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
August 2017

Acknowledgments

I'd like to thank my thesis sponsor Dr. Gary Kalmanovich, for his advice and giving me the freedom to explore this passion project. I'd also like to thank Dr. David Gillman for his advice and editing help. And special thanks to Nikolas "Waka Flocka" Wojtalewicz for his mathematical expertise.

Lastly, thanks to New College of Florida for providing the funding and environment to make this thesis possible.

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 August 10, 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. An important step in being able to enter this new field is working with physical hardware. However, hands-on experimentation may prove too expensive for individuals at the undergraduate level, with common research platforms costing several thousand dollars. In this thesis, I present a design for a cheap ground vehicle capable of fusing sensor data into a local state estimate, the first step of navigation. Costs were minimized by using personal components most college students already own, namely a personal laptop and smartphone. This should be an effective base to allow wider entry into autonomous navigation research.

Professor Gary Kalmanovich ....

August 10, 2017

Introduction

Navigation of a robot involves localizing where the robot is with respect to its environment, finding a path from where it currently is to where it would like to be, executing commands to follow that path, and dynamically reacting to obstacles to avoid collisions. While the robot design presented in this thesis is capable of performing all of these tasks, only the first - localization - has been implemented.

In order for a robot to localize itself with respect to its environment, it must either already know its environment via a static map, or dynamically generate a map of its surroundings. This latter approach is known as the simultaneous localization and mapping (SLAM) problem, and the most common algorithms used to solve it involve the use of ranging sensors. Ranging sensors measure the distance between the robot and surrounding obstacles.

In this project a static map was assumed to exist, where the map used is the Universal Transverse Mercator (UTM) 2D projection of a coordinate system onto the Earth's surface. A GPS receiver was used for partial noisy global localization information, and localization was implemented with respect to this map. Ranging sensors giving landmark detection on a smaller scale were not used.

The ultimate goal of this project is to create a mobile robot capable of navigating the New College of Florida campus during the day, with limited financial resources. While that end goal has not yet been reached, appreciable progress in designing and constructing the rover and its associated control software has been made. Localization with respect to a static map was implemented, and integration of existing range data into dynamic map generation is the next logical step. It is the author's hope that further work with this hardware will be completed at New College of Florida.

The design itself, while not groundbreaking, is carefully laid out in the hope that other students or individuals interested in experimenting with autonomous navigation will be able to replicate the construction. The cheap material cost may help ease the financial burden for those who wish for hands-on experience with this material.

The rover design consists of a four-wheeled skid-steering aluminum rover base, an Arduino microcontroller, and motor driver. It is controlled by a laptop connected via USB. Sensors available include two motor rotary encoders, a mobile smartphone, and an ultrasonic distance sensor. The Arduino acts as a low-level robotic controller, sending wheel encoder and range data to the laptop, 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.

The sensors available are limited in precision, but were chosen for their cheap cost and wide availability.

Rather than purchasing separate IMU and GPS chips to interface with the Arduino microcontroller, the built-in sensors on a smartphone were utilized. These built-in sensors are less accurate than their external counterparts, but in many cases present no additional cost to acquire for a college student.

Infrared (IR) sensors are cheap ranging sensors common in small robotics projects. However, this rover was designed to work outdoors during the day, and the ambient IR radiation from the sun makes these sensors unusable. Meanwhile light detection and ranging (LIDAR) sensors are highly expensive ranging sensors which are considered the industry standard for high resolution, high accuracy point cloud mapping in a 360 degree arc.

In order to satisfy the robust and financial constraints of this thesis, IR and LIDAR range sensors are out of the picture. So a third option, the ultrasonic sensor, was chosen. Ultrasonic sensors are medium-distance ranging sensors which emit high-frequency sonic pulses and measure the echo time. They are capable of detecting objects in a cone shape in front of them. They have reduced accuracy in rain or snow, and can suffer from confusing double echoes. However, they are nearly as inexpensive as IR sensors, and by panning back and forth, can slowly give a 180 degree distance measurement.

The rest of this thesis is structured as follows.

Chapter 1 covers the probability theory behind the extended Kalman filter, which is an iterative 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 2 describes the hardware components used in this project, their electrical connections, and the general design. Chapter 3 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 4 gives a brief overview of the Robot Operating System (ROS), and how it's used in this project. It then touches on the distinct software processes that make up the system. Chapter 5 describes the results of an experimental localization test run conducted with the rover.

Chapter 1

Theory

This section will dive into the mathematical theory behind the recursive state estimator used in this project, the Extended Kalman Filter. Steps have been taken to carefully explain every computation made for the less mathematically inclined. This section is largely based on chapters 1-3 of the book Probabilistic Robotics, which the interested reader should view for a broader look at the same material. [16].

1.1 Probability Theory Background

Robots estimate their environment stochastically, and so probability theory is vital to understanding their inner workings. Here we will review some elementary probability theory results which will be needed.

Random variables are objects from which specific numbers may be observed. Let X be a continuous random variable - that is, observations of X are real numbers. We will represent the probability of observing any particular real number x from X as

$p(X = x) \equiv p(x)$. This defines a function $p : \mathbb{R} \rightarrow \mathbb{R}$ representing the distribution of probabilities for the real numbers. Throughout this chapter, we will refer to such functions as PDFs (probability density functions). Note the basic result that for every PDF p constructed from a random variable X, $\int p(x)dx = 1$, which is to say that the observed value from X will be a real number with 100% certainty.

An important tool for describing PDFs is the expectation. We will define $E[X]$ to be the expected value, or arithmetic mean of, X. Note that the expectation operator is linear: $E[\alpha X + \beta Y] = \alpha E[X] + \beta E[Y]$.

For two random variables X and Y, we'll define their covariance to be

$$cov(X, Y) = E[(X - E[X])(Y - E[Y])]$$

this gives a measure of the linear relationship between X and Y.

The same concepts may be scaled up to random vectors, which are n-dimensional collections of random variables. Then $X = (X_1, X_2, ..., X_n)^T$, and $p : \mathbb{R}^n \rightarrow \mathbb{R}$ still defines a PDF. The expectation becomes

$$E[X] = \begin{pmatrix} E[X_1] \\ \vdots \\ E[X_n] \end{pmatrix}$$

and we can define a covariance matrix Σ for the random vector:

$$\Sigma = E[(X - E[X])(X - E[X])^T] = \begin{pmatrix} cov(X_1, X_1) \ldots cov(X_1, X_n) \\ \vdots \ddots \vdots \\ cov(X_n, X_1) \ldots cov(X_n, X_n) \end{pmatrix}$$

Now consider three random variables: X, Y, and Z. 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

$$p(x \mid y, z) = \frac{p(x, y, z)}{p(y, z)} \tag{1.1}$$

The *Law of Total Probability* states that $p(x) = \int p(x, y)dy$. Extending this law to use a third random variable Z, and incorporating Equation 1.1, we end up with the following equation:

$$p(x \mid z) = \int p(x, y \mid z)dy = \int p(y, z)p(x \mid y, z)dy \tag{1.2}$$

Using equations 1.2 and 1.1, one can derive a formula known as Bayes' Rule.

$$p(x \mid y, z) = \frac{p(x, y, z)}{p(y, z)} = \frac{p(y, x, z)}{p(x, z)} * \frac{p(x, z)}{p(y, z)} = \frac{p(y \mid x, z)p(x, z)}{p(y \mid z)} \tag{1.3}$$

In the future this will prove to be a useful tool to update a PDF with new information. When presented with new information that $Y = y$, one may use Bayes' Rule to transform the current PDF $p(x)$ into the new posterior PDF $p(x \mid y)$.

1.2 Bayes Filter

1.2.1 Scenario

Now, back to the inner workings of our rover. Consider the general case of a robot which uses some array of sensors to gather information about its environment. Each of these measurements will have some amount of error. The robot wishes to use these measurements to estimate its current state.

For simplicity, let's consider the rover to operate in discrete time steps ($t = 0, 1, 2, ...$). We can encode all state variables of interest in the vector $x_t = (x_{1_t}, x_{2_t}, ..., x_{n_t})^T$. Similarly, let $z_t = (z_{1_t}, z_{2_t}, ..., z_{k_t})^T$ represent a sensor measurement at time t. For both of these vectors we will use the compact notation $a_{1:t} = a_1, a_2, ..., a_t$ to denote the set of all vectors up to and including time t.

The robot wishes to know x_t, however this true state is hidden. It only has access to raw noisy data in the form of $z_{1:t}$. It can use this information to estimate the most likely state, and construct a PDF assigning a probability to every possible realization of x_t. This PDF 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:

$$bel(x_t) = p(x_t \mid z_{1:t}) \tag{1.4}$$

1.2.2 Derivation

Now we must figure out how to compute $bel(x_t)$. Let's start by using Bayes' Rule to rewrite $bel(x_t)$:

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

In order to continue, we will need to simplify $p(z_t|x_t, z_{1:t-1})$. Here we will make an important assumption. We'll assume that the state x_t satisfies the Markov property, that is, x_t perfectly encapsulates all current and prior information. Thus if x_t is known, then $z_{1:t}$ are redundant, i.e. $p(z_t \mid x_t, z_{1:t-1}) = p(z_t \mid x_t)$.

This assumption lets us remove consideration of past sensor measurements, and to rewrite the belief distribution as:

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

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

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

Notice that $bel(x_t)$ is a PDF, so it must integrate to 1. η acts as a normalizing constant enforcing this constraint.

Now $bel(x_t)$ has been split into two distributions of interest. Looking closely one may notice that $p(x_t \mid z_{1:t-1})$ is simply our original belief distribution, Eq. 1.4, 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 the *Law of Total Probability* (Eq. 1.2) and our Markov assumption:

$$
\begin{aligned}
&\overline{bel}(x_t) = p(x_t \mid z_{1:t-1}) \\
&= \int p(x_t \mid x_{t-1}, z_{1:t-1}) p(x_{t-1} \mid z_{1:t-1}) dx_{t-1} && \text{(by Eq. 1.2)} \\
&= \int p(x_t \mid x_{t-1}) p(x_{t-1} \mid z_{1:t-1}) dx_{t-1} && \text{(by Markov assumption)} \\
&= \int p(x_t \mid x_{t-1}) bel(x_{t-1}) dx_{t-1} && \text{(by Eq. 1.4)}
\end{aligned}
$$

We have arrived at a recursive definition of $bel(x_t)$ with respect to $bel(x_{t-1})$! As long as $p(x_t \mid x_{t-1})$ and $p(z_t \mid x_t)$ are known, we can recursively calculate $bel(x_t)$ from some starting belief $bel(x_0)$.

$p(x_t \mid x_{t-1})$ defines a stochastic model for the robot's state, determining how the robot's state will naturally evolve over time. This PDF will be referred to 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 state. This PDF will be referred to as the *measurement probability* [16].

Once we assume the *state transition probability* and *measurement probability* PDFs are known, we can finally construct the algorithm known as Bayes' Filter:

Algorithm 1 Bayes Filter

function BAYESFILTERITERATE($bel(x_{t-1})$, z_t)
    $\overline{bel}(x_t) = \int p(x_t \mid x_{t-1}) bel(x_{t-1}) dx_{t-1}$
    $bel(x_t) = \eta p(z_t \mid x_t) \overline{bel}(x_t)$
    Set $\int bel(x_t) dx = 1$, and solve for η
    Use η to normalize $bel(x_t)$
    return $bel(x_t)$
end function

1.3 Extended Kalman Filter

The most widely used algorithm implementing Bayes Filter is the Extended Kalman Filter (EKF). This filter approximates the PDFs found in Bayes Filter with multivariate normal distributions. This class of PDFs are unimodal, and have the form

$$p(x) = N(x; E[X], \Sigma) = det(2\pi\Sigma)^{-\frac{1}{2}} * exp\{-\frac{1}{2}(x - E[X])^T \Sigma^{-1} (x - E[X])\} \quad (1.5)$$

where $E[X]$ and Σ are the mean and covariance matrix of the random vector X, as described in section 1.1. Notice that the mean and covariance matrix are sufficient to uniquely define any particular normal distribution, and so the EKF only needs to keep track of a vector and matrix per PDF. This keeps the space complexity of the algorithm low.

The belief distribution $bel(x_t)$ is given the form $N(x_t; \mu_t, \Sigma_t)$, where $\mu_t = E[X]$ is the mean or best estimate of the state vector, and Σ_t is the covariance matrix representing the uncertainty in that estimate.

Let

$$x_t = g(x_{t-1}) + \epsilon_t$$

be the underlying model for the state transition pdf ($p(x_t \mid x_{t-1})$), where $g : \mathbb{R}^n \to \mathbb{R}^n$ is an arbitrary function, and ϵ_t is a random vector of normal distribution with mean 0 and covariance matrix R_t.

We will linearize this model by a first-order Taylor series approximation around the current best estimate, μ_{t-1}.

$$g(x_{t-1}) \approx g(\mu_{t-1}) + \frac{\partial g(\mu_{t-1})}{\partial x_{t-1}}(x_{t-1} - \mu_{t-1})$$

$$= g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1})$$

where

$$G_t = \begin{bmatrix} \frac{\partial g_1(\mu_{t-1})}{\partial x_{1_{t-1}}} & \cdots & \frac{\partial g_1(\mu_{t-1})}{\partial x_{n_{t-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_n(\mu_{t-1})}{\partial x_{1_{t-1}}} & \cdots & \frac{\partial g_n(\mu_{t-1})}{\partial x_{n_{t-1}}} \end{bmatrix}$$

is the Jacobian of g evaluated at μ_{t-1}. Then rewriting the model, we have

$$x_t \approx g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}) + \epsilon_t$$

and the state transition PDF has mean

$$E[x_t] = E[g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}) + \epsilon_t]$$

$$= E[g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1})] + E[\epsilon_t] \qquad \text{(by linearity of expectation)}$$

$$= E[g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1})] \qquad \text{(by the definition of } \epsilon_t\text{)}$$

$$= g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}) \qquad (x_{t-1} \text{ is a constant w.r.t } x_t)$$

and covariance matrix

$$\Sigma_t = E[(x_t - E[x_t])(x_t - E[x_t])^T]$$

$$= E[(g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}) + \epsilon_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))$$

$$* \, (g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}) + \epsilon_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))^T]$$

$$= E[\epsilon_t \epsilon_t^T] = E[(\epsilon_t - 0)(\epsilon_t - 0)^T] = E[(\epsilon_t - E[\epsilon_t])(\epsilon_t - E[\epsilon_t])^T] = R_t$$

and so its Gaussian distribution is $p(x_t \mid x_{t-1},) = N(x_t; g(\mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}), R_t)$

1.3.1 Matrix Calculus

For the mathematical derivation of the EKF which lies ahead, we will be working with matrix calculus.

For the benefit of the unfamiliar reader, here we will restate some elementary results which will be needed to follow along.

For $A, B \in \mathbb{R}^{nxn}$:

$$(AB)^T = B^T A^T \tag{1.6}$$

$$(A + B)^T = A^T + B^T \tag{1.7}$$

$$A \text{ is symmetric} \implies A^T = A \tag{1.8}$$

$$A \text{ is symmetric } \implies A^{-1} \text{ is symmetric} \tag{1.9}$$

Lemma 1.3.1. *For $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{nxn}$, A symmetric,*

$$\frac{\partial(x^T Ax)}{\partial x} = 2Ax$$

Proof.

Define $f : \mathbb{R}^n \to \mathbb{R}, f(x) = x^T Ax$

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \; \frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}, \; e_i = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$

For $h \in \mathbb{R}$,

$$\frac{\partial f}{\partial x_i} = \lim_{h\to 0} \frac{f(x + he_i) - f(x)}{h} = \lim_{h\to 0} \frac{(x + he_i)^T A(x + he_i) - x^T Ax}{h}$$

$$= \lim_{h\to 0} \frac{x^T Ax + h * x^T Ae_i + h * e_i^T Ax + h^2 * e_i^T Ae_i - x^T Ax}{h}$$

$$= \lim_{h\to 0} \frac{h[x^T Ae_i + e_i^T Ax + h * e_i^T Ae_i]}{h} = \lim_{h\to 0}(x^T Ae_i + e_i^T Ax + h * e_i^T Ae_i)$$

$$= x^T Ae_i + e_i^T Ax = e_i^T A^T x + e_i^T Ax \qquad (x^T Ae_i \text{ is 1x1} \therefore x^T Ae_i \text{ is symmetric})$$

$$= e_i^T (A^T + A)x = e_i^T (A + A)x = e_i^T (2Ax) \qquad \text{(by symmetry of A)}$$

Which tells us that the i^{th} component of $\dfrac{\partial f}{\partial x}$ is equal to the i^{th} component of $2Ax$.

$$\therefore \frac{\partial f}{\partial x} = 2Ax$$

□

Lemma 1.3.2. *For* $x, a \in \mathbb{R}^n$, $\frac{\partial(x^T a)}{\partial x} = \frac{\partial(a^T x)}{\partial x} = a$

Proof.

$$\frac{\partial(a^T x)}{\partial x} = \begin{pmatrix} \frac{\partial(a^T x)}{\partial x_1} \\ \vdots \\ \frac{\partial(A^T x)}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial(a_1 x_1 + \cdots + a_n x_n)}{\partial x_1} \\ \vdots \\ \frac{\partial(a_1 x_1 + \cdots + a_n x_n)}{\partial x_n} \end{pmatrix} = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} = a$$

$a^T x$ has dimension 1 by 1, $\therefore a^T x = (a^T x)^T = x^T a$

$$\implies \frac{\partial(a^T x)}{\partial x} = \frac{\partial(x^T a)}{\partial x} = a$$

□

Lemma 1.3.3. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a quadratic function given by*

$$f(x) = x^T A x + Bx + C$$

where A is a symmetric matrix of dimension $n \times n$, B is a matrix of dimension $1 \times n$, and $C \in \mathbb{R}$. Then

$$f(x) = f(a) + \frac{1}{2}(x - x^*)^T \frac{\partial^2 f}{\partial x^2}(x - x^*)$$

where $\frac{\partial f}{\partial x}(x^*) = 0$. This is a higher-dimensional application of Taylor's Theorem.

Proof.

$$\frac{\partial f}{\partial x} = \frac{\partial(x^T Ax)}{\partial x} + \frac{\partial(Bx)}{\partial x} + \frac{\partial(C)}{\partial x} = 2Ax + B^T \qquad \text{(by prior Lemmas)}$$
$$\therefore \frac{\partial f}{\partial x} = 0 \implies x^* = -\frac{1}{2}A^{-1}B^T$$

$$\frac{\partial^2 f}{\partial x^2} = \frac{\partial(2Ax)}{\partial x} + \frac{\partial(B^T)}{\partial x} = 2A^T = 2A \qquad \text{(by symmetry of A)}$$

$$f(a) + \frac{1}{2}(x - x^*)^T \frac{\partial^2 f}{\partial x^2}(x - x^*)$$
$$= (-\frac{1}{2}A^{-1}B^T)^T A(-\frac{1}{2}A^{-1}B^T) + B(-\frac{1}{2}A^{-1}B^T) + C$$
$$+ \frac{1}{2}(x + \frac{1}{2}A^{-1}B^T)^T (2A)(x + \frac{1}{2}A^{-1}B^T)$$
$$= \frac{1}{4}BA^{-1}B^T - \frac{1}{2}BA^{-1}B^T + C + x^T Ax + \frac{1}{2}Bx + \frac{1}{4}BA^{-1}B^T + \frac{1}{2}x^T B^T$$
$$= x^T Ax + Bx + C = f(x)$$

□

Lemma 1.3.4. *Let $p : \mathbb{R}^n \to \mathbb{R}$ be a PDF given by*

$$p(x) = \eta_1 \; exp\{-L\}$$

where $L : \mathbb{R}^n \to \mathbb{R}$, $L(x) = x^T Ax + Bx + C$. Then p defines a normal distribution with mean equal to the minimum of L, and covariance matrix equal to $(\frac{\partial^2 L}{\partial x})^{-1}$.

Proof. Using Lemma 1.3.3, we know that

$$L(x) = L(a) + \frac{1}{2}(x - x^*)^T \frac{\partial^2 L}{\partial x^2}(x - x^*)$$

where x^* is the minimum of L. Therefore we can rewrite p as

$$\begin{aligned} p(x) &= \eta_1 \ exp\{-L(a) - \frac{1}{2}(x - x^*)^T \frac{\partial^2 L}{\partial x^2}(x - x^*)\} \\ &= \eta_1 \ exp\{-L(a)\} exp\{-\frac{1}{2}(x - x^*)^T \frac{\partial^2 L}{\partial x^2}(x - x^*)\}\} \\ &= \eta_2 \ exp\{-\frac{1}{2}(x - x^*)^T \frac{\partial^2 L}{\partial x^2}(x - x^*)\}\} \end{aligned}$$

Compare this form of p to the definition of the normal distribution given in Eq. 1.5. This is just a normal distribution, where $E[X] = x^*$ and $\Sigma^{-1} = \frac{\partial^2 L}{\partial x^2}$. Therefore the mean of p is the minimum of L, and the covariance matrix of p is the inverse of the second derivative of L. □

Lemma 1.3.5. ***Inversion Lemma [16]***

For $R \in \mathbb{R}^{nxn}, Q \in \mathbb{R}^{kxk}, P \in \mathbb{R}^{nxk}$ where R, Q, and P are all invertible:

$$(R + PQP^T)^{-1} = R^{-1} - R^{-1}P(Q^{-1} + P^T R^{-1} P)^{-1} P^T R^{-1}$$

Proof.

$$
\begin{aligned}
&(R^{-1} - R^{-1}P(Q^{-1} + P^T R^{-1}P)^{-1}P^T R^{-1})(R + PQP^T) \\
&= [\underline{R^{-1}R}] + [R^{-1}PQP^T] - [R^{-1}P(Q^{-1} + P^T R^{-1}P)^{-1}P^T \underline{R^{-1}R}] \\
&- [R^{-1}P(Q^{-1} + P^T R^{-1}P)^{-1}P^T R^{-1}PQP^T] \\
&= [I] + [R^{-1}PQP^T] - [R^{-1}P(Q^{-1} + P^T R^{-1}P)^{-1}P^T] \\
&- [R^{-1}P(Q^{-1} + P^T R^{-1}P)^{-1}P^T R^{-1}PQP^T] \\
&= I + R^{-1}P[QP^T - (Q^{-1} + P^T R^{-1}P)^{-1}P^T \\
&- (Q^{-1} + P^T R^{-1}P)^{-1}P^T R^{-1}PQP^T] \\
&= I + R^{-1}P[QP^T - (Q^{-1} + P^T R^{-1}P)^{-1}\underline{Q^{-1}Q}P^T \\
&- (Q^{-1} + P^T R^{-1}P)^{-1}P^T R^{-1}PQP^T] \\
&= I + R^{-1}P[QP^T - \underline{(Q^{-1} + P^T R^{-1}P)^{-1}(Q^{-1} + P^T R^{-1}P)}QP^T] \\
&= I + R^{-1}P[\underline{QP^T - QP^T}] = I
\end{aligned}
\tag{1.10}
$$

□

1.3.2 Derivation

Note that in the following mathematical derivation, because cov(X,Y) = cov(Y,X), the covariance matrix R_t is symmetric by definition.

We can now begin by rewriting Line 2 of the Bayes Filter using our normal PDF definitions:

$$
\begin{aligned}
\overline{bel}(x_t) &= \int p(x_t \mid x_{t-1}) bel(x_{t-1}) dx_{t-1} \\
&= det(2\pi R_t)^{-\frac{1}{2}} det(2\pi \Sigma_{t-1})^{-\frac{1}{2}} \int [exp\{-\frac{1}{2}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))^T \\
&* R_t^{-1}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))\} * exp\{-\frac{1}{2}(x_t - \mu_{t-1})^T \Sigma_{t-1}^{-1}(x_t - \mu_{t-1})\} dx_{t-1}] \\
&= \eta \int exp\{-L_t\} dx_{t-1} \qquad (1.11)
\end{aligned}
$$

where we've defined

$$
\begin{aligned}
L_t &= \frac{1}{2}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))^T R_t^{-1}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1})) \\
&+ \frac{1}{2}(x_t - \mu_{t-1})^T \Sigma_{t-1}^{-1}(x_t - \mu_{t-1}) \qquad (1.12)
\end{aligned}
$$

By assumption, $\overline{bel}(x_t)$ is a normal distribution. We will define its mean and covariance to be $\overline{\mu}$ and $\overline{\Sigma}$ respectively. In order to compute these matrices, we would like to use Lemma 1.3.4. However, to turn $\overline{bel}(x_t)$ into the proper form, we will have to get rid of the integral over x_{t-1}. To do so, we will decompose L_t into two terms like so:

$$
L_t = L_t(x_{t-1}, x_t) + L_t(x_t) \qquad (1.13)
$$

where all terms containing x_{t-1} are collected in $L_t(x_{t-1}, x_t)$. This will allow us to move $L_t(x_t)$ outside of $\overline{bel}(x_t)$'s integral, as it will not depend on x_{t-1}. Then, as long as we choose our decomposition so that the integral of $L_t(x_{t-1}, x_t)$ equals a constant, we will be left with $\overline{bel}(x_t)$ in a form where we can apply Lemma 1.3.4.

Now to proceed with this decomposition. Notice that $L_t : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is quadratic with respect to x_{t-1}. By Lemma 1.3.3 we know that we can rewrite L_t using the minimum with respect to $x_{t-1} : x^*_{t-1}$

$$L_t = \underline{L_t(x^*_{t-1})} + \frac{1}{2}(x_{t-1} - x^*_{t-1})\frac{\partial^2 L_t}{\partial x^2_{t-1}}(x_{t-1} - x^*_{t-1})$$

where the underlined term is L_t evaluated at x^*_{t-1}. This will give us our desired decomposition, turning L_t into the sum of another quadratic expression w.r.t x_{t-1}, and a constant term w.r.t x_{t-1}. Let us now calculate the minimum x^*_{t-1}. To do so, we will first have to find $\frac{\partial L_t}{\partial x_{t-1}}$.

Define

$$L_{t_1} = \frac{1}{2}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))^T R_t^{-1}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))$$
$$L_{t_2} = \frac{1}{2}(x_t - \mu_{t-1})^T \Sigma_{t-1}^{-1}(x_t - \mu_{t-1})$$

then

$$\frac{\partial L_t}{\partial x_{t-1}} = \frac{\partial(L_{t_1} + L_{t_2})}{\partial x_{t-1}} = \frac{\partial L_{t_1}}{\partial x_{t-1}} + \frac{\partial L_{t_2}}{\partial x_{t-1}}$$

and

$$\frac{\partial L_{t_1}}{\partial x_{t-1}} = \frac{\partial(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))}{\partial x_{t-1}}$$
$$* \frac{\partial L_{t_1}}{\partial(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))} \qquad \text{(by Chain Rule)}$$
$$= (-G_t^T) * \frac{\partial L_{t_1}}{\partial(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))} \qquad \text{(by Lemma 1.3.2)}$$
$$= -G_t^T R_t^{-1}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1})) \qquad \text{(by Lemma 1.3.1)}$$

$$\frac{\partial L_{t_2}}{\partial x_{t-1}} = \frac{\partial L_{t_2}}{\partial(x_{t-1} - \mu_{t-1})} * \frac{\partial(x_{t-1} - \mu_{t-1})}{\partial x_{t-1}} = \frac{\partial L_{t_2}}{\partial(x_{t-1} - \mu_{t-1})}$$
$$= \frac{1}{2}(2\Sigma_{t-1}^{-1})(x_{t-1} - \mu_{t-1}) = \Sigma_{t-1}^{-1}(x_{t-1} - \mu_{t-1}) \qquad \text{(by Lemma 1.3.1)}$$

which means

$$\frac{\partial L_t}{\partial x_{t-1}} = -R_t^{-1}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))G_t + \Sigma_{t-1}^{-1}(x_{t-1} - \mu_{t-1})$$

and

$$\frac{\partial^2 L_t}{\partial x_{t-1}^2} = \frac{\partial(R_t^{-1}G_t x_{t-1} G_t + \Sigma_{t-1}^{-1} x_{t-1})}{\partial x_{t-1}}$$
$$= \frac{\partial(R_t^{-1}G_t x_{t-1} G_t)}{\partial x_{t-1}} + \Sigma_{t-1}^{-1} \qquad \text{(by Lemma 1.3.2)}$$
$$= \frac{\partial(R_t^{-1}G_t x_{t-1})}{\partial x_{t-1}} * G_t + (R_t^{-1}G_t x_{t-1})\frac{\partial G_t}{\partial x_{t-1}} + \Sigma_{t-1}^{-1} \qquad \text{(by product rule)}$$
$$= G_t^T R_t^{-1} G_t + \Sigma_{t-1}^{-1} \equiv \Phi_t^{-1} \qquad \text{(by Lemma 1.3.2)}$$

Now we can compute the minimum:

$$\frac{\partial L_t}{\partial x_{t-1}} = 0$$

$$\implies (x_{t-1}^* - \mu_{t-1})^T \Sigma_{t-1}^{-1} = (x_t - g(\mu_{t-1}) - G_t(x_{t-1}^* - \mu_{t-1}))^T R_t^{-1} G_t$$

$$\implies \Sigma_{t-1}^{-1}(x_{t-1}^* - \mu_{t-1}) = G_t^T R_t^{-1}(x_t - g(\mu_{t-1}) - G_t(x_{t-1}^* - \mu_{t-1}))$$

$$\implies \Sigma_{t-1}^{-1} x_{t-1}^* + G_t^T R_t^{-1} G_t x_{t-1}^* = G_t^T R_t^{-1}(x_t - g(\mu_{t-1}) + G_t \mu_{t-1}) + \Sigma_{t-1}^{-1}\mu_{t-1}$$

$$\implies \Phi_t^{-1} x_{t-1}^* = G_t^T R_t^{-1}(x_t - g(\mu_{t-1})) + \Phi_t^{-1}\mu_{t-1}$$

$$\implies x_{t-1}^* = \Phi_t[G_t^T R_t^{-1}(x_t - g(\mu_{t-1})) + \Phi_t^{-1}\mu_{t-1}] \tag{1.14}$$

Thus we can construct our decomposition:

$$L_t(x_{t-1}, x_t) = \frac{1}{2}(x_{t-1} - x_{t-1}^*)^T \frac{\partial^2 L_t}{\partial x_{t-1}^2}(x_{t-1} - x_{t-1}^*)$$

$$= \frac{1}{2}(x_{t-1} - \Phi_t[G_t^T R_t^{-1}(x_t - g(\mu_{t-1})) + \Phi_t^{-1}\mu_{t-1}])^T$$

$$* \Phi_t^{-1}(x_{t-1} - \Phi_t[G_t^T R_t^{-1}(x_t - g(\mu_{t-1})) + \Phi_t^{-1}\mu_{t-1}])$$

and

$$
\begin{aligned}
L_t(x_t) &= L_t - L_t(x_{t-1}, x_t) \\
&= \frac{1}{2}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1}))^T R_t^{-1}(x_t - g(\mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1})) \\
&+ \frac{1}{2}(x_t - \mu_t)^T \Sigma_{t-1}^{-1}(x_t - \mu_t) - \frac{1}{2}(x_{t-1} - \Phi_t[G_t^T R_t^{-1}(x_t - g(\mu_{t-1})) + \Phi_t^{-1}\mu_{t-1}])^T \\
&* \Phi_t^{-1}(x_{t-1} - \Phi_t[G_t^T R_t^{-1}(x_t - g(\mu_{t-1})) + \Phi_t^{-1}\mu_{t-1}]) \\
&= \frac{1}{2}(x_t - g(\mu_{t-1}) + G_t\mu_{t-1})^T R_t^{-1}(x_t - g(\mu_{t-1}) + G_t\mu_{t-1}) + \frac{1}{2}\mu_{t-1}^T \Sigma_{t-1}^{-1}\mu_{t-1} \\
&- \frac{1}{2}[G_t^T R_t^{-1}(x_t - g(\mu_{t-1})) + \Phi_t^{-1}\mu_{t-1}]^T \Phi_t [G_t^T R_t^{-1}(x_t - g(\mu_{t-1})) + \Phi_t^{-1}\mu_{t-1}]
\end{aligned} \tag{1.15}
$$

Notice that the quadratic and linear x_{t-1} terms cancel out in $L_t(x_t)$. This is to be expected from our decomposition, since $L_t(x_t)$ should be a constant with respect to x_{t-1}. Also, $L_t(x_{t-1}, x_t)$ is the negative of the exponent in a normal distribution, with covariance matrix Φ_t. Because all PDFs integrate to 1, we know:

$$
\int det(2\pi\Phi_t)^{-\frac{1}{2}} exp\{-L_t(x_{t-1}, x_t)\} dx_{t-1} = 1 \tag{1.16}
$$

We now have all the pieces to further simplify $\overline{bel}(x_t)$:

$$\begin{aligned}
\overline{bel}(x_t) &= \eta_1 \int exp\{-L_t\} dx_{t-1} && \text{(by Eq. 1.11)} \\
&= \eta_1 \int exp\{-L_t(x_{t-1}, x_t) - L_t(x_t)\} dx_{t-1} && \text{(by Eq. 1.13)} \\
&= \eta_1 \int exp\{-L_t(x_{t-1}, x_t)\} exp\{-L_t(x_t)\} dx_{t-1} \\
&= \eta_1 \ exp\{-L_t(x_t)\} \int exp\{-L_t(x_{t-1}, x_t)\} dx_{t-1} \\
&= \eta_1 \ exp\{-L_t(x_t)\} * det(2\pi\Phi_t)^{\frac{1}{2}} && \text{(by Eq. 1.16)} \\
&= \eta_2 \ exp\{-L_t(x_t)\} && (1.17)
\end{aligned}$$

Notice that the $det(2\pi\Phi_t)^{\frac{1}{2}}$ term was absorbed into the normalizing constant η, since it is a constant with respect to x_t.

Because $L_t(x_t)$ is quadratic, $\overline{bel}(x_t)$ now satisfies Lemma 1.3.4 (based on Eq. 1.17 and 1.15). So we know that the mean and covariance of the normal distribution representing $\overline{bel}(x_t)$ will be equal to the minimum of $L_t(x_t)$, and the inverse of its second derivative. We will now compute these values.

$$\frac{\partial L_t(x_t)}{\partial x_t} = \frac{\partial[\frac{1}{2}x_t^T R_t^{-1} x_t]}{\partial x_t} + \frac{\partial[\frac{1}{2}x_t^T R_t^{-1}(-g(\mu_{t-1}) + G_t\mu_{t-1})]}{\partial x_t}$$

$$+ \frac{\partial[\frac{1}{2}(-g(\mu_{t-1}) + G_t\mu_{t-1})R_t^{-1} x_t]}{\partial x_t} + \frac{\partial[-\frac{1}{2}x_t^T R_t^{-1} G_t \Phi_t G_t^T R_t^{-1} x_t]}{\partial x_t}$$

$$+\frac{\partial[-\frac{1}{2}x_t^T R_t^{-1} G_t \Phi_t(-G_t^T R_t^{-1} g_t + \Phi_t^{-1}\mu_{t-1})]}{\partial x_t}+\frac{\partial[-\frac{1}{2}(-G_t^T R_t^{-1} g_t + \Phi_t^{-1}\mu_{t-1})^T \Phi_t G_t^T R_t^{-1} x_t]}{\partial x_t}$$

$$= R_t^{-1}x_t - (R_t^{-1}G_t\Phi_t G_t^T R_t^{-1})^T x_t \qquad \text{(by Lemma 1.3.1)}$$

$$+ R_t^{-1}(-g(\mu_{t-1}) + G_t\mu_{t-1}) \qquad \text{(by Lemma 1.3.2)}$$

$$- R_t^{-1}G_t\Phi_t(-G_t^T R_t^{-1} g(\mu_{t-1} + \Phi_t^{-1}\mu_{t-1}) \qquad \text{(by Lemma 1.3.2)}$$

$$= (R_t^{-1} - R_t^{-1}G_t\Phi_t G_t^T R_t^{-1})x_t$$

$$+ R_t^{-1}(-g(\mu_{t-1}) + G_t\mu_{t-1}) - R_t^{-1}G_t\Phi_t(-G_t^T R_t^{-1} g(\mu_{t-1}) + \Phi_t^{-1}\mu_{t-1})$$

$$= (R_t + G_t\Sigma_{t-1}G_t^T)^{-1}x_t \qquad \text{(by Lemma 1.3.5)}$$

$$+ R_t^{-1}(-g(\mu_{t-1}) + G_t\mu_{t-1}) - R_t^{-1}G_t\Phi_t(-G_t^T R_t^{-1} g(\mu_{t-1}) + \Phi_t^{-1}\mu_{t-1})$$

$$\frac{\partial L_t(x_t)}{\partial x_t} = 0$$

$$\implies \overline{\mu_t} = (R_t + G_t\Sigma_{t-1}G_t^T)[R_t^{-1}G_t\Phi_t(-G_t^T R_t^{-1} g(\mu_{t-1}) + \Phi_t^{-1}\mu_{t-1})$$

$$- R_t^{-1}(-g(\mu_{t-1}) + G_t\mu_{t-1})]$$

$$= (R_t + G_t\Sigma_{t-1}G_t^T)[-R_t^{-1}G_t\Phi_t G_t^T R_t^{-1} g(\mu_{t-1}) + R_t^{-1}G_t\mu_{t-1}$$

$$+ R_t^{-1}g(\mu_{t-1}) - R_t^{-1}G_t\mu_{t-1}]$$

$$= (R_t + G_t\Sigma_{t-1}G_t^T)[(R_t^{-1} - R_t^{-1}G_t(G_t^T R_t^{-1}G_t + \Sigma_{t-1}^{-1})G_t^T R_t^{-1})g(\mu_{t-1})$$

$$+ R_t^{-1}G_t\mu_{t-1} - R_t^{-1}G_t\mu_{t-1}]$$

$$= (R_t + G_t\Sigma_{t-1}G_t^T)[(R_t + G_t\Sigma_{t-1}G_t^T)^{-1}g(\mu_{t-1})]$$

$$= g(\mu_{t-1})$$

$$\frac{\partial^2 L_t(x_t)}{\partial x_t^2} = \frac{(R_t + G_t\Sigma_{t-1}G_t^T)^{-1}x_t}{\partial x_t^2}$$

$$= [(R_t + G_t\Sigma_{t-1}G_t^T)^{-1}]^T = (R_t + G_t\Sigma_{t-1}G_t^T)^{-1}$$

$$\implies \overline{\Sigma_t} = R_t^{-1} + G_t\Sigma_{t-1}G_t^T$$

Therefore $\overline{bel}(x_t) = N(x_t; g(\mu_{t-1}), (R_t + G_t\Sigma_t G_t^T))$. So Line 2 of the Bayes Filter, which propagated the belief forward in time based on the state transition PDF, can be translated into the following two computations:

$$\overline{\mu_t} = g(\mu_{t-1}) \qquad \overline{\Sigma_t} = (R_t + G_t \Sigma_{t-1} G_t^T)$$

We're halfway done! The next line of Bayes filter is the update step, which updates the belief distribution based on the most recent sensor measurement:

$$bel(x_t) = \eta p(z_t \mid x_t)\overline{bel}(x_t)$$

As before, let the underlying model of the measurement PDF be given by

$$z_t = h(x_t) + \delta_t$$

where $h : \mathbb{R}^n \rightarrow \mathbb{R}^k$ is an arbitrary function, and δ_t is a random vector of normal distribution with mean 0 and covariance matrix Q_t.

Then we will once again use a first-order Taylor series expansion to approximate h, only now we will expand around our new best estimate of the robot's state: $\overline{\mu_t}$.

$$h(x_t) \approx h(\overline{\mu_t}) + \frac{\partial h(\overline{\mu_t})}{\partial x_{t-1}}(x_t - \overline{\mu_t})$$

$$= h(\overline{\mu_t}) + H_t(x_t - \overline{\mu_t})$$

where

$$H_t = \begin{bmatrix} \frac{\partial h_1(\overline{\mu_t})}{\partial x_{1_{t-1}}} & \cdots & \frac{\partial h_1(\overline{\mu_t})}{\partial x_{n_{t-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_k(\overline{\mu_t})}{\partial x_{1_{t-1}}} & \cdots & \frac{\partial h_k(\overline{\mu_t})}{\partial x_{n_{t-1}}} \end{bmatrix}$$

is the Jacobian of h evaluated at $\overline{\mu_t}$.

Then in the same manner as we did with the state transition pdf, we can calculate the mean of the measurement pdf $E[Z]$ to be $h(\overline{\mu_t}) + H_t(x_t - \overline{\mu_t})$, with covariance matrix Q_t. Thus the measurement pdf has the form

$$p(z_t \mid x_t) = N(z_t; h(\overline{\mu_t}) + H_t(x_t - \overline{\mu_t}), Q_t)$$

Note that just as before, the covariance matrix Q_t is symmetric by definition.

We will begin by rewriting the update step using the normal distribution definitions.

$$bel(x_t) = \eta_1\ p(z_t \mid x_t)\overline{bel}(x_t)$$

$$= \eta_1\ det(2\pi Q_t)^{-\frac{1}{2}} det(2\pi\overline{\Sigma_t})^{-\frac{1}{2}} exp\{-\frac{1}{2}(z_t - h(\overline{\mu_t}) - H_t(x_t - \overline{\mu_t}))^T$$

$$* Q_t^{-1}(z_t - h(\overline{\mu_t}) - H_t(x_t - \overline{\mu_t}))\} * exp\{-\frac{1}{2}(x_t - \overline{\mu_t})^T \overline{\Sigma_t}^{-1}(x_t - \overline{\mu_t})\}$$

$$= \eta_2\ exp\{-J_t\}$$

where

$$J_t = \frac{1}{2}(z_t - h(\overline{\mu_t}) - H_t(x_t - \overline{\mu_t}))^T Q_t^{-1}(z_t - h(\overline{\mu_t}) - H_t(x_t - \overline{\mu_t})) + \frac{1}{2}(x_t - \overline{\mu_t})^T \overline{\Sigma_t}^{-1}(x_t - \overline{\mu_t})$$

$bel(x_t)$, as all PDFs, must be normally distributed. We will denote its mean and covariance matrix by μ and Σ. This time there is no integral, and so we can immediately apply Lemma 1.3.4. Therefore the mean and covariance of the normal distribution representing $bel(x_t)$ will be equal to the minimum of J_t, and the inverse of its second derivative. We will now compute these values. Let us compute these values.

$$\begin{aligned}
\frac{\partial J_t}{\partial x_t} &= \frac{\partial(\frac{1}{2}x_t^T)}{\partial x_t} - \frac{\partial(\frac{1}{2}x_t^T H_t^T Q_t^{-1}(z_t - h(\overline{\mu_t}) + H_t\overline{\mu_t}))}{\partial x_t} \\
&- \frac{\partial(\frac{1}{2}(z_t - h(\overline{\mu_t}) + H_t\overline{\mu_t})^T Q_t^{-1} H_t x_t)}{\partial x_t} + \frac{\partial(\frac{1}{2}x_t^T \overline{\Sigma_t}^{-1} x_t)}{\partial x_t} \\
&- \frac{\partial(\frac{1}{2}x_t^T \overline{\Sigma_t}^{-1}\overline{\mu_t})}{\partial x_t} - \frac{\partial(\frac{1}{2}\overline{\mu_t}\,\overline{\Sigma_t}^{-1} x_t)}{\partial x_t} \\
&= H_t^T Q_t^{-1} H_t x_t - H_t^T Q_t^{-1}(z_t - h(\overline{\mu_t}) + H_t\overline{\mu_t}) \\
&+ \overline{\Sigma_t}^{-1} x_t - \overline{\Sigma_t}^{-1} - \overline{\mu_t} \\
&= (H_t^T Q_t^{-1} H_t + \overline{\Sigma_t}^{-1})(x_t - \overline{\mu_t}) + H_t^T Q_t^{-1}(h(\overline{\mu_t} - z_t)
\end{aligned}$$

$$\frac{\partial^2 J_t}{\partial x_t^2} = H_t^T Q_t^{-1} H_t + \overline{\Sigma_t}^{-1}$$

$$\therefore \Sigma_t = (H_t^T Q_t^{-1} H_t + \overline{\Sigma_t}^{-1})^{-1}$$

$$\frac{\partial J_t}{\partial x_t} = 0$$

$$\implies (H_t^T Q_t^{-1} H_t + \overline{\Sigma_t}^{-1})(\mu_t - \overline{\mu_t}) = H_t^T Q_t^{-1}(z_t - h(\overline{\mu_t}))$$

$$\implies \Sigma_t^{-1}(\mu_t - \overline{\mu_t}) = H_t^T Q_t^{-1}(z_t - h(\overline{\mu_t}))$$

$$\implies \mu_t = \overline{\mu_t} + \Sigma_t H_t^T Q_t^{-1}(z_t - h(\overline{\mu_t}))$$

Thus the update step of the Bayes Filter, which incorporated the most recent sensor measurement using the measurement PDF, can be translated into the following two computations:

$$\mu_t = \overline{\mu_t} + \Sigma_t H_t^T Q_t^{-1}(z_t - h(\overline{\mu_t})) \qquad \Sigma_t = (H_t^T Q_t^{-1} H_t + \overline{\Sigma_t}^{-1})^{-1}$$

The only other steps of Bayes' Filter involve normalizing the belief distribution, which we need not perform since we already have the mean and covariance matrix of both $\overline{bel}(x_t)$ and $bel(x_t)$, which are the only pieces of information we need to fully describe their normal distributions. Therefore we can construct a new algorithm, the EKF.

Algorithm 2 Extended Kalman Filter

1: **function** EKF_ITERATE(μ_{t-1}, Σ_{t-1}, z_t)
2: $\quad \overline{\mu_t} = g(\mu_{t-1})$
3: $\quad \overline{\Sigma_t} = (R_t + G_t \Sigma_{t-1} G_t^T)$
4: $\quad \Sigma_t = (H_t^T Q_t^{-1} H_t + \overline{\Sigma_t}^{-1})^{-1}$
5: $\quad \mu_t = \overline{\mu_t} + \Sigma_t H_t^T Q_t^{-1} (z_t - h(\overline{\mu_t}))$
6: $\quad$ **return** (μ_t, Σ_t)
7: **end function**

We could stop here and be done with it. However, the computational complexity of inverting a d by d matrix using today's methods is approximately $\mathrm{O}(d^{2.8})$ []. Line 4 of the current algorithm inverts the n by n matrix $\overline{\Sigma_t}$, where n is the dimension of the state vector x_t. So the computation of Σ_t given here is $\mathrm{O}(n^{2.8})$. It turns out that we can rewrite Line 4 and avoid inverting the state covariance matrix as so:

$$
\begin{aligned}
\Sigma_t &= (H_t^T Q_t^{-1} H_t + \overline{\Sigma_t}^{-1})^{-1} \\
&= \overline{\Sigma_t} - \overline{\Sigma_t} H_t^T (Q_t + H_t \overline{\Sigma_t} H_t^T)^{-1} H_t \overline{\Sigma_t} && \text{(by Lemma 1.3.5)} \\
&= [I - \overline{\Sigma_t} H_t^T (Q_t + H_t \overline{\Sigma_t} H_t^T)^{-1} H_t] \overline{\Sigma_t} \\
&= [I - K_t H_t] \overline{\Sigma_t}
\end{aligned}
$$

where we make the useful definition

$$
K_t = \overline{\Sigma_t} H_t^T (Q_t + H_t \overline{\Sigma_t} H_t^T)^{-1}
$$

This shifts the $\mathrm{O}(n^{2.8})$ computation of Σ_t to an $\mathrm{O}(k^{2.8})$ computation of K_t, where k is the dimension of the measurement vector z_t. Note that in order to keep this new computation of Σ_t at $\mathrm{O}(n^2)$, one must multiply matrices in the proper order:

$$\Sigma_t = \overline{\Sigma_t} - K_t(H_t\overline{\Sigma_t})$$

using the fact that the naive approach to multiplying two matrices of size $n \times m$ and $m \times o$ takes $\mathrm{O}(nmo)$ time.

For estimation problems involving state spaces of large dimensionality, n is often much larger than k, and so this change leads to greater computational efficiency, which is important since we would like to run the EKF in real-time.

Lastly, note that

$$\begin{aligned}
K_t &= \overline{\Sigma_t}H_t^T(Q_t + H_t\overline{\Sigma_t}H_t^T)^{-1} = (\Sigma_t\Sigma_t^{-1})\overline{\Sigma_t}H_t^T(Q_t + H_t\overline{\Sigma_t}H_t^T)^{-1} \\
&= (\Sigma_t(H_t^TQ_t^{-1}H_t + \overline{\Sigma_t}^{-1}))\overline{\Sigma_t}H_t^T(Q_t + H_t\overline{\Sigma_t}H_t^T)^{-1} \quad \text{(by Line 4 of Algorithm 2)} \\
&= \Sigma_t(H_t^TQ_t^{-1}H_t\overline{\Sigma_t}H_t^T + \overline{\Sigma_t}^{-1}\overline{\Sigma_t}H_t^T)(Q_t + H_t\overline{\Sigma_t}H_t^T)^{-1} \\
&= \Sigma_t(H_t^TQ_t^{-1}H_t\overline{\Sigma_t}H_t^T + H_t^T)(Q_t + H_t\overline{\Sigma_t}H_t^T)^{-1} \\
&= \Sigma_t(H_t^TQ_t^{-1}H_t\overline{\Sigma_t}H_t^T + H_t^TQ_t^{-1}Q_t)(Q_t + H_t\overline{\Sigma_t}H_t^T)^{-1} \\
&= \Sigma_tH_t^TQ_t^{-1}(H_t\overline{\Sigma_t}H_t^T + Q_t)(Q_t + H_t\overline{\Sigma_t}H_t^T)^{-1} \\
&= \Sigma_tH_t^TQ_t^{-1} \qquad (1.18)
\end{aligned}$$

which means the computation of μ_t may also be rewritten using K_t. Thus we can construct our new, more optimal EKF algorithm.

Algorithm 3 Extended Kalman Filter

function EKF_ITERATE(μ_{t-1}, Σ_{t-1}, z_t)
    $\overline{\mu_t} = g(\mu_{t-1})$
    $\overline{\Sigma_t} = (R_t + G_t \Sigma_{t-1} G_t^T)$
    $K_t = \overline{\Sigma_t} H_t^T (Q_t + H_t \overline{\Sigma_t} H_t^T)^{-1}$
    $\Sigma_t = (I - K_t H_t)\overline{\Sigma_t}$
    $\mu_t = \overline{\mu_t} + K_t(z_t - h(\overline{\mu_t}))$
    return (μ_t, Σ_t)
end function

1.3.3 Remarks

In total the EKF algorithm has a computational complexity of $max(\mathrm{O}(n^{2.8}), \mathrm{O}(k^{2.8}), \mathrm{O}(g), \mathrm{O}(h))$ per iteration, where the complexity of executing the functions g and h is unknown, but in practice are often O(1).

Chapter 2

Hardware

Now that we have covered the necessary mathematical background of the main algorithm used in this thesis, we will move on to an overview of the rover's design.

2.1 Specific Hardware Used

The hardware used in this project was chosen to minimize cost while producing a vehicle capable of navigating rough 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 rather than its sensors.

Figure 2-1: [11]

The mobile base chosen was the Lynxmotion A4WD1 Rover, shown in Figure 2-1. It was purchased as a kit including four 200 RPM DC gear motors, four 100 PPR motor encoders, and four 4.75" diameter wheels. The chassis consists of four aluminum side brackets, and polycarbonate top and bottom panels. This kit made up the bulk of the rover's cost. It was chosen for its large wheels and strong motors, increasing the rover's robustness to uneven terrain and ramps. The chassis can support up to 5 lbs overall, and a second stackable level is available as an extension. While not purchased, this second level would be an ideal place to put a tablet or small laptop.

One downside of this 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 don't see the distance traveled. This ultimately causes greater error buildup in the robot's localization. Choosing a different base which only uses two wheels would avoid this issue, and likely be cheaper. However, such a choice must be balanced with the robot's suitability for outdoor use, and room for all on-board components.

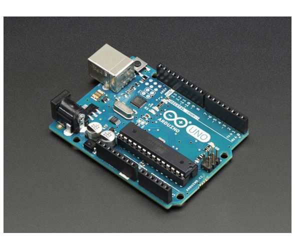
The motors are controlled by a Sabertooth 12A 6V-24V regenerative motor driver. There is a 5A version of this motor driver, which should have been purchased to save $20. This driver is dual-channeled, i.e. it controls two separate motor channels. Two DC motors are attached to each channel, so that the left and right side wheels of the rover are each controlled by a single channel. The Sabertooth 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. So the motor driver manages speed by modifying the voltage supplied to each channel. To do so it uses a technique called 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. To change the direction of rotation, an on-board circuit called the H-bridge is used to flip the polarity of voltage supplied. [7]

The motor driver is powered by two LG 18650 HE2 rechargable lithium ion cells, which sit in a battery case. This case was made out of an 18650 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. The Sabertooth motor driver has built-in lithium ion over-discharge protection, which ensures that the cells won't be discharged beyond their maximum capacity.

Situated on top of the rover is the PING))) ultrasonic distance sensor (see Figure 2-2), which is attached to a 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 2-2: [14]

Figure 2-3: [9]



At the heart of the rover is an Arduino Uno R3, shown in Figure 2-3. This microcontroller board handles low level control of the rover. It tells the motor driver what speed to set its two output channels to, and directly controls the panning motion of the Parallax servo. It also transmits sensor measurements to the computer. It's connected to this computer via a USB cable, which powers the board and allows communication over a serial port. Motor encoder values and ultrasonic range data are transmitted, and motor speed commands are received. An Arduino prototyping shield was stacked on top for soldering connections, to allow re-usability of the board.

Only two of the four available motor encoders are used, due to a limit of the microcontroller board used. Choosing a different model with more hardware interrupt pins would improve localization accuracy, without drastically increasing the price.

The computer used is a Dell Inspiron 3531 laptop, 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.

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

Both USB cables used are just under 10 feet long. 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 connectivity problems given the current length, but the cables cannot be extended much further.

Connecting electronics on the rover to the laptop via USB means the processing laptop must be manually kept within 10 feet of the rover as it navigates. Thus the rover is not truly autonomous. It could be made so by including wireless or radio communication with a server, or by using a larger chassis which would be able to simply carry the laptop. However, USB cables are cheap, and the current design still functions as a proof of concept for an autonomous rover.

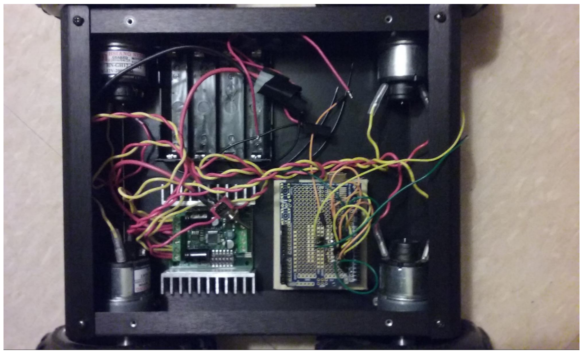
2.2 Construction

Figure 2-4 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 2-4: Constructed Chassis

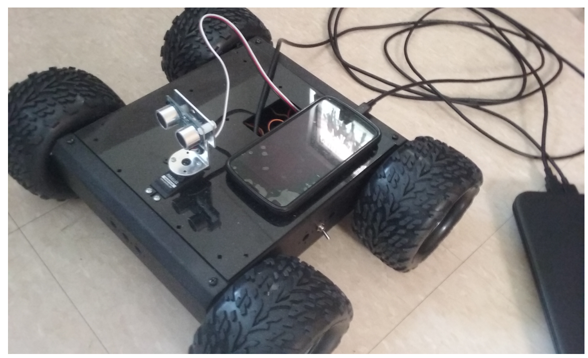
The Arduino Uno was screwed to a 2.5" x 3" x 0.5" wooden poplar block, with non-conductive nylon washers placed between the screw head and the Uno, and between the Uno and the wooden block. The wooden Arduino mounting board and the Sabertooth were both attached via double-sided foam mounting tape to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

Figure 2-5: Pieces Mounted



The servo fits conveniently into a precut 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 2-6: Construction Finished

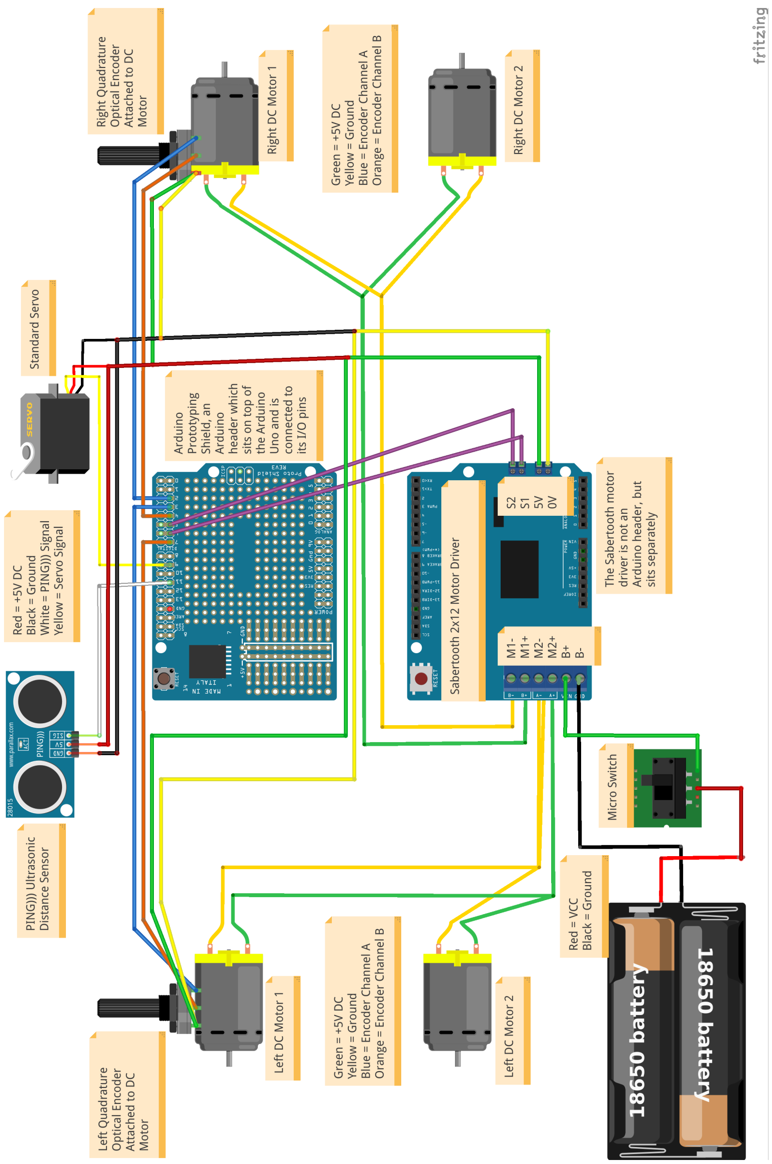


Another opening in the top panel allows the PING))) sensor to connect to the Arduino inside the body of the rover. This opening also allows the type A/B USB cable connected to the Arduino to extend out and reach the laptop. The smartphone sits on the top panel just to the left of this opening, secured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

Figure 2-7 is a schematic specifying the overall design for the rover.

Connections were made with flexible stranded core, 22 AWG breadboard wires.

Figure 2-7: Electrical Connections



This image was created with Fritzing

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.

2.2.1 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 3

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.

3.1 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.

3.1.1 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.

3.2 Arduino Uno Connections

Refer back to Figure 2-7 for a visual representation of how the Arduino Uno is connected to the other rover components.

3.2.1 Hardware Interrupt Pins

As one can see in Figure 2-7, 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 3.4.3 for more detail.

3.2.2 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 3.4.3 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.

3.3 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 3.1.1. [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.

3.4 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 3.1.1. 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.

3.4.1 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.

3.4.2 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.

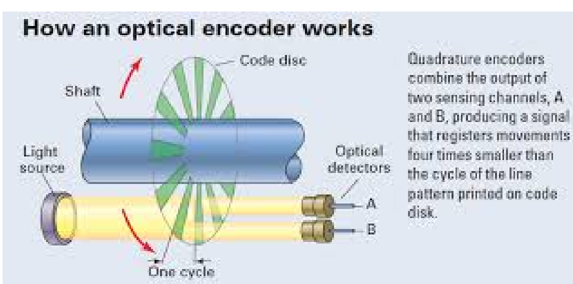
millis() is an Arduino built-in function which uses a hardware timer to count how many milliseconds have passed since the board was turned on. The callback handling that occurs while waiting reads the incoming serial buffer for any data, and if motor commands are found, an appropriate callback function is executed. publishSensorMessages() sends over serial the ping echo time, the servo's current angle, and the tick counts of both encoders. Section 4.2 will further describe how this data is passed between laptop and Arduino.

3.4.3 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 3-1: [1]

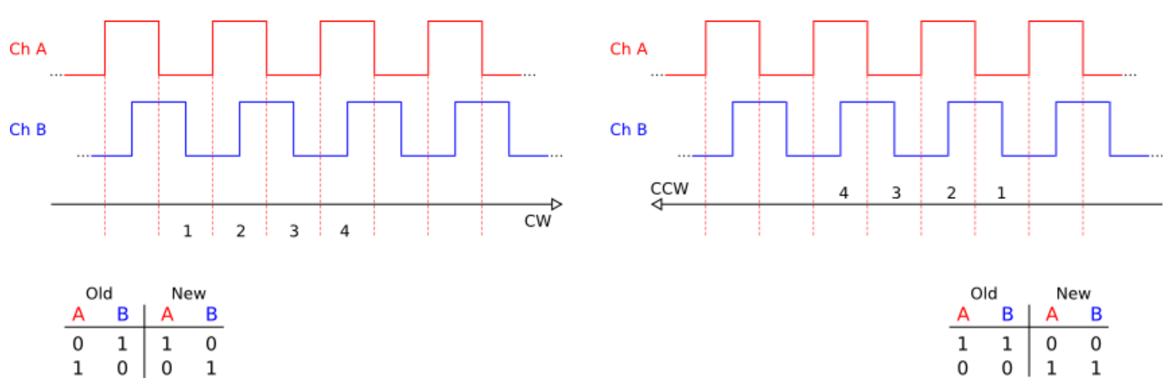


The motor encoders which came with the Lynxmotion rover kit are optical quadrature encoders. This type of encoder attaches a flat disk with thin slits known as the code disk to the motor's gear shaft. Two photodiodes, components which transform light into electric current, are placed above the disk side by side. A light source shines light through the disk from the other side. See Figure 3-1 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 3-2 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 3-2: [5]



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 3-2, 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 μs, 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 4

ROS

The software system controlling the rover and processing the incoming sensor data is built on top of the Robot Operating System (ROS). ROS is a meta operating system for open-source robotics. It provides an asynchronous messaging framework for multiple processes across multiple machines to communicate, package management for shared robotics libraries, and a method of dealing with multiple coordinate systems.

4.1 ROS Basics

4.1.1 Messaging

Processes in ROS are known as nodes. Nodes communicate by publishing and subscribing to certain communication channels called topics. The data passed over these topics are various classes of rigidly defined data structures called messages. Messages often contain meta-data such as a timestamp and sequence number, in addition to the data of interest.

When a node publishes a message to a particular topic, all nodes subscribed to that topic receive a copy. Any number of processes may publish or subscribe to a topic, making this a many-to-many communication protocol. Under the hood, messaging between nodes is handled by the ROS Master node, which acts like a DNS server and must be running for the ROS system to function.

Let's use a simple example to understand. Assume node A wants to publish messages to topic "/foo". It registers this intent with the Master. Then, say node B subscribes to topic "/foo". It too will register this with the Master. When node B does so, the Master node will notify all registered publishers on the "/foo" topic. Node A will receive the TCP/IP socket address of Node B, and store it. Then each time node A publishes a message to "/foo", it will iterate through its list of subscribed nodes, and send message data to each node's address.

4.1.2 Packages

Packages are collections of related ROS resources which all work together to perform some specific task. They often include source code for nodes, executable utilities, message definitions, and launch files. There are also meta-packages which collect various related packages.

Launch files are XML files which define several nodes to be run at once. Robotics systems grow large quickly, and require many nodes. This project ended up using 12 nodes throughout its launch files. Manually starting all of those nodes would be tedious, and launch files automate that process. They also allow convenient parameter specification for each node. Parameters are stored separately in files using the YAML syntax, and loaded dynamically into the launch files.

4.1.3 Frames and Transforms

Robotics systems include many moving parts. In order to keep track of those parts and their relation to one another, many different coordinate axes are needed. In ROS, these coordinate axes are called frames.

The frame names and definitions used in this project follow the ROS standards specified in REP-103 and REP-105 [].

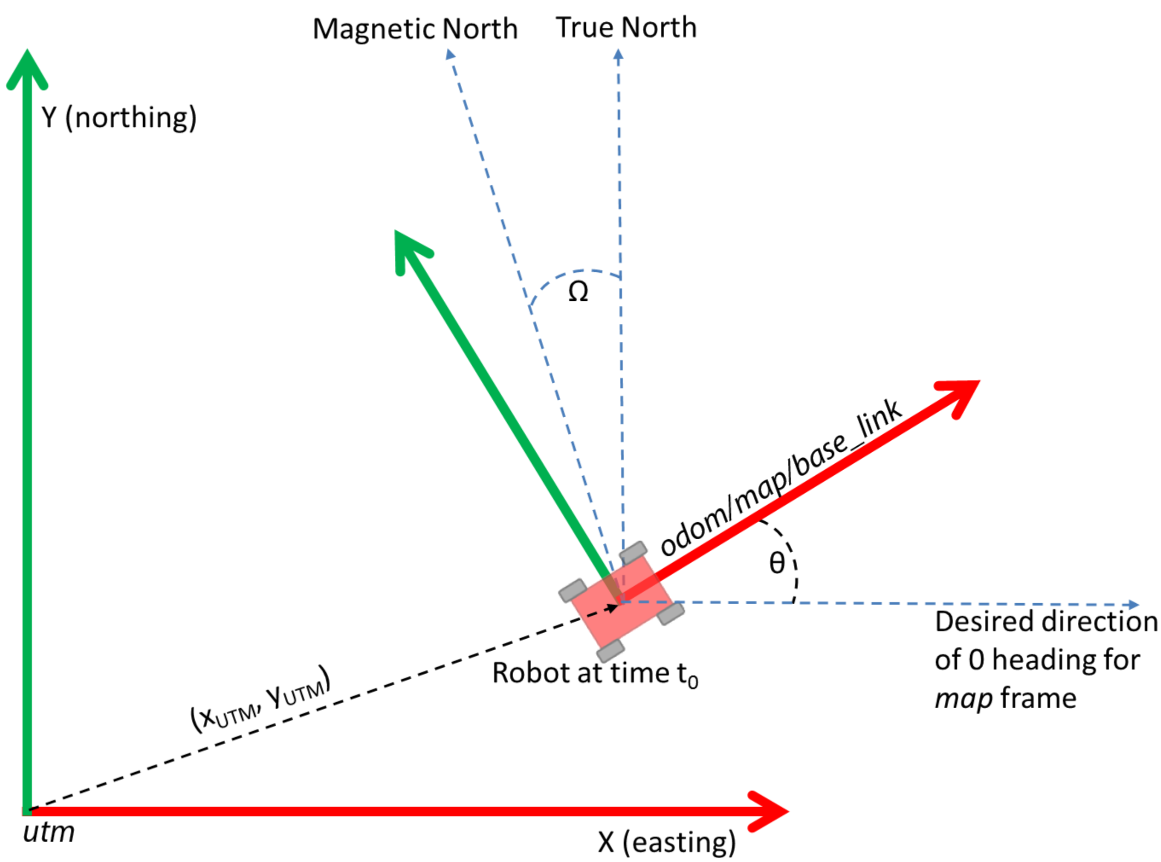
A common starting frame is the base_link frame, which has its origin at the center of a robotic chassis. This frame is attached to the rover, and moves as it moves. The X axis of the base_link frame points forward, the y-axis points to the robot's left, and the z-axis goes straight up above the robot.

Next we define a frame for the smartphone, simply called phone. The phone frame has its origin at the center of the phone, roughly two inches to the right of the base_link frame. The frame is once again attached, and moves with the smartphone. However, while the origin shifts, the orientation of the axes stays constant with respect to the earth. The X axis always points east, its Y axis points north, and its Z axis points up towards the sky, or tangential to the earth's surface. This axis orientation is referred to as the ENU convention (East-North-Up).

Another frame needed is a locally fixed frame. The origin and orientation of this frame will be the same as the base_link frame initially: origin at the center, X axis forward, Y to the left, and Z up. However, this frame will not be rigidly attached to the rover body, and will not move or rotate as the rover moves. Thus it will be able to keep track of the rover's total movement from its starting position.

We will define two such locally fixed frames: odom and map. We will use the map frame to keep track of the position estimate of the rover using all available sensors, including GPS. The GPS measurements filtered through the EKF will cause this estimate to jump around erratically, making it discontinuous. We will use the odom frame to keep track of the position estimate of the rover, using all sensors except for the GPS. This position estimate will be smooth and continuous, but the position error will grow unbounded over time.

Figure 4-1: [13]

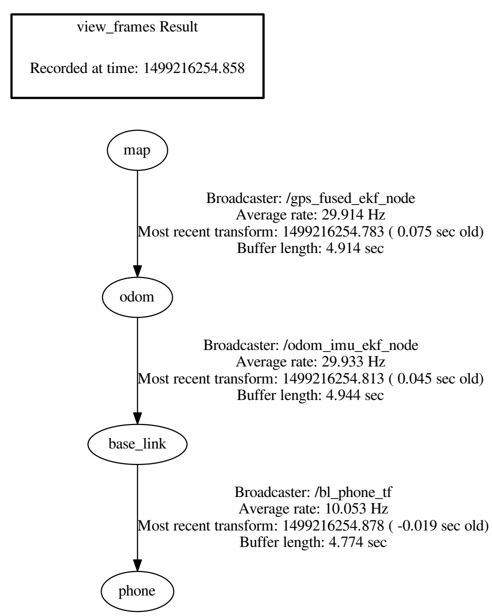


The UTM (Universal Transverse Mercator) frame defines a frame equal to the axes used by the UTM zone that the rover is currently in. The origin is at the (0,0) point of the UTM zone, and the axes are oriented according to the ENU convention.

See Figure 4-1 for a graphical representation of the rover right when it begins localizing. Notice that at that time, the base_link, odom, and map frames are all aligned. The phone frame, not pictured, would have the same origin as those three, but its axes would be oriented parallel to the UTM frame's axes. As the rover moves, the base_link and phone frames would move with it. The base_link axes would rotate as the rover rotates, and the phone axes would remain oriented as ENU. The odom and map frames would stay fixed at where they began.

Transforms are conversions between frames. They are broadcast through the ROS network to all nodes. See Figure 4-2 for a handy map of all transforms broadcast in the system, where arrows indicate an available transform from a parent frame to child frame.

Figure 4-2: Frames



4.2 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.

4.2.1 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.

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.

4.2.2 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".

4.3 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.

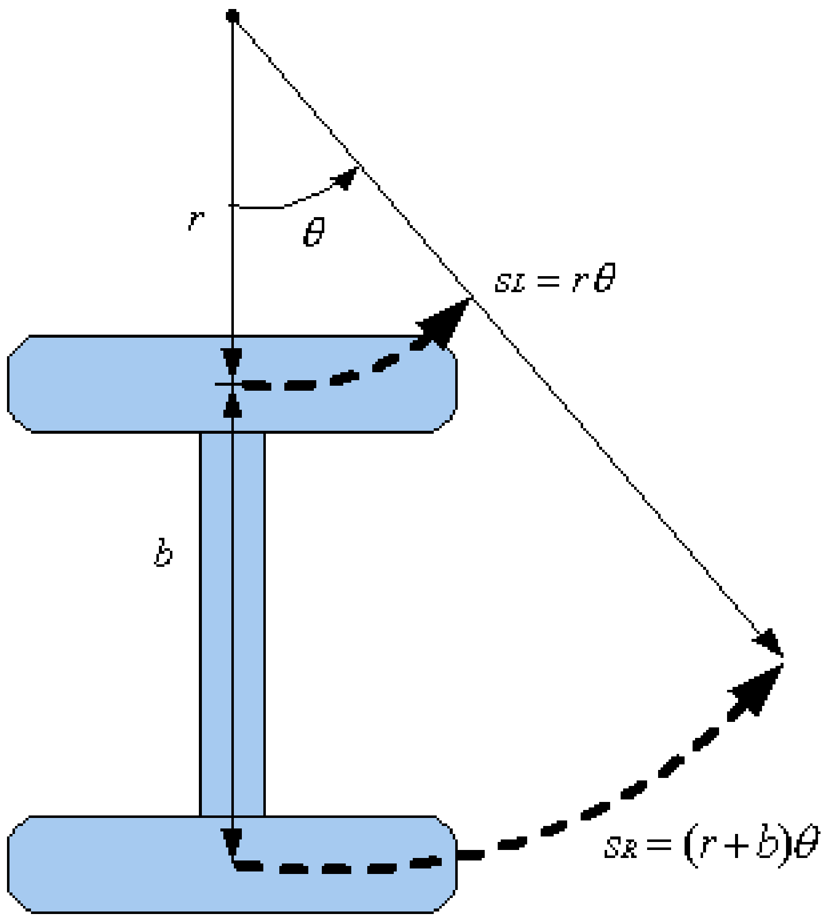
4.3.1 diff_odom

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 4.5.

First, let's take a look at the standard theory for differential wheeled robots. Figure 4-3 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

Figure 4-3: [10]



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.

$$s_M = ((r + (b/2)) * \theta = (s_L + s_R)/2 \tag{4.1}$$

$$\theta = (s_R - s_L)/b \tag{4.2}$$

Equation 4.1 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 4.2 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 computing 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 encoders' specification and the diameter of the wheels, it is a good idea to manually calibrate the number of encoder ticks per meter, and specify this as a configurable parameter to this node. This helps account for sources of error in the physical system. The ticks per meter can easily be calibrated by moving the Arduino one meter, and comparing the start and end tick count.

4.3.2 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 4.1, we know that

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

and from equation 4.2 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.

4.3.3 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.

4.3.4 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.

4.4 Ros Sensors App

In order to access the smartphone's IMU and GPS data, an Android app was written to act as a ROS node. All sensor data messages published are from the phone's frame, which is centered 2 inches to the right of the base_link frame.

The app makes use of phone tethering supported by the Android OS, which allows one to access the internet over a smartphone's data plan. However, this feature is used only as a convenient way for the phone and laptop to communicate locally over USB.

4.4.1 GPS

The Global Positioning System (GPS) was developed by the U.S. Department of Defense, and has been operational since 1995. It currently involves 31 satellites orbiting the earth in such a way that four or more satellites are always visible from most land masses. A standard GPS receiver uses a process called trilateration to pinpoint its location. []

The coastguard broadcasts fix message for gps, however the smartphone used in this project does not have the required hardware to access that.

The GPS coordinates reported by the phone node are actually using assisted GPS

(AGPS). In AGPS, cell phone towers use a phone's signal strength to help determine its position, and they also transmit almanac and ephemeris info to phones using a data connection. The hardware also reports an uncertainty estimate.[]

The phone node publishes gps messages roughly every five seconds. The covariance matrix is only filled in along the diagonal, using the uncertainty estimate reported by the Android OS.

4.4.2 IMU

Inertial Measurement Units (IMUs) are small chips that measure linear and angular movement. A digital accelerometer measures linear acceleration in all three dimensions, while a digital gyroscope measures angular velocity around each axis. Lastly, a digital magnetometer measures magnetic strength along each axis.

IMU chips often report in NED frame, however the Android API works within the ENU frame, as expected.

A covariance matrix was once again filled out on only the diagonals, with accelerometer and gyroscope variances hard-coded at the values at 0.04 $\frac{m}{s^2}$ and 0.004 $\frac{rad}{s}$. [**Nexus4Paper**] Magnetometer variance was guessed at 0.01 $\frac{deg}{s}$

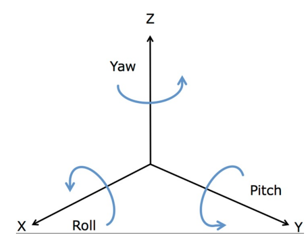
All of this information is encapsulated in a ROS IMU message, which is published at approximately 5 Hz.

4.5 robot_localization

Table 4.1: Sensor Configurations [12]

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

Figure 4-4: Roll, Pitch, and Yaw [3]



This package created by Tom Moore implements the extended Kalman Filter, the mathematical details of which were described in section 1.3. The filter 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, and Ψ represent roll, pitch, and yaw, respectively. These Euler angles describe rotation about the X, Y, and Z axes. See Figure 4-4. [12]

This project takes advantage of the fact that the rover is a ground vehicle and makes the assumption that all terrain is perfectly flat, and that the rover is moving in a two-dimensional environment. Thus z,roll, pitch, and their derivatives in the state vector are fixed at 0. This configuration should be changed for more elaborate testing, however this eliminates an entire axis of noise from GPS altitude and IMU measurements.

Table 4.1 shows which state variables each sensor affects, where a 1 indicates that that sensor gives a reading for the corresponding state variable, and a 0 indicates that it does not.

The rover's system runs three nodes from this package.

The first node runs an EKF which fuses wheel odometry from the diff_odom node (section 4.3.1) with IMU data from the phone node. It produces a state estimate in the local odom frame.

The second node provides a helper service that transforms gps coordinates into the local frame, and vice versa. Its main use is to transform gps messages coming from the phone node into coordinates in the map frame.

The third and final node fuses the odometry output from the first and second nodes together, producing a final state estimate in the map frame. This estimate is discontinuous, as the output from the second node is positional coordinates, which this node uses to instantaneously adjust the rover's x and y state variables.

Chapter 5

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].

5.1 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 4.3.4. 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 5-1 for two representations of the path taken. Figure 5-1a displays the path actually traversed, while Figure 5-1b 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 4.1 for a review of which state variables each sensor affects.

5.2 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 [15]. Figure 5-2 displays those plots.

Figure 5-2a 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 5-2b 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.

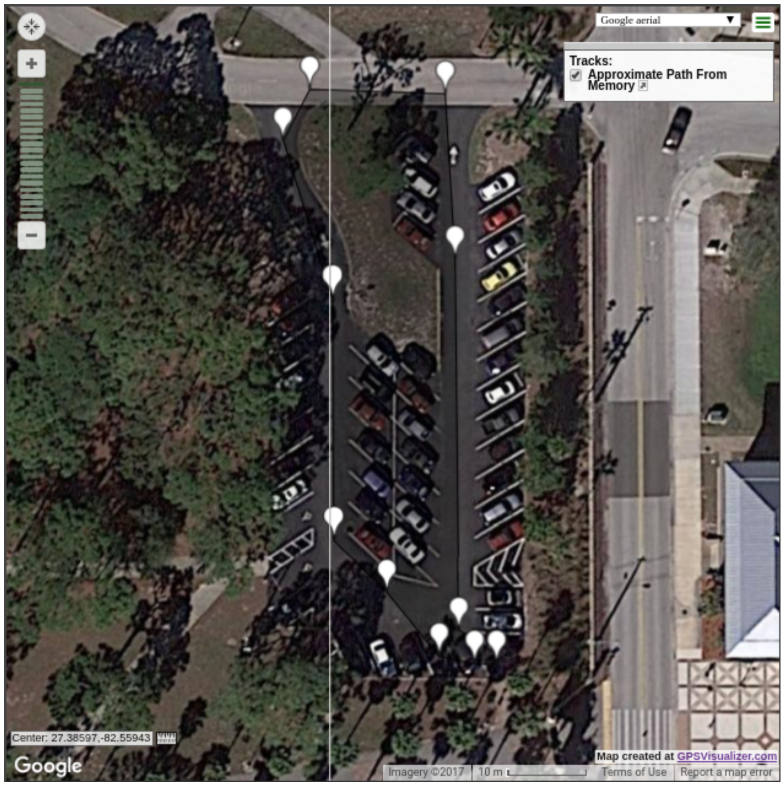
Lastly, Figure 5-2c shows the result of fusing both previous sensors with GPS fixes. This plot looks much like the raw gps plot in Figure 5-1b, however upon close inspection one can see jagged jumps in position. These jumps are instantaneous and actually lead to a discontinuous position estimate, though 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 5.1: Errors for Different Sensor Fusions [12]

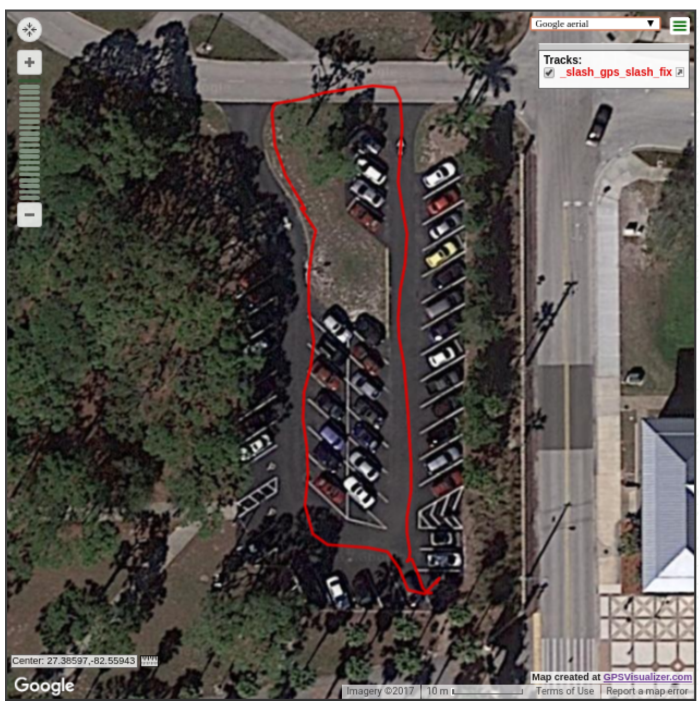
Sensors Fused	**Loop Closure Error x,y (m)**	**Filter's Std. Dev. x,y (m)**
Wheel Encoders	-88.37, -43.10	45.64, 126.36
Encoders + IMU	-12.90, -11.89	52.80, 52.02
Encoders + IMU + GPS	-0.97, -0.50	4.68, 4.56

Table 5.1 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, which puts it in the -X and -Y direction, according to ROS standards.

Figure 5-1: The rover's path.

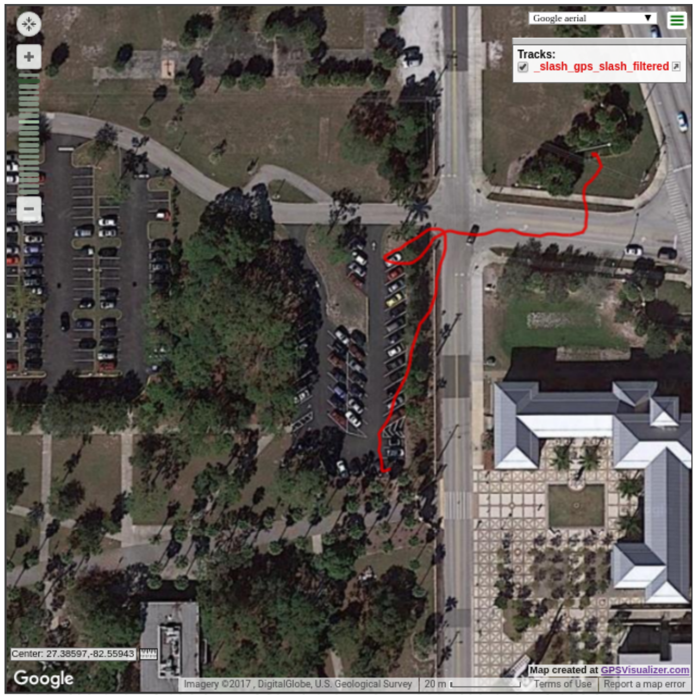


(a) Path Manually Mapped

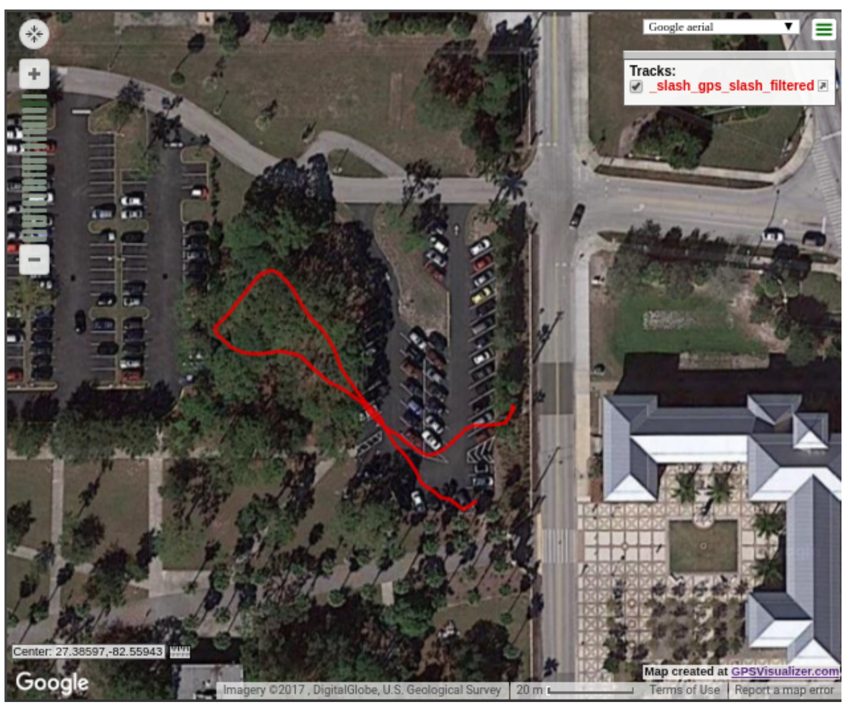


(b) Path According to Phone's GPS

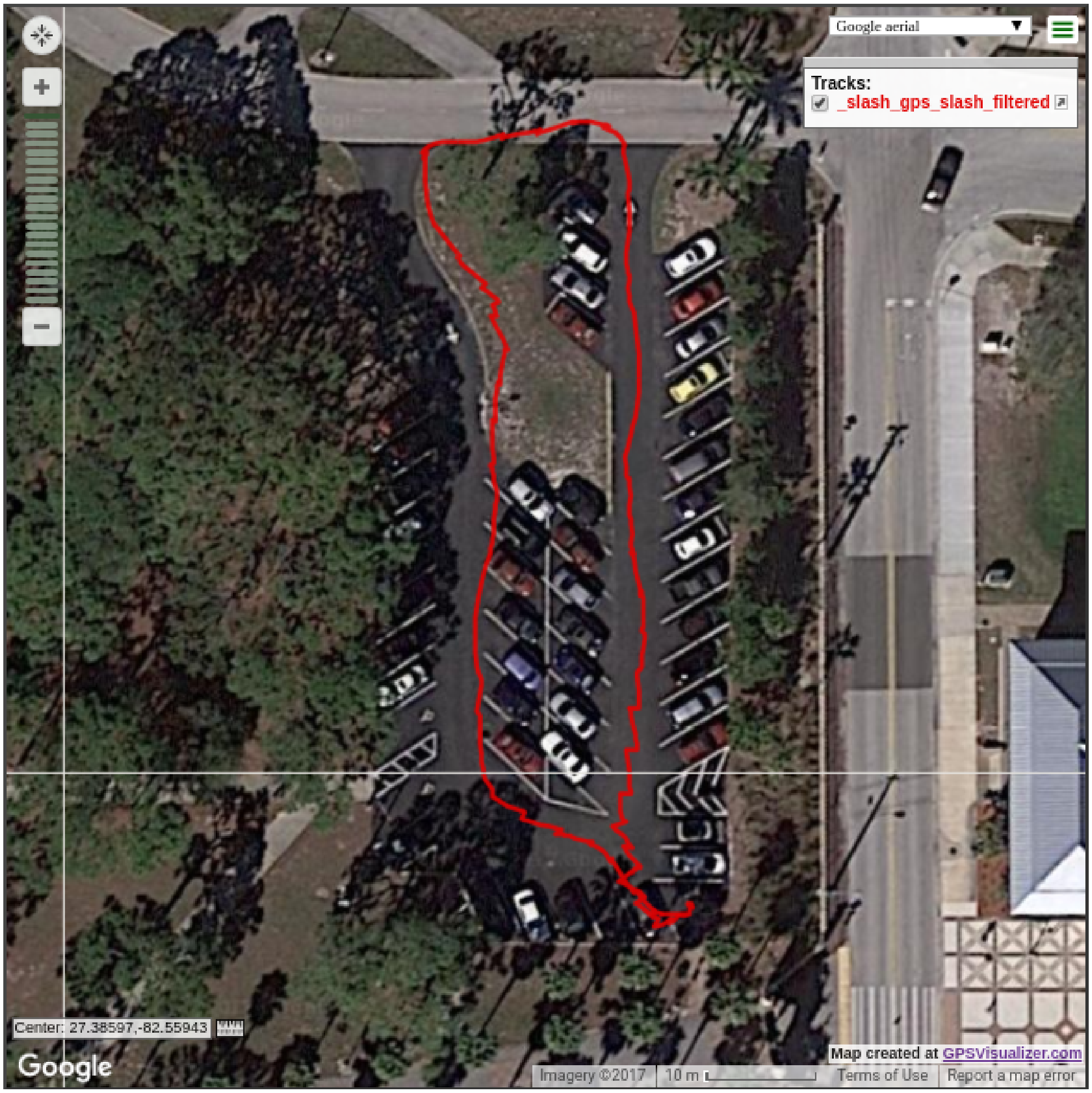
Figure 5-2: Filter Outputs For Different Sensor Fusions



(a) Raw Wheel Odometry



(b) Wheel Odometry + IMU



(c) Wheel Odometry + IMU + GPS

Conclusion

The rover's design has several flaws.

The differential drive model is a rough and messy approximation to the skid steering base, causing large errors in the yaw state variable to accumulate quickly in the wheel odometry.

The covariance matrices should be further refined, especially the wheel odometry and magnetometer variances, which were guessed at.

There were several factors which impacted the performance of the phone's magnetometer. The field test was carried out in a parking lot with running cars, which generate magnetic fields. The DC motors when running at high currents also produce a noticeable impact on the phone. If a second level were to be added on top of the rover, the extra distance from the motors could fix this issue.

This project is hoped to be a jumping off point for future development.

The logical next step to improve this project would be to integrate the ultrasonic sensor's range and angle data into the project, as a slower approximation of LIDAR range data, which will allow the system to dynamically generate a map of the immediate area around the rover. Localization can then be performed in real-time with respect to this partial map.

Another possible source of sensor data which could be added to the design is a video feed, either via the laptop's webcam or the smartphone's built-in camera. However, any camera rigidly attached to the frame would suffer from wobbling video frames due to the lack of shock absorbers. Perhaps video stabilization could be used to compensate here, in which case visual odometry could prove useful.

The Arduino microcontroller board chosen in this design could be replaced with a larger model, or by a Raspberry Pi which is a system on a chip which costs - at the time of this writing -roughly $15 more than the Arduino board, but with additional features such as built-in WiFi and Bluetooth for short-range communication.

On the software side, the differential_drive package should eventually be replaced by the diff_drive_controller package, which performs the same function but integrates more naturally into the ROS navigation stack.

The collection of software used in this project is available online at the following url: <https://github.com/NoahRJohnson/AutoRover>.

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] Tom Moore. *robot_localization wiki*. [Online: accessed July 17, 2017]. URL: http://docs.ros.org/kinetic/api/robot_localization/html/integrating_gps.html.

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

[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.