STATE ESTIMATION OF AN UNMANNED GROUND VEHICLE USING INEXPENSIVE SENSORS

by

Noah Johnson

A Thesis

Submitted to the Division of Natural Sciences
New College of Florida
in partial fulfillment of the requirements for the degree
Bachelor of Arts
under the sponsorship of Professor Gary Kalmanovich

Sarasota, Florida
May 2017

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

State Estimation of an Unmanned Ground Vehicle Using Inexpensive Sensors

Noah Johnson

New College of Florida, 2017

Submitted to the Division of Natural Sciences
on May 18, 2017, in partial fulfillment of the
requirements for the degrees of
Bachelor of Arts in Computer Science
and
Bachelor of Arts in Applied Mathematics

Abstract

Autonomous navigation is an important emerging technology with applications in warehouse automation, shipping, and personal navigation. Hands on experimentation often proves too expensive for individuals at the undergraduate level, with common research platform costing thousands of dollars. In this thesis, I present a design for a cheap ground vehicle capable of fusing sensor data into a local state estimation. Costs were minimized by using personal components owned by most students, namely a personal laptop and smartphone. This should be an effective base for research into various aspects of autonomous navigation.

Professor Gary Kalmanovich .....

May 18, 2017

Chapter 1

Introduction

This rover Create a cheap outdoors autonomous robot. Given a 'map' of the New College campus, this robot should be able to navigate from one outdoors location to another using footpaths. In doing so, it should dynamically avoid obstacles such as people, and recalculate alternate routes when a route is unexpectedly blocked.

intro to the concept of navigation

Since a map with GPS coordinates is provided, this is not Simultaneous Localization and Mapping (SLAM), but a simplified navigation problem.

The rover base consists of a Lynxmotion rover, an Arduino, and a Sabertooth motor driver. Sensors available include two quadrature rotary encoders, a mobile phone, and an ultrasonic distance sensor. The Arduino acts as a low-level robotic controller, publishing wheel encoder and range data, and accepting motor velocity commands. An Android app publishes IMU and GPS data from the mobile phone, and the laptop fuses these readings into a state estimation of pose and velocity using an extended Kalman filter.

IR, Microsoft Kinect can't use because the rover will be outdoors during the day and the sun gives off ambient IR radiation.

LIDAR - state of the art

And LIDAR sensors would be too costly. constrained by cheap hardware, and the inability to use IR or LIDAR sensors.

The rest of this thesis is organized as follows.

Chapter 2 covers the probability theory behind the extended kalman filter, which is an iterative update algorithm used to fuse noisy sensor data into a local state estimate for the rover. It may be skipped if one is not interested in the mathematical details.

Chapter 3 describes the hardware components used in this project, their electrical connections, and the general design.

Chapter 4 continues the description of physical connections with respect to the Arduino circuit board, and also describes the software which runs on that board and how it interfaces with the rest of the system.

Chapter 5 gives a brief overview of the Robot Operating System (ROS), and how it's used in this project. It then meanders through every software process used, and how they communicate through ROS.

Chapter 6 describes the results of an experimental test conducted with the rover. Limitations of and extensions to the current design are also mentioned.

Chapter 2

Theory

EKFs are important to this thesis

Robots estimate their environment stochastically, and so probability theory is vital to understanding their inner workings. First we will review the necessary theory, and then we will examine a class of algorithms for recursive state estimation. For a more in-depth look at this material, see Probabilistic Robotics [16].

2.1 Probability Background

Discrete random variables have a finite output space of possible values that may be observed. Let X be a random variable, then we define the probability that we observe value x from X as $p(X = x) \equiv p(x)$. Since x is arbitrary, this defines a probability distribution. For every random variable X, we have

$$\sum_{x \in X} p(x) = 1$$

Given two more random variables Y and Z, we'll define the joint distribution $p(X = x \text{ and } Y = y \text{ and } Z = z) \equiv p(x, y, z)$, and the conditional probability $p(X = x \text{ given that } Y = y \text{ and } Z = z) \equiv p(x \mid y, z)$. The conditional probability is defined to be

$$p(x \mid y, z) = \frac{p(x, y, z)}{p(y, z)} \tag{2.1}$$

The *Law of Total Probability* states that $p(x) = \sum_{y \in Y} p(x, y)$. Extending this law to use a third random variable Z, and incorporating the definition of conditional probability, we end up with the following equation:

$$p(x \mid z) = \sum_{y \in Y} p(x, y, z) = \sum_{y \in Y} p(y, z) p(x \mid y, z) \tag{2.2}$$

Lastly, we can use equation 2.1 to derive a version of Bayes' Theorem.

$$p(x \mid y, z) = \frac{p(x, y, z)}{p(y, z)} = \frac{p(y, x, z)}{p(x, z)} * \frac{p(x, z)}{p(y, z)} = \frac{p(y \mid x, z) p(x, z)}{p(y \mid z)} \tag{2.3}$$

In the future this will prove to be a useful tool to compute a posterior probability distribution $p(x \mid y)$ from the inverse conditional probability $p(y \mid x)$ and the prior probability distribution $p(x)$.

2.2 Bayes Filter

2.2.1 Scenario

Consider the general case of a robot which uses sensors to gather information about its environment. These sensors provide readings at discrete time steps $t = 0, 1, 2, ...$ Some amount of noise is associated with each of these readings. At each time step t, the robot may execute commands to affect its environment, and wishes to know its current state. [16]

Let's encode the robot's current state at time t in the vector x_t. Similarly, z_t will represent a sensor measurement at time t, and u_t will represent the commands issued by the robot at time t. For each of these vectors we will use the notation $z_{1:t} = z_1, z_2, ..., z_t$. [16]

The robot only has access to data in the form of z_t and u_t. Thus it cannot ever have perfect knowledge of its state x_t. It will have to make do by storing a probability distribution assigning a probability to every possible realization of x_t. This posterior probability distribution will represent the robot's belief in its current state, and should be conditioned on all available data. Thus we'll define the robot's belief distribution to be [16]:

$$bel(x_t) = p(x_t \mid z_{1:t}, u_{1:t}) \tag{2.4}$$

2.2.2 Derivation

We can use equation 2.3 to rewrite $bel(x_t)$:

$$bel(x_t) = p(x_t \mid z_{1:t}, u_{1:t}) = \frac{p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) p(x_t \mid z_{1:t-1}, u_{1:t})}{p(z_t \mid z_{1:t-1}, u_{1:t})}$$

In order to simplify $p(z_t \mid x_t, z_{1:t-1}, u_{1:t})$, we'll have to make an important assumption. We'll assume that the state x_t satisfies the Markov property, that is, x_t perfectly encapsulates all prior information. Thus if x_t is known, then $z_{1:t}$ and $u_{1:t}$ are redundant. This assumption lets us remove consideration of past sensor measurements and commands, and to rewrite the belief distribution as:

$$bel(x_t) = \frac{p(z_t \mid x_t) p(x_t \mid z_{1:t-1}, u_{1:t})}{p(z_t \mid z_{1:t-1}, u_{1:t})}$$

Notice that $p(z_t \mid z_{1:t-1}, u_{1:t})$ is a constant with respect to x_t. Thus it makes sense to let $\eta = (p(z_t \mid z_{1:t-1}, u_{1:t}))^{-1}$ and rewrite the belief distribution as:

$$bel(x_t) = \eta p(z_t \mid x_t) p(x_t \mid z_{1:t-1}, u_{1:t})$$

Now we are left with two distributions of interest. Looking closely one may notice that $p(x_t \mid z_{1:t-1}, u_{1:t})$ is simply our original belief distribution, equation 2.4, but not conditioned on the most recent sensor measurement, z_t. Let us refer to this distribution as $\overline{bel}(x_t)$, and break it down further using equation 2.2 and our Markov

assumption [16]:

$$\begin{aligned}
\overline{bel}(x_t) &= p(x_t \mid z_{1:t-1}, u_{1:t}) \\
&= \sum_{x_{t-1}} p(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}) p(x_{t-1} \mid z_{1:t-1}, u_{1:t}) \\
&= \sum_{x_{t-1}} p(x_t \mid x_{t-1}, u_t) p(x_{t-1} \mid z_{1:t-1}, u_{1:t}) \\
&= \sum_{x_{t-1}} p(x_t \mid x_{t-1}, u_t) bel(x_{t-1})
\end{aligned}$$

We have arrived at a recursive definition of $bel(x_t)$ with respect to $bel(x_{t-1})$! As long as $p(x_t \mid x_{t-1}, u_t)$ and $p(z_t \mid x_t)$ are known, we can recursively calculate $bel(x_t)$.

$p(x_t \mid x_{t-1}, u_t)$ defines a stochastic model for the robot's state, defining how the robot's state will evolve over time based upon what commands it issues. This probability distribution is known as the *state transition probability*. [16]

$p(z_t \mid x_t)$ also defines a stochastic model, modeling the sensor measurements z_t as noisy projections of the robot's environment. This distribution will be referred to as the *measurement probability*. [16]

Once we have models for both the *state transition probability* and *measurement probability*, we can finally construct the algorithm known as Bayes' Filter [16]:

2.3 Extended Kalman Filter

[15]

Algorithm 1 Bayes Filter

1: **function** BAYESFILTERITERATE($bel(x_{t-1})$, u_t, z_t)
2: **for** each possible state $x_t^* \in x_t$ **do**
3: $\overline{bel}(x_t^*) = \sum_{x_{t-1}^* \in x_{t-1}} p(x_t^* \mid x_{t-1}^*, u_t) bel(x_{t-1}^*)$
4: $bel(x_t^*) = \eta p(z_t \mid x_t^*) \overline{bel}(x_t^*)$
5: **end for**
6: Set $\sum_{x_t^* \in x_t} bel(x_t^*) = 1$, and solve for η
7: Use η to compute $bel(x_t)$
8: **return** $bel(x_t)$
9: **end function**

Chapter 3

Hardware

3.1 Specific Hardware Used

The specific hardware used in this project was chosen to minimize cost while still producing a vehicle capable of navigating rough, uneven outdoors terrain. Parts that were already on hand, and that most college students would reasonably have access to, such as a personal laptop and an Android smartphone, were used over superior alternatives. In total these parts were purchased for less than $500, with most of that cost coming from the rover's base.

The mobile base chosen was the Lynxmotion A4WD1 Rover, shown in Figure 3-1. It was purchased as a kit including four 200 RPM DC gear motors, 100 PPR motor encoders, and 4.75" diameter wheels. The chassis consists of four aluminum side brackets, and two polycarbonate panels on the top and bottom. This kit makes up the bulk of the cost of

Figure 3-1: Lynxmotion 4WD Rover [11]

the project. It was chosen for its larger wheels with relatively
strong motors, making it robust to uneven terrain and ramps,
ideal for outdoor use. The rover can support up to 5 lbs overall, and a second level which stacks on top is available as an extension. This second level would be an ideal place to put a tablet or small laptop.

One downside of this robot base is the fixed position of the four wheels. The lack of a turnable axis means the rover must steer by varying the speed of its motors. This causes wheel slippage in one or more of the wheels when the rover turns, meaning the motor encoders won't see the distance traveled. This ultimately causes errors in the robot's localization. Choosing a different base which uses only two driven wheels would avoid this issue, and likely be even cheaper. However, such a choice must be balanced with the robot's suitability for outdoor use, and room for all on-board components.

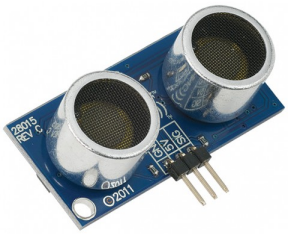
The motors are controlled by a Sabertooth dual-channel 12A 6V-24V regenerative motor driver. The lithium ion battery over-discharge protection. protects from back EMF causing a current flow

There is a 5A version of this motor driver, which should have been purchased to save $20.

This motor driver is powered by two LG 18650 HE2 rechargable lithium ion cells, which sit in an 18650 battery case. This case is made out of a battery holder soldered in series to act as a pack. The battery cells were individually charged before each use by a NiteCore-i2-V2014 li-ion charger.

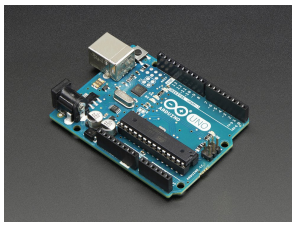
On top of the rover is the PING))) ultrasonic distance sensor (see Figure 3-2), which is attached to a standard Parallax servo which pans back and forth 180 degrees. This range sensor emits an ultrasonic chirp, and times how long it takes for that chirp to echo back. Based on that time, the distance from the sensor to an obstacle can be calculated. The PING))) sensor can be used to detect objects from 2cm to 3 meters away. [8]

Figure 3-2: PING))) Ultrasonic Sensor [13]



At the center of the rover is an Arduino Uno R3, see Figure 3-3. This microcontroller board handles several important tasks. It tells the motor driver what speed to set its two output channels to, and directly controls the panning motion of the standard Parallax servo. It also acts as a go-between for the digital output of the sensors on the rover and a laptop. It's connected to this laptop via a USB cable, which powers the board and allows communication over a serial port. Motor encoder values and ultrasonic range data are transmitted to the laptop, and motor power commands are received. An Arduino prototyping shield is stacked on top to allow re-usability of the board.

Figure 3-3: Arduino Uno R3 [9]



Only two encoders end up being used, due to a limit of the microcontroller board used. Choosing a different model with more hardware interrupt pins would improve accuracy, without drastically increasing the price.

The specific laptop used in this project is the Dell Inspiron 3531, which has a

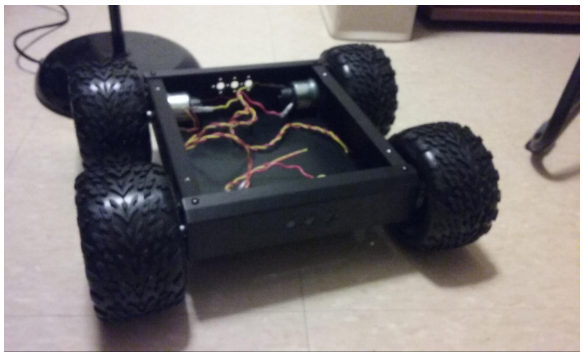
quad core 2.16 GHz processor, and 4 GB of RAM. Any personal laptop running Ubuntu or Debian could be used here, and additional computational resources would be beneficial. However, this laptop was a personal work machine and already available to use at no additional cost. The laptop is used as the main processing unit for the navigation logic.

The last component is a Nexus 4 smartphone placed on the top panel of the rover, which is also connected to the laptop by USB. Inside this phone is an MPU-6050 chip which contains a gyroscope and accelerometer. Elsewhere on the phone's logic board are a magnetometer, otherwise known as a digital compass, and a GPS receiver. This was also a personal device already available, and acts as a cheap Inertial Measurement Unit (IMU) and GPS receiver for the robot.

3.2 Construction

Figure 3-4 shows the base just after assembly. The aluminum side brackets' mounting holes did not line up properly with the motors, so a Dremel drill was used to widen them.

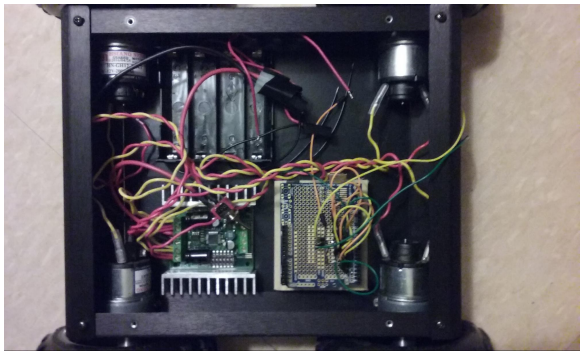
Figure 3-4: Constructed Chassis



The Arduino Uno was screwed to a 2.5" x 3" x 0.5" wooden poplar block, with non-conductive nylon washers placed between the screw head and the Uno, and between the Uno and the wooden

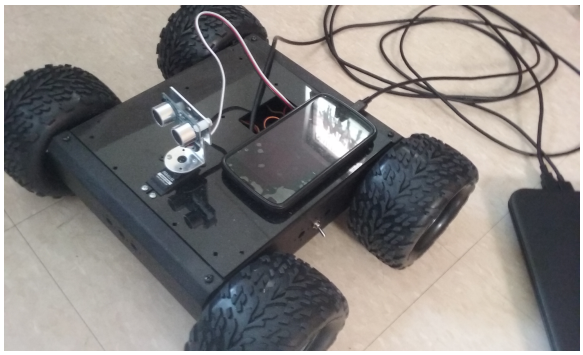
block. The wooden Arduino mounting board and the Sabertooth were both attached via double-sided foam mounting tape to the bottom panel of the rover. The battery holder was attached with glue dots to make removal easier.

Figure 3-5: Pieces Mounted



The servo fits conveniently into a pre-cut opening in the top chassis panel, and is held in place with four 3mm x 6mm screws and corresponding washers. A mounting bracket is attached to the servo, and the PING))) sensor is screwed to that mounting bracket, using non-conductive washers and screws to separate the circuit board and the metal mounting bracket.

Figure 3-6: Construction Finished



Another opening in the top panel allows the PING))) sensor to connect to the Arduino inside the body of the rover. This opening also allows the type A/B USB cable connected to the Arduino to extend out and reach the laptop. The smartphone sits on the top panel just to the left of this opening, secured in place by removable

glue adhesive dots. It is also connected to the laptop via a micro-USB to USB cable.

Both USB cables are long, stretching to just under 10 feet. The USB 2.0 specification limits the length of cable between two 2.0 USB devices to less than five meters, or about 16 feet [17]. Thus there should be no problem with the current length, but extensions in the future could not go much further.

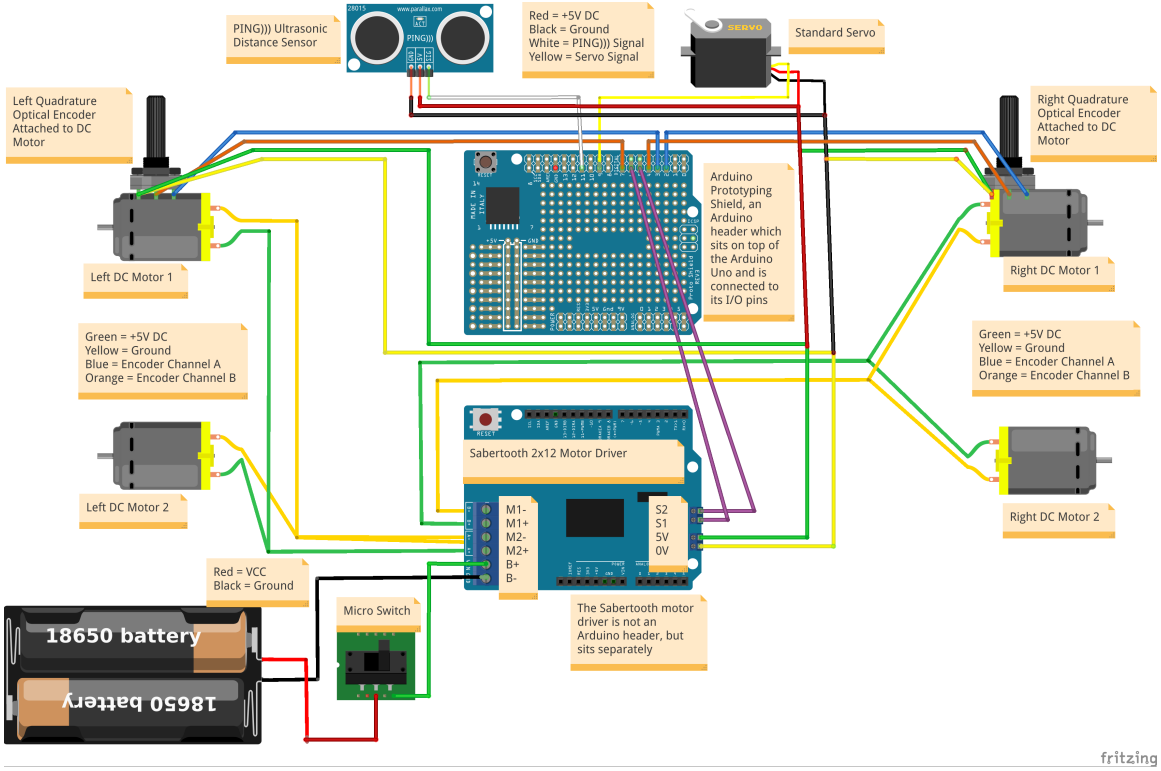
Connecting electronics on the rover to the laptop via USB means the processing laptop must manually be kept within 10 feet of the rover as it navigates. This design could easily be extended to include wireless or radio communication with a server, and a larger chassis would simply be able to carry the laptop on it. However, USB cables are cheap and still function as a proof of concept for an autonomous design.

Figure 3-7 is a schematic specifying the overall design for the rover.

Connections were made with flexible stranded core, 22 AWG breadboard wires. Connections to the Arduino's digital pins were made indirectly. A prototyping shield was stacked on top of the Arduino, and connected to its digital pins via pin headers. Signal pins were then soldered to the prototyping shield. Breadboard wires needing direct connection ideally should use terminal block connectors. However, these were difficult to find at a reasonable price, and so wires were soldered together tip to tip, and then wrapped in electrical tape.

The Sabertooth motor driver controls two motor channels. It drives DC motors from these channels in a relatively simple way. The speed of DC motors is proportional to the voltage supplied to them, and the direction of rotation can be flipped simply by flipping the polarity of the supplied voltage. The motor driver manages the voltage supplied to each channel by using pulse-width modulation, which involves switching

Figure 3-7: Connections Made



This image was created with Fritzing

the power on and off at a high frequency. This approximates a smooth waveform of the average voltage and current. An on-board H-bridge is used to flip the polarity. [7]

3.2.1 Power

Besides the Arduino Uno, which is powered separately by a USB connection to the laptop, most of the rover's components are powered by the battery pack. This pack contains two individual 18650 lithium-ion cells placed into a battery holder and connected in series. The battery pack is then connected to the motor driver's battery terminals B+ and B-. The positive B+ output goes through a microswitch, which is attached to a side bracket in the rover, and is accessible from the outside. This acts as a kill switch for the battery pack.

Each 18650 cell holds 4.2V at full charge, and discharges down to a minimum of 2.7V. The motor driver has a lithium cutoff mode which shuts the driver down when the average voltage of cells in the battery pack reaches 3.0V. Thus the voltage supplied to the four motors through the motor driver's output channels will range from 8.4V to 6.0V, which is within their acceptable operating range.

The specific li-ion cells being used can supply up to 20A continuously, and the motor driver can handle up to 12A per channel. The motors each draw a maximum current of 1.5A, and two are used per channel, putting the total possible current draw of 3A per channel well below the limits of the motor driver and battery pack.

The motor driver has an onboard battery eliminator circuit (BEC) which is an

efficient 5V voltage regulator capable of supplying up to 1 amp of continuous current, with 1.5 Amps at peak. The ultrasonic sensor, its servo, and the two rotary encoders combined use less than 500 mA, and are all powered through this BEC. The Arduino is also connected to this BEC's ground, as the Uno and Sabertooth must share a common ground plane in order for the control signals to be interpreted correctly [4]. It's important to note that a BEC of some kind is essential in this project, as the Arduino's on-board 5V regulator can not handle the peak amperage draw of the servo, and if used risks overheating.

The standard servo has the potential to draw peak currents of up to 1A, if it hits a snag and is stopped from moving. Therefore the input wires to the 5V and 0V BEC terminal connectors should be capable of handling those peaks. Since we are using 22 AWG wires, we are close to the limit, but a 22 AWG wire with 43 or more internal cores is rated to handle 1A. And the expected consistent draw is much lower, less than 500mA.

Note that the sensors attached to the microcontroller should be powered off before the Arduino, else the Arduino may try to power the whole Mega chip via its input pins. The sensors are powered from the BEC on the motor driver, so they may be turned off by using the microswitch between the battery holder and the motor driver.

Chapter 4

Arduino

The Arduino Uno in this project acts as a bridge between hardware and software, allowing the the laptop to read sensor data from the rover, and control the speed of its wheels.

4.1 Background

Arduino development boards are printed circuit boards capable of running small embedded programs. They contain an on-board microcontroller, timing crystal, USB port, I/O pins and more. The specific board used in this project, an Arduino Uno, uses the ATmega328P microcontroller with a 16 MHz quartz timing crystal and 14 digital I/O pins. It also has 6 analog-to-digital converter I/O pins, but we won't make use of them in this project.

Digital I/O pins can be configured to either read signals as input or generate them as output. Digital pins read input signals at specific times as binary values, i.e. the

connected signal's voltage is read as either on or off compared to a certain threshold voltage. Following the standard Arduino literature, we will refer to these signals as either HIGH or LOW. When digital pins are configured to generate HIGH or LOW signals, they produce a relative output voltage above or below the threshold voltage.

4.1.1 Servo Control Pulses

An important use-case which pops up often when using the Arduino is that of interfacing with RC electronics. In this project's design, both the Sabertooth motor driver and the standard servo require their signal inputs to use the standard R/C transmission protocol.

This protocol involves sending brief HIGH pulses of variable width, between one and two milliseconds. There is a fixed delay between pulses, commonly about 20 ms of LOW signal. The width of the HIGH pulse communicates to a servo the desired position. Its internal components then drive its DC motors until the servo is rotated to the commanded position. In the case of the Sabertooth motor driver, the position is interpreted as a speed to drive the motors at.

4.2 Arduino Uno Connections

Refer back to Figure 3-7 for a visual representation of how the Arduino Uno is connected to the other rover components.

4.2.1 Hardware Interrupt Pins

As one can see in Figure 3-7, only two optical quadrature encoders are used, placed on the front motors on the left and right side of the rover. This is due to a hardware limitation of the Arduino Uno. The ATmega328P microcontroller has only two interrupt pins, which are mapped to digital pins 2 and 3 on the Uno. These pins can trigger unique Interrupt Service Routines (ISRs) whenever the input signals change from LOW to HIGH voltage, or vice versa.

While it is possible to react to a change in any digital pin's voltage, it would be significantly slower than a hardware interrupt. An ISR is necessary to keep up with the fast rate of pin voltage changes that occur in the output of quadrature encoders.

If a different Arduino board such as the Mega were used, there would be sufficient hardware interrupt pins for all four encoders. Using a board with plentiful interrupts, one could even attach both channel outputs of the encoders to interrupt pins, rather than only one. This would double the encoders' resolution. See section 4.4.3 for more detail.

4.2.2 Digital Pin Connections

Each motor encoder has two output channels, A and B. Both encoders attach one of their output channels, channel A, to a hardware interrupt pin. In section 4.4.3 we will see why this configuration was chosen. The right motor's encoder connects channel A to pin 2, and channel B to pin 4. The left motor's encoder connects its channel A output to pin 3, and its channel B output to pin 7.

The S1 and S2 signal input terminals on the Sabertooth motor driver are connected to digital pins 5 and 6. The control signal for the hobby servo is connected to digital pin 9. The signal pin on the ultrasonic sensor is connected to digital pin 11. The Arduino's ground pin is connected to the ground of the motor driver's BEC, to ensure a common ground plane.

Most digital pin numbers used are arbitrary, and connections may be permuted without issue. The exceptions are pins 0-3, which must not be modified. Pins 0 and 1 must be left unattached for serial data transfer to work properly over USB. And pins 2 and 3 are hardware interrupt pins which must be used to handle the quadrature encoders' output.

4.3 Motor Driver's Configuration

The Sabertooth motor driver has two signal input terminals, S1 and S2, which allow the Arduino to issue instructions specifying how to drive the motors. The protocols used to communicate with the motor driver over these signal inputs are specified by six DIP switches on-board the driver. These DIP switches are manually flipped either up or down.

Setting switch 1 down and switch 2 up places the driver into R/C input mode, which configures S1 and S2 to expect servo control pulses, Ãă la R/C controllers. This protocol was briefly explained in section 4.1.1. [4]

Turning switch 3 down selects the lithium cutoff mode, which detects the number of lithium cells in series powering the driver, and shuts off when the battery pack's

voltage drops below 3.0V per cell, or 6.0V for the two cell battery pack this project uses. This prevents accidental damage to the 18650 cells which may be caused by over-discharge.

Flipping switch 4 down selects independent (differential) drive, which allows S1 and S2 to each independently control the speed of one motor channel. Using this mode, turning of the vehicle is achieved by lowering the relative speed of the motors on one side of the vehicle compared to the other.

Switch 5 is flipped up to ensure a linear rather than exponential response of the motors to the Arduino's input signal. Switch 6 is flipped down to select "microcontroller mode", which turns off auto-calibration of the zero-velocity input signal, and turns off an automatic timeout. Thus if the signal connection is somehow lost the motor driver will continue driving the motors according to the last signal received. This is necessary for smooth performance of the motors since the Arduino may slightly delay control pulses. Though this introduces a risk of loss of control should wires come disconnected, it is a small one that should only occur during a catastrophic crash.

4.4 Arduino Sketch

A sketch is Arduino-speak for an embedded program written for an Arduino board. There is an Arduino IDE which supports development of sketches in C or C++, and allows one to take advantage of a software library for common I/O interactions. After the code is written in this IDE, it is uploaded to the board over a USB serial

connection. The board will then continuously execute the code found in the sketch's main loop as long as the board is powered. This embedded software interacts with the various sensors and other electronics on a low level, through reading from and writing to the Arduino's digital I/O pins.

One of the standard Arduino libraries is the Servo library. This library allows one to configure a digital pin to output RC control pulses, as explained in section 4.1.1. The sketch used in this project uses this library to specify the speed of each set of wheels driven by the Sabertooth motor driver, and to control the position of the standard servo aiming the ultrasonic range sensor. The motor driver is sent pulses every 20 ms, with HIGH pulses from 1 ms to 2 ms. The standard servo is also sent pulses every 20 ms, with HIGH pulses from 0.75 ms to 2.25 ms as its datasheet specifies.

4.4.1 Ultrasonic Sensor

The PING))) ultrasonic distance sensor works by emitting a short burst of 40 kHz sound waves and timing the delay before an echo response. The Arduino triggers a ping by generating a brief 5 μs (microsecond) pulse on the sensor's bi-directional signal pin. The sensor then generates a HIGH output pulse, which continues until either the echo is received or the maximum amount of time, 18.5 ms, has passed. This time may then be multiplied by the speed of sound in air to calculate an estimated distance of the first object in front of the sensor. [8]

The sketch uses the NewPing library to handle this protocol [2]. This library

provides a convenient method, ping() which returns the echo time in μs. The sketch avoids costly floating point computations by sending this echo time over serial rather than a distance.

4.4.2 Servo

The ultrasonic sensor can only detect objects which are roughly straight in front of it. Thus for the rover to have a better approximation of its surroundings, the sensor needs to be panned back and forth. This is what the standard servo it is attached to allows. The sketch makes use of the Servo library to control the servo with R/C pulses. An angular degree from 0 to 180 is written to a Servo object, and the Servo library handles generating the output signal corresponding to that position on the appropriate digital pin.

The sketch sweeps the servo back and forth one degree at a time, and at each step an ultrasonic ping is emitted. The echo time for that ping is measured, and the current values of all sensors are published. This means that the delay between servo steps defines the publishing frequency of sensor data on the Arduino. This delay is currently set to 100 ms, which corresponds to a 10 Hz publishing frequency. The frequency could be increased for future work, but a bare minimum delay of 30 ms is necessary to give the servo time to finish moving, and the PING))) sensor time to recover and prepare for the next ping.

The basic idea is shown in the following code for an arbitrary servo step size:

```
while (servoPos < SERVO_LEFT) {
sonicServo.write(servoPos); // Set servo position
timer = millis(); // current time in ms
while (millis() - timer < SERVO_STEP_DELAY) {
nh.spinOnce(); // handle callbacks
}
ping_time_uS = sonar.ping(); // Get echo time
publishSensorMessages(servoPos, ping_time_uS);
servoPos += SERVO_STEP_SZ;
}
```

This code fragment is run inside the sketch's main loop, with a similar while-loop running right after, decrementing the servo position back to SERVO_RIGHT.

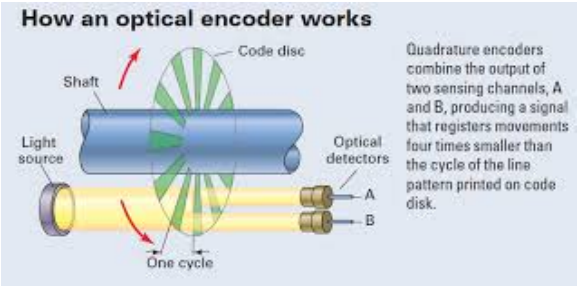
millis() is an Arduino built-in function which uses a hardware timer to count how many milliseconds have passed since the board was turned on. The callback handling that occurs while waiting reads the incoming serial buffer for any data, and if motor commands are found, an appropriate callback function is executed. publishSensorMessages() sends over serial the ping echo time, the servo's current angle, and the tick counts of both encoders. Section 5.2 will further describe how this data is passed between laptop and Arduino.

4.4.3 Quadrature Encoders

An important function of the Arduino sketch is to track the movement of the motors. Our system may command the motor driver to move the rover's wheels with a certain fraction of the maximum available power, but it is difficult to predict with precision the resulting angular velocity. For one thing, the RPM of DC motors is proportional to the supplied voltage. But the voltage supplied to the motors through the motor driver is coming from an external li-po battery pack, which generates variable voltage. It starts at 8.4V and drops to a minimum of 6.0V before the motor driver shuts off. Thus even if the same servo control pulse is continuously sent to the motor driver, the motors' angular velocity will decrease over time.

In order to determine the true angular velocity of the motors, rotary encoders are attached to them. These feedback devices are incremental position encoders, meaning they monitor the change in the motor shaft's position compared to some starting position.

Figure 4-1: [1]



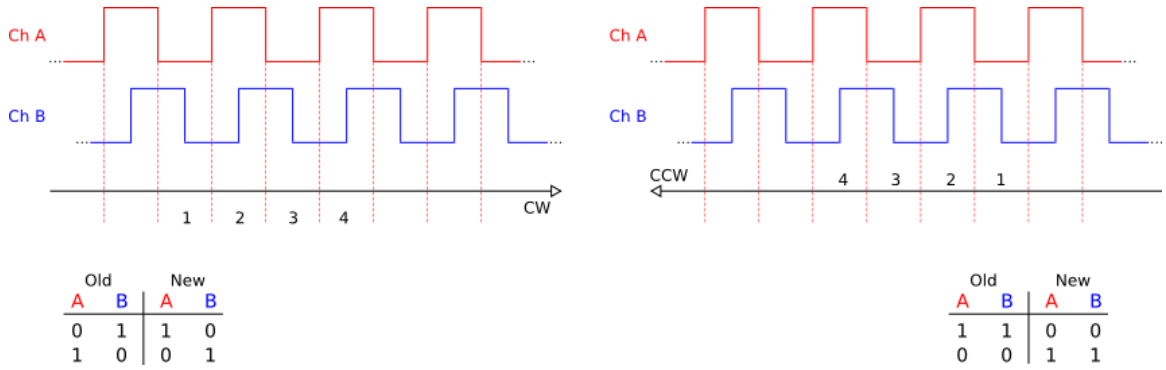
The motor encoders which came with the Lynxmotion rover kit are optical quadrature encoders. This type of encoder attaches a flat disk with thin slits known as the code disk to the motor's gear shaft. Two photodiodes, components which transform light into electric current, are placed above the disk side by side. A light source shines light through

the disk from the other side. See Figure 4-1 for a visual illustration.

As the motor spins the gear shaft, the code disk turns with it. This produces an on-off pattern of light on the photodiodes, which produce two square waves as signal outputs. These two channels of output pulses are referred to as channels A and B. Depending on the direction of rotation, channel A's square wave will either lag behind or be ahead of channel B. This can be seen in Figure 4-2 which shows example output pulses as a motor turns clockwise (CW) or counter-clockwise (CCW). [5]

The number of slits in the code disk corresponds directly to how many pulses each channel will produce in one revolution of the DC motor. This is known as the pulses per revolution (PPR), and is given by the manufacturer. By counting how many pulses occur in one second, and using the PPR, one can calculate the angular speed of the motor. If one wishes for greater resolution, they can watch each square wave for a change in voltage from LOW to HIGH or HIGH to LOW. This gives a maximum resolution of $4 * PPR$ detectable position increments per revolution.

Figure 4-2: [5]



Old		New	
A	B	A	B
0	1	1	0
1	0	0	1

Old		New	
A	B	A	B
1	1	0	0
0	0	1	1

Handling rapid changes in voltage is exactly what hardware interrupts are designed for. Unfortunately, for maximum resolution each encoder needs two hardware

interrupt pins, one for each channel. The Arduino Uno only has two hardware interrupt pins, and our rover has two sides. It would be nice if we could at least use two encoders, one for each side.

We achieve this by reacting to changes in voltage in only one channel per encoder. In the two tables in Figure 4-2, LOW voltage values are encoded as 0, and HIGH voltage values as 1. The first plot in the figure shows the output of the two channels when the motor is moving in the CW direction. When channel A transitions from the section labeled 1 to section 2, it is rising from 0 to 1, and channel B has value 0. That information alone tells us that the motor's gear shaft is turning, but not in what direction. However, at the next transition between sections 2 and 3, channel A falls from 1 to 0, and channel B has value 1. Now we are confident that channel B began its pulse after channel A. This means that the photodiode generating channel B detected light after channel A's photodiode, i.e. the code disk is turning in the direction of photodiode A to B. Datasheet specifications will tell us that this translates to the CW direction. It turns out that for each channel A transition event, the previous and new channel values are sufficient to uniquely determine the direction of rotation of the motor. Thus, while only monitoring one of the channels lowers our resolution to $2 * PPR$ counts per revolution, it allows us to use hardware interrupts for two quadrature encoders rather than only one. [5]

The sketch uses an implementation described in [5], which creates a lookup table using the four binary digits representing the previous and current channel states. These digits form a four-bit binary number, which indexes into a sixteen element array. Each element of this array stores either 1, -1, or 0, where 1 represents a

movement in the CW direction, -1 represents a movement in the CCW direction, and 0 represents an indeterminate transition. This lookup table is then used by the sketch when it reacts to a hardware interrupt caused by the channel A output of one of the encoders. When this interrupt occurs, the code reads the values of channels A and B from the corresponding digital pins, and combines them with the previous values to find the appropriate index in the lookup table. The value at that index is then added to a global counter variable, which keeps track of the net number of incremental movements from the motor's starting position. A net negative number indicates how far the motor has rotated in the CCW direction since it started, and a net positive number indicates how far the motor has rotated in the CW direction. [5]

The implementation described above is shown in the following C code of the ISR for the left encoder. [5]

```
volatile long encLeftCount = 0L;
const int8_t encoder_lookup_table[] =
{0,0,0,-1,0,0,1,0,0,1,0,0,-1,0,0,0};
void encoderLeft_isr() {
static uint8_t encLeft_val = 0;
encLeft_val = encLeft_val << 2;
encLeft_val = encLeft_val |
  ( ((PIND & 0b100) >> 1) |
  ((PIND & 0b10000) >> 4) );
```

```
    encLeftCount = encLeftCount +
      encoder_lookup_table[encLeft_val & 0b1111];
    }
```

When the digital pin connected to the left encoder's Channel A output changes, the main loop of the sketch is interrupted, and this ISR is executed. Until this ISR finishes, no other code is run, including other ISRs, though they may be flagged for future execution. Therefore ISRs must be as fast as possible, to not cause any interrupt events to be dropped, and to ensure that the main loop continues running smoothly.

To this end, this ISR makes use of constants and low-level C and avr microcontroller commands to ensure a speedy execution, at the price of readability. PIND is an avr command which returns input readings from digital pins 0-7 encoded into a byte. Bit shifting and masking are then used to extract and store the values of pins 2 (channel A) and 4 (channel B) into the two least significant bits of the encLeft_val variable. This variable is static, and so retains its value between ISR executions. The count is then incremented according to the lookup table.

When the sketch's main loop publishes sensor readings, it needs to publish the current encoder tick count for the right and left encoders. However, reading from multi-byte variables which are accessed within and without an ISR risks data corruption in the event that the ISR interrupts the main thread in-between readings of bytes. Since the encoder tick count variables are four bytes long, interrupt guards must be used around a read to make it atomic. These guards temporarily stop the custom ISR

from executing while the global variables are copied over to local ones. This is shown for the left encoder count in the following snippet from the sketch:

```
detachInterrupt(digitalPinToInterrupt(encLeftAPin));
encMsg.leftTicks = encLeftCount;
attachInterrupt(digitalPinToInterrupt(encLeftAPin),
  encoderLeft_isr, CHANGE);
t = 0;
```

Similar guards are used for the right encoder count variable.

Because we are turning interrupts off briefly, there is the risk that we could miss an interrupt event on one of the channel A pins. Missing an edge pulse from one of the encoders would not only lose that tick, but would also throw off the next value we index into the lookup table. Luckily, the Arduino has a single-bit interrupt event flag for every interrupt event. Therefore in the worst case scenario, an encoder interrupt event occurs just after the ISR handler is detached, and that event is flagged. As long as the ISR is reattached and handles that flag before a second interrupt event occurs, there won't be a problem. After reattaching the ISR, the next program instruction is guaranteed to be executed before handling any flagged events. To ensure speedy handling of a flagged event, a meaningless assignment of zero is made to the local variable t.

Let's calculate how often each interrupt event occurs. Each encoder generates 100 pulses per revolution. We will only be watching one of the square waves (output

channel A), so that's 200 edge transitions per revolution. The motors have a maximum speed of 200 RPM, so each encoder is guaranteed to revolve less than 3.4 times per second. Thus there will be at most

$$3.4\ rev/sec * 200\ events/rev = 680\ events/sec$$

And we are guaranteed to have at least $1/680 \approx 1.4\ ms$ between encoder interrupt events.

So the interrupt guards need to take significantly less than $1.4\ ms$ in order to allow a flagged interrupt event to be handled before the next event occurs. Each global encoder count variable is four bytes, so assignment compiles to four machine instructions. The assignment of zero to the one-byte variable t takes one machine instruction. The helper macro digitalPinToInterrupt() is a preprocessor #define, and so takes zero machine instructions. Therefore there are five total machine instructions executed before the ISR is finished. The Uno uses a 16 MHz quartz timing crystal, so executing one instruction takes

$$1/(16000000\ Hz) = 62.5\ nanoseconds$$

Thus to run the five instructions takes $5 * 62.5\ ns = 0.3125\ \mu s$. Then the flagged event must be handled. External interrupt calling has an overhead of $5.125\ \mu s$, to enter and leave the function [6]. Thus as long as the ISR executes in less than $1.4\ ms - 0.0003125\ ms - 0.005125\ ms = 1.3945625\ ms$, the sketch will never miss an

encoder event. Testing of the ISR indicates that this is an order of magnitude more time than needed.

Chapter 5

ROS

5.1 Overview

The Robot Operating System (ROS) is a meta operating system for open-source robotics.

It provides package management, an asynchronous messaging framework between processes

All software components in this project communicate using the asynchronous, distributed framework supported by ROS.

5.1.1 Nodes

Processes in ROS are known as nodes.

5.1.2 Messages

Messages are data structures which contain some amount of named data fields. publishing and subscribing messages

5.1.3 Topics

Messages are published to certain topic names. Processes subscribe to these topic names, and receive all messages sent on those names. Any number of processes may publish or subscribe to a topic, making this a many-to-many communication protocol.

The ROS Master acts like a DNS server, giving nodes references to each other. Let's use an example. Assume node A starts publishing messages to topic "/foo". When node A begins publishing, it will register that fact with the ROS master. If node B wants to subscribe to that same topic, it will communicate its intent to the ROS master. The master will then go through its list of all publishers to that topic, and inform them of node B's desire. In this case, node A is the only publisher, so the master will tell node A what TCP/IP socket to send its data to when it publishes to the "/foo" topic. From then on, when node A publishes a message to "/foo", it will go through its list of nodes subscribed to it, and send a copy of its message to each of those sockets.

Running nodes create a graph based on which nodes send data to which other nodes.

5.1.4 Packages

Packages are collections of related ROS resources to perform some function. They often include node source code, built executables, message definitions, and launch files.

There are also meta-packages which are, simply enough, collections of related packages.

5.1.5 Launch Files

Launch files are simply XML files which define several nodes to be run at the same time. Robotics systems grow large quickly, and dozens of nodes may need to be run to perform some task. Manually starting all of those nodes would be tedious, and launch files save that time. They also allow easy configuration and parameter specification for each node. Parameters are often stored separately in files using the YAML syntax, and loaded dynamically into the launch files. This allows one to bring a node up or down simply by modifying a launch file, but never losing all of the configuration settings for that node.

5.1.6 Frames

Frames are a set of reference coordinate axes.

A common starting frame is the base_link frame, which has its origin at the center of a robotic chassis. The x-axis of the base_link frame points forward, the y-axis points to the robot's left, and the z-axis goes straight up above the robot.

position and orientation quaternion vectors represent orientation messages have frame_ids detailing which frame they are in

5.1.7 Transforms

conversions between frames handled by the ROS package tf broadcasting transforms

5.1.8 Data Format Conventions

REP-103 and REP-105 for conventions true north, magnetic declination

ENU convention

5.1.9 User defined packages

auto_rover package Defines launch files, configuration files, message definition for EncCount, and a single range_converter node.

5.2 rosserial

Communication between the Arduino and laptop is handled with the ROS meta-package rosserial. Different client packages support different client machines, such as embedded linux devices, or different microcontroller boards. These client packages create local support libraries or header files on those machines, which use a serialization protocol to send and receive ROS messages over a serial port. On the other side of the serial connection, host packages run a bridging node which communicates with the ROS network on behalf of the client machine. Subscribed topics have their

messages serialized and sent to the client machine, and outgoing messages from the client are de-serialized and published.

5.2.1 rosserial_arduino

rosserial_arduino is one client package of rosserial, which creates an Arduino library to provide bare-bones ROS support to sketches. The sketch running on this project's Uno board uses this library to publish sensor data as messages, and subscribe to motor command topics.

Every time the sketch wishes to update the laptop with its newest sensor readings, it publishes three messages. First the servo angle in degrees is published to the "/ping/angleDeg" topic, as a standard Int8 message. This message just contains a single data field: an 8-bit signed integer. Next the echo time in microseconds is published to the "/ping/timeUS" topic, as a standard UInt16 message which contains a single 16 bit data field representing an unsigned integer. Lastly the two encoder tick counts are both placed into a single custom message called EncCount, and published to the "/odom/encTicks" topic. This custom message type has two 32 bit fields, one for each encoder. This message type will be further explained in section 5.6.1.

When the sketch is waiting between updates, it continually listens to the serial port for motor commands. These commands are Int8 messages on the "/cmd/left" or "/cmd/right" topics, which the sketch subscribes to. When these messages are found in the serial input buffer, a short callback function is executed, which writes the R/C pulse command to the proper motor channel.

Arduino boards use different types of memory. Flash memory is used to store sketch code, and static random access memory (SRAM) is used to store dynamic variables at runtime. The Uno has 32kB of flash memory, but only 2kB of SRAM. The rosserial Arduino library is large, and takes up quite a lot of SRAM space. Its input and output serial buffers alone use 560 bytes. This makes running out of space for local variables quite easy, which can lead to instability and crashes when running the sketch. To save space, a modified version of rosserial_arduino which supports storing constant strings in flash memory rather than SRAM has been used. Since topic names and error messages use long descriptive strings, this saves several hundred kB of space in SRAM and ensures the sketch's stability.

The rosserial Arduino library abstracts away most of the serial communication protocol, but does allow the baud rate to be specified. In this use case, baud rate is equivalent to bits per second. The more bits per second sent over serial, the more frequently the microcontroller needs to sample the incoming and outgoing line. So the baud rate cannot be set arbitrarily high, as the Uno has a limited clock speed. If it is set too low, however, then the stream of sensor data being published would overwhelm the connection. Significantly less data will be streaming in than transmitted out, so the amount of outgoing data is the deciding factor. Thus to calculate an appropriate baud rate, the amount of sensor data transmitted per second must be known.

rosserial uses a serial protocol with 8 bytes of overhead for every message. Each sensor update publishes three messages: an eight-byte EncCount message, a one-byte Int8 message, and a two-byte UInt16 message. This means that each update pushes 11 bytes of data in three messages, with 24 bytes of overhead. Thus a total of 35

bytes are sent over serial.

Since the PING))) sensor requires a minimum delay of 30 ms between pings, the sketch cannot publish its sensor values at a rate higher than 33 Hz. Therefore the sketch will not push more than:

$$33\ Hz * 35\ Bytes = 1155\ Bytes\ per\ second\ (Bps)$$

The Uno uses one start bit and one stop bit to surround each byte of information sent over serial. Thus it takes 10 bits to send one byte of information. Therefore the minimum baud rate required is:

$$1155\ Bps * 10\ bits\ per\ byte = 11550\ bits\ per\ second$$

We'll choose a standard baud rate of 28,800 to more than double that for some breathing room, and to account for the fact that the rosserial Arduino library occasionally transmits time-keeping and synchronization messages of its own.

5.2.2 rosserial_python

rosserial_python is one host package of rosserial, which acts as a bridge between the Arduino and the ROS network. It runs a node on the laptop which communicates with the Arduino using the rosserial protocol. It automatically handles setup, communication with the ROS master, subscription, and publishing on behalf of the Arduino. When launched, the serial node must be configured to use the same baud

rate as the Arduino: 28,800. It must also be configured to connect to whichever serial port name the Arduino uses. For simplicity, a symbolic link was created using a udev rule on the laptop, to ensure that the port name will always be accessible as "/dev/arduino".

5.3 differential_drive

The differential_drive package was created by Jon Stephan to create a simple interface for controlling a differential wheeled robot []. Such a robot uses a two-wheeled system where both wheels are on a common axis, but each wheel is driven independently. Turning is achieved by lowering the velocity of one wheel compared to the other.

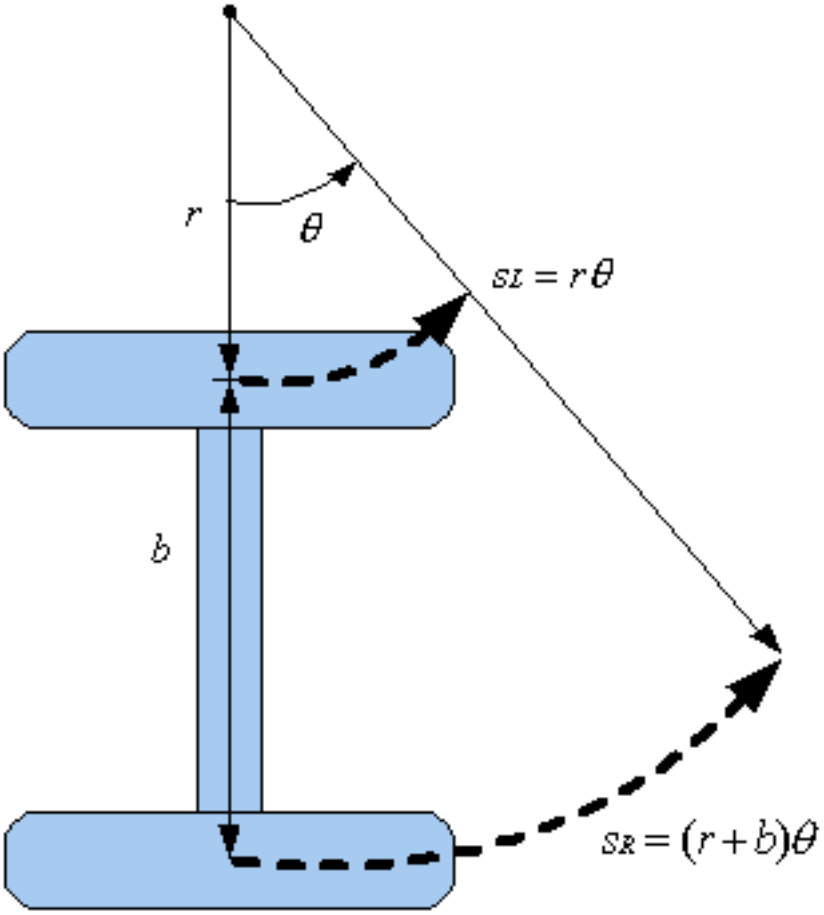
Because the rover has four wheels, turning necessarily involves slippage of one or more wheels. This is known as a skid-steering system, due to the skidding of the wheels. When wheels slip, they move without rotating. This causes error in the quadrature encoder values, and makes it difficult to properly estimate the distance the rover has traveled. Despite this flaw, this project's rover is modeled as a differentially steered robot for the purposes of dead reckoning, due to the simplicity of the kinematic model.

5.3.1 diff_odom

Odometry messages are a type of ROS message used for navigation. They represent an estimate of the position and velocity of the rover at a certain time. The diff_odom node subscribes to encoder tick data, and uses that data to calculate and

publish an Odometry message to the "odometry/wheel" topic. The Odometry messages contain estimates of the rover's position, orientation, and linear and angular velocity with a timestamp. This node is a modified version of the diff_tf node from the differential_drive package, changed to use the custom EncCount message type, set appropriate covariance values, and not publish an odom->base_link transform. This transform is published by the EKF node after fusing all sensor data, which is described in section 5.5.

Figure 5-1: [10]



First, let's take a look at the standard theory for differential wheeled robots. Figure 5-1 shows a simple two-wheeled robot making a left turn of θ radians around

some point. We assume the robot is one rigid body, and that each wheel maintains a constant velocity along the turn. This assumption of zero acceleration is obviously violated in the real world, but robots with a small mass and relatively powerful motors are able to approximate it well. [10]

r is the turning radius for the left wheel, and $(r + b)$ is the turning radius for the right wheel, where b is the distance between wheels. Using the formula for arc length, we know the distance traveled by the right and left wheels. Define a point M to be at the midpoint of the two wheels. This point will then travel an arc length of $(r + (b/2)) * \theta$. We can manipulate s_L and s_R to produce the following two equations.

$$s_M = ((r + (b/2)) * \theta = (s_L + s_R)/2 \tag{5.1}$$

$$\theta = (s_R - s_L)/b \tag{5.2}$$

Equation 5.1 gives the distance the center of the robot travels over the turn, in terms of the distance the two wheels traveled. Dividing this distance by the elapsed time it took to make the turn, gives an estimate of the instantaneous velocity of the robot at the end of the turn. Similarly, equation 5.2 calculates the angle of the turn using the distance the two wheels traveled. Dividing θ by the elapsed time gives the angular velocity of the robot.

Both calculations make use of the distance traveled by the two wheels. The two quadrature motor encoders attached to the rover's front wheels give a certain number of counts per revolution, and the wheel circumference is given from the diameter.

Thus the distance traveled can be surmised from the difference between the encoder ticks counted in the last sensor update, and those in the most recent update. This node uses these values to calculate the rover's angular and linear velocity. Only the angular velocity along the rover's z-axis is reported; the angular velocities along the other axes are assumed to be zero. Because a differentially steered robot can only move in the direction of its fixed wheels, the linear velocity is reported along the base_link x axis, which faces forward from the rover's midpoint.

The Odometry message includes a six by six covariance matrix for the linear and angular velocities along all three axes. These covariances give the EKF node an idea of how much to trust these velocity estimates. Since only two velocities are calculated, reasonable constant variances squared for those two values are filled in along the matrix's diagonal. Every other element is set to zero, since they are ignored by the filter.

Because dead reckoning estimates are naturally noisy and drift quickly, this project's EKF ignores the pose element of this node's Odometry message entirely. Therefore we don't bother with calculating the positional update estimate, or its covariance matrix.

This node is configured to publish Odometry messages at a rate equal to the Arduino's sensor update rate. If it published at a slower rate, then some resolution would be lost as the distance traveled is calculated from the difference between the most recent tick count, and the tick count used in the calculation of the prior Odometry estimate.

Though the number of encoder ticks per meter may be calculated from the en-

coders' specification and the diameter of the wheels, it is a good idea to manually calibrate the number of encoder ticks per meter, and pass this as a configurable parameter to this node. This helps account for sources of error. The ticks per meter can easily be calibrated by moving the Arduino one meter, while counting the number of encoder ticks.

5.3.2 twist_to_motors

Many ROS navigation packages produce Twist messages to command robotic platforms. The Twist message includes a linear and angular velocity, which the rover is expected to match, as a subfunction of some path following algorithm. The differential_drive package uses the twist_to_motors node to translate Twist messages into individual motor velocities for each motor channel.

Taking into account the differentially steered conditions, this node only considers the linear velocity along the rover's x axis, and the angular velocity around the rover's z-axis. Let's refer to these as x' and θ', respectively. Let b once again be the distance between the rover's wheels. Let L' and R' be the velocity of the left and right wheels. From equation 5.1, we know that

$$x' = (L' + R')/2$$

and from equation 5.2 we know that

$$\theta' = (R' - L')/b$$

This is a system of two equations with two unknowns, L' and R'. Solving this system gives us:

$$L' = x' - (b * \theta')/2$$

$$R' = x' + (b * \theta')/2$$

This node uses these equations to calculate the appropriate wheel velocities from the incoming Twist message, and publishes those velocities to the "lwheel_vtarget" and "rwheel_vtarget" topics.

5.3.3 pid_velocity

The pid_velocity node creates a proportional-integral-derivative (PID) controller which uses encoder feedback to translate motor velocities to actual motor R/C pulse commands. While the appropriate R/C servo command to reach a desired velocity could be estimated from the Sabertooth motor driver's datasheet, this would be the theoretical value and wouldn't take into account real-world sources of error such as uneven terrain, high traction, wind drag, etc. Therefore a control loop which utilizes real-time feedback is preferable.

Two of these nodes are run, one for each motor channel. One node subscribes to the topic "lwheel_vtarget", and publishes commands to "/cmd/left", while the other node subscribes to "rwheel_vtarget", and publishes to "/cmd/right". The Arduino node, through its bridge, is subscribed to these two topics and handles their messages appropriately.

PID controllers work by adjusting their output according an error term, which is the difference between the current feedback and the desired value. This error, the integral of all past error terms, and the estimated error of this derivative are all combined into a weighted sum. This sum then acts as the new output of the system.

The weights in this sum are three constant parameters: K_p, K_i, and K_d. These parameters must be manually tuned to the target system for optimal use of the controller. The tuning procedure involves zeroing out K_i and K_d, and slowly increasing K_p until oscillation is observed in the control loop. Once a limit is found, set K_p to half of it. Then tune K_i and lastly K_d, in the same fashion.

5.3.4 virtual_joystick

For manual driving of the rover, the differential_drive package provides a joystick node, which brings up a simple GUI. This app allows the user to drag their mouse along a simple two-dimensional axis representing a desired linear and angular velocity, and publishes the corresponding Twist message. This is useful for manual testing of the rover, but will not be needed once autonomous navigation is fully functional. This node requires installation of PySide, the python binding for the Qt framework.

5.4 Ros Sensors App

Android app to publish IMU and GPS data from a smartphone.

phone frame

5.4.1 GPS

GPS receivers background coastguard broadcasts fix message for gps, don't have hardware to access that though

navsatfix message covariance matrix

5.4.2 IMU

IMU chips background: magnetometers, accelerometers, gyroscopes what a quaternion is

publishes IMU message

IMU messages are expected to be in ENU reference frame, instead of NED.

Android TYPE_ROTATION_VECTOR sensor fuses magnetometer and accelerometer data to produce an quaternion representation of an orientation in the ENU frame.

Android accelometer readings tell us the linear acceleration of the rover, but are reported with respect to the local orientation of the phone, and so the node converts them to the ENU orientation before publishing them, in the phone frame (which just specifies a static translation)

Android gyroscope readings tell us the angular velocity of the rover, but are reported with respect to the local axes of the phone. However, our rover operates solely in 2D, and so we aren't interested in angular velocities of roll or pitch. And since the phone lays face-up on top of the rover, the phone's local z axis is the same as the ENU up axis, and so we don't need to transform the gyroscope reading for rad/sec rotation of yaw.

covariance matrices

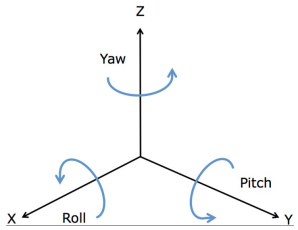
5.4.3 How to use

usb tethering on linux

5.5 robot_localization

This package created by Tom Moore implements the extended Kalman Filter, the mathematical details of which were described in section 2.3. The package keeps track of a 15-dimensional vector describing the rover's state: $(x, y, z, \Phi, \theta, \Psi, x', y', z', \Phi', \theta', \Psi', x'', y'', z'')$. In this state vector Φ, θ, Ψ represent roll, pitch, and yaw, respectively. These Euler angles describe rotation about the X, Y, and Z axes. See Figure **??**.

Figure 5-2: Roll, Pitch, and Yaw [3]



[12]

5.6 auto_rover

custom package this is where custom message type, EncCount is defined, and header files are created

5.6.1 EncCount

5.6.2 range_converter

ping sensor at ultrasound frame

Adjust code for conversion of ultrasonic sensor ping times to distance, to take into account the ambient air temperature If an echo has been received, then the speed of sound in air in m/s, C_{air}, is calculated using the current air temperature in Celsius T_C:

$$C_{air} = 331.5 + (0.6 * T_C)*$$

Multiplying C_{air} by the duration of the timed output pulse gives the estimated distance of the first object in front of the sensor.

Subscribes to /ping/timeUS and /ping/angleDeg topics, reads servo angle and ping time from those topics. Those messages sent from arduino to minimize amount of data being sent over serial. Then this node converts that to a distance in meters and an ultrasound -> base_link transform. transform uses static translation from center of rover to center of PING))) sensor, as well as instantaneously changing rotation at each ping snapshot.

5.6.3 EKF Configuration

Contains launch files which control rover's movement, and state estimation

Assumes a two-dimensional environment, which cause z,roll, pitch, roll', pitch', z',

Table 5.1: Sensor Configurations [12]

State Variable / Sensor	x	y	z	Φ	θ	Ψ	x'	y'	z'	Φ'	θ'	Ψ'	x''	y''	z''
Wheel Encoders	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0
IMU	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1
GPS	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

z" to all be fixed at 0. IMU roll, pitch, and GPS altitude measurements are ignored.

First ekf_localization node takes in odometry estimate from wheel encoders via diff_odom node (section 5.3.1), and IMU message from phone node. Produces a fused estimate of odometry.

navsat_transform_node takes in odometry message from first ekf node, which is the robotâĂŹs current position estimate in the frame specified by its start location. It also takes in the navsatfix and imu messages from the phone, and fuses all these to produce a different odometry estimate which is the gps data converted to the coordinates of the robot's world frame.

Second ekf_localization node fuses the gps and odometry outputs from the previous two nodes into a final odometry message which is the final estimation of the robot's current state.

[12]

Chapter 6

Field Test

In order to test the rover's ability to localize itself, a simple field test was conducted in a parking lot. Inspiration for this experiment comes from Moore and Stouch [12].

6.1 Experiment Design

The rover was initially placed in a parking lot oriented facing west. It was then driven in a roughly rectangular shape around the lot using the virtual_joystick node, described in section 5.3.4. During this time the raw sensor data streaming in from the Arduino and phone was recorded and saved into a ROS bag file. The rover was driven in a loop, such that its initial and ending position and orientation were roughly equal. Total collection time was five and a half minutes.

See Figure 6-1 for two representations of the path taken. Figure 6-1a displays the path actually traversed, while Figure 6-1b is constructed from the phone's GPS readings. At no point did the rover go onto the grass.

The ROS *rosbag* utility was then used to repeatedly simulate the recorded sensor messages, while the EKF was run in different configurations. The rover's state was computed from raw wheel odometry, from wheel odometry fused with the IMU data, and from wheel odometry, IMU data, and GPS fixes all fused together. Refer to Table 5.1 for a review of which state variables each sensor affects.

6.2 Results

During each filter computation, a state estimate was produced at 30 Hz in a local frame, and that output was then transformed into a global frame, where position is given as latitude and longitude. These gps coordinates were then plotted using the handy GPS Visualizer tool [14]. Figure 6-2 displays those plots.

Figure 6-2a shows the estimated path when fusing only the wheel encoder output. The initial upward trajectory and left turn are tracked reasonably well, but the second left turn rotates too far and throws the rest of the estimate off.

Figure 6-2b shows the path generated when fusing wheel odometry with IMU data. In this case the shape of the path is much closer to truth, though the initial right turn is under-estimated.

Lastly, Figure 6-2c shows the result of fusing both previous sensors with GPS fixes. This plot looks much like the raw gps plot in Figure 6-1b, however upon close inspection one can see jagged jumps in position. These jumps are instantaneous and actually lead to a discontinuous position estimate, though the visualizing tool connects every point. They are caused by the filter instantaneously adjusting the

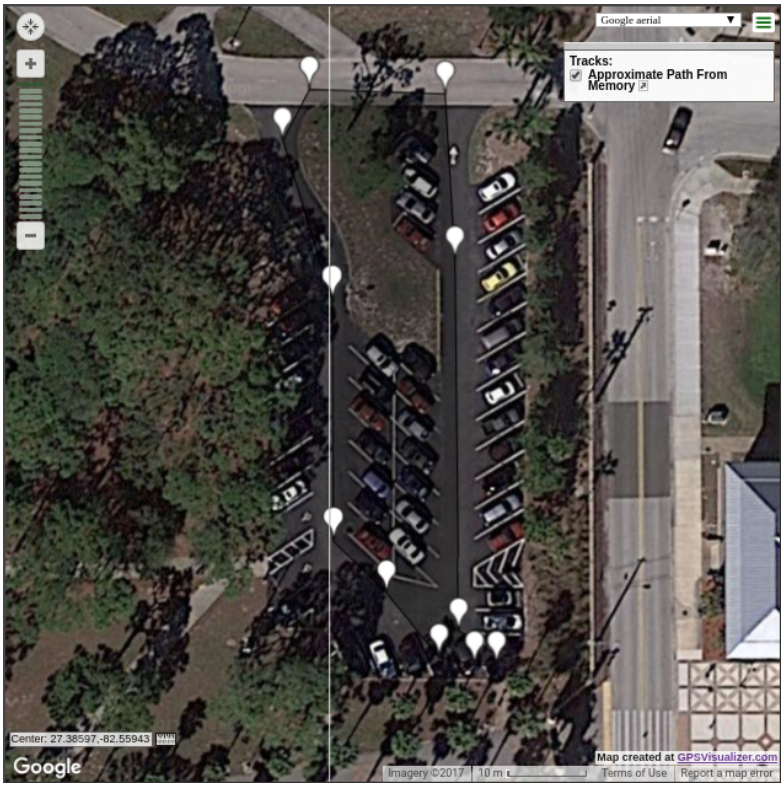
position estimate based on an incoming gps fix. The filter does give weight to the current estimate, so the new adjusted position lies in between the gps fix and the old position estimate. Due to the frequent gps fixes and slow velocity of the rover, the estimated path does not vary too far from the raw gps path.

Table 6.1: Errors for Different Sensor Fusions [12]

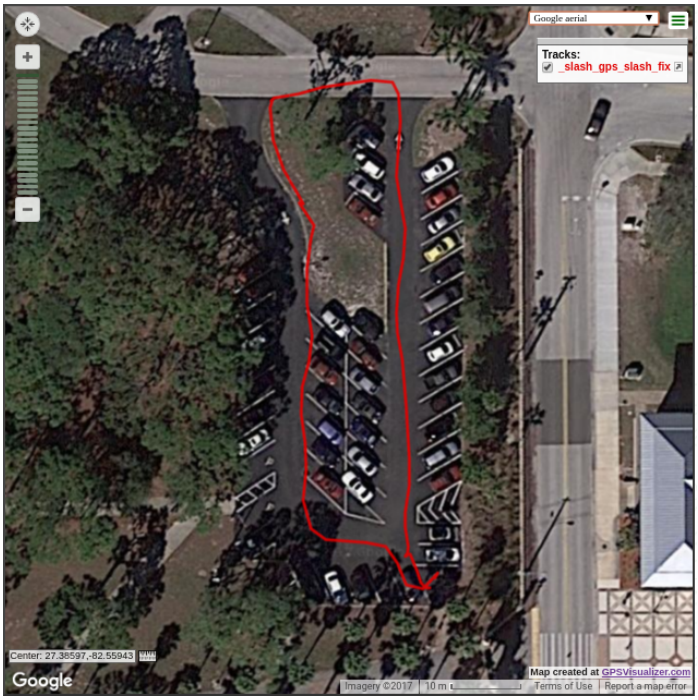
Sensors Fused	**Loop Closure Error x,y (m)**	**Filter's Std. Dev. x,y (m)**
Wheel Encoders	-88.37, -43.10	45.64, 126.36
Encoders + IMU	-12.90, -11.89	52.80, 52.02
Encoders + IMU + GPS	-0.97, -0.50	4.68, 4.56

Table 6.1 shows the position error between the rover's start and end positions for each filter configuration. Because the rover's local frame has its origin at the start point, this error is simply the last state estimate produced by the filter. The standard deviation for each dimension is also reported, giving an idea of the filter's confidence in its location. Note that the position errors are negative because the filter considers the end point to be behind and to the right of the rover's starting orientation.

Figure 6-1: The rover's path.

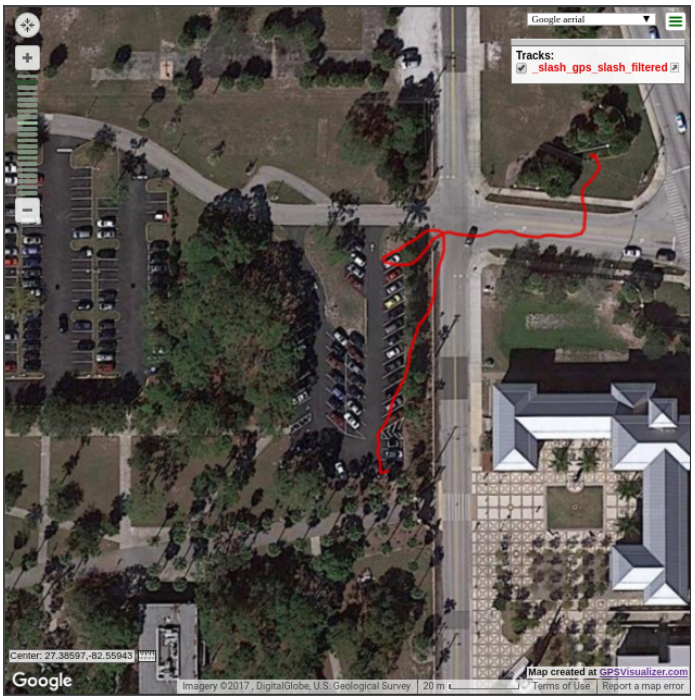


(a) Path Manually Mapped



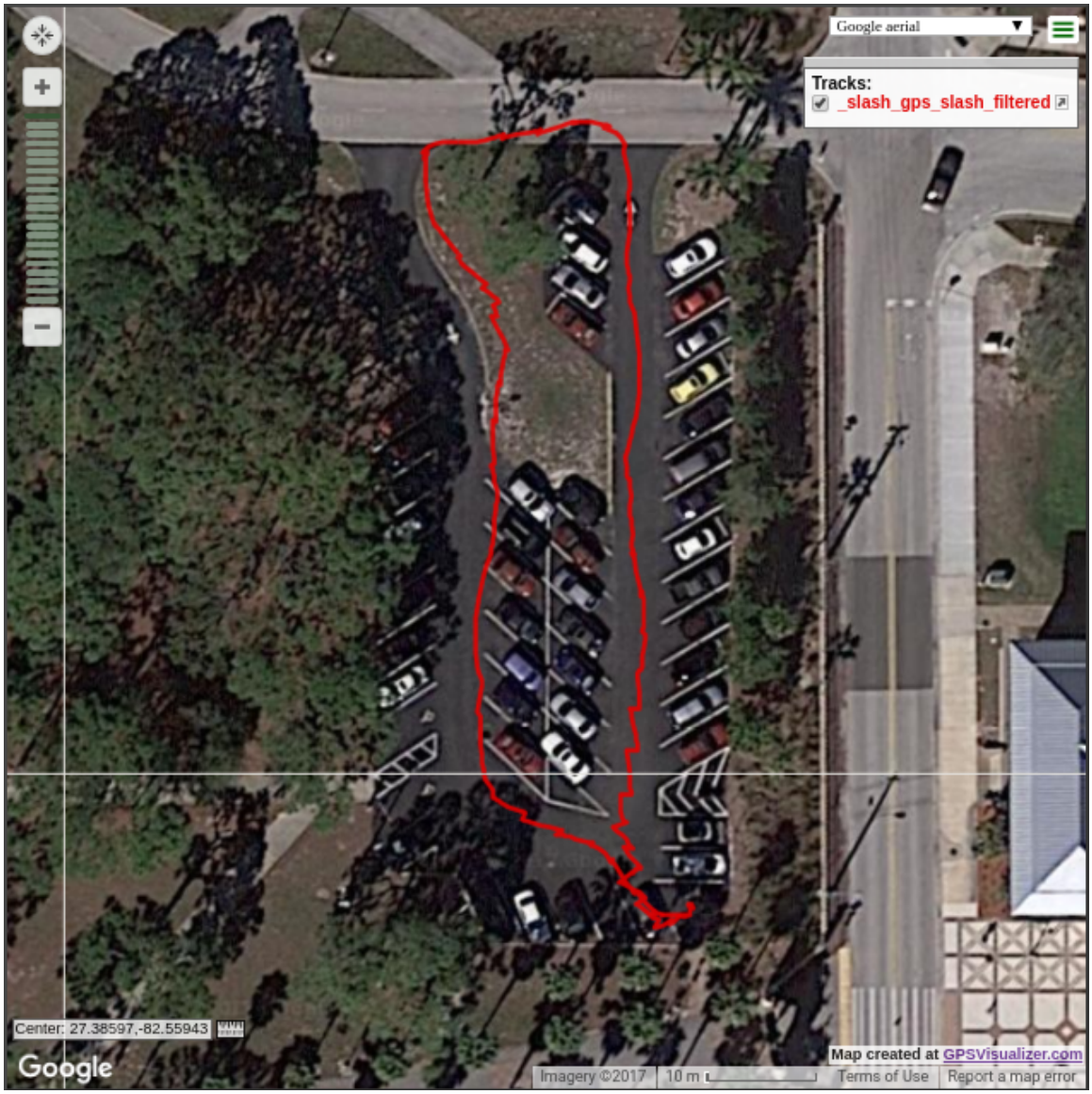
(b) Path According to Phone's GPS

Figure 6-2: Filter Outputs For Different Sensor Fusions



(a) Raw Wheel Odometry

(b) Wheel Odometry + IMU



(c) Wheel Odometry + IMU + GPS

Bibliography

[1] Machine Design. *quadrature-signals-encoders-basics*. [Online: accessed April 24, 2017]. 2004. URL: http://www.machinedesign.com/sites/machinedesign.com/files/uploads/2013/04/quadrature-signals-encoders-basics.png.

[2] Tim Eckel. *NewPing Library for Arduino*. Library. 2016. URL: http://playground.arduino.cc/Code/NewPing.

[3] Katherine Ellis et al. *Average roll, pitch, and yaw angles*. [Online: accessed July 7, 2017]. URL: https://www.researchgate.net/publication/262055313_Identifying_Active_Travel_Behaviors_in_Challenging_Environments_Using_GPS_Accelerometers_and_Machine_Learning_Algorithms.

[4] Dimension Engineering. *Sabertooth 2x12 User's Guide*. English. 2012. 21 pp. URL: http://web.archive.org/web/20160324141917/http://www.robotshop.com/media/files/pdf/user-guide-sabertooth-2-12.pdf.

[5] Nich Fugal. *Efficiently Reading Quadrature With Interrupts*. Blog. 2013. URL: https://web.archive.org/web/20140515005614/http://makeatronics.blogspot.com/2013/02/efficiently-reading-quadrature-with.html.

[6] Nick Gammon. *Interrupts*. Online. 2012. URL: https://web.archive.org/web/20170218142257/gammon.com.au/interrupts.

[7] Tom Igoe. *Controlling DC Motors*. Blog. 2017. URL: http://web.archive.org/web/20160915053759/http://www.tigoe.com:80/pcomp/code/circuits/motors/controlling-dc-motors/.

[8] Parallax Inc. *PING))^{TM} Ultrasonic Distance Sensor (#28015) Documentation*. English. 2009. 12 pp.

[9] Adafruit Industries. *Arduino Uno R3 (Atmega328 - assembled)*. [Online: accessed April 13, 2017]. 2017. URL: http://web.archive.org/web/20170321022401/https://www.adafruit.com/product/50.

[10] G.W. Lucas. *A Tutorial and Elementary Trajectory Model for the Differential Steering System of Robot Wheel Actuators*. URL: http://web.archive.org/web/20170120154230/http://rossum.sourceforge.net/papers/DiffSteer/.

[11] Lynxmotion. *Lynxmotion Aluminum A4WD1 Rover Kit (w/ Encoders)*. [Online: accessed April 11, 2017]. URL: https://web.archive.org/web/20151208053255/http://www.robotshop.com/en/lynxmotion-aluminum-a4wd1-rover-kit-w-encoders.html.

[12] T. Moore and D. Stouch. "A Generalized Extended Kalman Filter Implementation for the Robot Operating System". In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, July 2014.

[13] Parallax. *Parallax PING Ultrasonic Sensor*. [Online: accessed April 13, 2017]. 2017. URL: http://web.archive.org/web/20161208012823/http://www.robotshop.com/en/parallax-ping-ultrasonic-sensor.html.

[14] Adam Schneider. *GPS Visualizer*. URL: http://web.archive.org/web/20170704153136/http://www.gpsvisualizer.com/.

[15] Gabriel A. Terejanu. *Extended Kalman Filter Tutorial*. University at Buffalo. URL: https://web.archive.org/web/20151224081427/https://homes.cs.washington.edu/~todorov/courses/cseP590/readings/tutorialEKF.pdf.

[16] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. The MIT Press, 2005.

[17] *USB 2.0 Frequently Asked Questions*. Online. 2017. URL: http://web.archive.org/web/20170225172623/http://www.usb.org/developers/usbfaq#cab1.