Characterization parameters for a three degree of freedom mobile robot

Fitzgerald, Jessica L.

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/38929
CHARACTERIZATION PARAMETERS FOR A THREE DEGREE OF FREEDOM MOBILE ROBOT

by

Jessica L. Fitzgerald

December 2013

Thesis Advisor: Richard Harkins
Second Reader: Andres Larraza

Approved for public release; distribution is unlimited
CHARACTERIZATION PARAMETERS FOR A THREE DEGREE OF FREEDOM MOBILE ROBOT

Jessica L. Fitzgerald

Naval Postgraduate School
Monterey, CA 93943

Department of the Navy

Approved for public release; distribution is unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A

Control and Navigation logic was developed for a 3-Degree of Freedom Surf-Zone Robot to assist in the identification and characterization of platform parameters for use in the Shuey Dynamic Model. These parameters included, primarily platform rotational inertia and wheel slip. Data was collected in various track scenarios including benign flat terrain and more complicated beach runs. Track lengths spanned short straight paths of no more than 10 meters to full-run point-to-point autonomous navigation paths of up to 80 meters. The longer runs included turns of up to 180 degrees and terrain inclines of 2 degree or less. As expected the Shuey model proved reliable for short runs of no more than 10 meters. For long length runs in the beach environment the Dynamic Model diverged quickly. This is attributed to, primarily, wheel slip conditions and the fact that the Shuey Model is open loop. Motor current was monitored under load conditions to identify wheel slip and simple algorithms were implemented to account for this with little success. However, closed loop heading input resulted in significant improvement to the model.

Robots, Robotics, Amphibious Vehicles, Mobility, Surf-zone, Dynamic Model, Kinematic Model, Simulation, Lagrangian Mechanics, Slip, Skid, Autonomous
CHARACTERIZATION PARAMETERS FOR A THREE DEGREE OF FREEDOM MOBILE ROBOT

Jessica L. Fitzgerald
Lieutenant, United States Navy
B.S, Systems Engineering, United States Naval Academy, 2009

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN APPLIED PHYSICS

from the

NAVAL POSTGRADUATE SCHOOL
December 2013

Author: Jessica L. Fitzgerald

Approved by: Richard Harkins
Thesis Advisor

Andres Larraza
Second Reader

Andres Larraza
Chair, Department of Combat Systems Science
Control and Navigation logic was developed for a 3-Degree of Freedom Surf-Zone Robot to assist in the identification and characterization of platform parameters for use in the Shuey Dynamic Model. These parameters included, primarily platform rotational inertia and wheel slip. Data was collected in various track scenarios including benign flat terrain and more complicated beach runs. Track lengths spanned short straight paths of no more than 10 meters to full-run point-to-point autonomous navigation paths of up to 80 meters. The longer runs included turns of up to 180 degrees and terrain inclines of 2 degree or less. As expected the Shuey model proved reliable for short runs of no more than 10 meters. For long length runs in the beach environment the Dynamic Model diverged quickly. This is attributed to, primarily, wheel slip conditions and the fact that the Shuey Model is open loop. Motor current was monitored under load conditions to identify wheel slip and simple algorithms were implemented to account for this with little success. However, closed loop heading input resulted in significant improvement to the model.
# Table of Contents

1 Introduction 1
   1.1 The Surf-Zone Project ........................................... 1
   1.2 Objectives .......................................................... 2

2 The Robot 3
   2.1 Robot Components .................................................. 3

3 Algorithm and Control 7
   3.1 Navigation ........................................................... 7
   3.2 Data Extraction ...................................................... 9

4 Theory 15
   4.1 Shuey Model .......................................................... 15

5 Experiment and Results 19
   5.1 Control Coefficient Experiments ................................. 19
   5.2 Evaluation of Shuey Model Using DARc and Beach Environment ........................................ 20

6 Conclusion 27
   6.1 Shuey Model Assesment ............................................. 27
   6.2 Future Work .......................................................... 27

Appendices 29

A Control and Navigation Code 29
List of Figures

Figure 1.1 Shuey Model Results ........................................... 2
Figure 2.1 DARc Structure .................................................. 4
Figure 2.2 Wheg Design ....................................................... 4
Figure 2.3 View Inside Pelican Case ...................................... 5
Figure 2.4 External Sensors ................................................... 6
Figure 3.1 Control Code Flow Diagram ................................... 8
Figure 3.2 Costatemets Example .......................................... 11
Figure 3.3 Distance and Heading Calculation from Dynamic C Code ........ 12
Figure 3.4 Distance and Heading Example ............................. 13
Figure 3.5 PID Feedback Loop ............................................. 14
Figure 5.1 Underdamped Proportional Control ......................... 19
Figure 5.2 Experimental Platform and Testing Environment Comparison ........ 21
Figure 5.3 Beach Run Between Two Way Points ....................... 22
Figure 5.4 Beach Run Between Three Way Points ...................... 23
Figure 5.5 Difference Between Motor Currents ....................... 24
Figure 5.6 Slip Condition Flow Chart ................................... 24
Figure 5.7 Slip Identification ............................................... 25
Figure 5.8 Slip Reduction Results ....................................... 26
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1</td>
<td>Garmin Data String Example</td>
<td>9</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Compass Data String Example</td>
<td>9</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>DAR-C Dimensions</td>
<td>17</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Ziegler-Nichols Parameters</td>
<td>20</td>
</tr>
</tbody>
</table>
List of Acronyms and Abbreviations

DARc  Durable Autonomous Robotic Crustacean
NMEA  National Marine Electronics Association
NPS  Naval Postgraduate School
PID  Proportional Integral Derivative
Acknowledgements

I would like to thank my friends and family who have supported me through my time at the Naval Postgraduate School and fueled me in my off hours. I was extremely blessed to be placed in the Physics department where I found supportive classmates and inspirational professors.

To my thesis advisor, thank you for your unmatched encouragement, guidance, and patience. To Steve Jacobs, thank you for your dedication, support and all the time you put into helping me.
CHAPTER 1: Introduction

The crucial role unmanned vehicles play in both civilian and military fields is becoming difficult to dispute. These platforms are appealing because of their amplified on station time, sensor capabilities and the removal of human risk. Extensive work and successful developments have been made in both unmanned land and sea platforms, however, little work has been done to develop a robot to operate in and transition between the two environments. The combination of these two domains is referred to as the littoral region, or Surf-zone, which encompasses but is not limited to missions like surveillance, environmental monitoring, intelligence, reconnaissance, and mine clearance.

This region has not been extensively explored or well developed for good reason. The Surf-zone requires a platform able to adapt to a variety of environmental and dynamic factors which are dramatically different between platforms operating exclusively on land or in the sea. Several prototype mechanical platforms have been developed utilizing biologically inspired locomotion to traverse the Surf-zone.

1.1 The Surf-Zone Project

Prior work on the surf-zone project included a negatively buoyant Foster Miller Lemming tracked platform [1] and several Wheg™ based platforms, AGBOT [2] and MONTe [3].

Collaboration with Case Western produced multiple prototypes and programs in attempts to meet the demands of this complicated dual environment [4]. Details and a more comprehensive history of the Surf-zone collaboration can be found in the Shuey Thesis. Shuey focused on creating a model for the movement of a four wheeled surf-zone robot [5].

The sole input to this open loop model was time stamped motor shaft encoder data. Encoder data sent into the model produced time stamped position $X, Y$ and heading $\phi$ as output seen in Figure1.1. The figure shows three tracks. Red is ground truth as identified by VICON data. Green is the kinematic physical model and blue is the dynamical model result.
1.2 Objectives

This thesis supports Surf-zone research by creating a new platform and employing it to test the validity of the Shuey model [5]. We address the following questions:

- Is there a mobile and autonomous platform that can be modeled accurately to predict performance and thus inform the design process?
- What are the key considerations for autonomous control design?

The scope of research and model design was limited to the beach, defined as the waterline and inland for a distance of 100 meters. Operation in the waterborne environment is excluded. The robot was modeled with three degrees of freedom: two perpendicular direction coordinates and one orientation angle.
CHAPTER 2:  
The Robot

The Surf-zone robot is designed to be durable, portable, multi-mission capable and easily manufactured. Our design, similar to previous Surf-zone projects, takes ideas and concepts from nature to develop solutions to the mobility problem. The design of our platform, a Durable Autonomous Robotic Crustacean (DARc), was inspired by a crab’s mobility and will later include a lobster like tail for stability on the beach.

2.1 Robot Components
DARc is designed to be modular as discussed in the following sections.

2.1.1 Platform
The skeleton, in Figure 2.1a holds a pelican case, Figure 2.1b, containing mission specific sensors, motors, batteries and micro-controller, that will be discussed in Section 2.1.3. This allows for quick transition between operations requiring dissimilar sensors or programs. Imagine switching between a surveillance mission and a mine hunting assignment. Each utilizes different sensor and communication suites. Also, each has a different likelihood of return. DARc’s adaptability allows unnecessary but valuable components to be removed when there is a higher mission risk.

The Whegs, as seen in Figure 2.2, were produced using a three-dimensional printer. This fabrication technique contributes to the ease and speed of manufacturing, especially in isolated regions. A unit in the field need simply hit the print button and assemble the Wheg. DARc’s design will later include a lobster tail similar to the one created for the 2011 Slatt thesis [6].

2.1.2 Hardware
For autonomy we used a BL2600 single board computer, which incorporates a Rabbit 3000 microprocessor, digital IO, flash and RAM memory and both RS-232 and RS485 serial ports as seen if Figure 2.3. These features allow smooth integration with the sensors discussed in Section 2.1.3 and the data storage discussed in Section 3.2.
Figure 2.1: Hardware components are enclosed in the pelican case (Figure 2.1b) to protect them from the elements. The pelican case is then placed in the Skeleton (Figure 2.1a). The case holds a single board computer, power, and sensors. The skeleton contains the essential components not expected to change between missions. This includes the motor-controllers, the Whegs, and the aluminum structure that houses the pelican case.

Figure 2.2: DARc Wheg design printed by a three-dimensional printer.

2.1.3 Sensors
The Garmin GPS unit (Figure 2.4b), is used to assist in way point navigation. It is capable of tracking up to twelve satellites and has differential GPS capability providing accuracy within three meters [7]. The design is well suited for DARc because it is waterproof and rugged. Position data is provided via an RS232 connection in a GPS Fix Data Format string, including but not limited to, time of fix, position details, fix accuracy and number of satellites used for the fix.

To measure heading, we used a Honeywell HMR3000 Magnetic Compass seen in Figure 2.4a. This selection was made because of the Honeywell’s durability, accuracy, availability and ease of use. Parsing of both the GPS and heading strings are discussed in Section 3.1.2.
(a) Upper Level Components

(b) Lower Level Components

Figure 2.3: The pelican case houses two levels of components as identified above.
Figure 2.4: DARc employed two external sensors; a Honeywell magnetic compass, Figure 2.4a, and a Garmin GPS 18, Figure 2.4b.
Before modeling and performance assessments could be conducted, we needed to create basic control and navigation code and achieve autonomy. Dynamic C is a non-standard programming language designed to operate the Rabbit micro-controller and has been extensively used in each of the previous Naval Postgraduate School (NPS) surf zone projects.

3.1 Navigation

Figure 3.1 is a flowchart of DARc’s basic costatements and functions. Costatements, or costates, are Dynamic C additions to the C language which simplify cooperative multitasking. Costates are usually nested in a while loop in order of priority. The program continues to cycle through them as illustrated in Figure 3.2. Each costate may contain a "pause" where the program must wait for an event, a specified passage of time or a set of information. During a pause, the program will leave the costate and continue to the next costate in the loop and return to that position when the necessary event has occurred.

Within each costatement is an ordered list of tasks or functions grouped together in support of a program operation like the navigation costate in Figure 3.1. The functions take in data from outside sensors and calculate the required parameters including but not limited to position, heading, desired heading and distance. Priority for each statement is driven by operational requirements. Before entering the loop of costates, the program initializes the board, serial ports, and creates a file for data storage as seen in Appendix A.

3.1.1 Sensor Data

Data from the compass and GPS are sent to the BL2600 through the F and E serial ports. Each produces a data string that uses the format from the National Marine Electronics Association (NMEA), a US organization that sets communications standards for marine electronics. An example of each string with the respective explanations are listed in Tables 3.1 and 3.2.

String parsing for both the GPS and Heading functions is very similar. In each case characters are individually evaluated based on the NMEA standards. This information, attained in the compass and GPS costates, is then sent to the navigation costate where desired heading and distance are calculated.
3.1.2 Position and Heading Calculations

Compared to the radius of the earth, the distances between our way points were relatively close together. This allowed for a flat earth approximation. The navigation function, found in Figure 3.1, is where distance and heading are calculated. First, we converted the GPS coordinate into Cartesian space where West and South were treated as the negative parts of the X and Y plane shown in Figure 3.4. Equation 3.1 calculates the distance, \( d \), between two way points using the distances in the x and y planes. Equation 3.2 calculates the heading, \( h \), to the desired way point. Figure 3.4 visualizes the geometry and assumptions made for these calculations.

\[
    d = \sqrt{\Delta x^2 + \Delta y^2} \tag{3.1}
\]

\[
    h = 180^\circ - \arctan\left(\frac{\Delta y}{\Delta x}\right) \tag{3.2}
\]
### Table 3.1: The NMEA GGA string format meaning for GARMIN 18 using the example string \'$GPGGA,123456,3635.7058,N,12152.4939,W,2,5\'.

<table>
<thead>
<tr>
<th>String</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GGA</td>
<td>Global Positioning System Fix Data</td>
</tr>
<tr>
<td>123456</td>
<td>Fix taken at 12:34:56 UTC</td>
</tr>
<tr>
<td>3635.7058,N</td>
<td>Latitude 36 deg 35.7058’ N</td>
</tr>
<tr>
<td>12152.4939,W</td>
<td>Longitude 121 deg 52.4939’ E</td>
</tr>
<tr>
<td>2</td>
<td>Fix quality: 0 = invalid</td>
</tr>
<tr>
<td></td>
<td>1 = Non Differential fix</td>
</tr>
<tr>
<td></td>
<td>2 = Differential fix</td>
</tr>
<tr>
<td>5</td>
<td>Number of satellites being tracked</td>
</tr>
</tbody>
</table>

### Table 3.2: Honeywell Compass format using the example string \'$PNTNTHPR,301\'.

<table>
<thead>
<tr>
<th>String</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>301°True</td>
</tr>
</tbody>
</table>

Figure 3.3 is a sample of the navigation code for calculating distance and heading. Latitude and Longitude are converted to Cartesian coordinates and then used to determine distance and heading to the next way point.

#### 3.1.3 PID Control

While a Wheg design is excellent for traversing in the surf zone environment [6], the shape tends to exacerbate the need for course correction. In order to steer a steady course we implemented proportional-integral-derivative(PID) control. Figure 3.5 is a diagram of this feedback loop and Equation 3.1.3 is the formula which calculates the signal, \(u(t)\), sent to the plant. \(K_p\), \(T_i\) and \(T_d\) are the proportional, integral and derivative gain values, \(e(t)\) is the error as a function of time, and \(s\) is time in seconds. The experiment to obtain the compensator values is discussed in Chapter 5.

\[
u(t) = K_p (1 + \frac{1}{T_i s} - T_d s) e(t) \tag{3.3}\]

#### 3.2 Data Extraction

Once able to drive a reliable path, our focus turned to pulling information for later use. Data Extraction can be conducted in a variety of ways. Previous thesis work kept the platform tethered to a laptop computer and extracted data by printing directly to the screen. This method had two
disadvantages. First, tethering proves difficult when attempting to make visual observations, prevent cables from getting caught up in the wheels, and executing robust paths. Secondly, the print statements necessary for the tether system slows the frequency of data steps and reduces model accuracy. A wireless connection is also a viable solution, however, more coding and components would be needed for the platform.

Instead of a tether or using a wireless connection, we took advantage of the file library in dynamic C and developed code to extract and store data which could later be retrieved when the robot was connected again in the dry and stationary lab environment. The developed extraction code is found in Appendix B. For each step we recorded the time, encoder counts, current readings for each motor, latitude, longitude and the heading. After extraction, analysis was conducted, as will be discussed in Chapter 5.
Figure 3.2: Here are three iterations of an example program with costates. The first goes through each statement completely and returns to the top of the program. In the second loop, the program finds a pause partway through statement one and continues to the next costate. On the third loop, the program goes back to the pause point, finishes costate 1 from that point on and then continues to costate 2.
Figure 3.3: A sample from my Dynamic C code found in appendix A. Latitude and Longitude are converted to Cartesian coordinates and then used to determine distance and heading to the next way point.

```c
current_lat = (double)current_pos.lat_degrees + (current_pos.lat_minutes / 60.0);
if (current_pos.lat_direction == 'S')
    current_lat = 0 - current_lat; //negative values in Southern hemisphere

current_lon = (double)current_pos.lon_degrees + (current_pos.lon_minutes / 60.0);
if (current_pos.lon_direction == 'W')
    current_lon = 0 - current_lon; //negative values for West hemisphere

// Desired Location
wypt_lat = desired_WP.lat;
wypt_lon = desired_WP.lon;
err_distance = distance(current_lon, current_lat, wypt_lon, wypt_lat); //Error distance between goal and starting position
desired_heading = (180.0/PI)*normalize2PI(M_PI/2 - headingTo(current_lon, current_lat, wypt_lon, wypt_lat));
```
Figure 3.4: Distance and heading are calculated using cartesian coordinates and basic geometry principles as shown here. $\Delta x$ is the change in latitude and $\Delta y$ is the change in longitude from one way point to another. The heading and distance are continually calculated which influences motor-controller speeds. DAR-C operates in the Northern hemisphere therefore true desired heading is obtained by subtracting the cartesian heading from 180°. In this figure the true heading is roughly $180° - (-45°) = 225°$. 
Figure 3.5: PID feedback loop: A desired heading is sent to the controller that calculates error by comparing it to the desired heading. The lower illustration details the calculations of the compensator.
CHAPTER 4:
Theory

The methods we used for developing a robotic simulation centered on fundamental physics theories including energy calculations, inertia tensors, traditional mechanics and Lagrangian mechanics. The first set of calculations and programming originates from the Shuey Thesis. This chapter will briefly explain that work.

4.1 Shuey Model
The Shuey model is composed of two parts; a kinematic model and a dynamic model. The first uses basic wheel rotation and movement principals while the second uses Lagrangian mechanics to develop a more complex and accurate model. For each case, the following assumptions were made:

- The platform is modeled as a two wheeled differentially steered robot rather than a four wheel skid-steered robot.
  - no lateral slippage
- The robot operates with three degrees of freedom(two for position and one for orientation).
  - no changes in elevation

4.1.1 Kinematic Model
The first model is a simple kinematic description of the robot’s movement. The dead reckoning portion calculates the average velocity of the two wheels, $\dot{\phi}$, using encoder counts, the times step and wheel radius. The model iterates at such a slight time change that at each step we are able to get a very small account of forward and angular velocity. Equation 4.1 determines forward speed, $\dot{x}$, of the robot in it’s own frame of reference, where $r$ is the radius of the wheel and $\dot{\phi}$ is the wheel velocity for the left or right wheel.

$$\dot{x} = r \frac{\dot{\phi}_L + \dot{\phi}_R}{2},$$

Equations 4.2, 4.3, and 4.4 are the final results from the Shuey kinematic model where $\Delta \theta$ is the platform’s change in heading, $\Delta W$ is the change in number of revolutions for a right or left
wheel, and \( d \) is the axle length. The change in heading, \( \Delta \theta \) is summed to compute the heading at each time step. The last equation provides the platform’s position in the global frame, \( X_W \) and \( Y_W \).

\[
\Delta \theta = 2\pi r \frac{\Delta W_L - \Delta W_R}{d}.
\] \hfill (4.2)

\[
\Delta X_W = \pi r \cos(\theta)(\Delta W_L + \Delta W_R)
\] \hfill (4.3)

\[
\Delta Y_W = \pi r \sin(\theta)(\Delta W_L + \Delta W_R).
\] \hfill (4.4)

While the kinematic model is simple to create and understand, it does not accurately account for acting forces. A more complex set of equations is required to more precisely capture motion.

### 4.1.2 Dynamic Model

The dynamic equations, developed using Lagrangian mechanics, yielded Equation 4.5 then expanded in Equation 4.6. A new force, torque and

\[
L = T_B + T_{wL} + T_{wR} - U,
\] \hfill (4.5)

\[
L = \frac{1}{2} mx^2 + \frac{1}{2} my^2 + \frac{1}{2} I_R \dot{\theta}^2
\] \hfill (4.6)

Next, using the Euler-Lagrange equation and two orders of integration the equations of motion were produced, found in Equations 4.7, 4.8, and 4.9. All three equations require current velocities, forces and torque. These values change throughout the run, however, for simplicity the model treats them as a constant at each time step. At the next time step, new velocity, force and torque values are calculated with equations 4.1, 4.11 and 4.12 then to the equations of motion.

\[
x(t) = \frac{1}{2} \frac{F_x}{m} t^2 + x_0t + x_0
\] \hfill (4.7)

\[
y(t) = \frac{1}{2} \frac{F_y}{m} t^2 + y_0t + y_0
\] \hfill (4.8)
\[ \theta(t) = \frac{1}{2} \frac{\tau}{I_R} t^2 + \dot{\theta}_0 t + \theta_0 \]  

(4.9)

The above equations are also governed by constants including mass, dimensions and inertia. Table ??; DARc’s dimensions which contributed to the inertia, force, torque and position equations. We calculated body inertia as 1.93\text{kg} \cdot \text{m}^2 and wheel inertia as 0.101\text{kg} \cdot \text{m}^2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (front to back)</td>
<td>0.445 m</td>
</tr>
<tr>
<td>Width (side to side)</td>
<td>0.458 m</td>
</tr>
<tr>
<td>Height</td>
<td>0.188 m</td>
</tr>
<tr>
<td>Wheel Separation</td>
<td>0.285 m</td>
</tr>
<tr>
<td>Wheel to COM separation</td>
<td>0.316 m</td>
</tr>
<tr>
<td>Wheel Radius Outer/Inner</td>
<td>0.127 m .085 cm</td>
</tr>
<tr>
<td>Wheel Mass</td>
<td>0.7055 kg</td>
</tr>
<tr>
<td>Body Mass</td>
<td>13.542 kg</td>
</tr>
</tbody>
</table>

Table 4.1: DAR-C Dimensions

The force on each wheel, Equation 4.10 is calculated from the wheel’s angular velocity, inertia and density, \( \rho \). These are then combined to find forward force, \( F_x \), and torque, \( \tau \).

\[
F_L = \frac{\omega_{L} I_{w}}{\rho} \quad F_R = \frac{\omega_{R} I_{w}}{\rho}
\]

\[
F_x = F_L + F_R
\]

\[
\tau = \frac{b}{2} (F_R - F_L)
\]

(4.10)

(4.11)

(4.12)

The Shuey model assumes no slip, therefore, lateral force, \( F_y \), is considered zero. These calculations are then submitted to MATLAB’s ode45 solver to produce position and orientation in the world frame.
CHAPTER 5:  
Experiment and Results  

There were two experimental portions for this thesis. First, we designed a test to obtain control coefficients. Second, data was collected by navigating between waypoints. The information was then used to test the Shuey model.

5.1 Control Coefficient Experiments

As discussed in Section 3.1.3, plant signals were determined using a PID controller. We employed the Ziegler-Nichols method to calculate each gain value. First we implemented a controller with only proportional gain ($K_p$). We then increased the value for $K_p$ until we achieved steady oscillation as seen in Figure 5.1. This value became our critical gain, $K_0$, and the oscillation period for this run became $T_0$.

![Figure 5.1: Here, the robot was ordered to steer a steady course of 190° using proportional control where $K_p = 1.45$. Average frequency is 4.6 seconds. We used these two values to calculate our PID gain values by the Ziegler-Nichols method.](image)
These values were then used to calculate PID gains with the Ziegler-Nichols equations found in Table 5.1. Equation 5.1 is the final PID controller equation with the calculated gains from the experiment and the equations in Table 5.1.

<table>
<thead>
<tr>
<th>Controller</th>
<th>$K_p$</th>
<th>$T_i$</th>
<th>$T_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>$K_0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PI</td>
<td>$K_0$</td>
<td>$\frac{1}{1.2T_0}$</td>
<td></td>
</tr>
<tr>
<td>PID</td>
<td>$K_0$</td>
<td>$T_0$</td>
<td>$\frac{T_0}{8}$</td>
</tr>
</tbody>
</table>

Table 5.1: Ziegler-Nichols Parameters

\[ u(t) = .85\left(1 + \frac{1}{2.5s} + .63s\right)e(t) \] (5.1)

5.2 Evaluation of Shuey Model Using DARc and Beach Environment

After constructing DARc and developing reliable control we were able to test the Shuey model with the new platform. There are several important differences between the Shuey tests and ours. The first testing environment was in a controlled lab space with a smooth, flat, cement surface where the second was in the beach environment with diverse surface composition and varied elevation as seen in Figure 5.2.

The platforms were also significantly dissimilar with the first, Sandie, having round rubber wheels and employing deterministic path execution. Contrarily, DARc employed polycarbon WHEGS and autonomous navigation.

5.2.1 Initial Results

The experiment was conducted at the Del Monte Beach in Monterey California on terrain reaching inclines up to two degrees and terrain type ranging from hard packed to medium packed sand. DARc was ordered to conduct navigation between way points while simultaneously stor-
(a) DARc at Del Monte Beach in Monterey, California.

(b) Sandie at the NPS Vicon laboratory.

Figure 5.2: Above are the two trial platforms in their respective testing environments. DARc, Figure 5.2a, is on the beach terrain comprised of varying surface composition and inclines. Sandie, Figure 5.2b, is observed from above in the indoor NPS VICON laboratory.

Experimenting in RAM for later extraction. Figure 5.3 shows the results from one of our beach runs. Note that the dynamic model predicts position well within the first ten to twenty meters, however, drifts significantly to the right roughly half way through the run. It was observed that during this test the left wheel experienced slip. The model assumes a no slip environment, as it has no way to account for loss of traction, thus while the left wheel was spinning the model interpreted this as a right turn.

All runs conducted yielded similar results and were unable to account for slip resulting in significant deviation from platform’s true heading and notable error in final estimated position. Figure 5.4 is a beach run where the platform navigated between three way points. Significant slip was experienced resulting in a poor model estimate of platform location and orientation.

5.2.2 Accounting for Slip

Original testing of the Shuey model was conducted in an environment where a no slip assumption had much less effect on model accuracy. The change in platform type, going from a round wheel to a WHEG, and the varying and rugged terrain made this assumption impossible when looking to accurately model DARc’s motion.
Current and Slip
To identify slip we took a closer look at the motor currents [8]. We first identified time steps with large differences in current between the two motors as seen in Figure 5.5. Next, these time steps were evaluated such that wheels experiencing low current were identified as being in a slip condition, as seen if Figure 5.7. The slip threshold for both the current difference and each individual current was determined using a best fit method for the multiple beach runs. Assumed speed for identified slip time steps was reduced for the slipping wheel. Figure 5.8 compares the original model to the modified slip reduction model for two runs. The calculated orientation and position is improved, however, this modification did not completely correct the model.

Heading Modification
Next we employed real data and closed the dynamic model loop by periodically updating the heading. Results from the modification are found in Figure 5.9. These results encourage future work to focus on preventing significant orientation changes during slip conditions.
Figure 5.4: This run was conducted at the beach where DARc navigated between three way points. Halfway to the second waypoint the platform experienced left wheel slip. The Shuey model interpreted this as a right turn which caused significant error in the model orientation.
Figure 5.5: This graph identifies large differences between motor currents. We used a best fit method for multiple beach runs to determine the threshold for large current differences indicating slip. Time steps where the current difference was above the threshold is then examined more closely for low currents to determine slip (as illustrated in the decision chart in Figure 5.6).

Figure 5.6: Decision chart for identifying slip: The thresholds for determining a high current difference and a motor with low current was determined using a best fit method.
Figure 5.7: The run shown in Figure 5.3 was evaluated for time steps experiencing slip using the decision chart in Figure 5.6. Red lines indicate a time step where the left wheel experienced slip and green lines indicate the same for the right wheel. Velocities for time steps and wheels experiencing slip were then reduced.
Figure 5.8: Two runs assessed using the original Shuey model, left, and the new slip reduction model, right.

Figure 5.9: Two runs assessed using the original Shuey model, left, and the closed loop model, right.
CHAPTER 6: Conclusion

While the Shuey model produced a fairly accurate account of the robot’s real world position, ignoring skid and slip prevented the model from precisely calculating heading and position.

6.1 Shuey Model Assessment

The original Shuey model is best suited for situations where little to no slip exists. This would include platforms with round wheels, especially truly deferentially steered platforms. Also, environments with flat, smooth surfaces and no elevation change produce more accurate model results. The surf zone platforms and the environment they are intended for meet neither of these two requirements, thus significant work needs to be conducted to account for slip and skid.

6.2 Future Work

Future thesis work should explore modifying the model to reflect the platform steering method. Skid-steering, as opposed to differential steering assumed in the model, is a more accurate way to describe our platform.

6.2.1 Skid and Differentially Steered

Identifying the mode for steering is a crucial first step in creating an accurate model. Ackerman steering, skid steering and differential steering are three very common methods used in wheeled robotics. Most of the surf zone platforms have employed skid steering. The only exception is AGBOT which used ackeman steering, a technique where wheels are able to rotate relative the robot body. The easiest example of a differentially steered platform to visualize is the wheel chair. The assumptions from the Shuey thesis allowed for this model to come very close to describing the robot path accurately. Specifically, because Shuey assumed no slip and no lateral force, the model produced was very close. Skid steering is a more accurate way to describe our platform. Visualize a tank. Unlike your car, to turn the wheels must either slip or skid as the heading changes. Moving from a concrete to a gravel or sand environment dramatically increases the need to account for slip and friction. While differential steering is more accurate for less slip prone environments like concrete, for loose gravel, like sand, where slipping is very common we need to account for slippage. The surf zone is a slip prone sand environment and
thus sliding and skidding must be taken into account by more accurately modeling the steering method.

The relation of slip and current should be examined more thoroughly. Testing should be conducted comparing ordered speed, expected speed, actual speed, and current.

While autonomous platforms have been developed and increasingly utilized in exclusively land, sea or air environments, there has yet to be a defense design employed in the littoral region. Proficiency in the surf zone environment is crucial to the Navy’s ability to transition between land and sea while maintaining command and control in both regions. This platform would be excellent for mine clearance and detection or removal of other integrated defenses. Having an integrated sensor suite would be invaluable in mission planning and intelligence gathering. In order to reap the benefits that autonomous platforms have yielded in other environments it is essential that accurate models and simulations be developed and perfected to inform the design process and accurately predict performance.
APPENDIX A:
Control and Navigation Code

/* Fitzgerald_Thesis_2013.c
* Programmers: Jessica L. Fitzgerald
* Programs Referenced/Used: Steve Jacob's '2 SERVO POSITION & AD VALUE .C'
* and PC4015 Winter 2012's BENDER2012V120.C
* Supervisor: Prof Harkins
*
* Program Description: This program consists of a number of functions followed
* by the main() file. Thus far the following have been implemented:
* 1. Desired speeds, accelerations and predetermined paths can be sent to
* and executed by the Rabbit.
* 2. Data is recorded to the Rabbit's RAM.
*   a. Data can be extracted using either:
*      i. PRINTFILE .C (preferred)
*      ii. DUMPFILE .C
*   b. Data Recorded includes time(mS), encoder count, current
*      for both motors, heading(from compass), Latitude, and
*      Longitude(from GPS).
* 3. Platform navigates using sensors:
*   a. Steer a course using drive_Heading() (employs PID control)
*   b. Drive between Waypoints (uses drive_heading() function)
* 4. Program currently designed to run between 3 waypoints on the Del
*   Monte Beach.
* */

/////////////////////////////////////////////////////////////////////
// File and basic commands declarations
/////////////////////////////////////////////////////////////////////
#define TESTFILE 1
#define TESTFILE2 2
#define PIC_STEP
#define PIC_SERVO
#define CINBUFSIZE 511
#define COUTBUFSIZE 511
#define nmc.lib
#define DEBUG_PRINT 0 // 1 to print, 0 for NO print
#define DATA_PRINT 0 // 1 to print, 0 for NO print
#define Velocity 30000
#define Acceleration 100
#define sixteenFootTicks 27500
#define oneRev 1961.5
#define turnRev 3675
#define WAIT 2 // should be dependant on speed and operating method
#define ms2wait 12000 // 6000 = 3 12000 = 6 18000 = 9 24000 = 12
#define pi 3.1416
#define sideOfBox oneRev *2 // 3 then 6 revs 9, 12 / 2013 1-24

int iterator, inter, sit, alt2, M1AD, M2AD, successFlag;
char ch;
long t0, timer, leftPosition, rightPosition, averagePosition, distanceTilNext;

29
long spiralIn, spiralOut;
long pivotTicks;
long information[5];
long leftVelocity, rightVelocity;

static char buffer1[10];

// Initialization Methods
void robotInit(void);
void read2stats(void);
void setServoParameters(void);

// Basic Motor Control Methods
void setLeftSpeed(long vel, long accel);
void setRightSpeed(long vel, long accel);

// Advanced Motor Control Methods
void goStraight(long vel, long accel, int dir);
void carTurn(long vel, long accel, int dir);
void stop(void);
void runLine(File *file, int baseSpeed);
void runCircle(File *file, int baseSpeed);
void runSpiral(void);
void makePivotCCW(void);

// Logging Methods
void logInformation(long distanceTilNext);
void logData(File *f);

// Compass declarations
#define MAX_SENTENCE 100
#define FINBUFFSIZE 255
#define FOUTBUFFSIZE 255
#define COMPASS_DELAY 10
#define READDELAY 15
#define COMPASS_WAIT_TIME 1
#define EXT_SENSORS 1
#define HEADING_PRINT 0 // 1 to print, 0 for NO print

int compass(char sentence[MAX_SENTENCE]);
void msDelay(long sd);

void GetCompassString(char *RetCompassStr);

const char init_str[] = "#BAD=11*7A\r\n\t"; //5 times per second
float curr_hdg;
char compass_sentence[MAX_SENTENCE];
int compass_error;

unsigned long compass_wait_time;
const int compass_timeout = 1;

int Compass_update;
//Montery is East 14 so add 14 to the compass mag to get true head
float magVar;
float heading;

/////////////////////////////////////////////////////////////////////
// Navigation declarations
// You can implement a P, PI or PID controller by commenting out the
// unused sections.
/////////////////////////////////////////////////////////////////////
#define BASE_SPEED 150
#define k_0 1.45
#define T_0 5

//P Controller
#define k_p (.5*k_0) // Proportional Control Parameter
#define k_i 0 // Integral Control Parameter
#define k_d 0 // Derivative Control Parameter

// PI Controller
#define k_p .45*k_0 // Proportional Control Parameter
#define k_i (1/(1.2*T_0)) // Integral Control Parameter
#define k_d 0 // Derivative Control Parameter

// PID Controller
#define k_p .6*k_0 // Proportional Control Parameter
#define k_i .5*T_0 // Integral Control Parameter
#define k_d .125*T_0 // Derivative Control Parameter

#define PID_PRINT 0 // 1 to print, 0 for NO print
#define NAV_PRINT 0 // 1 to print, 0 for NO print
#define M_PI 3.14159
#define DISTANCE_ERROR .00005 // Error distance allowed (Robot considers itself "at" the waypoint if within this

float desired_heading; // defined in main() or in functions
float error_sum, previous_heading, previous_error;
float error_distance; // Distance from current position to intended waypoint
void drive_heading(float requested_heading);
float PID_Controller(void);

// uses flat Earth approximation around robot’s current position
// Define polarity for robot, N hemisphere (+) and East hemisphere (+)
const float esquared = .0066943800; // e^2 = f*(2-f) where f = 1/298.257223563 for WGS84
const float a = 6378.13700; // radius Earth in km for WGS84
float R1, R2; // meridional radius of curvature and radius of curvature for prime vertical in km
const float position_accuracy = 5; // 5 m accuracy for target position
const float heading_accuracy = 10; // +/- 10 degrees accuracy
float spd; // Forward velocity of the robot in meters per second
float turnrate; // Turn rate of the robot in radians per second, positive is clockwise

double myLAT;
double myLONG;
int WPcount;

/***************************
* Robot Utility Functions *
***************************/

double normalize2PI(double angle) {
while (angle >= (M_PI * 2.0)) angle -= (M_PI * 2.0);
while (angle < 0.0) angle += (M_PI * 2.0);
}
```c
return angle;
}

/* normalizePI ( double angle )
 * Normalizes an angle so that it is between -PI and PI
 * @param theta : angle ( radians ) to be normalized
 * @return an equivalent angle on a range of (-PI, PI]
 */
double normalizePI ( double angle ) {
    while ( angle > M_PI ) angle -= ( M_PI * 2.0 ) ;
    while ( angle <= -M_PI ) angle += ( M_PI * 2.0 ) ;
    return angle ;
}

/* distance ( double x1 , double y1 , double x2 , double y2 )
 * Computes the linear distance between two Cartesian points.
 * @param x1 : Cartesian X coordinate of the first point
 * @param y1 : Cartesian Y coordinate of the first point
 * @param x2 : Cartesian X coordinate of the second point
 * @param y2 : Cartesian Y coordinate of the second point
 * @return the linear distance between (x1 , y1) and (x2 , y2)
 */
double distance ( double x1 , double y1 , double x2 , double y2 ) {
    return sqrt ( (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1) ) ;
}

/* headingTo ( double x1 , double y1 , double x2 , double y2 )
 * Computes a heading to go from one point in Cartesian space to another
 * @param p1 : Cartesian X coordinate of the first point
 * @param y1 : Cartesian Y coordinate of the first point
 * @param x2 : Cartesian X coordinate of the second point
 * @param y2 : Cartesian Y coordinate of the second point
 * @return the bearing (heading) from (x1 , y1) to (x2 , y2)
 */
double headingTo ( double x1 , double y1 , double x2 , double y2 ) {
    return atan2 ( (y2 - y1) , (x2 - x1) ) ;
}
```

```c
/////////////////////////////////////////////////////////////////////
// GPS declarations
/////////////////////////////////////////////////////////////////////
#define GPS_Reset " $PGRMI ,,,,,,,R\r\n"
#define GPS_GGA_Enable " $PGRMO , GPGGA ,1\r\n"
#define GPS_Enable_Avg " $PGRMC1 ,,,2,,,,,,,,,,\r\n"
#define GPS_Disable_All " $PGRMO , ,2\r\n"
#define GPS_9600_Bd " $PGRMC ,,,,,,,,,,5,,,,\r\n"

typedef struct { float lat ; float lon ; char action ; } WP;
```
WP waypoints[13]; // 10 user waypoints + 3 for collision avoidance
WP desired_WP;
WP current_WP;
int current_wp_count; // a counter to keep track of waypoints

/////////////////////////////////////////////////////////////////////
// Functions
/////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////
// Basic Patterns
/////////////////////////////////////////////////////////////////////
void runCircle(File *file, int baseSpeed)
{
    long startPosition;
    int counter;
    startPosition = StepGetPos(2);
    counter = 10;
    t0 = MS_TIMER;
    // module address, goal position, maximum velocity, maximum acceleration, wait_for_reply

    while(counter < baseSpeed)
    {
        ServoSetPwm(1, counter, 1);
        ServoSetPwm(2, counter*(1/2), 1);
        delay(250);
        counter = counter + 10;
        logData(file);
    }
    ServoSetPwm(1, baseSpeed, 1);
    // slt = ServoSetVel(1,10000, 10, 0); // motor # 1
    delay(WAIT);
    // module address, goal position, maximum velocity, maximum acceleration, wait_for_reply
    ServoSetPwm(2, baseSpeed*(1/2), 1);
    // slt = ServoSetVel(2, 30000, 10, 0); // motor # 2
    delay(WAIT);
    ServoStartMove(255, 0);

    while(t)
    {
        timer = MS_TIMER;
        logData(file);
        delay(WAIT);
        averagePosition = (leftPosition + rightPosition)/2;
        rightPosition = StepGetPos(2);

        if(rightPosition > 15000 + startPosition)
        {
            stop();
            break;
        }
    }
    while(counter > 0)
    {
        ServoSetPwm(1, counter, 1);
        ServoSetPwm(2, counter, 1);
    }
delay(250);
counter = counter - 10;
logData(file);
}

void runLine(File *file , int baseSpeed)
{
  long startPosition;
  int counter;
  startPosition = StepGetPos(2);
  counter = 10;
t0 = MS_TIMER;

  // module address, goal position, maximum velocity, maximum acceleration, wait_for_reply
  s1t = ServoSetVel(1,70000, 10, 0); // motor # 1
  /* while(counter < baseSpeed) */
  {
    ServoSetPwm(1,counter, 1);
    ServoSetPwm(2,counter, 1);
    delay(250);
    counter = counter + 10;
    logData(file);
  }

  ServoSetPwm(1, baseSpeed, 1); /*
  delay(WAIT);
  */

  // module address, goal position, maximum velocity, maximum acceleration, wait_for_reply
  s1t = ServoSetVel(2,70000, 10, 0); // motor # 2
  // ServoSetPwm(2, baseSpeed, 1);
  delay(WAIT);

  // ServoStartMove(255,0);

  while(1)
  {
    timer = MS_TIMER;
    logData(file);
    delay (WAIT);
    averagePosition = (leftPosition + rightPosition)/2;
    rightPosition = StepGetPos(2);
    s1t = ServoSetVel(2, 70000, 10, 0); // motor # 2
    s1t = ServoSetVel(1,70000, 10, 0); // motor # 1

    if( rightPosition > 30000+ startPosition )
    {
      step();
      break;
    }
  }

  while(counter>0)
  {
    ServoSetPwm(1,counter, 1);
    ServoSetPwm(2,counter, 1);
    delay(250);
    counter = counter - 10;
    logData(file);
  } /*
  */

  //** robotInit ******************/
/**
 * Description: Initializes the key components of the robot,
 * including the board, motors, and any variables that need
 * setting.
 */

/***parameter descriptions:
**no parameters taken.

void robotInit(void)
{
    brdInit();
    serCrdFlush();
    ch = NmcInit();
    setServoParameters();
    
#if DEBUG_PRINT
    printf("Modules found on network: %d\n\n", ch);
#endif

    distanceTilNext = 0;

    // initialize gps and compass variables
    gps_error_count = 0;
    magVar = 14;

#if EXT_SENSORS
    // gps initialization
    serEopen(19200);
    serEwrFlush();
    // send configuration sentences to GARMIN GPS
    serEputs(GPS_9600_Bd);
    serEputs(GPS_Reset);// need reset for baud rate change to take effect
    serEputs(GPS_Enable_Avg);
    serEputs(GPS_Disable_All);
    serEputs(GPS_GGA_Enable);
    
    // Compass intialization
    serFopen(19200);//19200 baud default for Honeywell
    serFwrFlush();
    serFputs(init_str);
#endif

    }

void setServoParameters()
{
    // module address, position gain, derivative gain, integral gain, integration limit, output limit,
    // current limit, error limit, servo rate divisor, deadband compensation, wait_for_reply status (1-individual, 0-group)
    successFlag = ServoSetGain(255, 100, 10, 2, 1400, 255, 0, 4000, 1, 1, 0); // 0 for group addr
delay(WAIT);
    
#if DEBUG_PRINT
    printf("ServoSetGain success ?: %d\n\n", successFlag);
#endif

    // module address, control, wait_for_reply (1-individual, 0-group)
    successFlag = ServoSetIOMode(255, 12, 0); // for 4 motors group addresses
delay(WAIT);
    
#if DEBUG_PRINT
    printf("ServoSetIOMode success?: %d\n\n", successFlag);
#endif

    // This is necessary, initializes mode.
}
successFlag = ServoStopMotor(255,11,0);  // group address for all 4 motors;
delay(WAIT);
#if DEBUG_PRINT
printf("ServoStopMotor success?: %d\n\n", successFlag);
#endif

#if DEBUG_PRINT
printf("Initializing servos\n\n");
#endif

/** BASIC MOTOR COMMANDS *******************/
void setLeftSpeed (long velocity , long acceleration )
{
  // module address , goal position , maximum velocity , maximum acceleration , wait_for_reply
  slt = ServoSetVel(1 , velocity , acceleration , 1); // motor # 1
  delay(WAIT);

  //slt2 = ServoSetVel(3 , velocity , acceleration , 1); // motor # 3
  // delay(WAIT);

  slt = 0;
  //slt2 = 0;
}

void setRightSpeed (long velocity , long acceleration )
{
  // module address , goal position , maximum velocity , maximum acceleration , wait_for_reply
  slt = ServoSetVel(2 , velocity , acceleration , 1); // motor # 2
  delay(WAIT);

  //slt2 = ServoSetVel(4 , velocity , acceleration , 1); // motor # 4
  // delay(WAIT);

  slt = 0;
  //slt2 = 0;
}
/** END BASIC MOTOR COMMANDS***************/

/** ADVANCED MOTOR COMMANDS **************/
void stop ()
{
  //setLeftSpeed(0 , Acceleration);
  //setRightSpeed(0 , Acceleration);
  //delay(2000);
  ServoStopMotor(255 , 11 , 0);
}

void goStraight (long velocity , long acceleration , int direction ) //1 for forward , -1 for reverse
{
  //distance you want to go , velocity , acceleration
  setLeftSpeed(direction*(velocity -78) , acceleration); // was -75
  setRightSpeed(direction*velocity , acceleration);
  ServoStartMove(255,0);
}
/** END ADVANCED MOTOR COMMANDS*************/

/** DATA STORAGE COMMANDS***************/
void logData(File *f)
{
    char buffer1[7];
    char buff1[7], buff2[5], buff3[5], buff4[5], buff5[5], buff6[8], buff7[15], buff8[15];

    // TIMER
    timer = MS_TIMER - t0;
    sprintf(buff1, "%d", timer);
    fwrite(f, buff1, sizeof(timer));
    if(DATA_PRINT)
        printf("%s", buff1);
    fwrite(f, ",", 1);

    // LEFT POSITION
    leftPosition = StepGetPos(1);
    sprintf(buff2, "%d", leftPosition);
    fwrite(f, buff2, sizeof(leftPosition));
    if(DATA_PRINT)
        printf("%s", buff2);
    fwrite(f, ",", 1);

    // LEFT CURRENT
    M1AD = StepGetAD(1);
    sprintf(buff3, "%d", M1AD);
    fwrite(f, buff3, sizeof(M1AD));
    if(DATA_PRINT)
        printf("%s", buff3);
    fwrite(f, ",", 1);

    // RIGHT POSITION
    rightPosition = StepGetPos(2);
    sprintf(buff4, "%d", rightPosition);
    fwrite(f, buff4, sizeof(rightPosition));
    if(DATA_PRINT)
        printf("%s", buff4);
    fwrite(f, ",", 1);

    // RIGHT CURRENT
    M2AD = StepGetAD(2);
    sprintf(buff5, "%d", M2AD);
    fwrite(f, buff5, sizeof(M2AD));
    if(DATA_PRINT)
        printf("%s", buff5);

    // HEADING
    sprintf(buff6, "%3.0f", heading);
    fwrite(f, buff6, sizeof(heading));
    if(DATA_PRINT)
        printf("%s", buff6);

    // LAT
    sprintf(buff7, "%f", myLAT);
    fwrite(f, buff7, sizeof(myLAT));
    if(DATA_PRINT)
        printf("%s", buff7);
// LONG

    sprintf(buff8," %f ", myLONG);
fwrite(f,buff8,10); // sizeof(myLONG)
if( DATA_PRINTF )
    printf(" %s \n",buff8,sizeof(myLONG));

fwrite(f,"\n",1);
}

/** END DATA STORAGE COMMANDS ***********/

/////////////////////////////////////////////////////////////////////
//4. compass (RS332)
/////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////
// Function : msDelay
// Input : desired time delay in ms
// Output : Nil
// Description : To delay the processor with the input time in ms, primarily
// for serial and I2C communications
/////////////////////////////////////////////////////////////////////
void msDelay(long sd) {
    unsigned long t1; t1 = MS_TIMER ;
    for (t1 = MS_TIMER ; MS_TIMER < (sd + t1);) ;
}

/////////////////////////////////////////////////////////////////////
// Function : GetCompassString
// Input : char ptr to CompassString
// Output : Nil
// Description : Function fills buffer with serial data from serialB data
// from the Honeywell compass. Parses the buffer and returns the compass
// sentence between $ and *.
/////////////////////////////////////////////////////////////////////
void GetCompassString(char * RetCompassStr) {
    char error_buf[100];
    // waitFor ( DelayMs(compass_delay));
    msDelay(COMPASS_DELAY);
    serFrdFlush();
    string_pos = 0;
#if EXT_SENSORS
    input_char = serFgetc();
    // find begining of sentence
    while (input_char != ' $') {
        input_char = serFgetc();
    }
#endif
    // DEBUG_PRINT
    printf("%c", input_char);
#endif
    msDelay(READDELAY);
}
#if DEBUG_PRINT
    printf("\n");
#endif
    // read the sentence
    while (input_char != ' *') {
compass_sentence[string_pos] = input_char;
string_pos++;
if (string_pos == MAX_SENTENCE)
    string_pos = 0; // reset string if too large

input_char = serFgetc();
#endif
    printf("%c", input_char);
#endif
msDelay(READDELAY);
}
compass_sentence[string_pos] = 0; // add null
if ((compass_error = compass(compass_sentence)) != 0) {
    sprintf(error_buf, "$Compass Error: %d
", compass_error);
    // send error to the COMM module
    // strcat(ErrString, error_buf);
#endif
    printf("$Compass Error: %d
 %s
", compass_error, compass_sentence);
#endif
#else
return -1;
#endif
    printf("Current heading: %03.0f\n", heading);
#endif
Compass_update = 1; // global variable

//sprintf(RetCompassStr, "% %03.0f\n", heading);
#endif
#else
strcpy(compass_sentence, "$PTNTHPR,301,N");
#endif

/////////////////////////////////////////////////////////////////////
// Function : compass
// Input : char sentence
// Output : int
// Description : Sets the floating point heading global variable
// after parsing the sentence passed to the function.
/////////////////////////////////////////////////////////////////////
int compass(char sentence[MAX_SENTENCE]) {
    auto int i;
    char *err, *hdg, *type;
    char error;
    if (strlen(sentence) < 4)
        return -1;
    if (strncmp(sentence, "$PTNTHPR", 8) == 0) {
        // parse hpr sentence
        type = strtok(sentence, " ,");
        hdg = strtok(NULL, " ,");
        err = strtok(NULL, " ,");
        if (hdg == NULL)
            return -2;
        // pull out data - update global variable and add magVar to get true heading
        heading = atof(hdg) + magVar;
        if(heading >=360.0)
            heading = heading - 360.0;
        error = (int) err;
        if (strncmp(&error, "N", 1) == 0)
            return -2;
    } else
        return -1;
return 0;
}

void gps() {
    char sentence[MAX_GPS_SENTENCE];
    int input_char;
    int string_pos;
    int gps_error;
    unsigned long t;

    t = MS_TIMER;
    serErdFlush();
    string_pos = 0;
    
#ifndef EXT_SENSORS
    input_char = serEgetc();
    //printf("c: %c 
", input_char);
    while (input_char != '$') // find beginning of sentence
    {
        input_char = serEgetc();
        msDelay(READDELAY_GPS);
        //printf("c: %d 
", input_char);
    }
    while ((input_char != '\'r') && (input_char != '\n'))
    {
        sentence[string_pos] = input_char;
        string_pos++;
        if (string_pos == MAX_GPS_SENTENCE)
            string_pos = 0; // reset string if too large
        input_char = serEgetc();
        msDelay(READDELAY_GPS);
    }
#endif

#ifndef EXT_SENSORS
    // pass a dummy GPS packet if there are no external sensors set up
    strcpy(sentence, "$GPGGA,123456,3635.7058,N,12152.4939,W,2,5*");
#endif

#ifndef DEBUG_PRINT
    printf("%s
", sentence);
#endif

    // SendToCommModule(&GpsChan, sentence);
    gps_error = gps_get_position(&current_pos, sentence); // parse lat and long position with gps lib function

    if (gps_error == 0) {
        // get horizontal nav error
        if (!gps_horz_error(sentence)) // if no error, then GPS accuracy is acceptable
        {
            gps_error_count = 0;
            //dr_mode_flag = 0;
        } else // sentence is OK, but horizontal dilution of precision exceeds limit
        {
            // or gps reported a sentence with no position fix
        }
    }
gps_error_count ++;
#if(gps_error_count > 3)
  //dr_mode_flag = 1; //start dead reckoning nav.
#else // gps parsing error or invalid sentence
  gps_error_count ++;
  //if(gps_error_count > 3)
  // dr_mode_flag = 1; //start dead reckoning nav.
#endif

sentence[string_pos] = 0;

/////////////////////////////////////////////////////////////////////
// Function : gps_horz_error
// Input : char * sentence
// Output : int result 0 = no error , 1 = error in horizontal positioning
// Description : This function parses the gps GGA sentence to extract the
// horizontal dilution of precision and quality of the gps fix . dop
// greater than roughly 6.0 will result in poor fix accuracy.
/////////////////////////////////////////////////////////////////////
int gps_horz_error (char * sentence) {
  auto int i;
  char * charPtr;
  char * dummy;

  char s[6]; // temp string for the dop chars
  float dop; // dilution of precision

  strcpy(s, ""); // set to NULL

  if (strncmp(sentence, "$GPGGA", 6) == 0) {
    // parse GGA sentence
    for (i = 0; i < 9; i++) {
      sentence = strchr(sentence, ',');
      if (sentence == NULL)
        return 1; // return true , horz. error exists
      sentence ++; // first character in field
      // pull out data
      if (i == 5) // link quality
        {
          if (*sentence == '0') // indicates no fix
            {
              #if DEBUG_PRINT
              printf(" No fix \n");
              #endif
              return 1; // return true , horz. error exists
            }
        }
    }
  }

  // DGPS fix can still have excessive horizontal errors, while
  // GPS can have acceptable fix accuracy if good satellite geometry.
  // Test satellite geometry (Horizontal Dilution of Precision, HDOP)
  // before accepting fix.

  // get the HDOP measure of the geometry. If it's OK,
  // consider the fix GOOD. If HDOP too big, return 1 = error.
  // Based on experimental observation in Quad...
  // Typical HDOP for excellent, 6 satellites DGPS fix is 1.1 to 1.7
  // Adequate GPS fixes, HDOP varies from 2.0 to up to 5.5
  // 4 satellite GPS fixes with HDOP = 8 or 9 are about 20 to 30 m off
  if (i == 7) {
    // get the chars from sentence to the next ','
    charPtr = strchr(sentence, ',');
    // find the location of ,
    strcat(s, sentence, (charPtr - sentence));
    // convert the chars to float
    dop = strtod(s, &dummy);
  }
#if DEBUG_PRINT
    printf("gps dop %f\n", dop);
#endif

// test dop
if (dop > 5.6)
    return 1; // horizontal error too big
}

} // end for loop
} // end if good GGA sentence
else // error not GGA sentence
{
    return 1;
}
return 0; // everything else is OK, so no error

/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////
// Function: navigation
// Input: Nil
// Output: Nil
// Description: Uses a potential field algorithm which simulates waypoints as linear points of attractions and obstacles as 1/r^2 points of repulsion.
// This approach encompasses both waypoint driving and object avoidance simultaneously.
//
void navigation(void) {

    float err_angle; // Difference between current heading and intended heading
    float angle_to_goal; // Bearing to waypoint from GPS location, independent of robot heading
    float turn_gain; // Gain for proportional turn rate
    float speed_gain; // Gain for proportional speed rate
    float min_speed; // Minimum speed for search motion
    float max_speed; // Maximum speed for search motion

    float repgain; // Gain factor for rep force
    float repmag; // Magnitude for rep force
    float repdir; // Polar direction for direction of rep force
    float force_rep_x; // Computed x force component of rep force
    float force_rep_y; // Computed y force component of rep force
    float sumforce_rep_x; // Summation of x force component of rep force
    float sumforce_rep_y; // Summation of y force component of rep force

    float attgain; // Gain factor for att force
    float attmag; // Magnitude for att force
    float attdir; // Polar direction for direction of att force
    float force_att_x; // Computed x force component of att force
    float force_att_y; // Computed y force component of att force
    float sumforce_att_x; // Summation of x force component of att force
    float sumforce_att_y; // Summation of y force component of att force

    float totalforce_x; // Addition of att and rep force x component
    float totalforce_y; // Addition of att and rep force y component

    double current_lat, current_lon;
    double wypt_lat, wypt_lon; // lat and long for waypoint

// find Origin of the flat Earth. Let Origin correspond to robot's current position

//current_lat = current_WP.lat;
//current_lon = current_WP.lon;
current_lat = (current_pos.lat_minutes / 60); // current_pos.lat_degrees
if (current_pos.lat_direction == 'S')
    current_lat = 0 - current_lat; // negative values in Southern hemisphere

if (current_pos.lon_direction == 'W')
    current_lon = 0 - current_lon; // negative values for West hemisphere

#if NAV_PRINT
    printf("CURRENT: %f %f\n", current_lat, current_lon);
#endif

// find the lat, lon of the point to navigate to from the Origin
wypt_lat = desired_WP.lat;
wypt_lon = desired_WP.lon;

#if NAV_PRINT
    printf("DESIRED: %f %f\n", wypt_lat, wypt_lon);
#endif

myLAT = current_lat;
myLONG = current_lon;

error_distance = distance(current_lon, current_lat, wypt_lon, wypt_lat); // Error distance between goal and starting position

if (WPcount == 2 && error_distance < DISTANCE_ERROR)
    WPcount = 3;
if (WPcount == 1 && error_distance < DISTANCE_ERROR)
    WPcount = 2;

desired_heading = (180.0/M_PI)*normalize2PI(M_PI/2 - headingTo(current_lon, current_lat, wypt_lon, wypt_lat)); // Bearing to waypoint from gps location, independent of robot heading
#if NAV_PRINT
    printf("D HEADING: %f\n", desired_heading);
#endif

} } 

/////////////////////////////////////////////////////////////////////
// Function: drive_heading
// Programmer: Jess Fitzgerald
// Description: navigates to and maintains requested course
// 
/////////////////////////////////////////////////////////////////////
void drive_heading(float requested_heading)
{
    float modification;
    float Lvel, Rvel;

desired_heading = requested_heading;
#if READING_PRINT
    printf("Heading %03.0f\n", heading);
#endif

float PID_Controller(void)
{
  float error, d_heading; // d_heading is the difference between the current
  // heading and the last heading
  float returnValue;
  // Proportional Control
  #if PID_PRINT
    printf("%d A:%03.0f D:%03.0f\n", MS_TIMER, heading, desired_heading);
  #endif
  error = desired_heading - heading;
  if(error < -180) // gives relative bearing from -180 to 180
    error = error + 360;
  if(error>180)
    error = error - 360;
  //if PID_PRINT
  //  printf(" E:%03.0f", error);
  //#endif
  if(error<2.0 && error > -2.0)
    error_sum=0;
  else
    error_sum+=error; //1;
  if(error_sum>180 || error_sum<-180)
    error_sum = 0;
  // Derivative Control
  d_heading = heading - previous_heading; //heading derivative
  previous_heading = heading;
  previous_error = error;

  returnValue = (k_p*error + (1/k_i)*error_sum - (k_d)*d_heading);
  //if PID_PRINT
  //  printf(" R: %1.3f ", returnValue);
  //#endif

  return returnValue;
}

main()
{
  File file1;
}
File file2;
char CompassString[100]; // for comms to control
char dir_string; // for gps print
char val[20];

robotInit();
fs_get_ram_lx();
fs_init(0,0);

desired_WP.lat = .603010;
desired_WP.lon = -.873840; // longitudes from laptop are inverse polarity, so multiply by -1

desired_WP.action = 'a';

WPcount = 1;

// current_WP.lat = 36.5904889;
// current_WP.lon = -121.8716511;
// current_WP.action = 'a';

error_sum=0;

    if (!fcreate(&file1, TESTFILE) && fopen_wr(&file1,TESTFILE))
    {
      printf("\n\nerror opening TESTFILE %d\n", errno);
      return -1;
    }

    if (!fcreate(&file2, TESTFILE2) && fopen_wr(&file2,TESTFILE2))
    {
      printf("\n\nerror opening TESTFILE2 %d\n", errno);
      return -1;
    }

    fclose(&file1);
    fclose(&file2);
    fdelete(1);
    fdelete(2);

    if (!fcreate(&file1, TESTFILE) && fopen_wr(&file1,TESTFILE))
    {
      printf("\n\nerror opening TESTFILE %d\n", errno);
      return -1;
    }

    if (!fcreate(&file2, TESTFILE2) && fopen_wr(&file2,TESTFILE2))
    {
      printf("\n\nerror opening TESTFILE2 %d\n", errno);
      return -1;
    }

    //runLine(&file1, 240);
    //delay(500);
    //runCircle(&file1, 200);
    //stop();
    //desired_heading=190;

    /*ServoSetPwm(1,250, 1);
    ServoSetPwm(2,50, 1);
delay(10000); */

    t0 = MS_TIMER;
    desired_heading=50;

    while(!kbhit()) // (MS_TIMER<t0+6000){
costate { //compass
  GetCompassString(CompassString);
  compass_error = compass(compass_sentence);
  SendToCommModule(&CompassChan, CompassString);
  SendCompassToControlStation();
  tcp_tick(NULL);
  waitfor(DelayMs(50));
} // end costate

costate { //gps
  gps();
  #if DEBUG_PRINT
    dir_string = current_pos.lat_direction;
    printf(" Latitude : %d %g %c\n", current_pos.lat_degrees,
      current_pos.lat_minutes, dir_string);
    dir_string = current_pos.lon_direction;
    printf(" Longitude : %d %g %c\n", current_pos.lon_degrees,
      current_pos.lon_minutes, dir_string);
  #endif
  SendGpsToControlStation();
  tcp_tick(NULL);
  waitfor(DelayMs(1000));
} // end costate

} // end costate

costate { //Drive to desired Heading
  if(WPcount == 2)
  {
    desired_WP.lat = .603254; // .59508418; // 36.59038613;
    desired_WP.lon = -.873187; // -.87496038; // longitudes from laptop are inverse polarity, so
    multiply by -1
    desired_WP.action = 'a';
  }
  if(WPcount == 3)
  {
    desired_WP.lat = .603150; // .59508418; // 36.59038613;
    desired_WP.lon = -.873288; // -.87496038; // longitudes from laptop are inverse polarity, so
    multiply by -1
    desired_WP.action = 'a';
  }
  navigation();
  drive_heading(desired_heading); // This will drive to the most
  // recently calculated desired
  // heading stored in the global
  // variable desired_heading.
  logData(&file1); // LOG data from current run
  // printf("%f\n", error_distance);
  if (kbhit() || (error_distance<DISTANCE_ERROR && (WPcount == 2)) || MS_TIMER> t0+60000)
  {
    fclose(&file1);
    fclose(&file2);
    stop();
    abort;
  }
  tcp_tick(NULL);
  waitfor(DelayMs(120));
} // end costate
fclose(&file1);
close(&file2);
delay(1000);
stop();

if( DEBUG_PRINT || DATA_PRINT)
printf(" Ending program \n");

) //end while
APPENDIX B:
Data Extraction Code

/* Fitzgerald_Thesis_2013.c
 * Programmers: Jessica L. Fitzgerald
 * Programs Referenced/Used: Z-world's 'DUMPFILE.C' from file system sample files
 * Supervisor: Prof Harkins
 * Program Description: Used to extract Data stored in RAM by 'Fitzgerald_Thesis_2013.c' which formats the string before storage.
 */

#include auto
#define MY_FORCE_FORMAT 0
#define FS2_USE_PROGRAM_FLASH 16

/**
 * LX_2_USE - You can use fs_get_flash_lx() (for the 2nd flash);
 * or fs_get_other_lx() (for program flash if configured in \BIOS\RABBITBIOS.C).
 */
define LX_2_USE fs_get_ram_lx()

#define FS_DEBUG
#define FS_DEBUG_FLASH
#define memmap xmem
#define "fs2.lib"
defuse "errno.lib"

FSLXnum fs_ext;

/* START FUNCTION DESCRIPTION ********************************************
mem_dump
SYNTAX: void mem_dump( const char * real_ptr , long addr , long d_size )
KEYWORDS: debug
DESCRIPTION: Formats bytes in root memory. If addr ! = -1L, then bytes fetched from real_ptr but displayed as if from addr.
0 1 2 3 4 5 6 7 8 9 A B C D E F
543210: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0123456789 ABCDEF
RETURN VALUE: none.
SEE ALSO: printf()
END DESCRIPTION **********************************************************/

// Separator between byte 7 and 8.
define CH_BETWIX '\'

nodebug void
mem_dump( char * real_ptr , long addr , long d_size )
{
    static const char hexdigs[] = "0123456789ABCDEF";
    auto char row[16];
    auto char str[80];
}
auto char pstr[80];
auto char * bp;    /* hex byte ptr */
auto char * ap;    /* ASCII rendering pointer */
auto char j;

/* If not faking address, set it... */
if(-1L == addr) {
    addr = (long)real_ptr;
}

while( d_size > 0 ) {
    /* The magic constants here define where the hex and ASCII areas are. */
    sprintf(str, "%06lx: ", addr & 0xFFFFFFF0L);
    memset(str +7, ' ', sizeof(str)-1-7);
    str[31] = CH_BETWIX;
    str[57] = '|';
    str[74] = '|';
    str[75] = '0';
    j = ((char)addr) & 0x0F;
    bp = str + 8 + j * 3;
    ap = str + 58 + j;
    memcpy(row + j, real_ptr, 16 - j);
    /* ADVANCE */
    real_ptr += (16 - j);
    addr += (16 - j);

    /* Convert to printable ASCII, if applicable, default is '.' */
    for(; d_size > 0 && j < 16 ; ++j, --d_size ) {
        *bp++ = hexdigs[row[j] >> 4];
        *bp++ = hexdigs[row[j] & 15];
        *ap = '.';
        if( ' ' <= row[j] && row[j] <= '~') {
            *ap = row[j]; /* If printable ASCII, then display it. */
        }
        ap++;
        bp++;  // skip space between hex numbers.
    }
    printf("%s\n", str);
}

} /* end mem_dump() */

/**
 * Dump the file contents.
 */
void
do_dump(int fnum)
{
    char contents[128];
    File f;
    int want, got, count;
    long addr;
    char buffer[1], time[5], leftE[5], rightE[5], leftI[2], rightI[2];

    count = 1;
    if( 0 != fopen_rd(& f, fnum ) ) {
        printf("%sError: file %d doesn't exist (errno %d)\n", fnum, errno);
        exit(2);
    }
    addr = 0L;
    while(fread(&f,buffer,1)>0)
    {

    }
printf("%s",buffer);

//if(count%5 == 0)
//printf("\n");
count = count +1;
}

while( 0 != (got=fread( & f, contents, sizeof(contents) ))) {
    mem_dump( contents, addr, got );
    addr += got;
} /*
fclose( & f );
} /* end do_dump() */

/**
 * Create a file with some contents. Lots of error checking here.
 */
void
do_create(int fnum)
{
    auto char contents[128];
    auto File  f;
    auto int   want, got;
    auto long  addr;

    /* fopen_wr() if file exists, or fcreate() if doesn't exist. */
    if( 0 != fopen_wr( & f, fnum ) && 0 != fcreate( & f, fnum ) ) {
        printf("\nError : file #%d not writable ( errno %d)\n", fnum, errno);
        exit(2);
    }

    sprintf( contents, "This is a test 1234 of FileSystem MkII\n");

    if( 0 == fwrite( & f, contents, strlen(contents) ) ||
        0 == fwrite( & f, contents, strlen(contents) ) ||
        0 == fwrite( & f, contents, strlen(contents) ) ) {
        printf(" error : writing ( errno %d)\n", errno);
    }
}
fclose( & f );
} /* end do_create() */

/**
 * Try opening (for reading) every file. Print those we can do!
 */
void
scan_for_existing_files()
{
    auto File  f;
    auto int   fnum;
    auto int   count;

    printf("Found these existing files: ");
    for( fnum=1, count = 0 ; fnum < 128 ; ++ fnum ) {
        if( 0 != fopen_rd( & f, fnum ) ) {
            fclose( & f );
            printf(" #%d ", fnum );
            ++count;
        }
    }

    if( 0 == count ) {
        printf(" ... none ...");
    }
    printf("\n");
} /* end scan_for_existing_files() */
void main()
{
    char contents[128];
    int rc;
    int fnum;
    errno = 0;
    fs_ext = LX_2_USE;
    if (!fs_ext) {
        printf("The specified device (LX# %d) does not exist. Change LX_2_USE.
", (int)fs_ext);
        exit(1);
    } else {
        printf("Using device LX# %d...
", (int)fs_ext);
    }
    /*
     * Step 2: format the filesystem if requested. Note that formatting
     * must be done _after_ the call to fs_init().
     * This demo has compile-time constant of whether the flash should be
     * formatted or not.
     */
    rc = fs_init(0, 0);
    if( rc != 0 ) {
        printf("Error: can't get filesystem initialized.
");
        exit(2);
    }
    if( MY_FORCE_FORMAT ) {
        printf("Note: File System MkII not present.
");
        if( 0 != lx_format(fs_ext, 0) ) {
            printf("ABORT: Can't Format MkII File System, errno=%d
", errno);
            exit(2);
        }
    }
    /* Show the user some possible solutions: */
    scan_for_existing_files();
    /* Prompt for the file number to dump or create. */
    printf("Which file number to affect? ");
    gets(contents);
    fnum = atoi(contents);
    if( fnum <= 0 ) {
        printf("Error: bad input
");
        exit(2);
    }
    printf("\n--------------\n");
    do_dump(fnum);
}


Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia
2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California