CHEAP OUTDOORS AUTONOMOUS NAVIGATION WITH ROS

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

Cheap Outdoors Autonomous Navigation with ROS

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

In this thesis, I designed and constructed an unmanned ground vehicle for the purpose of autonomous navigation.

Professor Gary Kalmanovich . . . .

May 18, 2017

Chapter 1

Introduction

1.1 Goal

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.

1.2 Navigation

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.

1.3 Choice of Sensors

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.

Chapter 2

Theory

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.

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 $p(x) = \sum\limits_{y \in Y} p(x, y)$. 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 $t = 0, 1, 2, \ldots$. 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. [15]

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 $z_{1:t} = z_1, z_2, \ldots, z_t$. [15]

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 [15]:

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

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 [15]:

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

$$= \sum_{x_{t-1}} p(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}) p(x_{t-1} \mid z_{1:t-1}, u_{1:t})$$

$$= \sum_{x_{t-1}} p(x_t \mid x_{t-1}, u_t) p(x_{t-1} \mid z_{1:t-1}, u_{1:t})$$

$$= \sum_{x_{t-1}} p(x_t \mid x_{t-1}, u_t) bel(x_{t-1})$$

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.* [15]

$p(z_t \mid x_t)$ 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.* [15]

Once we have models for both the *state transition probability* and *measurement probability*, we can finally construct the algorithm known as Bayes' 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 $\eta$
7:     Use $\eta$ to compute $bel(x_t)$
8:     return $bel(x_t)$
9: end function
```

2.2.3 Example

2.3 Kalman Filter

2.3.1 Extended Kalman Filter

Chapter 3

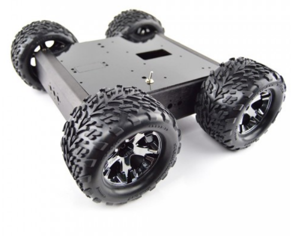
Hardware

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.

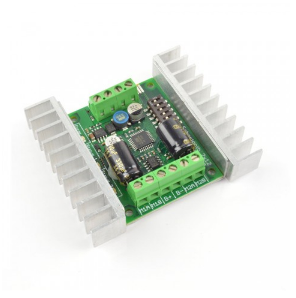
The mobile base used is the Lynxmotion A4WD1 Rover, see Figure 3-1. This kit comes with four 200 RPM DC gear motors, 100 PPR motor encoders, and 4.75" diameter wheels. The chassis consists of four aluminum side brackets, and two polycarbonate panels on the top and bottom. The rover is rated to carry up to 5 pounds.

Figure 3-1: Lynxmotion 4WD Rover [12]



The motors are controlled by a Sabertooth dual-channel 12A 6V-24V regenerative motor driver, see Figure 3-2. This motor driver is powered by two LG 18650 HE2 rechargable lithium ion cells, which sit in an 18650 battery case which has been soldered to act as a battery pack with two 18650 cells in series. The battery cells are individually charged before use with a NiteCore-i2-V2014 li-ion charger.

Figure 3-2: Sabertooth 2x12 [5]

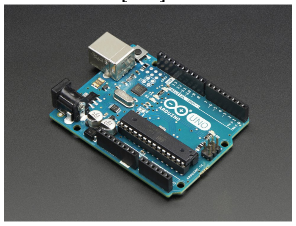


On top of the rover is the PING))) ultrasonic distance sensor (see Figure 3-3), which is attached to a standard Parallax servo which pans back and forth 180 degrees. This range sensor emits an ultrasonic chirp, and times how long it takes for that chirp to echo back. Based on that time, the distance from the sensor to an obstacle can be calculated. The PING))) sensor can be used to detect objects from 2cm to 3 meters away. [10]

Figure 3-3: PING))) Ultrasonic Sensor [14]

At the center of the rover is an Arduino Uno R3, see Figure 3-4. 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 3-4: Arduino Uno R3 [11]



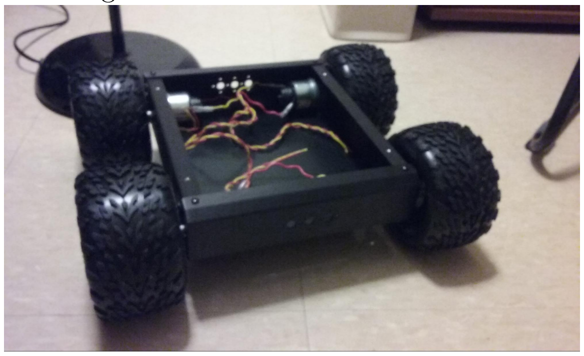
The specific laptop used in this project is the Dell Inspiron 3531, which has a quad core 2.16 GHz processor, and 4 GB of RAM. Any personal laptop running Ubuntu or Debian could be used here, and additional computational resources would be beneficial. However, this laptop was a personal work machine 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

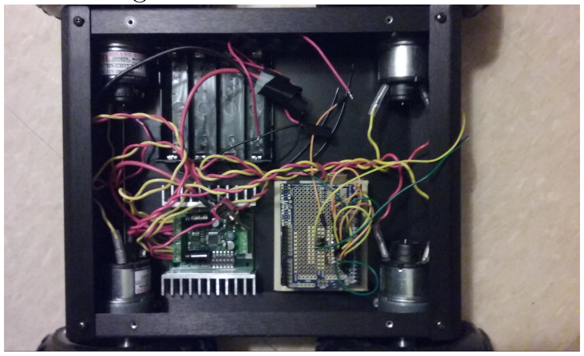
Figure 3-5 shows the base just after assembly. The aluminum side brackets' mounting holes did not line up properly with the motors, so a Dremel drill was used to widen them.

Figure 3-5: Constructed Chassis



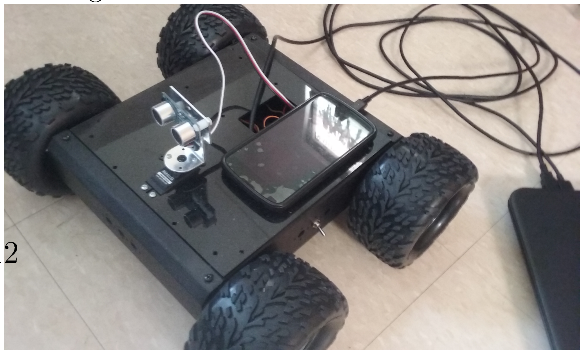
The Arduino Uno was screwed to a 2.5" x 3" x 0.5" wooden poplar block, with non-conductive nylon washers placed between the screw head and the Uno, and between the Uno and the wooden block. The wooden Arduino mounting board and the Sabertooth were both attached via double-sided foam mounting tape to the bottom panel of the rover. The battery holder was attached with glue dots to make removal easier.

Figure 3-6: 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.

Figure 3-7: Construction Finished



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.

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.

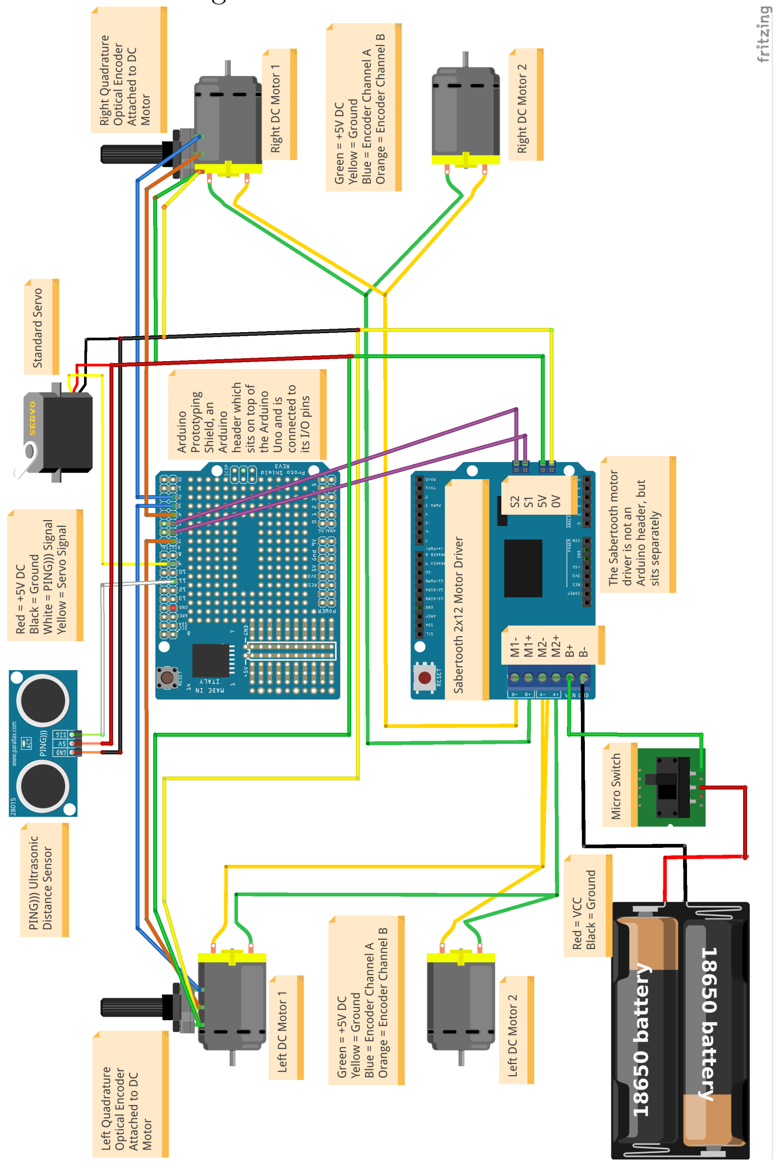
Both USB cables are long, stretching to just under 10 feet. The USB 2.0 specification limits the length of cable between two 2.0 USB devices to less than five meters, or about 16 feet [16]. Thus there should be no problem with the current length, but extensions in the future could not go much further.

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 3-8 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.

Figure 3-8: Connections Made


Right Quadrature Optical Encoder Attached to DC Motor
Right DC Motor 1
Green = +5V DC
Yellow = Ground
Blue = Encoder Channel A
Orange = Encoder Channel B
Right DC Motor 2
Standard Servo
Arduino Prototyping Shield, an Arduino header which sits on top of the Arduino Uno and is connected to its I/O pins
Red = +5V DC
Black = Ground
White = PING))) Signal
Yellow = Servo Signal
Sabertooth 2x12 Motor Driver
S2
S1
5V
0V
M1-
M1+
M2-
M2+
B+
B-
The Sabertooth motor driver is not an Arduino header, but sits separately
Micro Switch
Red = VCC
Black = Ground
PING))) Ultrasonic Distance Sensor
Left Quadrature Optical Encoder Attached to DC Motor
Left DC Motor 1
Green = +5V DC
Yellow = Ground
Blue = Encoder Channel A
Orange = Encoder Channel B
Left DC Motor 2
18650 battery
18650 battery
fritzing

This image was created with Fritzing

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 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. [9]

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 navigation stack on 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 (PCBs) with an on-board microcontroller, timing crystal, USB port, I/O pins and more. The board used in this project, an Arduino Uno, uses the ATmega328P microcontroller with a 16 MHz quartz timing crystal and has 14 digital I/O pins. It also has 6 analog 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, that is the connected signal's voltage is read as either LOW or HIGH. They are also capable of generating binary output voltage, HIGH and LOW signals.

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. In fact, the standard servo contains among other components an internal motor driver, and so

This protocol involves sending brief pulses of a HIGH signal, 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 3-8 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 3-8, only two optical quadrature encoders are used, with one placed on the front motor for both 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 create interrupts which react to a change in a pin's voltage for any digital pin, these will be slower than ISRs from hardware interrupts. A hardware interrupt 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 [6].

4.2.2 Digital Pin Connections

Each motor encoder has two output channels, channel A and channel B. Both encoders attach one of their output channels, channel A, to a hardware interrupt pin. In section 4.4.1 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 DIP switches on-board the driver. There are six of these DIP switches, and they can be 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.

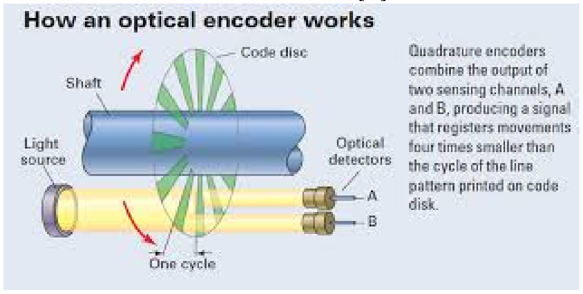
Part of the standard Arduino library is the Servo library. This library allows one to configure a digital pin to output RC control pulses, as explained in section 4.1.1. The sketch used in this project uses this library to specify the speed of each set of wheels driven by the Sabertooth motor driver, and to control the position of the standard servo aiming the ultrasonic range sensor.

4.4.1 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: [2]


How an optical encoder works
Code disc
Shaft
Light source
Optical detectors
A
B
One cycle
Quadrature encoders combine the output of two sensing channels, A and B, producing a signal that registers movements four times smaller than the cycle of the line pattern printed on code disk.

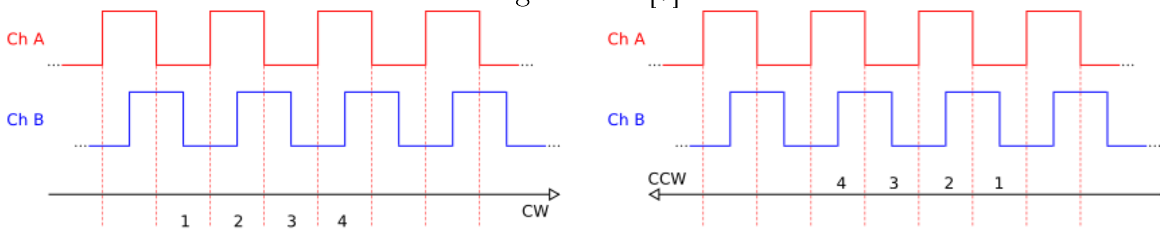
The motor encoders which came with the Lynxmotion rover kit are optical quadrature encoders. This type of encoder attaches a flat disk with thin slits known as the code disk to the motor's gear shaft. Two photodiodes, components which transform light into electric current, are placed above the disk side 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). [6]

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.

Handling rapid changes in voltage is exactly what hardware interrupts are de-

Figure 4-2: [7]



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

signed 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. [6]

The sketch uses an implementation described in [6], 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. [6]

TODO TODO TODO TODO TODO TODO text in this chapter below this line is not finished

When one of the digital pins connected to Channel A change, the main loop of the sketch is interrupted, and the corresponding interrupt service routine (ISR) is executed. Until this ISR finishes, no other code is run, including other interrupt events, 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.

PIND returns input readings from pins 0-7 as a byte. The below code uses bit shifting and masking to put the values of digital input pins 4 (output B) and 2 (output A) into the least significant bit and second-least significant bit of the enc_val variable, respectively.

```
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 enc_val = 0;
enc_val = enc_val << 2; // Store the previous 2-bit code
enc_val = enc_val | ( ((PIND & 0b100) >> 1) |
    ((PIND & 0b10000) >> 4) );
encLeftCount = encLeftCount +
    encoder_lookup_table[enc_val & 0b1111];
}
```

4.4.2 Timing

The serial communication between the Arduino and laptop is setup at a baud rate (equivalent to bit rate per second) of 115,200. The USB connection between the Arduino and laptop uses a cable supporting a data transfer rate of 480 Mbps (480 million bits per second). Ten bits are used per byte sent over the serial port (one start bit, 8 data bits, one stop bit), so 480 Mbps / 10 bits = 48 MBps. LetâĂŹs calculate the maximum data transfer rate I will need for all my sensors + motor commands (should be much less than 53 MBps).

The Arduino Uno has 3 timers, timer0, timer1, timer2. Timer0 controls millis() and delay() in Arduino. Timer1 is used by the Servo library. Timer2 is used by the NewPing library.

Each encoder generates 100 pulses per revolution. We will only be watching one of the square waves (output channel A), so thatâĂŹs 200 edge transitions per revolution. The motors are 200 rpm, so at maximum speed the two motors with encoders attached would see less than 3.4 revolutions per second. So there will be at most 3.4 * 200 = 680 counts per second. So there will be at least 1 sec / 680 = 1.4 milli seconds between counts. 9600 baud = 960 bytes / sec is more than enough for (14 byte per sensor update) * (30 updates per second (limit on ultrasonic sensor)) = 420 bytes per sec. At 960 bytes/sec, there is about a millisecond between byte transmissions.

Interrupt guards have been placed around code copying two long encoder counting variables (4 bytes) that are modified within ISRs. Each assignment to a local temp variable should translate to 4 machine instructions, so there will be 8 machine instructions between interrupt guards. The Arduino Uno uses a 16 MHz quartz crystal, so it can process 16 million instructions per second, which means it takes 1 sec / 16 million instructions = 6.25e-8 of a second per instruction. Thus to run the 8 instructions inside the guard takes 8 * 6.25e-8 = 5e-7 or half a micro second.

The worst case scenario would be for all possible interrupts to flag their events (serial RX, two hardware pins for quadrature encoders, plus Arduino background interrupts like the one for millis) right after interrupt handling is disabled. In this case, as long as interrupts are enabled and handled before another interrupt event occurs, no events will be lost. External interrupt calling has an overhead of 5.125 microseconds (5.125e-6), and each ISR should not take many microseconds [8]. Thus the interrupt guards should not cause a problem for either the 1400 micro seconds of time until the next possible quadrature encoder pin change or for the 1000 micro seconds until the next incoming serial byte. After exiting the interrupt guard, all interrupts are guaranteed to be handled before the next event arrives.

4.4.3 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 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. [10]

The sketch uses the NewPing library to handle this protocol [3]. This library provides a convenient method, ping() which returns the echo time in μs. The sketch reports this time rather than a distance, since the speed of sound in air depends on the current air temperature. The appropriate conversion is made by the repeater node on the laptop, see section 5.4.

4.4.4 Servo

The ultrasonic sensor can only detect objects which are roughly straight in front of it. Thus for the rover to have a greater view of its surroundings, the sensor needs to be panned back and forth, to "look" straight ahead, left, and right. This is what the standard servo it is attached to allows. The sketch makes use of the standard Servo library to control the servo with PPM signals. An angular degree from 0 to 180 is written to a Servo object, and the servo library handles generating the output signal for the digital pin corresponding to that object.

The sketch sweeps the servo back and forth one degree at a time, and at each step a range reading (in μs) from the PING))) sensor is taken. This reading, along with the servo's position in degrees, and the current motor encoder values, are then transmitted over serial.

This is shown in the following code for an arbitrary step size:

```
void setup() {
sonicServo.attach(sonicServoPin);
sonicServo.write(SERVO_RIGHT);
}
void loop() {
servoPos = SERVO_RIGHT; // initial servo position
while (servoPos < SERVO_LEFT) {
sonicServo.write(servoPos);
delay(SERVO_STEP_DELAY);
ping_time_uS = sonar.ping();
pushSensorUpdate(servoPos, ping_time_uS);
servoPos = min(servoPos + SERVO_STEP_SZ, SERVO_LEFT);
}
while (servoPos > SERVO_RIGHT) {
sonicServo.write(servoPos);
delay(SERVO_STEP_DELAY);
ping_time_uS = sonar.ping();
pushSensorUpdate(servoPos, ping_time_uS);
servoPos = max(servoPos - SERVO_STEP_SZ, SERVO_RIGHT);
}
}
```

4.4.5 Communication Protocol

Serial communication over usb Hexadecimal ASCII - why? sends ping time, servo position, and both encoders tick counts at each servo position when it stops to take a range reading. This occurs at roughly 20 Hz.

receives instructions, sets channel 1 and 2 accordingly

* Reading from multi-byte variables which are accessed within * and without an ISR risks data corruption, so interrupt guards * must be used around a read to make it atomic.

// worst case scenario, two encoder interrupt flags and serial RX flag are set right after noInterrupts();

long is 4 bytes, so assignment translates to 4 machine instructions.

// turn all interrupts back on, run next statement, then handle any that were flagged in-between guards

```
noInterrupts(); // turn off all interrupts
encLeftCount_temp = encLeftCount;
encRightCount_temp = encRightCount;
interrupts(); // turn all interrupts back on
```

Send sensor data to processing unit over serial port, with descriptive start bytes.

None of the start bytes should be in the range '0' - '9' or 'a' - 'f', since those chars are used by the hexadecimal ASCII encoding. L == Left motor's quadrature encoder's position value R == Right motor's quadrature encoder's position value S == Angular degree of servo P == Ultrasonic ping time (in micro seconds) measured at the given servo angle

Chapter 5

ROS

5.1 ROS

The Robot Operating System (ROS)

5.1.1 Nodes

5.1.2 Frames

5.1.3 Topics

5.1.4 Messages

publishing and subscribing messages

5.2 Overview

5.3 Ros Sensors App

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

5.3.1 Sensor Background

GPS Receivers IMU chips: magnetometers, accelerometers, gyroscopes what a quaternion is

5.3.2

Covariance matrices

5.3.3 How to use

5.4 Arduino Repeater Node

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.

Differential drive (skid steering) wheels can slip (high covariance?)

5.5 robot_localization package

odometry estimate from wheel encoders

[13]

5.6 Results

results of state estimation / sensor fusion test in parking lot

Chapter 6

Future Steps

6.1 ROS Navigation Stack

6.1.1 ros_controller

6.2 Field Test scenario

6.2.1 GPS Waypoints

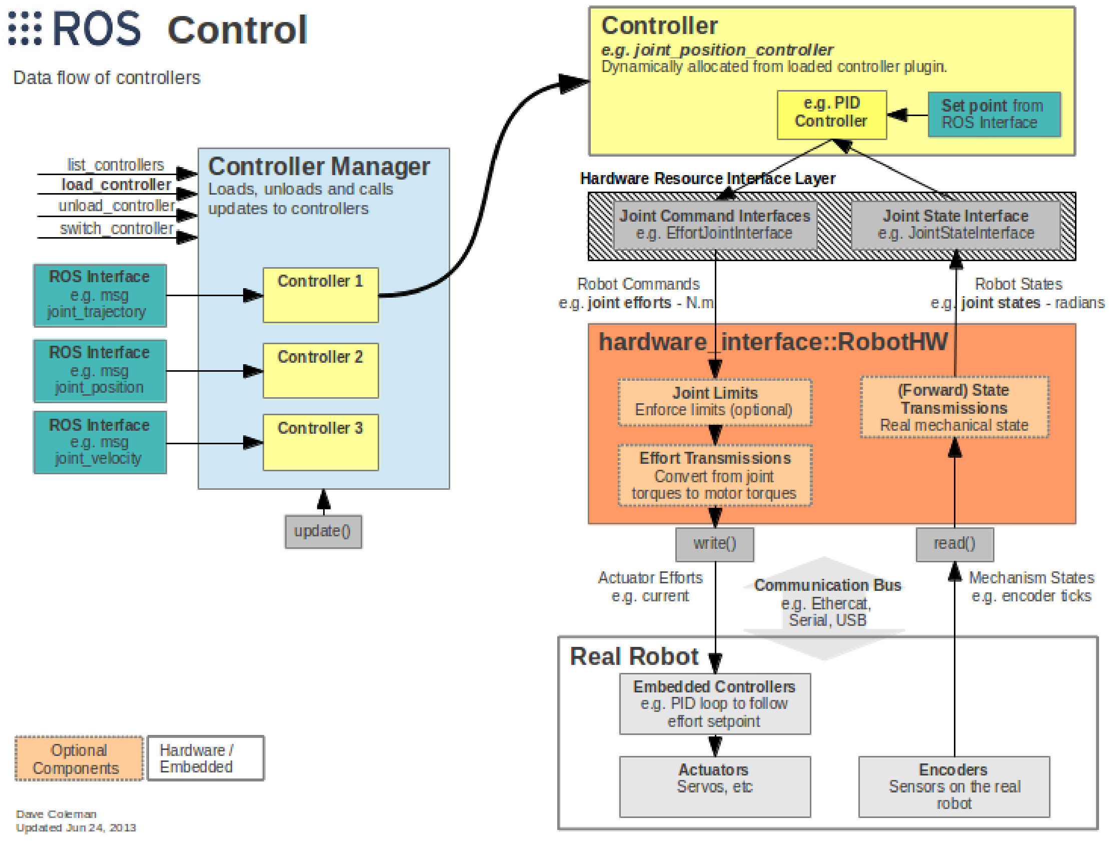
GPS markers form a linearized path travel in a straight line from one GPS node to another

6.3 Project Limitations

how this project is limited

introduction of a camera, but attaching a webcam directly to the chassis of the

Figure 6-1: ROS control overview [1]



ROS Control
Data flow of controllers
list_controllers
load_controller
unload_controller
switch_controller
Controller Manager
Loads, unloads and calls updates to controllers
ROS Interface e.g. msg joint_trajectory
ROS Interface e.g. msg joint_position
ROS Interface e.g. msg joint_velocity
Controller 1
Controller 2
Controller 3
update()
Controller
e.g. joint_position_controller
Dynamically allocated from loaded controller plugin.
e.g. PID Controller
Set point from ROS Interface
Hardware Resource Interface Layer
Joint Command Interfaces e.g. EffortJointInterface
Joint State Interface e.g. JointStateInterface
Robot Commands e.g. joint efforts - N.m
Robot States e.g. joint states - radians
hardware_interface::RobotHW
Joint Limits Enforce limits (optional)
Effort Transmissions Convert from joint torques to motor torques
(Forward) State Transmissions Real mechanical state
write()
read()
Actuator Efforts e.g. current
Communication Bus e.g. Ethercat, Serial, USB
Mechanism States e.g. encoder ticks
Real Robot
Embedded Controllers e.g. PID loop to follow effort setpoint
Actuators Servos, etc
Encoders Sensors on the real robot
Optional Components
Hardware / Embedded
Dave Coleman
Updated Jun 24, 2013

robot would be troublesome to work with, sense there would be no shock absorption and the video frames would wobble.

6.4 Conclusion

Bibliography

[1] Dave Coleman. *ros_control*. [Online: accessed April 10, 2017]. 2013. URL: https://web.archive.org/web/20161019152656/http://wiki.ros.org/ros_control.

[2] 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.

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

[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] Dimension Engineering. *Sabertooth Dual 12A 6V-24V Regenerative Motor Driver*. [Online: accessed April 11, 2017]. URL: https://web.archive.org/web/20160402162832/http://www.robotshop.com/en/sabertooth-dual-regenerative-motor-driver.html.

[6] 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`.

[7] Nich Fugal. *quadrature*. [Online: accessed April 24, 2017]. URL: `http://3.bp.blogspot.com/-9WVs1ImbheQ/U9w53Hm10FI/AAAAAAAAOv4/H313k8qFmnw/s1600/quadrature_half_res.png`.

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

[9] 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/`.

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

[11] 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`.

[12] 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`.

[13] 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.

[14] 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`.

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

[16] *USB 2.0 Frequently Asked Questions*. Online. 2017. URL: `http://web.archive.org/web/20170225172623/http://www.usb.org/developers/usbfaq%5C#cab1`.