              = Params(9)*U/rho/L^4;
Thetas_max      = Params(10);
Thetap_max      = Params(10);
OmegasDotMax    = Params(11)*L/U;
OmegapDotMax    = Params(11)*L/U;


Thetas_dot_max = pi/2;
Thetap_dot_max = Thetas_dot_max;


u =         State(1)/U;     %Surge Velocity.
v =         State(2)/U;     %Sway velocity.
r =         State(3)/U*L;   %Yaw velocity.
psi =       State(4);       %Heading.
Omegap =    State(7)/U*L;       %Current port thruster rotation [rad/s].
Omegas =    State(8)/U*L;       %Current port thruster rotation [rad/s].
Thetap =    State(9);
Thetas =    State(10);
Omegap_c =  block.InputPort(1).Data(1)/U*L;     %Commanded port thruster rotation [].
Omegas_c =  block.InputPort(1).Data(2)/U*L;     %Commanded starboard thruster rotation
[].
Thetap_c =  block.InputPort(2).Data(1);
Thetas_c =  block.InputPort(2).Data(2);


Xudot = -840e-5;
Yvdot = -1546e-5;
Nvdot = 23e-5;

%non-dimensional hydrodynamic coefficients
Xu    = -184e-5;
Xuu   = -110e-5;
Xuuu  = -215e-5;
Xvv   = -899e-5;
Xrr   = 18e-5;
Xrv   = 798e-5;


Yrdot = 9e-5;
Yv    = -1160e-5;
Yr    = -499e-5;
Yvvv  = -8078e-5;
Yvvr  = 15356e-5;
Yvu   = -1160e-5 ;
Yru   = -499e-5;


Nrdot = -83e-5;
Nv    = -264e-5;
Nr    = -166e-5;
Nvv  = 1636e-5;
Nvvr  = -5483e-5;
Nvu   = -264e-5;
Nru   = -166e-5;


% Masses and moments of inertia


m11 = m-Xudot;
m22 = m-Yvdot;
m23 = m*xG-Yrdot;
m32 = m*xG-Nvdot;
m33 = Izz-Nrdot;


Omegap_dot = Omegap_c - Omegap;
Omegas_dot = Omegas_c - Omegas;


if abs(Omegap_dot) >= OmegapDotMax
```

```
    Omegap_dot = sign(Omegap_dot)*OmegapDotMax;
end
if abs(Omegas_dot) >= OmegasDotMax
    Omegas_dot = sign(Omegas_dot)*OmegasDotMax;
end

% Thrust saturation and dynamics (VERY simple).
if abs(Thetas_c) > Thetas_max
    Thetas_c = Thetas_max;
end

if abs(Thetap_c) > Thetap_max
    Thetap_c = Thetap_max;
end

Thetap_dot = Thetap_c - Thetap;
Thetas_dot = Thetas_c - Thetas;

if abs(Thetap_dot) >= Thetap_dot_max
    Thetap_dot = sign(Thetap_dot)*Thetap_dot_max;
end
if abs(Thetas_dot) >= Thetas_dot_max
    Thetas_dot = sign(Thetas_dot)*Thetas_dot_max;
end

% Forces and moments
X = Xu*u + Xuu*u^2 + Xuuu*u^3 + Xvv*v^2 + Xrr*r^2 + Xrv*r*v + ...
               + Ct*Omegap*abs(Omegap)*cos(Thetap) +...
               + Ct*Omegas*abs(Omegas)*cos(Thetas);

Y = Yv*v + Yr*r + Yvvv*v^3 + Yvvr*v^2*r + Yvu*v*u + Yru*r*u + ...
               + Ct*Omegap*abs(Omegap)*sin(Thetap) ...
               + Ct*Omegas*abs(Omegas)*sin(Thetas);

N = Nv*v + Nr*r + Nvv*v^3 + Nvvr*v^2*r + Nvu*v*u + Nru*r*u ...
               - Ct*Omegas*abs(Omegas)*cos(Thetas)*Dy + ... %
               + Ct*Omegap*abs(Omegap)*cos(Thetap)*Dy + ... %
               + Ct*Omegas*abs(Omegas)*sin(Thetas)*Dx + ...
               + Ct*Omegap*abs(Omegap)*sin(Thetap)*Dx;

% Dimensional stateh derivative
block.Derivatives.Data(1) =     X*(U^2/L)/m11;
block.Derivatives.Data(2) =     -(-m33*Y+m23*N)/(m22*m33-m23*m32)*U^2/L;
block.Derivatives.Data(3) =     (-m32*Y+m22*N)/(m22*m33-m23*m32)*U^2/L^2;
block.Derivatives.Data(4) =     r*U/L;
block.Derivatives.Data(5) =     (cos(psi)*(U0/U+u)-sin(psi)*v)*U;
block.Derivatives.Data(6) =     (sin(psi)*(U0/U+u)+cos(psi)*v)*U;
block.Derivatives.Data(7) =     Omegap_dot*U^2/L^2;
block.Derivatives.Data(8) =     Omegas_dot*U^2/L^2;
block.Derivatives.Data(9) =     Thetap_dot*U/L;
block.Derivatives.Data(10) =    Thetas_dot*U/L;
%end Derivatives.

%
% Terminate:
%   Functionality    : Called at the end of simulation for cleanup
%   Required         : Yes
%   C-MEX counterpart: mdlTerminate
%
function Terminate(block)

%end Terminate
```

6.2.2 **MATLAB Data Analysis Code**

parse_mission_data.m

```
clear, clc, clf, close all

%Some options.
PRINT = 0; %Comment out to suppress saving the files.
handles = [];
windowPos = [1 1 6 4.5];

%% Read all the data (that we might care about).
% Now read in some data. The directories that we care about are stored in
% a file who has the directory names for all the runss. It was made using
%
% $ find -name "*.dat" -printf "%h\n" | uniq > dirNamesComplete.txt
%
% from a directory above all the files. It could be more robust but I
% had already pared down the directory tree to be pretty minimal, so
% there isn't any trash. After the COMPLETE list was made, and I plotted
% some things, then I pared down that list even further with only the
% particular ones I cared about.

%Get directory names of all the runss.
    fileId = fopen('fileInventory.txt');

    if(fileId == -1)        %If it broke...
        error('Something is wrong with your directory list file.');
    end

    fgetl(fileId); %Read off the first line because it has a header.

    runs = struct(   'time', {}, ...
                'day', {}, ...
                'type', {}, ...
                'dirname', {},...
                'hasGps', {}, ...
                'gps', {}, ...
                'hasCmp', {}, ...
                'cmp', {}, ...
                'hasAdc', {}, ...
                'adc', {}, ...
                'hasSsc', {},...
                'ssc', {}, ...
                'hasMti', {},...
                'mti', {}, ...
                'hasIni', {}, ...       %Should be all ones.I made INI files for the
blanks.
                'ini', {},...
                'x', {}, ...
                'y', {}, ...
                'z', {}, ...
                'd', {}, ...
                'u', {}, ...
                'v', {});
    ctr = 1;
    while(~feof(fileId))
        C = textscan(fileId, '%s %d %d %d %d %d %d %d\r\n', 1);
        if(C{8} == 1)
           [runs(ctr).dirname runs(ctr).hasGps,...
               runs(ctr).hasCmp runs(ctr).hasAdc,...
               runs(ctr).hasSsc runs(ctr).hasMti,...
               runs(ctr).hasIni toss] = C{:};
           runs(ctr).dirname = runs(ctr).dirname{:};
```

```
            [a b c d] = fileparts(runs(ctr).dirname);
            runs(ctr).time = b;
            [a b c d] = fileparts(runs(ctr).dirname);
            runs(ctr).day  = b;
            clear a b c d;
            ctr = ctr + 1;
        end
    end
    fclose(fileId);
    numRuns = length(runs);

for ii = 1:numRuns

    %INI file.
    if(runs(ii).hasIni)
        ini_file = dir(fullfile(runs(ii).dirname, '*.ini'));
        runs(ii).ini = parseIniFile(fullfile(runs(ii).dirname, ini_file.name));
        runs(ii).type = runs(ii).ini.missionType;
    else
        disp('This doesn''t have an INI file. Beware.');
    end

    %Grab the filenames for the runs.
    %adc_file = dir(fullfile(runs(ii).dirname, 'adc_*.dat'));

    %GPS
    if(runs(ii).hasGps) %Semi dangerous if we had both MTi and GPS runsning. We did not.
        runs(ii).gps.file = dir(fullfile(runs(ii).dirname, 'gps_*.dat'));
        runs(ii).gps = parseGps(fullfile(runs(ii).dirname, runs(ii).gps.file.name));
        startingPos = [runs(ii).gps.lat(1) runs(ii).gps.lon(1) 0];
        for jj = 1:length(runs(ii).gps.lat)
            [runs(ii).x(jj) runs(ii).y(jj) runs(ii).z(jj) runs(ii).d(jj)] =
lla2ned(startingPos(1), startingPos(2), startingPos(3),...
                                        runs(ii).gps.lat(jj), runs(ii).gps.lon(jj), 0);

        end
    end

    %TCM5
    if(runs(ii).hasCmp)
        runs(ii).cmp.file = dir(fullfile(runs(ii).dirname, 'cmp_*.dat'));
        runs(ii).cmp = parseTCMFile(fullfile(runs(ii).dirname, runs(ii).cmp.file.name));
        runs(ii).cmp.yaw = unwrap(runs(ii).cmp.yaw - 180);
    end

    %SSC
    if(runs(ii).hasSsc)
        runs(ii).ssc.file = dir(fullfile(runs(ii).dirname, 'ssc_*.dat'));
        runs(ii).ssc = parseSscFile(fullfile(runs(ii).dirname, runs(ii).ssc.file.name));
    end

    %MTi-G
    if(runs(ii).hasMti)
        runs(ii).mti.file = dir(fullfile(runs(ii).dirname, 'mtig_*.dat'));
        runs(ii).mti = parseMtiFile(fullfile(runs(ii).dirname, runs(ii).mti.file.name));
        runs(ii).startingPos = [runs(ii).mti.lat(1) runs(ii).mti.lon(1)
runs(ii).mti.alt(1)];
        for jj = 1:length(runs(ii).mti.lat)
            [runs(ii).x(jj) runs(ii).y(jj) runs(ii).z(jj) runs(ii).d(jj)] =
lla2ned(runs(ii).startingPos(1), runs(ii).startingPos(2), runs(ii).startingPos(3),...
                                        runs(ii).mti.lat(jj), runs(ii).mti.lon(jj),
runs(ii).mti.alt(jj));
            vel_ned = [runs(ii).mti.xvel(jj); runs(ii).mti.yvel(jj);
runs(ii).mti.zvel(jj)];
```

```
            orientation = pi/180*[runs(ii).mti.yaw(jj); runs(ii).mti.roll(jj);
runs(ii).mti.pitch(jj)];
            vel_bf = ned2bf(vel_ned, orientation);
            runs(ii).u(jj) = vel_bf(1);
            runs(ii).v(jj) = vel_bf(2);
        end
    end
end

%% Do work.
%  Depending on the runs type, do different analysis, different things,
%  etc.
clc, close all
handles = [];
windowPos = [1 1 6 6];

straightPlot = [12 13 15 17 24];
circlePlot   = [42];
headingPlot  = [44];
waypointPlot = [49 51];

interestingRuns = [straightPlot circlePlot headingPlot waypointPlot];
interestingRuns = waypointPlot;

for kk = interestingRuns
    switch runs(kk).type
        case 'OPEN_STRAIGHT' %I don't think I'll use any plots from this, I
                             %have no good speed information.

            %Plot trajectory
            handles = [handles figure]; hold on; legendCell = {}; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos);
            text(0,-0.3,[runs(kk).dirname ': ' strrep(runs(kk).type, '_', ' ')] ,
'Units', 'inches', 'FontSize', 6)
            plot(runs(kk).y, runs(kk).x, 'x'), axis equal, set(gca, 'FontSize', 9)
            title('GPS-Measured USV Trajectory for Straight Line Test');
            xlabel('{\it x}-Position [m]'), ylabel('{\it y}-Position [m]');

            %Calculate speed using GPS and compass (GPS gives EF velocity,
            %cmp gives orientation)
            %Resample so dt lines up.
            interpYaw = interp1(runs(kk).cmp.t, runs(kk).cmp.yaw, runs(kk).gps.t(1:end-
1), 'cubic');
            orientation = pi/180 * [interpYaw; zeros(2,length(interpYaw))];
            vel_ned = [ diff(runs(kk).x)./(diff(normSerialDate(runs(kk).gps.t)));
                        diff(runs(kk).y)./(diff(normSerialDate(runs(kk).gps.t)));
                        zeros(1,length(runs(kk).gps.t) - 1)];
            vel_bf = ned2bf(vel_ned, orientation);
            runs(kk).u = vel_bf(1,:);
            runs(kk).v = vel_bf(2,:);

            %Plot calculated speed, along with ssc command.
            handles = [handles figure]; hold on; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos), set(gca, 'FontSize', 9);

            subplot(4,1,1:3), set(gca, 'FontSize', 9),
            plot(normSerialDate(runs(kk).gps.t(1:end-1)), runs(kk).u, 'b',...
                normSerialDate(runs(kk).gps.t(1:end-1)), runs(kk).v, 'r');



            xlim(normSerialDate([min(runs(kk).gps.t(1:end-1)) max(runs(kk).gps.t(1:end-
1))]))
            title({'Body-Fixed Velocities For Straight Line Test'});
```

```
            ylabel('Velocity, [m/s]')
            legend( 'Surge, \it{u}',...
                    'Sway, \it{v}',...
                    'Location', 'NorthWest');
            subplot(4,1,4), set(gca, 'FontSize', 9)
            plot(normSerialDate(runs(kk).ssc.t), 100*runs(kk).ssc.u, 'm-');
            text(0,-0.3,[runs(kk).dirname ': ' strrep(runs(kk).type, '_', ' ')] ,
'Units', 'inches', 'FontSize', 6)
            ylabel('Surge Command, {\it\delta_r}');
            xlabel('Time, {\it t } [s]');

            %Plot heading information.

        case 'OPEN_CIRCLE'

            %Plot trajectory
            handles = [handles figure]; hold on; legendCell = {}; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos);
            text(0,-0.3,[runs(kk).dirname ': ' strrep(runs(kk).type, '_', ' ')] ,
'Units', 'inches', 'FontSize', 6)
            plot(runs(kk).y, runs(kk).x, 'x'), axis equal, set(gca, 'FontSize', 9)
            title('GPS-Measured USV Trajectory for Straight Line Test');
            xlabel('{\it x}-Position [m]'), ylabel('{\it y}-Position [m]');

            %Calculate speed using GPS and compass (GPS gives EF velocity,
            %cmp gives orientation)
            %Resample so dt lines up.
            interpYaw = interp1(runs(kk).cmp.t, runs(kk).cmp.yaw, runs(kk).gps.t(1:end-
1), 'cubic');
            orientation = pi/180 * [interpYaw; zeros(2,length(interpYaw))];
            vel_ned = [ diff(runs(kk).x)./(diff(normSerialDate(runs(kk).gps.t)));
                        diff(runs(kk).y)./(diff(normSerialDate(runs(kk).gps.t)));
                        zeros(1,length(runs(kk).gps.t) - 1)];
            vel_bf = ned2bf(vel_ned, orientation);
            runs(kk).u = vel_bf(1,:);
            runs(kk).v = vel_bf(2,:);

            %Calculate yaw rate from CMP.
            r = pi/180*diff(runs(kk).cmp.yaw)./diff(normSerialDate(runs(kk).cmp.t));



            %Plot heading with dr command.
            handles = [handles figure]; hold on; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos), set(gca, 'FontSize', 9);

            subplot(2,1,1), plot(normSerialDate(runs(kk).cmp.t), runs(kk).cmp.yaw, 'b'),
hold on;
            xlim(normSerialDate([min(runs(kk).cmp.t(1:end-1)) max(runs(kk).cmp.t(1:end-
1))]))
            set(gca, 'FontSize', 9);
            title({'Heading Response for Circle Test'});
             plot(normSerialDate(runs(kk).ssc.t), -100*runs(kk).ssc.dr, 'r:',...
                normSerialDate(runs(kk).ssc.t), 100*runs(kk).ssc.u, 'm-.');

            legend({'Heading, \psi [deg]', ...
                'Rudder \delta_r', ...
                'Forward \delta_u'},...
                'Location', 'NorthWest');

            subplot(2,1,2);
            plot(normSerialDate(runs(kk).cmp.t(1:(end-1))), r, 'b'); set(gca, 'FontSize',
9);
            ylabel('Yaw Rate, {\it r} [rad/s]');
```

```
            text(0,-0.3,[runs(kk).dirname ': ' strrep(runs(kk).type, '_', ' ')] ,
'Units', 'inches', 'FontSize', 6)
            xlabel('Time, {\it t } [s]');

        case 'OPEN_ZIGZAG'

            %Calculate speed using GPS and compass (GPS gives EF velocity,
            %cmp gives orientation)
            %Resample so dt lines up.
            interpYaw = interp1(runs(kk).cmp.t, runs(kk).cmp.yaw, runs(kk).gps.t(1:end-
1), 'cubic');
            orientation = pi/180 * [interpYaw; zeros(2,length(interpYaw))];
            vel_ned = [ diff(runs(kk).x)./(diff(normSerialDate(runs(kk).gps.t)));
                        diff(runs(kk).y)./(diff(normSerialDate(runs(kk).gps.t)));
                        zeros(1,length(runs(kk).gps.t) - 1)];
            vel_bf = ned2bf(vel_ned, orientation);
            runs(kk).u = vel_bf(1,:);
            runs(kk).v = vel_bf(2,:);

            %Calculate yaw rate from CMP.
            r = pi/180*diff(runs(kk).cmp.yaw)./diff(normSerialDate(runs(kk).cmp.t));

            %Plot heading with dr command.
            handles = [handles figure]; hold on; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos), set(gca, 'FontSize', 9);

            subplot(5,1,1:2), plot(normSerialDate(runs(kk).cmp.t), runs(kk).cmp.yaw -
runs(kk).cmp.yaw(1), 'b'), hold on;
            xlim(normSerialDate([min(runs(kk).cmp.t(1:end-1)) max(runs(kk).cmp.t(1:end-
1))]))
            set(gca, 'FontSize', 9);
            xlimSave = xlim;
            title({'Heading Response for Zig-zag Test'});

            ylabel({'Heading, \psi [deg]'});

            subplot(5,1,3:4);
            plot(normSerialDate(runs(kk).cmp.t(1:(end-1))), r, 'b'); set(gca, 'FontSize',
9);
            ylabel('Yaw Rate, {\it r} [rad/s]'), xlim(xlimSave);

            subplot(5,1,5);
            plot(normSerialDate(runs(kk).ssc.t), 100*runs(kk).ssc.dr, 'm-.'),set(gca,
'FontSize', 9);
            ylabel('Rudder \delta_r'), ylim([-100 100]); xlim(xlimSave);
            text(0,-0.3,[runs(kk).dirname ': ' strrep(runs(kk).type, '_', ' ')] ,
'Units', 'inches', 'FontSize', 6)
            xlabel('Time, {\it t } [s]');

        case 'CLOSED_HEADING'

            %Plot trajectory
            handles = [handles figure]; hold on; legendCell = {}; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos);
text(0,-0.3,strrep([runs(kk).dirname ': ' runs(kk).type], '_', ' ') , 'Units', 'inches',
'FontSize', 6);
            plot(runs(kk).y, runs(kk).x, 'x'), axis equal, set(gca, 'FontSize', 9)
            title('MTiG-Measured USV Trajectory for Heading Controller Test');
            xlabel('{\it x}-Position [m]'), ylabel('{\it y}-Position [m]');

            %Calculate yaw rate from CMP.
            r = pi/180*diff(runs(kk).cmp.yaw)./diff(normSerialDate(runs(kk).cmp.t));
```

```
            %find the gains being used.
            k_p = [runs(kk).ini.segment(findMaxSeg(runs(kk).ini)).kp_heading];

            heading_segs = findMaxSeg(runs(kk).ini);

            psi_cmd = [runs(kk).ini.segment(heading_segs).headingCmd];

            t_cmd   = [0 cumsum([runs(kk).ini.segment.duration])];

            t_plot = zeros(1,length(heading_segs)*2);

            %this is the ugliest thing I have ever done.
            for mm = 1:length(psi_cmd)
                t_plot(2*mm-1:2*mm) = [t_cmd(heading_segs(mm))
t_cmd(heading_segs(mm)+1)];
                psi_plot(2*(mm)-1:2*mm) = [psi_cmd(mm) psi_cmd(mm)];
            end

            %calculate the error (this wasn't saved, so we need to
            %recalculate.
            t_span = normSerialDate(runs(kk).mti.t);
            t_tweak = [(diff(t_plot) == 0) 0].* ...
                        (t_plot(find(t_plot == t_plot(~diff(t_plot)), 1, 'first')) -
0.001) + ...
                        [(diff(t_plot) ~= 0) 1].*t_plot;

            psi_c_span = interp1(t_tweak, psi_plot, t_span);

            e_psi = psi_c_span - runs(kk).mti.yaw;

            %Plot heading with dr command.
            handles = [handles figure]; hold on; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos), set(gca, 'FontSize', 9);

            subplot(2,1,1), plot(normSerialDate(runs(kk).mti.t), ...
                    runs(kk).mti.yaw, 'b'), hold on;
                    plot(t_plot, psi_plot, 'r--'),
            xlim(normSerialDate([min(runs(kk).mti.t(1:end-1)) max(runs(kk).mti.t(1:end-
1))]))
            set(gca, 'FontSize', 9);
            xlimSave = xlim;
            title(['Heading Response for Heading Controller Test, k_p = '
num2str(k_p(1))]);
            legend({'Heading, \psi [deg]', 'Command, \psi_c [deg]'}, 'FontSize', 9);

            subplot(2,1,2);
            plot(normSerialDate(runs(kk).ssc.t), -100*runs(kk).ssc.dr, 'm-.'), hold on,
            set(gca, 'FontSize', 9);
            plot(normSerialDate(runs(kk).ssc.t), 100*runs(kk).ssc.u, 'b:');

            hold on;
            plot(t_span, e_psi, 'b');

            legend({'Rudder Cmd \delta_r',...
                'Speed Cmd, \delta_u',...
                'Heading Error, {\it e_\psi}'}, 'FontSize', 9)

            ylim([-100 100]); xlim(xlimSave);
            text(0,-0.3,strrep([runs(kk).dirname ': ' runs(kk).type], '_', ' ') ,
'Units', 'inches', 'FontSize', 6);
```

```
            xlabel('Time, {\it t } [s]');

        case 'CLOSED_WAYPOINT'
            clear x_cmd y_cmd
            %Plot trajectory
            handles = [handles figure]; hold on; legendCell = {}; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos);
            text(0,-0.3,strrep([runs(kk).dirname ': ' runs(kk).type], '_', ' ') ,
'Units', 'inches', 'FontSize', 6);
            plot(runs(kk).y, runs(kk).x, 'x'), axis equal, set(gca, 'FontSize', 9), hold
on
            title('MTiG-Measured USV Trajectory for Waypoint Controller Test');
            xlabel('{\it x}-Position [m]'), ylabel('{\it y}-Position [m]');

            %Calculate yaw rate from CMP.
            r = pi/180*diff(runs(kk).cmp.yaw)./diff(normSerialDate(runs(kk).cmp.t));


            %find the gains being used.
            k_p = [runs(kk).ini.segment(findMaxSeg(runs(kk).ini)).kp_heading];


            %figure out what our waypoints are.
            heading_segs = findMaxSeg(runs(kk).ini);
            lat_cmd = [runs(kk).ini.segment(heading_segs).latitude];
            lon_cmd = [runs(kk).ini.segment(heading_segs).longitude];


            maxDistance = unique([runs(kk).ini.segment(heading_segs).maxDistance]);


            x_cmd = zeros(size(heading_segs));
            y_cmd = x_cmd;
            z_cmd = x_cmd;
            d   = x_cmd;
            for jj = 1:length(heading_segs)
                [x_cmd(jj) y_cmd(jj) z_cmd(jj) d] = lla2ned(runs(kk).startingPos(1), ...
                                                runs(kk).startingPos(2), ...
                                                runs(kk).startingPos(3), ...
                                                lat_cmd(jj), lon_cmd(jj), 0);


            end


            %plot the waypoints
            plot(y_cmd(1:end-1), x_cmd(1:end-1), 'rd', 'markersize', 7,
'markerfacecolor', 'r');


%             t_cmd   = [0 cumsum([runs(kk).ini.segment.duration])];
%             t_plot = zeros(1,length(heading_segs)*2);
%
%             %this is the ugliest thing I have ever done.
%             for mm = 1:length(psi_cmd)
%                 t_plot(2*mm-1:2*mm) = [t_cmd(heading_segs(mm))
t_cmd(heading_segs(mm)+1)];
%                 psi_plot(2*(mm)-1:2*mm) = [psi_cmd(mm) psi_cmd(mm)];
%             end


            %calculate the error (this wasn't saved, so we need to
            %recalculate.
%             t_span = normSerialDate(runs(kk).mti.t);
%             t_tweak = [(diff(t_plot) == 0) 0].* ...
%                         (t_plot(find(t_plot == t_plot(~diff(t_plot)), 1, 'first')) -
0.001) + ...
%                         [(diff(t_plot) ~= 0) 1].*t_plot;
%
%             psi_c_span = interp1(t_tweak, psi_plot, t_span);
%
%             e_psi = psi_c_span - runs(kk).mti.yaw;
```

```
%
            %Plot heading with dr command.
            handles = [handles figure]; hold on; %#ok<*NASGU,*AGROW>
            set(gcf, 'Units', 'inches', 'Position', windowPos), set(gca, 'FontSize', 9);

            subplot(2,1,1), plot(normSerialDate(runs(kk).mti.t), ...
                    runs(kk).mti.yaw, 'b'), hold on;

            xlim(normSerialDate([min(runs(kk).mti.t(1:end-1)) max(runs(kk).mti.t(1:end-
1))]))
            set(gca, 'FontSize', 9);
            xlimSave = xlim;
            title(['Heading Response for Waypoint-Tracking Controller Test, k_p = '
num2str(k_p(1))]);
            ylabel('Heading, \psi [deg]', 'FontSize', 9);

            subplot(2,1,2);
            plot(normSerialDate(runs(kk).ssc.t), -100*runs(kk).ssc.dr, 'm-.'), hold on,
            set(gca, 'FontSize', 9);
            plot(normSerialDate(runs(kk).ssc.t), 100*runs(kk).ssc.u, 'b:');

            hold on;
%             plot(t_span, e_psi, 'b');

            legend({'Rudder Cmd \delta_r',...
                'Speed Cmd, \delta_u'}, 'FontSize', 9)

            ylim([-100 100]); xlim(xlimSave);
            text(0,-0.3, [runs(kk).dirname ': ' strrep(runs(kk).type, '_', ' ')] ,
'Units', 'inches', 'FontSize', 6)
            xlabel('Time, {\it t } [s]');
        otherwise
    end
end

%% Special circle test plotter.
clc, close all
handles = [];

t = (-5:0.1:25)';
psi_moved = zeros(length(t), length(interestingRuns));
psiCtr = 1;
colorStr = {'r', 'b', 'g', 'm', 'k', 'c', 'b.', 'g.', 'm.', 'k.'};
for kk = interestingRuns
            i_start = find(diff(runs(kk).ssc.dr), 1, 'first');
            t_start = normSerialDate(runs(kk).ssc.t);
            t_start = t_start(i_start);
            psi_start = interp1(normSerialDate(runs(kk).cmp.t), runs(kk).cmp.yaw,
t_start);
            psi_moved(:,psiCtr) = interp1(normSerialDate(runs(kk).cmp.t) - t_start,
runs(kk).cmp.yaw - psi_start, t);
            psi_dr(psiCtr) = -1*runs(kk).ini.segment(findMaxSeg(runs(kk).ini)).rudder;
            psi_u(psiCtr) = -1*runs(kk).ini.segment(findMaxSeg(runs(kk).ini)).sscSpeed;
%             plot(t, psi_moved(:,psiCtr), lineStr{psiCtr}), hold on
%             legendStr{end+1} = (sprintf('%d', psi_dr(:,psiCtr)));
            r_avg(psiCtr) = mean(diff(runs(kk).cmp.yaw((i_start+200):500))./...
                        diff(normSerialDate(runs(kk).cmp.t((i_start+200):500))));
            psiCtr = psiCtr + 1;
end

%Plot the rates.
handles = [handles figure]; hold on; legendStr = {};
set(gcf, 'Units', 'inches', 'Position', windowPos), set(gca, 'FontSize', 9);
plot(psi_dr, pi/180*r_avg, 'sb', 'markersize', 8, 'markerfacecolor', 'b'), hold on
```

```
% p = polyfit(psi_dr, pi/180*r_avg,2);
% plot(unique(psi_dr), polyval(p, unique(psi_dr)), 'b');
title('Average Turning Rates During Turning Trials');
ylabel('Yaw rate, {\it r} [rad/s]');
xlabel('Virtual Rudder Input, {\it \delta_r}');
axis([40 135 0.045 0.14]);
text(0,-0.3,strrep([runs(kk).dirname ': ' runs(kk).type], '_', ' ') , 'Units', 'inches',
'FontSize', 6);

%Plot the responses together.
handles = [handles figure]; hold on; legendStr = {};
set(gcf, 'Units', 'inches', 'Position', windowPos), set(gca, 'FontSize', 9);
text(0,-0.3,strrep([runs(kk).dirname ': ' runs(kk).type], '_', ' ') , 'Units', 'inches',
'FontSize', 6);

[psi_dr psi_dr_iSort] = sort(psi_dr, 'descend');
dr_range = fliplr(unique(psi_dr));
psi_moved(:,psi_dr_iSort);
marks = {'o-', '<-', 's-', '^-', 'd-', 'v-'};
for jj = 1:length(dr_range)
    hSLines{jj} = plot(t(1:20:length(psi_moved)), ...
                       psi_moved(1:20:length(psi_moved),psi_dr == dr_range(jj)), ...
        [colorStr{jj} marks{jj}], 'markersize', 6, 'markerfacecolor', colorStr{jj}); hold
on
    hSGroup{jj} = hggroup;
    set(hSLines{jj},'Parent', hSGroup{jj})

    % Include these hggroups in the legend:
    set(get(get(hSGroup{jj},'Annotation'),'LegendInformation'),...
        'IconDisplayStyle','on');
end
legend( '{\it\delta_r} = 130', ...
    '{\it\delta_r} = 120', ...
    '{\it\delta_r} = 100', ...
    '{\it\delta_r} = 80', ...
    '{\it\delta_r} = 75', ...
    '{\it\delta_r} = 50', ...
    'Location', 'northwest');
ylabel('Heading, \psi [deg]'),
title('Heading Response for Various Circle Trials');
text(0,-0.3,[runs(kk).dirname ': ' strrep(runs(kk).type, '_', ' ')] , 'Units', 'inches',
'FontSize', 6)
xlabel('Time, {\it t } [s]');

%% Save Plots.
plotDir = 'C:\Users\Tom\Desktop\Dropbox STuff\TF\Thesis Docs\Figures\data_analyzed';
if(PRINT == 1)
    for ii = [2 3 4]
        figure(handles(ii)),
        set(gcf, 'Units', 'inches', 'Position', windowPos)
        set(gcf, 'PaperPositionMode', 'auto')
        set(gca, 'FontSize', 9)
        print(sprintf('%s\\heading_test%d.png', plotDir, handles(ii)), '-dpng', '-r400');
%         print(sprintf('%s\\%s_%s_%s_%d.png', plotDir,...
%                             runs(kk).day, ...
%                             runs(kk).time, ...
%                             runs(kk).type, ...
%                             handles(ii)), '-dpng', '-r300');
    end
end
```

parseGps.m:

```
function gps = parseGps(filename)
% Parse GPS (position) log on GUSSerino
```

```
%   Time[s]         Lat [deg]       Lon[deg]        S   SoG         CoG
% 1319751458.146    26.055770874    -80.113075256   1   0.750000    193.619995  1
% 1319751458.146    26.055770874    -80.113075256   1   0.700000    193.619995  1
% 1319751458.146    26.055770874    -80.113075256   1   0.680000    193.619995  1
% 1319751458.146    26.055772781    -80.113075256   1   0.670000    193.619995  1

raw = dlmread(filename);

%Also, sometimes the GPS doesn't update between readings. So, if neither
%lat or lon has moved, we will discard the entire reading. "Smoothing" the data
%sort of. (Really just getting rid of the quantization noise. If we do
%velocity calculations without this the boat appears to be stationary for
%much of the runs.

changed = (diff(raw(:,2)) ~= 0 ) | (diff(raw(:,3)) ~= 0);

%Note that at this point the times were messed up. So we have to do some
%manipulating to get the time back out. (We knowt he start and the dt).
t = nixTime2matlabTime(raw(1,1) + 0.2*(0:(length(raw)-1)));
gps(1).t = t(changed);
gps.lat = raw(changed,2);
gps.lon = raw(changed,3);
gps.sog = raw(changed,5);
gps.cog = raw(changed,6);
```

parseIniFile.m:

```
function m = parseIniFile(filename)
%This attempts to take the mission INI file and put it in a structure.
%We'll see how it goes.
%
%NOTE: This file will probably NOT work if the general schema of the INI
%file is changed on the vehicle. Partially, the structure defined here
%doesn't really do any checking, and secondly, there are some capabilities
%in the INI parser itself (which I did not write) that this won't be able
%to handle. Namely, if a major field, like [SEG0], has a value assigned to
%it, I make no guarantees. Originally I wanted the 'type' to be declared
%like "[SEG0] = type_name_here;", but for some reason (I can't remember
%why) it was more work than I really wanted to invest to get that to work
%correctly. Good luck.

%Open 'er up.
fileId = fopen(filename, 'r');

segCtr = 1;

%For some sketchiness later.
possibleLegs = {    'NEUTRAL', ...
                    'OPEN_START', ...
                    'OPEN_STRAIGHT', ...
                    'OPEN_CIRCLE', ...
                    'OPEN_ZIGZAG', ...
                    'CLOSED_SPEED', ...
                    'CLOSED_HEADING', ...
                    'CLOSED_WAYPOINT', ...
                    'CLOSED_TRACK'};
maxType = 1;
%Initialize the structure (is this even necessary?
s = struct(   'num', {}, ...
                    'type', {}, ...
                    'waitForGps', {}, ...
                    'gpsPeriod', {}, ...
                    'waitTime', {}, ...
```

```
                    'wait', {}, ...
                    'sscSpeed', {}, ...
                    'duration', {}, ...
                    'turnDelay', {}, ...
                    'rudder', {}, ...
                    'turnDir', {}, ...
                    'period', {}, ...
                    'turns', {}, ...
                    'speedCmd', {}, ...
                    'dt', {}, ...
                    'kp_speed', {}, ...
                    'ki_speed', {}, ...
                    'kd_speed', {}, ...
                    'headingCmd', {}, ...
                    'kp_heading', {}, ...
                    'ki_heading', {}, ...
                    'kd_heading', {}, ...
                    'latitude', {}, ...
                    'longitude', {}, ...
                    'maxDistance', {}, ...
                    'lookAhead', {}, ...
                    'time', {});

%Scan line-by-line.
while(~feof(fileId)) %While we aren't at the end of the file.
    %Get the line.
    line = fgetl(fileId);
    %Split it, if there are some comments.
    %Note here, the comments are discarded. I really wanted to
    %same them and add them in, but this structure is already getting
    %unwieldly.
    split = regexp(line, ';', 'split');
    data = split(1);

    %If it's a segment counter...
    if(strncmp(data,'[SEG', 4))
        segCtr = sscanf([data{:}], '[SEG%d]');
        s(segCtr).num = segCtr;
    end

    %If a value is assigned on that line, parse it out and put it away.
    if(ismember('=', data{:}))
        fieldInfo = textscan([data{:}], '%s = %s');
        if(any(isletter(fieldInfo{2}{:})))
            s(segCtr) = setfield(s(segCtr), fieldInfo{1}{:},  fieldInfo{2}{:});
        else
            s(segCtr) = setfield(s(segCtr), fieldInfo{1}{:}, str2num(fieldInfo{2}{:}));
        end
    end
end

%Here I do a bit of funny checking. I want to know what kind of mission
%this is, and while it could be heterogeneous, in general there is some
%sense of rank among leg types. So, I just search for the highest in the
%rank.

for ii = 1:segCtr
    for jj = 1:length(possibleLegs)
        if(strcmp(possibleLegs{jj}, s(ii).type))
            if(jj > maxType)
                maxType = jj;
            end
        end
    end
end
```

```
m = struct('segment', s, 'missionType', possibleLegs{maxType});
fclose(fileId);
```

parseTCMFile.m:

```
function tcm = parseTCMFile(filename)
    fileId = fopen(filename);
    firstLine = fgetl(fileId);
    fclose(fileId);
    decSecs = sscanf(firstLine, 'Parsed Compass Data, %*4d%*2d%*2d_%*2d%*2d%*2d%f');

    %Because we want the decimal seconds off of that stamp, but don't need
    %the seconds, because they are a LIE!. Instead we take the stamp from
    %the filename. (The assumption is that the conversion chopped off whole
    %numbers of seconds because of leap seconds.

    filename_nodir = filename(strfind(filename, 'cmp_'):end);
    ActualStartTime = sscanf(filename_nodir, 'cmp_%4d%2d%2d_%2d%2d%2d.dat');
    StartTime = ActualStartTime + [0 0 0 0 0 decSecs]';

    raw_dat = dlmread(filename,'\t', 2, 0);
    Npts = length(raw_dat) - 1; %Chop off last because sometimes it's incomplete.
    tr = raw_dat(1:Npts,1);
    %Datenum converts a vector (or matrix) of m-by-6 to
    % a decimal number of days since Jan 0, 0000 (why?)
    % this builds a matrxi of size Npts-by-6, combining the tag
    % in the first line of the file with the t values in the rows.

    tcm(1).t  = datenum(ones(Npts,6)*diag(StartTime) + [zeros(Npts,5) (tr)]);
    tcm.roll = raw_dat(1:Npts,2);
    tcm.pitch = raw_dat(1:Npts,3);
    tcm.yaw = raw_dat(1:Npts,4);
```

ned2bf.m:

```
function R_bf = ned2bf(R_ned, orientation)
%  NED2BF           Takes North-East-Down (NED) based vector and transforms
%                   it to the body fixed frame. The x,y,z Euler angles
%                   between the BF and NED frames is passed through
%                   orientation, whereas the column-vector to be
%                   transformed is passed as R_ned.

if(size(R_ned, 1) ~= 3)
    R_ned = R_ned';
end

if(size(orientation, 1) ~= 3)
    orientation = orientation';
end

R_bf = zeros(size(R_ned));

for jj = 1:size(R_ned,2)
    psi = orientation(1,jj);
    theta = orientation(2,jj);
    phi = orientation(3,jj);

    C_g2b = [   c(psi)*c(theta),...
                (-1*s(psi)*c(phi) + c(psi)*s(theta)*s(phi)),...
                (s(psi)*s(phi) + c(psi)*s(theta)*c(phi));
```

```
                c(theta)*s(psi),...
                (c(psi)*c(phi) + s(psi)*s(theta)*s(phi)),...
                (-1*c(psi)*s(phi) + s(psi)*s(theta)*c(phi));
                -1*s(theta),...
                c(theta)*s(phi),...
                c(theta)*c(phi)]';

    R_bf(:,jj) = C_g2b*R_ned(:,jj);
end
function out = c(in)
out = cos(in);

function out = s(in)
out = sin(in);
```

findMaxSeg.m:

```
function seg = findMaxSeg(runini)
seg = [];
for ii = 1:length(runini.segment)
    if(strcmp(runini.segment(ii).type, runini.missionType))
        seg = [seg ii];
    end
end
```

parseADCFile.m:

```
function [t Volts] = parseADCFile(filename)

    ADC_XRATE_CHAN = 2;
    ADC_YRATE_CHAN = 4;
    ADC_ZRATE_CHAN = 6;
    ADC_XACC_CHAN   = 8;
    ADC_YACC_CHAN   = 10;
    ADC_ZACC_CHAN   = 12;
    ADC_PTACH_CHAN  = 13;
    ADC_STACH_CHAN  = 15;

    raw_dat = dlmread(filename, '\t', 7, 0);
    t = raw_dat(:,1);
    Volts = raw_dat(:,2:end);
```

ecef2ned.m:

```
function R_ned = ecef2ned(R_ecef, lat, lon)

C_e2g = [   -1*s(lat)*c(lon),   -1*s(lat)*s(lon),   c(lat);
            -1*s(lon),          c(lon),             0;
            -1*c(lat)*c(lon),   -1*c(lat)*s(lon),   -1*s(lat)];

R_ned = C_e2g * R_ecef;

function out = c(in)
out = cosd(in);

function out = s(in)
out = sind(in);
```

fileInventorier.m:

```
clear, clc, clf, close all

%Some options.
SAVEFIGS = []; %Comment out to suppress saving the files.
colorStr = {'r', 'g', 'b', 'c', 'm', 'y', 'k'};
handles =[];

%% Load Data.
% Now read in some data. The directories that we care about are stored in
% a file who has the directory names for all the runs. It was made using
%
% $ find -name "*.dat" -printf "%h\n" | uniq > dirNamesComplete.txt
%
% from a directory above all the files. It could be more robust but I
% had already pared down the directory tree to be pretty minimal, so
% there isn't any trash. After the COMPLETE list was made, and I plotted
% some things, then I pared down that list even further with only the
% particular ones I cared about.

%Get directory names of all the runs.
    fileId = fopen('dirNames.txt');

    if(fileId == -1)        %If it broke...
        error('Something is wrong with your directory list file.');
    end

    dirname = {};           %Needed because of the concatenating in the loop.

    while(~feof(fileId))    %While we aren't at the end of the file...
        dirname = [dirname {fgetl(fileId)}]; %Read in the names.
    end

    numRuns = length(dirname);

%Prepare a structure to hold everything.
    run = struct(   'time', {}, ...
                    'type', {}, ...
                    'gps', {}, ...
                    'cmp', {}, ...
                    'adc', {}, ...
                    'ssc', {}, ...
                    'mti', {}, ...
                    'ini', {});

interestingRuns = 1:numRuns;
haveFile = zeros(numRuns, 6);
doICare = ~ismember(1:numRuns, [64 63 57 56 55 53 48 47 44 39 18 16 8 5 2 1]);

for ii = interestingRuns
    %Grab the filenames for the run.
    gps_file = dir(fullfile(dirname{ii}, 'gps_*.dat'));
    cmp_file = dir(fullfile(dirname{ii}, 'cmp_*.dat'));
    adc_file = dir(fullfile(dirname{ii}, 'adc_*.dat'));
    ssc_file = dir(fullfile(dirname{ii}, 'ssc_*.dat'));
    mti_file = dir(fullfile(dirname{ii}, 'mtig_*.dat'));
    ini_file = dir(fullfile(dirname{ii}, '*.ini'));

    haveFile(ii,:) = [~isempty(gps_file),...
                      ~isempty(cmp_file),...
                      ~isempty(adc_file),...
                      ~isempty(ssc_file),...
```

```
                    ~isempty(mti_file),...
                    ~isempty(ini_file)];
end

fileId = fopen('fileInventory.txt', 'w');
for ii = 1:numRuns
    fprintf(fileId, '%s\t\t%i\t\t%i\t\t%i\t\t%i\t\t%i\t\t%i\t\t%i\t\t%i\r\n', dirname{ii},
haveFile(ii,:), doICare(ii));
end
fclose(fileId);

%     handles = [handles figure];
%     hold on
%     %Parse the data.
%         %MTi-G
%
%     if(~isempty(mti_file))
%         mti(ii) = parseMtiFile(fullfile(dirname{ii}, mti_file.name));
%         startingPos = [mti(ii).lat(1) mti(ii).lon(1) mti(ii).alt(1)];
%         for jj = 1:length(mti(ii).lat)
%             [x(jj) y(jj) z(jj) d(jj)] = lla2ned(startingPos(1), startingPos(2),
startingPos(3),...
%                                         mti(ii).lat(jj), mti(ii).lon(jj),
mti(ii).alt(jj));
%         end
%         %handles = [handles figure];
%         %plot(y, x, 'x')
%         %axis equal, ylabel('x [m]'), xlabel('y [m]'), title('Trajectory');
%
%         %handles = [handles figure];
%         plot(normSerialDate(mti(ii).t), mti(ii).xvel, ...
%             normSerialDate(mti(ii).t), mti(ii).yvel);
%     end
%
%         %TCM5
%     if(~isempty(cmp_file))
%         cmp(ii) = parseTCMFile(fullfile(dirname{ii}, cmp_file.name));
%         cmp(ii).yaw = unwrap(cmp(ii).yaw - 180);
%         plot(normSerialDate(cmp(ii).t), cmp(ii).yaw);
%     end
%         %SSC
%     if(~isempty(ssc_file))
%         ssc(ii) = parseSscFile(fullfile(dirname{ii}, ssc_file.name));
%         %handles = [handles figure];
%         plot(normSerialDate(ssc(ii).t), 100*[ssc(ii).u ssc(ii).dr]);
%     end
%
%         %INI file.
%     if(~isempty(ini_file))
%         ini(ii) = parseIniFile(fullfile(dirname{ii}, ini_file.name));
%     end
%
% end
```

lla2ecef.m:

```
function R_ecef = lla2ecef(R_lla)

a = 6378137;            %Equatorial diameter.
b = 6356752.3142;       %Axial diameter.

f = (a-b)/a;            %Flattening ratio.
e_1 = 2*f - f^2;        %1st eccentricity

N = size(R_lla,1);
```

```
R_ecef = zeros(N,3);

for n = 1:N
    lambda = R_lla(n, 1);
    alpha  = R_lla(n, 2);
    h      = R_lla(n, 3);

    N = a/sqrt(1 - e_1*sind(lambda)^2);

    x = (N + h)*cosd(lambda)*cosd(alpha);

    y = (N + h)*cosd(lambda)*sind(alpha);

    z = (N*(1-e_1) + h)*sind(lambda);

    R_ecef(n,:) = [x y z];
End
```

lla2ned.m:

```
function [n,e,d,distance]=lla2ned(lat,lon,alt,lat2,lon2,alt2) %#codegen

% Converted from Thomas C. Furfaro - UVN HW7

T_ecef      =   lla2ecef([lat, lon, alt]); % Convert to ECEF.
T_ecef2     =   lla2ecef([lat2,lon2,alt2]);
T_ned       =   ecef2ned([T_ecef2 - T_ecef]', lat2, lon2);  % Convert to NED.
n           =   T_ned(1);            %Parse local x vector.
e           =   T_ned(2);            %Parse local y vector.
d           =   T_ned(3);            %Parse local z vector.
distance    =   sqrt(n^2+e^2+d^2);   %Spatial dif in ECEF coordinates
```

ned2bf.m:

```
function R_bf = ned2bf(R_ned, orientation)
%  NED2BF           Takes North-East-Down (NED) based vector and transforms
%                   it to the body fixed frame. The x,y,z Euler angles
%                   between the BF and NED frames is passed through
%                   orientation, whereas the column-vector to be
%                   transformed is passed as R_ned.

if(size(R_ned, 1) ~= 3)
    R_ned = R_ned';
end

if(size(orientation, 1) ~= 3)
    orientation = orientation';
end

R_bf = zeros(size(R_ned));

for jj = 1:size(R_ned,2)
    psi = orientation(1,jj);
    theta = orientation(2,jj);
    phi = orientation(3,jj);

    C_g2b = [   c(psi)*c(theta),...
                (-1*s(psi)*c(phi) + c(psi)*s(theta)*s(phi)),...
                (s(psi)*s(phi) + c(psi)*s(theta)*c(phi));
                c(theta)*s(psi),...
                (c(psi)*c(phi) + s(psi)*s(theta)*s(phi)),...
```

```
                (-1*c(psi)*s(phi) + s(psi)*s(theta)*c(phi));
                -1*s(theta),...
                c(theta)*s(phi),...
                c(theta)*c(phi)]';

    R_bf(:,jj) = C_g2b*R_ned(:,jj);
end
function out = c(in)
out = cos(in);

function out = s(in)
out = sin(in);
```

nixTime2matlabTime.m:

```
function matlabDateObject = nixTime2matlabTime(seconds)

nixOffset = datenum('01-Jan-1970');

% convert to a number, dived by number of seconds
% per day, and add offset
matlabDateObject = seconds/(24*60*60) + nixOffset;
```

normSerialDate.m:

```
function t = normSerialDate(serialDate)
%If the date is in matlab serial form, convert it to incrementing seconds.
t = (serialDate - serialDate(1))*24*60*60;
```

parseMtiFile.m:

```
function mti = parseMtiFile(filename)
raw_dat = dlmread(filename,'\t', 3, 0);
[Row Col] = size(raw_dat);
if(sum(raw_dat(Row,:) ~= 0 ) ~= Col)
    raw_dat = raw_dat(1:(Row - 1),:);
end

mti = struct('t', {zeros(Row,1)}, ...
             'pitch', [], ...
             'roll', [], ...
             'yaw', [],...
             'lat', [], ...
             'lon', [], ...
             'alt', [], ...
             'xvel', [], ...
             'yvel', [], ...
             'zvel', []);

mti(1).t    = nixTime2matlabTime(raw_dat(:,1));
mti.status  = raw_dat(:,2);
mti.roll    = raw_dat(:,3);
mti.pitch   = raw_dat(:,4);
mti.yaw     = unwrap(raw_dat(:,5));
mti.lat     = raw_dat(:,6);
mti.lon     = raw_dat(:,7);
mti.alt     = raw_dat(:,8);
mti.xvel    = raw_dat(:,9);
mti.yvel    = raw_dat(:,10);
mti.zvel    = raw_dat(:,11);
```

parseRMCFile.m:

```
function [t lat lon speed course stat] = parseRMCFile(filename)
    raw_dat = dlmread(filename,'\t');
    [Row Col] = size(raw_dat);
    if(sum(raw_dat(Row,:) ~= 0 ) ~= Col)
        raw_dat = raw_dat(1:(Row - 1),:);
    end
    dt = 0.2;
    t = raw_dat(:,1);
    t = t(1):dt:(t(1)+dt*(length(t)-1));
    lat = raw_dat(:,2);
    lon = raw_dat(:,3);
    stat = raw_dat(:,4);
    speed = raw_dat(:,5);
    course = raw_dat(:,6);
```

parseSscFile.m:

```
function ssc = parseSscFile(filename)
    %Between open loop and closed loop testing, I changed the format of the
    %SSC logfile to contains less useless information. Thus, the old and
    %new parsers have a hard time when looking at each other's data. This
    %just inspects the first line and makes a decision on which one to
    %call.

    %The time stamp for both is the same (*nix time plus the fractional
    %seconds).

    %The "old" format for the commands is 24 hex values, tab separated,
    % where they look like:
    %
    % 8 Channels of: 0xFF 0xYY 0xZZ
    %
    % 0xFF      - Start byte.
    % 0xYY      - Channel Number (0 - 11)
    % 0xZZ      - Value. (0x7F is "center", 0xFF is not a valid value)
    %
    % Sample Line: (wrapped to fit in the window)
    % 1319751191.600029 0xFF 0x07 0x7F 0xFF 0x01 0x7F 0xFF 0x02 0x7F 0xFF
    %    0x06 0x7F 0xFF 0x00 0x7F 0xFF 0x03 0x7F 0xFF 0x04 0x7F 0xFF 0x05 0x7F

    oldFmtStr = ['%f %*i %*i %*i %*i %*i %i %*i %*i %i %*i %*i %*i %*i' ...
                    '%*i %*i %*i %*i %*i %*i %*i %*i %*i %*i %*i %*i %*i \r\n'];

    %The "new" format is just two values, tab separated, that looks like:
    %
    % 1327855734.62921  P127    S127
    %

    newFmtStr = '%f P%i S%i\r\n';

    %Just to check which is which, we'll look for a "P" in the first line.
    fileId = fopen(filename);

    if(isempty(findstr('P', fgetl(fileId))))
        fmtStr = oldFmtStr;
    else
        fmtStr = newFmtStr;
    end

    output = fscanf(fileId, fmtStr, [3 inf]);
    fclose(fileId);
```

```
    Npts = size(output,2);

    %Make room for 6 channels of info, and a time vector.
    tNix    = output(1,:)';
    ssc(1).t       = nixTime2matlabTime(tNix);

    %Just because our signs are flipped.
    ssc.stbd = -1*(output(2,:)'-127)/127;
    ssc.port = -1*(output(3,:)'-127)/127;
    ssc.dr  = diff([ssc.stbd ssc.port], 1, 2);
    ssc.u   = mean([ssc.stbd ssc.port], 2);
```

vlength.m:

```
function l = vlength(R)
% VLENGTH Length of n-dimensional vector R, if basis is orthonormal.
% If R is a matrix, length of each row is taken.

l = sqrt(sum(R'.^2));
```

6.2.3 **Template mission INI file**

```
[SEG1]
type            = OPEN_START;
waitForGps      = 1; Wait for GPS before starting?
gpsPeriod       = 20; Time between GPS retries at init.
waitTime        = 5; How long to wait before starting (after GPS fix in waitForGPS = 1)

[SEG2]
type            = OPEN_STRAIGHT;
wait            = 0; How long to wait before move.
sscSpeed= 50; Set SSC speed (in percent max throttle).
duration        = 30; How long are we going straight for?

[SEG3]
type            = OPEN_CIRCLE;
wait            = 0; How long to wait before move.
sscSpeed        = 50; What the center velocity is before turn.
turnDelay       = 10; How long we go straight, before start turning.
rudder          = 30; Percent differential rudder to apply.
turnDir = PORT; Which direction the first turn is.
duration        = 30; How long we're turning for.

[SEG4]
type            = OPEN_ZIGZAG;
wait            = 0; How long to wait before we move.
sscSpeed        = 50; What the center velocity is before turn.
turnDelay       = 10; How long we go straight, before start turning.
rudder          = 30; Percent differential rudder to apply.
turnDir = PORT; Which direction the first turn is.
period          = 5; Duration of each cycle. (1/Hz = s = f)
turns           = 6; Number of turns (cycles);

[SEG5]
type            = CLOSED_SPEED;
wait            = 0; How long to wait before we move.
speedCmd        = 1.5; Command velocity [m/s]
```

```
dt                      = 0.2;  controller discrete dt
kp_speed        = 80;   Proportional speed gain.
ki_speed        = 20;   Integral speed gain.
kd_speed= -10;  Derivative speed gain.
duration        = 20;   Maximum run time.

[SEG6]
type            = CLOSED_HEADING;
wait            = 0; How long to wait before we move.
sscSpeed        = 40;   What the center velocity is before applying dr.
headingCmd      = 0; Command heading (degrees).
dt                      = 0.2;  controller discrete dt
kp_heading      = 80;   Proportional heading gain.
ki_heading      = 20;   Integral heading gain.
kd_heading      = -10;  Derivative heading gain.
duration        = 20;   Maximum run time.

[SEG7]
type            = CLOSED_WAYPOINT;
wait            = 0; How long to wait before we move.
latitude        = ; Target latitude in decimal degrees.
longitude       = ; Target longitude in decimal degrees.
speedCmd        = 0.5; Command velocity [m/s]
dt                      = 0.2;  controller discrete dt
kp_heading      = 80;   Proportional speed gain.
ki_heading      = 20;   Integral speed gain.
kd_heading      = -10;  Derivative speed gain.
kp_speed        = 80;   Proportional speed gain.
ki_speed        = 20;   Integral speed gain.
kd_speed= -10;  Derivative speed gain.
duration        = 20;   Maximum run time.
maxDistance = 5;Maximum acceptable distance to wpt, [m].

[SEG8]
type            = CLOSED_TRACK;
wait            = 0; How long to wait before we move.
latitude        = ; Target latitude in decimal degrees.
longitude       = ; Target longitude in decimal degrees.
speedCmd        = 1.5; Command velocity [m/s]
dt                      = 0.2;  controller discrete dt
kp_heading      = 80;   Proportional speed gain.
ki_heading      = 20;   Integral speed gain.
kd_heading      = -10;  Derivative speed gain.
kp_speed        = 80;   Proportional speed gain.
ki_speed        = 20;   Integral speed gain.
kd_speed= -10;  Derivative speed gain.
duration        = 20;   Maximum run time.
lookAhead       = 5;    Look ahead distance, [m].
maxDistance = 5;Maximum acceptable distance to wpt, [m].

[SEG9]
type            = NEUTRALtime           = 10; How long we're sitting idle for.
```

7 REFERENCES

[1] Volker Bertram, "Unmanned Surface Vehicles – A Survey," in *Skibsteknisk Selskab*, Copenhagen, Denmark, 2008.

[2] J.E. Manley, "Unmanned surface vehicles, 15 years of development," in *Proceedings of OCEANS'08*, Quebec City, QC, Canada, 2008, pp. 1-4.

[3] Stephanie Showalter and Justin Manley, "Legal and Engineering Challenges to Widespread Adoption of Unmanned Maritime Vehicles," in *OCEANS 2009*, Biloxi, MS, USA, 2009, pp. 1-5.

[4] Thor I. Fossen. (2010) Lecture Notes TTK 4190: Guidance and Control. PDF Slides.

[5] M. H. Tall, P. F. Rynne, J. M. Lorio, and K. D. von Ellenrieder, "Visual-Based Navigation of an Autonomous Tugboat," *MTS Journal*, vol. 44, no. 2, pp. 37-45, March/April 2010.

[6] Marine Advanced Research, Inc. (2011, October) WAM-V®: Marine Advanced Research, Inc. [Online]. http://wam-v.com/index.html

[7] M. H. Tall, P. F. Rynne, J. M. Lorio, and K. D. von Ellenrieder, "Visual-based Navigation of an Autonomous Tugboat," in *MTS/IEEE OCEANS 2009*, Biloxi, MS, USA, 2009.

[8] Paul Newman, "A mission oriented operating suite," MIT Department of Ocean Engineering, Cambridge, MA, Technical Report OE2007-07, 2003.

[9] Michael Benjamin, Henrik Schmidt, Paul Newman, and John Leonard, "Nested autonomy for unmanned marine vehicles with MOOS-IvP," *Journal of Field Robotics*, vol. 27, no. 6, pp. 834-875, 2010.

[10] Michael J. Matczynski, "A Distributed Embedded Software Architecture for Multiple Unmanned Aerial Vehicles," Massachusetts Institute of Technology, Cambridge, MA, Master's Thesis 2006.

[11] Andrew Y. Ng, Stephen Gould, Morgan Quigley, Ashutosh Saxena, and Eric Berger, "STAIR: Hardware and Software Architecture," in *AAAI 2007 Robotics Workshop*, Stanford, CA, USA, 2007.

[12] M. Quigley et al., "ROS: an open-source Robot Operating System," in *Proc. Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, 2009.

[13] Albert Huang, Edwin Olson, and David Moore, "Lightweight Communications and Marshalling for Low-Latency Interprocess Communication," Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, Technical Report MIT-CSAIL-TR-2009-041, 2009.

[14] A.S. Huang, E. Olson, and D.C. Moor, "LCM: Lightweight Communications and Marshalling," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Tapei, Taiwan, 2010, pp. 4057-4062.

[15] Brian S. Bingham, Jeffrey M. Walls, and Ryan M. Eustice, "Developemnt of a Flexible Command and Control Software Architecture for Marine Robotic Applications," *MTS Journal*, vol. 45, no. 3, pp. 25-36, May/June 2011.

[16] T. C. Furfaro, J. E. Dusek, and K.D. von Ellenrieder, "Design, construction, and initial testing of an autonomous surface vehicle for riverine and coastal reconnaissance," in *MTS/IEEE OCEANS 2009*, Biloxi, MS, USA, 2009, pp. 1-6.

[17] Matthew B. Greytak, "High Performance Path Following for Marine Vehicles Using Azimuthing Podded Propulsion," MIT, Cambridge, MA, USA, MS Thesis 2006.

[18] Serge Sutulo and C. Guedes Soares, "An algorithm for consistent linearization of ship manoeuvring mathematical models," in *Proceedings of the 7th IFAC CAMS*, Elaphusa, Croatia, 2007.

[19] Zhen Li, Jing Sun, and Soryeok Oh, "Design, analysis and experimental validation of a robust nonlinear path following controller for marine surface vessels," *Automatica*, vol. 45, no. 7, pp. 1649-1658, July 2009.

[20] W. Naeem, R. Sutton, and J. Chudley, "Soft Computing Design of a Linear Quadratic Guassian Controller for an Unmanned Surface Vehicle," in *14th Mediterranean Conference on Control and Automation*, Ancona, IT, 2006, pp. 1-6.

[21] Yan Peng and Jianda Han, "Tracking Control of Unmanned Trimaran Surface Vehicle: Using Adaptive Unscented Kalman Filter to Estimate the Uncertain Parameters," in *IEEE Conference on Robotics, Automation and Mechatronics*, Chegndu, China, 2008, pp. 901-906.

[22] Yan Peng, Bo Zhou, and Jianda Han, "Hardware Design and UKF-based Tracking Control Design of Unmanned Trimaran Surface Vehicle," in *Proceedings of ROBIO'07*, Sanya, China, 2007.

[23] Marine Advanced Research, Inc., "WAM-V USV Model 12-8," San Francisco, CA, USA, Key Specifications & Drawings 2009.

[24] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Phillippe Gerum, *Building Embedded Linux Systems*, Second Edition ed.: O'Reilly Media, 2008.

[25] Michael Opdenacker, Thomas Petazzoni, and Gilles Chanteperdrix. (2011, February) Free Electrons. [Online]. http://free-electrons.com/doc/embedded_linux_realtime.pdf

[26] Paul E. McKenney. (2005, August) Attempted summary of "RT patch acceptance" thread. Newsgroup: gmane.linux.kernel.

[27] Arther Siro, Carsten Emde, and Nicholas McGuire, "Assessment of the Realtime Preemption Patches (RT-Preempt) and their impact on the general purpose performance of the system," in *Ninth Real-Time Linux Workshop*, Linz, Austria, 2007.

[28] Morten Mossige, Pradyumna Sampath, and Rachana Rao, "Evaluation of Linux rt-preempt for embedded industrial devices for Automation and Power technologies - A case study," in *Ninth Real-Time Linux Workshop*, Linz, Austria, 2007.

[29] Thomas C. Furfaro, "Preliminary Development and Evaluation of a Linux-based Real-Time Operating System for Future Implementation in the Autonomous Mobile Periscope System," NSWCCD, Bethesda, MD, USA, Technical Paper 2009.

[30] Thomas C. Furfaro, "RTOS Linux Development Manual," NSWCCD, Bethesda, MD, USA, Technical Report 2009.

[31] Nicolas Devillard. (2011, December) iniParser: stand-alone ini parser library in ANSI C. [Online]. http://ndevilla.free.fr/iniparser

[32] Barnes W. McCormick, *Aerodynamics of V/STOL Flight*. Orlando, FL, USA: Academic Press, 1967, pp. 249-252.

[33] Karl von Ellenrieder and L.E.J. Ackermann, "Force/flow measurements on a low-speed, vectored-thruster propelled UUV," in *Proceedings of the Oceans'06*, Boston, MA, USA, 2006, pp. 1-6.

[34] PA Brandner and MR Renilson, "Iteraction between two closely spaced azimuthing thrusters," *The Journal of Ship Research*, vol. 42, no. 1, pp. 15-32, 1998.

[35] Dana R Yoerger, John G Cooke, and Jean-Jacques E Slotine, "The Influence of Thruster Dynamics on Underwater Vehicle Behavior and Their Incorporation Into Control System Design," *IEEE Journal of Oceanic Engineering*, vol. 15, no. 3, pp. 167-178, July 1990.

[36] Janine L. Mask, "System Identification Methodology for a Wave Adaptive Modular Unmanned Surface Vehicle," Florida Atlantic University, Dania Beach, FL, USA, Master's Thesis 2011.

[37] Michael S Triantafyllou and Franz S Hover. (2003, November) Maneuvering and Control of Marine Vehicles. MIT OpenCourseware Text.

[38] P. Krishnamurthy, F. Khorrami, and T. L. Ng, "Control Design for Unmanned Sea Surface Vehicles: Hardware-In-The-Loop Simulator and Experimental Results," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Dieg, CA, USA, 2007, pp. 3660-3665.

[39] Morten Breivik and Thor I. Fossen, "Path Following of Straight Lines and Circles for Marine Surface Vessels," in *Proceedings of the 6th IFAC CAMS*, Ancona, Italy, 2004, pp. 65-70.

[40] Morten Breivik and Thor I. Fossen, "Path Following for Marine Surface Vessels," in *Proceedings of the OTO'04*, Kobe, Japan, 2004, pp. 2282-2289.

[41] Gongxing Wu, Hanbin Sun, Jin Zou, and Lei Wan, "The basic motion control strategy for the water-jet-propelled USV," in *Proceedings of ICMA'09*, Changchun, China, 2009, pp. 611-616.

[42] Hashem Ashrfiuon, Kenneth R. Muske, and Lucas C. McNinch, "Review of Nonlinear Tracking and Setpoint Control Approaches for Autonomous Underactuated Marine Vehicles," in *Proceedings of ACC'10*, Baltimore, MD, USA, 2010, pp. 5203-5211.

[43] Lucas C. McNinch, Hashem Ashrafiuon, and Kenneth R. Muske, "Optimal Specification of Sliding Mode Control Parameters for Unmanned Surface Vessel Systems," in *Proceedings of the ACC'09*, St. Louis, MO, USA, 2009, pp. 2350-2355.

[44] Joao Almeida, Carlos Silvestre, and Antonio Pascoal, "Path-Following Control of Fully-Actuated Surface Vessel in the Presence of Ocean Currents," in *Proceedings of the 7th IFAC CAMS*, Elaphusa, Croatia, 2007.

[45] U.S. Navy, "The Navy Unmanned Surface Vehicle (USV) Master Plan," Department of the Navy, Washington DC, Technical Directive 2007.

[46] Keng Peng Tee and Shuzhi Sam Ge, "Control of Fully Actuated Ocean Surface Vessels Using a Class of Feedforward Approximators," *IEEE Transactions on Control Systems Technology*, vol. 14, no. 4, pp. 750-756, July 2006.

[47] M.W. Oppenheimer, D.B. Doman, and M.A. Bolender, "Control Allocation for Over-actuated Systems," in *Proceedings of MED'06,* Ancona, Italy, 2006, pp. 1-6.

[48] R.E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME -- Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35-45, 1960.

[49] Giovanni Indiveri, Alessandro Antonio Zizzari, and Valentina Giovanna Mazzotta, "Linear Path Following Guidance Control for Underactuated Ocean Vehicles," in *Proceedings of the 7th IFAC CAMS*, Elaphusa, Croatia, 2007.

[50] Gong-xing Wu, Jin Zou, Lei Wan, and Zai-bai Qin, "Design of the Intelligence Motion Control System for the High-speed USV," in *Proceedings of ICICTA '09*, Changsha, Hunan Province, China, 2009, pp. 50-53.

[51] M. S. Chislett and J. Støm-Tejsen, "Planar Motion Mechanism Tests and full-scale Steering and Maneuvering Predictions for a Mariner Class Vessel," Hydrodynamics Department, Hydro and Aerodynamics Laboratory, Lyngby, Denmark, Technical Report Hy-6, 1965.

[52] Thor I Fossen, *Guidance and Control of Ocean Vehicles*. Chichester, England: John Wiley & Sons, 2004.