Rochester Institute of Technology
RIT Scholar Works

Theses | Thesis/Dissertation Collections

6-2017

Sensor Fusion and Deep Learning for Indoor Agent Localization

Jacob F. Lauzon
jfl4577@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Lauzon, Jacob F., "Sensor Fusion and Deep Learning for Indoor Agent Localization" (2017). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Sensor Fusion and Deep Learning for Indoor Agent Localization

By

Jacob F. Lauzon
June 2017

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Approved by:

Dr. Raymond Ptucha, Assistant Professor — Date
Thesis Advisor, Department of Computer Engineering

Dr. Roy Melton, Principal Lecturer — Date
Committee Member, Department of Computer Engineering

Dr. Andres Kwasinski, Associate Professor — Date
Committee Member, Department of Computer Engineering

R·I·T | Kate Gleason *College of* ENGINEERING

Department of Computer Engineering

Acknowledgments

Firstly, I would like to sincerely thank my advisor Dr. Raymond Ptucha for his unrelenting support throughout the entire process and for always pushing me to better my work and myself. Without him, and his mentorship, none of this would have been possible.

I would also like to thank my committee, Dr. Roy Melton and Dr. Andres Kwasinski for their added support and flexibility.

I also must thank the amazing team of people that I have worked with on the Milpet project, both past and present, that has dedicated many long hours to making the project a success. In particular I would like to thank Rasika Kangutkar for her work on ROS, path planning, and obstacle avoidance, Alexander Synesael for his work on the hardware systems, and Nicholas Jenis for his work on the mobile applications. I would also like to thank Dr. Clark Hochgraf for his helpful advice.

Abstract

Autonomous, self-navigating agents have been rising in popularity due to a push for a more technologically aided future. From cars to vacuum cleaners, the applications of self-navigating agents are vast and span many different fields and aspects of life. As the demand for these autonomous robotic agents has been increasing, so has the demand for innovative features, robust behavior, and lower cost hardware. One particular area with a constant demand for improvement is localization, or an agent's ability to determine where it is located within its environment. Whether the agent's environment is primarily indoor or outdoor, dense or sparse, static or dynamic, an agent must be able to have knowledge of its location. Many different techniques exist today for localization, each having its strengths and weaknesses. Despite the abundance of different techniques, there is still room for improvement. This research presents a novel indoor localization algorithm that fuses data from multiple sensors for a relatively low cost. Inspired by recent innovations in deep learning and particle filters, a fast, robust, and accurate autonomous localization system has been created. Results demonstrate that the proposed system is both real-time and robust against changing conditions within the environment.

Contents

List of Figures

List of Tables

Acronyms

AC

Alternating Current

ADAM

Adaptive Moment Estimation

ANNs

Artificial Neural Networks

ASIFT

Affine Scale-Invariant Feature Transform

CNN

Convolutional Neural Network

DC

Direct Current

DPDT

Double Pull, Double Throw

EKF

Extended Kalman Filter

FIR

Finite Impulse Response

FNNs

Feed-forward Neural Networks

GPIO

General Purpose Input Output

GPS

Global Positioning System

IMU

Inertial Measurement Unit

LiDAR

Light Detection and Ranging

Milpet

Machine Intelligence Lab Personal Electronic Transport

PC

Personal Computer

PID

Proportion Integral Derivative

PWM

Pulse Width Modulation

ReLU

Rectified Linear Unit

RFID

Radio Frequency Identification

ROS

Robot Operating System

RSSI

Received Signal Strength Indicator

SIFT

Scale-Invariant Feature Transform

SLAM

Simultaneous Localization and Mapping

SPST

Single Pole, Single Throw

SURF

Speeded Up Robust Features

SVM

Support Vector Machine

Chapter 1

Introduction

Localization, or a agent's ability to determine its location, has been a topic of research for decades as it is fundamental to any autonomously or semi-autonomously navigating agent. Whether the agent is intended for indoor use, outdoor use, or both, it must have an idea of its location if the intention is to travel from one point to another. Indoor and outdoor environments each contain a unique set of challenges for the task of localization. Outdoor environments are very large while indoor environments are more constrained. Indoor environments face additional challenges such the lack of access to certain technologies like Global Positioning System (GPS). These challenges govern the implementation of localization. Qualities of the agent, such as what sensors the agent has access to and how much processing power the agent has, also factor into the localization implementation. No one system is perfect for all agents and environments, but creating a system that can generalize to as many conditions as possible is important.

Over the past years, many different techniques for localization have been developed and tested. Each method requires the agent to be equipped with at least one sensory device that can take meaningful data from the environment and a software algorithm that can interpret this data and make an estimation of location. The sensor choice is heavily dependent on the environment the agent will be deployed in, while the algorithm choice is dependent on the sensor choice. Both the sensor and algo-

rithm choices come with trade-offs. Some sensors, like Light Detection and Ranging (LiDAR) sensors and GPS sensors, can be very expensive. Others, such as ultrasonic range sensors, can be too coarse to achieve high accuracy. On the other hand, sensors like high-definition cameras can provide more data than certain agents are capable of processing in a timely manner. The optimal solution for most localization systems is to maximize the accuracy and speed of the location estimate, while minimizing the required processing power and cost.

The previous work in localization spans a wide range of sensor types and algorithms but can be mostly divided into three main categories: techniques that utilize external signals, techniques that utilize sensors external to the agent, and techniques that make use of sensors attached to the agent. Using external signals such as Wi-Fi or Bluetooth for proximity detection or triangulation-based localization has been the focus of much recent research [9, 10, 11]. While these methods demonstrate promising results, they rely on specific external signals to be present in the environment. For autonomous agents that must be able to be deployed in any environment, signal based solutions fall short. There has also been significant research in localization techniques that use sensors scattered throughout the environment and external to the agent to localize the agent [12, 13]. Other techniques similar to this involve augmenting the environment to make it easier for the agent to localize itself, such as Jeon et al. [14] who placed magnets around the environment for the agent to drive over. While these solutions are fairly accurate, they still rely on the agent being deployed in a specific environment or the environment to be outfitted with sensors prior. Again, this limits the areas an agent can travel.

The third category of localization techniques is the most appealing. By using only sensors attached to the agent and not relying on external signals, the agent should be able to be deployed into any environment. Within this category there are still many different proposed solutions that make use of different sensors. The most basic

solution is dead reckoning, which is localization using only odometry data such as that from wheel encoders. Dead reckoning techniques are simple but, alone, are insufficient as they suffer from sensor and mechanical inaccuracies which, when propagated over time, become unacceptable. Dead reckoning techniques were improved by the addition of filtering algorithms and other sensors such as in [15, 16]. More recent approaches to localization utilize other sensors such as ultrasonic, LiDAR, and vision to achieve higher accuracy and build a more robust system. Fontanelli et al. [17] was able to create an accurate localization system using just LiDAR data and a modified Random Sample Consensus (RANSAC) algorithm. Vision based localization has become very popular recently due to the feature rich data a camera can provide for a relatively low cost. A significant amount of work has been done in using vision to perform image matching for localization [18, 19, 20, 21]. These techniques involve extracting certain features from images of key landmarks during training and then performing a matching algorithm as the agent captures a new image.

Part of this research was to aid in developing an indoor agent to be able to test and demonstrate the localization algorithm. The agent that was created was an affective access wheelchair that was converted to drive and navigate autonomously via speech or touch input. The wheelchair, termed Machine Intelligence Lab Personal Electronic Transport (Milpet), was developed as a research platform for all aspects of research within autonomous navigation. The wheelchair was equipped with wheel encoders, LiDAR, an omni-vision camera, an Inertial Measurement Unit (IMU), and ultrasonic range sensors for sensing input. In addition, localization was combined with other algorithms for path planning, obstacle avoidance, speech recognition, and motor control to achieve fully autonomous operation.

This thesis presents a novel approach to indoor agent localization that utilizes data from a low-cost omni-vision camera, along with ultrasonic sensors and odometry from wheel encoders, to achieve a fast an accurate localization system. The algorithm

uses only data from these sensors and does not rely on any external sensors or signals. The algorithm makes use of state-of-the-art filtering and deep learning techniques to create a robust model capable of working under many indoor conditions. In addition, the system was tested and proven on a physical agent and could be extended to any agent with a similar sensor array.

The rest of this thesis is organized as follows: a chapter investigating previous work in this area and important technologies utilized followed by a chapter discussing the platform used for testing. Next, is a chapter that discusses the methodology of the localization system followed by a examination of the testing methodology, datasets, and results. Finally, there is a brief conclusion and a discussion on possible future work on this topic.

Chapter 2

Background

2.1 Localization

Localization is defined as an agent's ability to determine its location within an environment. It is an essential technology for most autonomous or semi-autonomous agents to be successful. Agent localization has been a topic of research for many years as the popularity and desire for more autonomous agents continues to grow. Localization is required for agents in any environment, whether it be indoor or outdoor. The general principles and requirements of localization hold true for almost all environments; however, different environments may require different implementations. For example, in most outdoor environments GPS is available, where it is not in indoor environments. This means that an outdoor localization algorithm that makes use of GPS would fail in most indoor environments. It is also important to consider the agent when considering localization. Different agents are equipped with different sensor technologies, movement mechanics, and amount of processing power; all of which must be considering when developing an algorithm for any specific subset of agents.

2.1.1 Dead Reckoning

The earliest, and most crude form of localization is called dead-reckoning. Dead-reckoning involves using odometry information, trigonometry, and robot kinematics to determine how far the agent travels from its initial position. Dead reckoning is plagued by two major issues. The first is the algorithm requires knowledge of the agent's initial position, and the second is over time measurement errors cause the accuracy to decrease to an unacceptable level. Ojeda et al. [22] presented a localization system for a humans that utilized an IMU to perform dead reckoning. While the system was fairly accurate at detecting linear displacement over short distances, it struggled to remain accurate with longer walks. To improve this form of localization, error correction techniques were added to account for the accumulating sensor error. Among the most popular techniques, Thrun et al. [23] used a probabilistic approach known as particle filtering, and Kwon et al. [15] utilized extended Kalman Filters for error correction. Others tried to improve dead reckoning techniques by including additional sensor data with the odometry information. Kim et al. [16] combined dead reckoning with data from ultrasonic sensors and used a Kalman Filter to improve localization performance.

Dead reckoning techniques, even the more advanced ones, still suffer from the need to know the agent's initial position. This prevents the agent from being able to solve, what is informally know as, the "kidnapped robot problem." The "kidnapped robot problem" describes randomly placing an agent into an environment and having it be able to determine its location without any prior knowledge. This is a common problem in robotics as most modern agents must be able to determine location without an initial position. Modern localization techniques work to solve the "kidnapped robot problem" by introducing different sensors to better gather information from the environment and by utilizing more advanced mathematical or machine learning algorithms to interpret the sensor data. Most research in this area has been into the

idea of Simultaneous Localization and Mapping (SLAM) [24, 25] where a map is built by the agent and then the agent can localize itself within that map. One of the most popular SLAM techniques was proposed by Montemerlo et al. [26] called FastSLAM. FastSLAM utilizes sensor data and a particle filtering algorithm to achieve a fast and robust SLAM algorithm.

2.1.2 Signal-Based Localization

A lot of modern research on localization has been focused on observing wireless signals within the environment The most popular signals are Wi-Fi, Bluetooth, and Radio Frequency Identification (RFID) [27]. Wi-Fi techniques usually utilize Received Signal Strength Indicator (RSSI) fingerprinting with existing Wi-Fi networks to determine agent location [9, 11, 28, 29]. Techniques based on Bluetooth and RFID usually require the environment to be augmented with beacons or tags to create the necessary signals to build a "radio map" [10, 30, 31]. These methods continue to be improved and are the topic of a significant amount of recent work [32, 33, 34]. While these methods are popular and show promising results, they require the agent to be deployed in an environment where the signals are present or require the environment to be curated with signal creating devices. This limits the environments these algorithms can be used in, and therefore prevents them from generalizing to any agent or situation.

2.1.3 Sensor Networks

Another popular approach to localization is to use sensors external to the agent to track the agent within its environment. This is sometimes referred to as using a sensor network. Sensor networks can be sensors, such as cameras, used to observe the agent within the environment, augmentations to the environment, such as placing RFID tags or magnets around, or a combination of both. Choi et al. [12] provides some good

examples of sensor network localization. He uses a combination of RFID tags spread throughout the environment and an external camera to monitor the agent. Bischoff et al. [35] combined odometry data with data from an external camera using a Kalman Filter to achieve higher accuracies than odometry alone. The camera acted as a means to correct the odometry as time went on. Ramer et al. [13] added ceiling mounted cameras to overcome some of the challenges with localization using basic LiDAR and odometry data. He used the camera to aid with the initial location estimate and updating the global map with obstacles. Jeon et al. [14] provided an example of using only environment augmentation by scattering magnets and orange traffic cones around and equipping the agent with sensors to observe these augmentations to determine location. Similar to the signal based methods, sensor network localization methods do not generalize well to many environments because sensors and artificial landmarks have to be added to the environment prior.

2.1.4 Vision-Based Localization

The third modern approach to localization is to use only sensors attached to the agent and use algorithms to interpret the data from these sensors. This is the most appealing form of localization as it can generalize to many different environments and can be applied to different agents. In outdoor environments fairly accurate localization can be achieved with only GPS but in indoor environments where GPS is not present, the problem is more challenging. For indoor environments LiDAR sensors [17] or vision sensors are typically utilized with vision being the most intriguing option due to the relatively low cost of cameras, especially when considering the amount of data they provide. Some even took advantage of depth cameras for their ability to provide both depth and vision information [36, 37].

2.1.4.1 Image Matching

The most common approach to vision-based localization is image matching. Image matching usually consists of two stages, an offline stage and an online stage. During the offline stage, images are gathered from the agent's environment in specific, known locations. These images are used to build a database of images to be used later during the online stage. Typically, instead of storing the raw images, certain features are extracted from the images, such as Scale-Invariant Feature Transform (SIFT) [38] or Speeded Up Robust Features (SURF) [39]. The online stage of image matching consists of taking images from the environment, while traveling, and comparing them to those already in the database. The way the images are compared, and a position is ultimately estimated, involves a mathematical approach to determine the similarity between feature maps. Both Piciarelli [20] and Guan et al. [19] implemented a successful localization system based on image matching using SURF features. Guan et al. [19] developed a slightly more sophisticated matching algorithm by also determining a "learning line" for each image. The learning line was a line somewhere in the image that provided a reference coordinate system within the image. This line could be used in the online stage to help determine the agent's position relative to the landmark in the image. Mostofi et al. [40] combined image matching using SURF and an IMU to perform SLAM in an indoor environment. The image matching technique was used to account for drift in the measurements of the IMU. Another interesting use of image matching was introduced by Sui et al. [41]. Sui combined image matching using Affine Scale-Invariant Feature Transform (ASIFT) [42] features with a scenario selection algorithm that used a Convolutional Neural Network (CNN). The CNN was trained to classify which scenario the agent was traveling through in order to limit the possible images that could be matched.

2.1.4.2 Omni-vision

An intriguing branch of vision-based localization is the use an omni-vision, or omni-directional, camera to capture the images. Omni-vision images are usually floor to ceiling images 360° around the agent. Figure 2.1 shows an example of an omni-vision image.

Figure 2.1: An example of an omni-vision 360° image.

Images, like the one in Figure 2.1 capture more information than conventional camera images as they see all around an agent. For localization purposes, this extra information can be very helpful in identifying an agent's environment. Many have used omni-vision images for the purposes of localization. Hesch et al. [43] used omni-vision images along with a Kalman filter for the purpose of SLAM, and Menegatti et al. [44] utilized an omni-vision camera as a range finder to achieve an accurate localization system. Most localization techniques using omni-vision images still rely on image matching using landmark features [45, 46]. Resch et al. [47] argued that conventional image features, such as SIFT or SURF, do not work with omni-vision images due to various distortions and developed a 3D orientation descriptor to be used in omni-vision matching tasks.

Wong et al. [1] took a different approach to omni-vision based localization by using the captured images to train a machine learning classifier to predict where the image was taken. This is similar to image matching, except that the classifier is essentially performing the matching algorithm. Wong did not use any feature extraction directly on the images; rather dimensionality reduction was used to extract a low dimension representation of the image to train the classifier. The classifier performed with an accuracy of around 80% which, when combined with a particle filtering algorithm, created a successful localization system. This work is intriguing as it eliminates the need to perform feature extraction for image matching during the offline stage of localization. In addition, most classifiers are able to make predictions very quickly after being trained which could be beneficial for real-time localization during the online stage.

2.2 Filtering

Filtering techniques are mathematical algorithms designed to mitigate the effect of noise or uncertainty from estimation or measurement tasks. This can be very beneficial in localization systems as there is typically a level of uncertainty with sensor measurements and/or pose estimation. Most localization algorithms developed in the past have used some form of filtering, including [15, 16, 23, 35, 43, 1] to name a few. Two of the most popular filtering algorithms used for localization are Kalman filters and particle filters.

2.2.1 Kalman Filters

Kalman filtering is a mathematical filtering algorithm that uses a series of measurements that contain measurement and process noise, to estimate unknown variables with more accuracy than would be possible with a single measurement. In general, Kalman filtering is an iterative algorithm that uses Bayesian inference to estimate

the probability distribution of the unknown variables. The algorithm can run in real-time and requires only the current input measurement and the previous state of the system. Kalman filters were first introduced by Rudolf Kalman in 1960 [48] to solve classical problems in control theory. They quickly became extremely popular and, today, are heavily used in many fields including navigation, signal processing, and robot control.

Kalman filters work by continuously estimating some output and an associated uncertainty, based on a measurement of the value and the uncertainty of that measurement, along with the previous output value and its uncertainty. The first step to using a Kalman filter is to ensure the system follows a stochastic linear model and to determine the parameters of this model. The equations of a typical Kalman filter model are shown in (2.1) and (2.2).

$$x_k = Ax_{k-1} + Bu_k + w_{k-1} \tag{2.1}$$

$$z_k = Hx_k + v_k \tag{2.2}$$

In (2.1) and (2.2), k denotes the current iteration step. So Equation (2.1) shows the equation for the current estimate of the unknown variable x_k which is a linear combination of the previous value of the variable x_{k-1}, the control signal u_k, and the **process noise** from the previous iteration w_{k-1}. Equation (2.2) shows the equation for the current measurement value z_k which is a linear combination of the unknown variable and the **measurement noise** v_k. The variables A, B, and H are generally matrices that determine the weight given to each component of the equation. These values are system dependent and must be determined beforehand; however, in most systems, they can be scalar values or even have a value of one. The noise values w_{k-1} and v_k are assumed to come from a Gaussian distribution with a mean of zero and a

covariance matrix of Q and R respectively. Q and R must be estimated beforehand, as well, but need not be perfect as the algorithm tries to converge to correct estimators.

If the system can fit into the Kalman filter model described above, the Kalamn filter process can be applied in the system. The process runs iteratively and consists of two steps: the **time update** and the **measurement update**. Each step has a set of equations that must be solved to determine the current state. The equations for the time update step are shown in (2.3) and (2.4).

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k \tag{2.3}$$

$$\hat{P}_k^- = AP_{k-1}A^T + Q \tag{2.4}$$

Equation (2.3) shows the calculation of the **prior estimate** $\hat{x}_k^-$ which is the estimate before the measurement update. This is a linear combination of the previous prior estimate $\hat{x}_{k-1}$ and the control signal. Equation (2.4) shows the calculation for what is called the **prior error covariance** $\hat{P}_k^-$ which is an estimation of the error before the measurement update. These prior values are used in the measurement update shown in (2.5), (2.6), and (2.7).

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \tag{2.5}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \tag{2.6}$$

$$P_k = (I - K_k H)\hat{P}_k^- \tag{2.7}$$

Equation (2.6) shows the equation for x at time k which is the desired output from the Kalman filter. It requires the prior estimate, the measurement z_k and the **Kalman gain** K_k. The Kalman gain is a weight given to the measurement at time

step k. It is a hidden value that is dependent on the noise estimation and the prior error covariance. After $\hat{x}_k$ is calculated, the final step of the measurement update is to update the error covariance, P_k. The error covariance is a measure of the variance associated with the calculated output value which is used and updated each iteration. The two steps, the time update step and the measurement update step, are repeated to iteratively estimate the variable value. The value may oscillate slightly in the first few iterations but will converge quickly as the error estimate becomes more accurate.

The drawback to this algorithm is that the system must adhere to the linear model described in (2.1) and (2.2). Many systems, however, cannot be modeled with a linear system. To solve this, researchers introduced the Extended Kalman Filter (EKF), an extension of Kalman filters that could be used with nonlinear systems. The math involved with EKFs is out of scope of this research but they work by linearizing model parameters using a Taylor approximation to fit to the linear model. In addition to EKFs, other algorithms have been introduced for nonlinear systems such as unscented Kalman filters [49, 50].

2.2.2 Particle Filters

Particle filters are another popular filtering technique used for localization tasks. Particle filters are a set of algorithms that work to estimate the internal state of a system based on repeated observations. They are a member of a more generic type of algorithm, called Monte Carlo methods, that utilize observations over many random samples to estimate a state. Particle filters, in their generic form, were introduced in the 1970s by Handschin [51] but were not considered for localization tasks until the modern form was developed by Gordon et al. [52]. Thrun et al. [25] popularized the use of particle filters for localization by using them to perform SLAM. Particles filters can be applied with data from almost any type of sensor, making them applicable for most localization applications. They have become very popular as they can provide

a quick location estimate, while handling ambiguities in sensor data and supporting re-localization without adding algorithm complexity.

More technically, particle filters iteratively approximate the posteriors from a partially observable, controllable Markov Chain with discrete time [53]. The goal is to estimate the state, x_t at time t based on the previous state x_{t-1} which follows some probability distribution $P(x_t|u_t, x_{t-1})$, referred to as the *motion model*, where u_t is some control over the time interval. In a robotic system the state is never known but can be measured. The measurement, z_t, is an estimation of x_t based on some probability distribution $P(z_t|x_t)$, usually referred to as the *measurement model*. The initial state, x_0, is drawn from another distribution, $P(x_0)$, which is typically random in nature. There can also be multiple controls, $u^t = u_0 \dots u_t$, and multiple measurements, $z^t = z_0 \dots z_t$, that are factored into the motion model. The challenging portion of solving Markov chain problems is estimating the state probability from the measurements. In general, this can be accomplished using Bayes filters [54] in the form of (2.8).

$$P(x_t|z^t, u^t) = A \times P(z_t|x_t) \int P(x_t|u_t, x_{t-1}) \times P(x_{t-1}|z^{t-1}, u^{t-1}) dx_{t-1} \tag{2.8}$$

Equation (2.8) shows the mathematical way to recursively compute the system state, x_t, given the current measurements, z^t, and controls, u^t. This equation works under the initial conditions $P(x_0|z^0, u^0) = P(x_0)$. This equation also has only a closed form solution in discrete cases and some specialized continuous cases. Most robotic systems are applied in continuous spaces that may not fit the specialized cases, therefore (2.8) cannot be used. Particle filters are a general way to solve for this distribution in any sample space. The idea is to estimate a distribution of sample states, $\{x_t^{[i]}\}$, known as particles, where x_t^i is one of N number of sample

states. Algorithm 1 shows the basic form of the particle filter algorithm.

Algorithm 1 Basic Particle Filter Algorithm.

1: Initialize N particles from $P(x_0)$, denote X_0
2: **while** True **do**
3: Generate $x_t^{[i]}$ for each $x_{t-1}^{[i]} \in X_{t-1}$ by drawing from $P(x_t|u_t, x_{t-1}^{[i]})$, denote $\bar{X}_t$
4: Draw N particles from $\bar{X}_t$ (with replacement) from $P(z_t|x_t^{[i]})$, denote X_t
5: **end while**

The particles are first initialized from the distribution $P(x_0)$. Next, in a loop, the particles from the previous time step are updated based on the motion model and then N new particles are sampled, with replacement, from the measurement model. In some systems, such as mobile agents, the motion model, $P(x_t|u_t, x_{t-1}^{[i]})$, is measured, rather than sampled, from wheel encoders or other similar sensors. The measurement model, $P(z_t|x_t^{[i]})$, usually comes from sensor data and the uncertainty associated with that data. This basic algorithm can generalize to many different types of problems, such as agent localization. With localization, the particle filter algorithm attempts to estimate location based on sensor information. The particles are initialized at random locations within the map and iteratively resampled based on measurement data. Algorithm 2 shows a general particle filter algorithm for agent localization.

Algorithm 2 General Particle Filter Algorithm for Localization.

1: Initialize particles with a random location
2: **while** True **do**
3: Update particles based on robot movement
4: Observe sensor measurements
5: Assign weight to each particle by comparing with sensor data
6: Resample particles based on weight
7: Estimate location from particle locations and weights
8: **end while**

Algorithm 2 shows the general flow of how a particle filter algorithm can be used for agent localization. The particles are initialized within the map and then iteratively updated with robot motion, weighted with measurement data, and resampled based

on weight. This matches the general form of particle filters in Algorithm 1 with the motion model coming from robot motion and the measurement model coming from sensor measurement. Figure 2.2 shows a localization example using a similar algorithm.

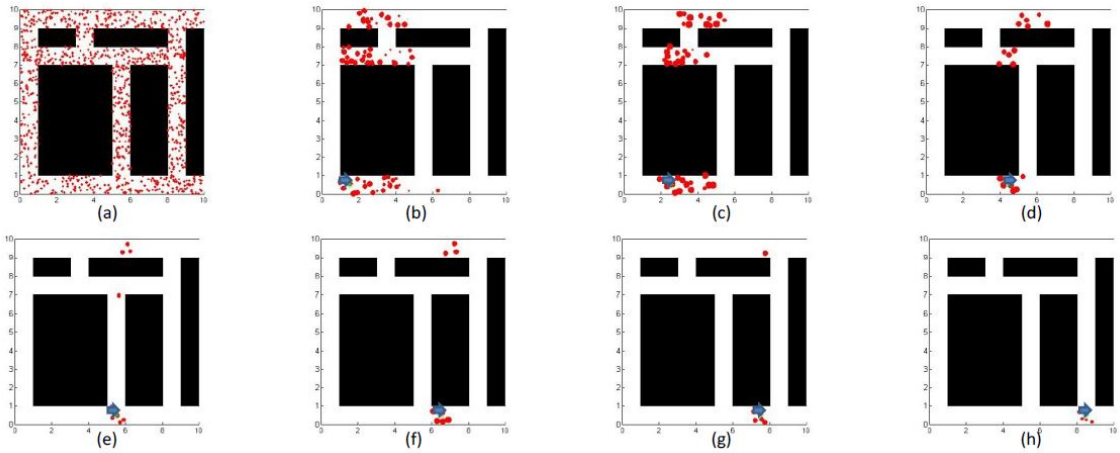

Figure 2.2: A particle filter localization example [1].

In Figure 2.2 each image shows a simulated map where white areas are free space and black areas are obstacles. The red dots are particles, and the blue arrow shows the actual position of the agent. As the agent is moving throughout the hallway from (a) to (h), the particles eventually converge on the actual location.

2.3 Support Vector Machines

A Support Vector Machine (SVM) is a very popular machine learning algorithm developed for the task of classification. SVMs were first introduced in 1995 by Cortes and Vapnik [55] and quickly became an extremely popular technique for many different tasks including pattern recognition [56], text categorization [57], and face detection [58]. SVMs became a pinnacle in the machine learning community as they performed well on many different tasks. An SVM is a binary classifier that, in general, works by finding a separating hyperplane between the data of two different classes. The al-

gorithm works to find the optimal hyperplane that maximizes the minimum distance from the plane to the nearest data point of each class.

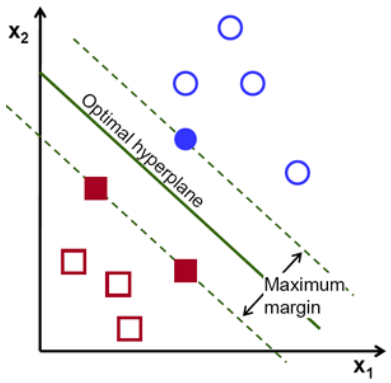

Figure 2.3: A basic 2D example of the separating plane created by an SVM [2].

Figure 2.3 shows an example of the optimal hyperplane computed by an SVM in a simple 2D example. The hyperplane is considered optimal when the distance from it to the nearest data points, known as the *margin*, is maximized. To make the mathematics more convenient, it is said that all points on the optimal hyperplane are equal to zero, all points on one margin are equal to one, and all points on the other margin are equal to negative one. This allows for the equation of the hyperplane to be written as it is in (2.9).

$$|w^T x_m + b| = 1 \tag{2.9}$$

In (2.9), x_m is the ground truth data of any dimension, w^T is the weights of the hyperplane and b is the intersection point of the hyperplane. To find the hyperplane, both w^T and b need to be solved for. Solving for w^T can be done with a constrained minimization problem. Part of solving this problem is to form a Lagrange formulation and solve for the Lagrange multipliers. Equation (2.10) shows the equation used to

solve for the Lagrange multipliers α.

$$\min_{\alpha} \frac{1}{2}\alpha^T \begin{vmatrix} y^{(1)}y^{(1)}x^{(1)^T}x^{(1)} & y^{(1)}y^{(2)}x^{(1)^T}x^{(2)} & \cdots & y^{(1)}y^{(n)}x^{(1)^T}x^{(n)} \\ y^{(2)}y^{(1)}x^{(2)^T}x^{(1)} & y^{(2)}y^{(2)}x^{(2)^T}x^{(2)} & \cdots & y^{(2)}y^{(n)}x^{(2)^T}x^{(n)} \\ \vdots & \vdots & \vdots & \vdots \\ y^{(n)}y^{(1)}x^{(n)^T}x^{(1)} & y^{(n)}y^{(2)}x^{(n)^T}x^{(2)} & \cdots & y^{(n)}y^{(n)}x^{(n)^T}x^{(n)} \end{vmatrix} \alpha + (-1)\alpha \quad (2.10)$$

The first step in solving for α in (2.10) is to form a matrix of each sample compared with every other sample. Each value in this matrix is the dot product of one training sample with another. This forms an $n \times n$ matrix of scalar values where n is the number of training samples. One of the limitations of SVMs is that if the training set grows very large, it becomes very difficult to compute the solution. To solve for α in (2.10) a standard convex quadratic programming package can be used. The equations and specific operations for solving this equation are out of scope of the this research. Once the values of α are known, they can be plugged into (2.11) to solve for w.

$$w = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} \quad (2.11)$$

In (2.11), $y^{(i)}$ is the ground truth label of each training sample, and $x^{(i)}$ is the data from each training sample. While this seems like a large summation if the number of training samples is large, most of the values of α are zero and can be ignored in this calculation. After w is known, the value of b can be solved using one training point that falls on a margin, known as a *support vector*, with (2.12).

$$y_{sv}(w^T x_{sv} + b) = 1 \quad (2.12)$$

In (2.12), y_{sv} and x_{sv} are the ground truth label and data from a single support vector respectively. Once w and b are known, they can be used to classify an unseen

test point x_{test} by using (2.13).

$$class(x_{test}) = sign(w^T x_{test} + b) \tag{2.13}$$

Equation (2.13) is used to classify an unseen test sample, x_{test}, by evaluating the sign of $w^T x_{test} + b$. If the sign is positive, x_{test} belongs to the $+1$ class; if it is negative it belongs to the -1 class. An SVM can be expanded to handle more than two classes by creating many one-versus-all classifiers where, for each class, a hyperplane is created to separate that class from all other classes.

SVMs were praised for their flexibility to be used on many different classification tasks and for being very fast at test time. While other classification methods have become more popular, such as neural networks and deep networks, SVMs still remain popular today for basic classification tasks, especially ones where only a small number of training samples are available. There have been advancements to the original SVM algorithm that have helped to keep it relevant. For example the Kernel Trick [59] helped to handle inseparable data and higher dimensional spaces. Also, in 2001, Chang and Lin introduced LibSVM [60], a software package for working with SVMs. The package handles all of the complicated math behind the scenes and lets the user focus on their classification problem. LibSVM quickly became the most popular tool for developing with SVMs with wrappers for most programming languages, including Java, C++, Python, and MATLAB. The introduction of this library aided in the widespread adoption of this technique by making it easy for anybody to train and deploy an SVM.

2.4 Deep Learning

Deep learning is a series of powerful machine learning algorithms that are used to learn abstract representations of data. The representations can be used to perform tasks

such as classification, detection, or other machine learning tasks. At a high level, deep learning techniques utilize a cascade of nonlinear units, usually called layers, where the output of one layer is the input to the next. This allows for a hierarchical representation to be learned, with the first layers learning low level features and later layers learning higher level features based on the previous layers. Like other machine learning algorithms, deep learning algorithms can be either supervised or unsupervised; however supervised is much more common. Deep learning techniques have become extremely popular over the past few years due to their performance on many different tasks. One of the first successful implementations of deep learning was in 2012 when Krizhevsky et al. [61] won the ImageNet classification competition by a significant margin using a deep Convolutional Neural Network. After that, researchers began applying deep learning to increasingly more sophisticated problems and out-performing conventional methods. In addition, a lot of effort has gone into designing better architectures and new techniques that continue to improve deep learning models. Today, deep learning can out-perform conventional methods on a large number of machine learning tasks and can be used to solve new problems that were not possible before.

2.4.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are one the most popular deep learning architectures for working with structured data such as images, videos, or audio, where the data can be easily filtered. The inspiration for CNNs comes from Artificial Neural Networks (ANNs), specifically Feed-forward Neural Networks (FNNs) where the output from one layer is the input to the next layer. In typical FNNs the layers are all fully-connected, meaning that each neuron in one layer has a connection, and an associated weight, to each neuron in the next layer. The number of required connections in such a network rapidly grows, as the input size increases, to an unreasonable

level. For example, if the input to such a network was from a VGA camera, meaning 640x480x3 pixels, there would be 921,600 weights between an input neuron and a single hidden neuron. Additionally, the first hidden layer would need to consist of thousands of neurons to handle the dimensionality of the input, leading to a model with billions of weights that all need to be learned. It is very difficult to work with this many weights, both in terms of memory requirements and the required computation power. CNNs mitigate this issue with the use of convolution filters instead of fully connected layers.

2.4.1.1 CNN Architecture

Like FNNs, CNNs are made up of many layers that feed from one to the next. The layers fall into four main types: convolution layers (CONV), nonlinearity layers (RELU), typically a Rectified Linear Unit (ReLU) is used for this, pooling layers (POOL), and fully connected layers (FC).

2.4.1.2 Convolution Layers

CONV layers are the most important layer of a CNN as these layers are where most of the learning occurs. The input to a CONV layer is a 3D structure with a height, width, and depth, rather than a 1D vector as used in FNNs. CONV layers replace the hidden neurons in standard FNNs with a family of convolution filters. Instead of learning connection weights, a CNN learns the filter values. These filters are 3D Finite Impulse Response (FIR) filters identical to those used in signal processing. Each filter in the filter bank is convolved with the input to that layer to form a filtered output. Figure 2.4 shows a comparison between a typical FNN architecture and that of a CNN. The 1D input (shown in red) of the FNN is replaced by a 3D structure. The input structure for an image would be $height \times width \times 3$ (red, green, and blue).

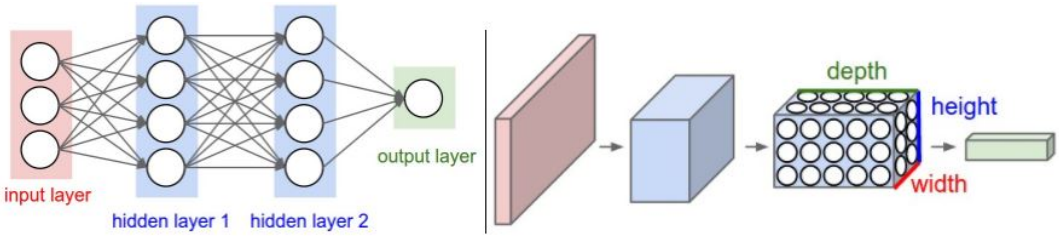

Figure 2.4: Comparison between a standard FNN structure (left) and a CNN structure (right) [3].

Convolution layers introduce many hyperparameters to a CNN architecture. Hyperparameters are parameters of the model that must be chosen ahead of time rather than learned. In typical FNNs the hyperparameters are parameters such as number of layers and number of neurons per hidden layer. With CNNs the hyperparameters have to do with the CONV layers and are usually different for each layer. The main hyperparameters are described below.

Receptive Field

For large input sizes it is difficult to connect every neuron to all neurons in the previous layer. Instead, neurons are connected to only certain regions of the input structure. The spatial extent of this connection is called the receptive field and is analogous to the convolution filter size within a CONV layer. Figure 2.5 depicts an example of how the receptive field connects a part of the input image to a single neuron in the next layer.

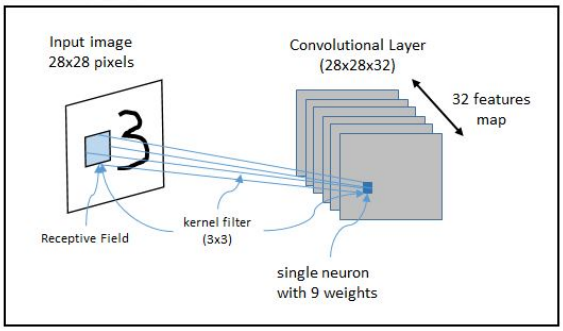

Figure 2.5: Example of the receptive field in a CNN [4].

While the depth of the filter must match the depth of the input structure, the height and width are hyperparameters. Choosing larger values will consider larger regions of the input but also have more parameters to learn. Typically, the height and width are chosen to be small odd numbers and equal to each other such as 7×7 or 3×3.

Number of Filters

The number of filters in a CONV layer refers to the number of separate filters that will be learned for that layer. Each filter can learn a different structure within the input image. For example, there may be a filter for vertical edges, another for horizontal edges, and a third for blobs within the image. The number of filters will determine how many different structures the layer looks for and will determine the depth of the output structure.

Stride

The stride is how many pixels each filter is moved while sliding it through the image structure. The stride, together with the filter size, will determine the size of the output structure. If the stride is too large important information may be lost between layers, and if the stride is too low, the filters may overlap and create redundancy between layers. Typically, values of 1 or 2 are used with small filter sizes.

Zero Padding

Zero padding involves adding zeros around all sides of the input structure for filtering purposes. The number of rows and columns of zeros to add is a hyperparameter and is typically based on the filter size. Usually zero padding is used to preserve the input size in the output size. For example, using a 3×3 filter would reduce the size of the output image by 2 pixels in each dimension as the filter is forced to start one row and

column from the edge of the input. Adding a single row or column of zeros to each side of the input would allow for the output structure to match the size of the input structure.

Output Structure Size

All of the above hyperparameters will determine the size of the output structure based on the size of the input structure. Equations (2.14), (2.15), and (2.16) show the height, width, and depth of the output structure, H_o, W_o, and D_o respectively, based on the height, width, and depth of the input structure, H_i, W_i, and D_i, the height and width of filter, the number of filters, the stride, and zero padding, H_{rf}, W_{rf}, K, S, and P respectively.

$$H_o = \frac{H_i - H_{rf} + 2 * P}{S} + 1 \tag{2.14}$$

$$W_o = \frac{W_i - W_{rf} + 2 * P}{S} + 1 \tag{2.15}$$

$$D_o = K \tag{2.16}$$

Figure 2.6 shows an example of a basic CNN architecture where the hyperparameters of each layer are specified. The figure shows the effect of these hyperparameters on the size of the output structure from layer to layer. The input to the network is a $64 \times 64 \times 3$ image, and the final output is a single 1D vector of length 10. This network also makes use of max pooling layers which are described later in Section 2.4.1.4.

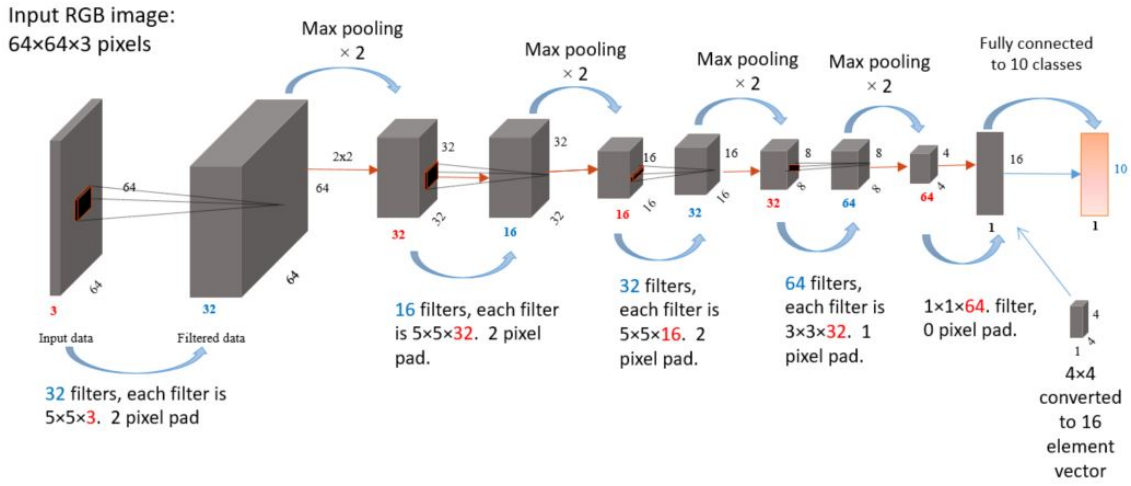

Figure 2.6: Example of hyperparameters of a CNN and their effect on image size [5].

2.4.1.3 Nonlinearity Layers

Each neuron in the fully connected layers of traditional FNNs contains a nonlinear activation function to allow the network to learn complex nonlinear functions. Since the CONV layers replace the fully connected layers, a nonlinear activation function must be introduced to the system in the form of another layer following each CONV layer. In most CNN architectures the ReLU activation function is used. The ReLU function, shown in (2.17), simply computes the maximum between the input and zero. ReLUs have become very popular as they are very easy to compute and drastically reduce training time.

$$f(x) = max(0, x) \tag{2.17}$$

2.4.1.4 Pooling Layers

Most CNN architectures work to reduce the spatial dimensionality when moving from the input layer to the final output. This is desired to reduce the number of learned parameters and to allow the model to learn many simple features in the beginning and fewer complex features towards the end. The popular way of reducing the spatial

dimensions is through pooling methods. Pooling downsamples each depth slice of the input to produce a smaller output. The goal is to reduce the dimensionality while maintaining structure and information.

Pooling works by sliding a window through the input and replacing the values in the window with a single new value. The size of the window, called the receptive field, and the stride used when sliding are both hyperparameters of the pooling operation. In addition, there are a few different techniques for determining the value to replace the receptive field with. The output height width and depth, H_o, W_o, and D_o respectively, can be determined as a function of the input height, width, and depth, H_i, W_i, and D_i, the size of the receptive field, $H_{rf} \times W_{rf}$, and the stride, S, using Equations (2.18), (2.19), and (2.20).

$$H_o = \frac{H_i - H_{rf}}{S} + 1 \tag{2.18}$$

$$W_o = \frac{W_i - W_{rf}}{S} + 1 \tag{2.19}$$

$$D_o = D_i \tag{2.20}$$

There are only two hyperparameter sets for pooling that are typically used. The most common is a receptive field size of 2×2 with a stride $S = 2$, but a receptive field size of 3×3 with a stride $S = 2$ is also sometimes used. Using larger receptive field sizes often destroys too much information to be useful. There are a few common ways to perform the pooling operation which are described below.

Max Pooling is the most common pooling technique. Max pooling simply replaces all of the elements within the receptive field with the single maximum value from the field. This is currently the most popular method as it is powerful while remaining very simple to compute. Figure 2.7 shows an example of image downsampling via pooling (left) and the max pooling technique (right).

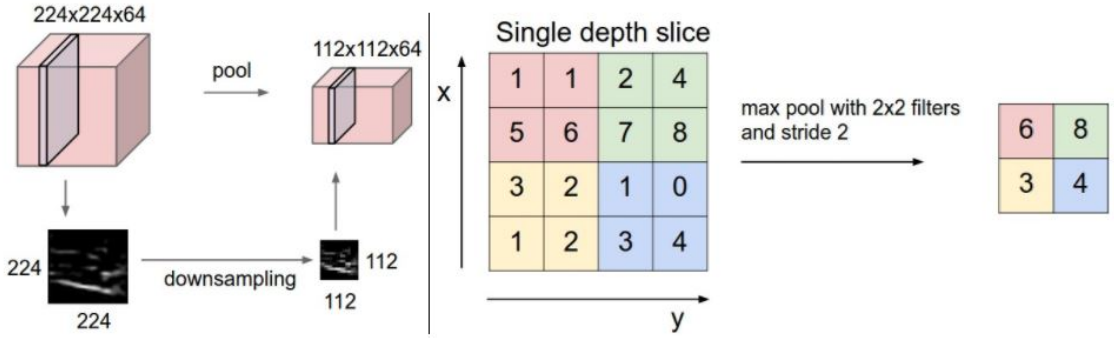

Figure 2.7: (left) Example of image downsampling using pooling. (right) Example of max pooling operation [3].

Average Pooling replaces all of the values in the receptive field with the mean value of the elements in the field. This method is no longer popular as it has been proven that max pooling outperforms average pooling in almost all cases and is easier to compute.

L2 Pooling replaces all values in the receptive field with the L2 norm of all of the elements in the field. Equation (2.21) shows how the L2 norm is computed.

$$norm = \sqrt{\sum_{k=1}^{n} x_k^2} \tag{2.21}$$

2.4.1.5 Fully Connected Layers

Fully Connected (FC) layers are identical to those in FNNs where there is a set of hidden neurons fully connected to the output of the previous layer. FC layers are typically used at the end of CNNs to make the final prediction. It is possible to convert fully connected layers to convolution layers. This can be accomplished by creating a convolution layer where the filter size is the exact size of the input structure which produces a 1×1 vector with a length that is equivalent to the number of neurons needed in an FC layer. This conversion is often done to reduce the number of learned parameters and increase the overall efficiency of the model.

2.4.1.6 CNN Example

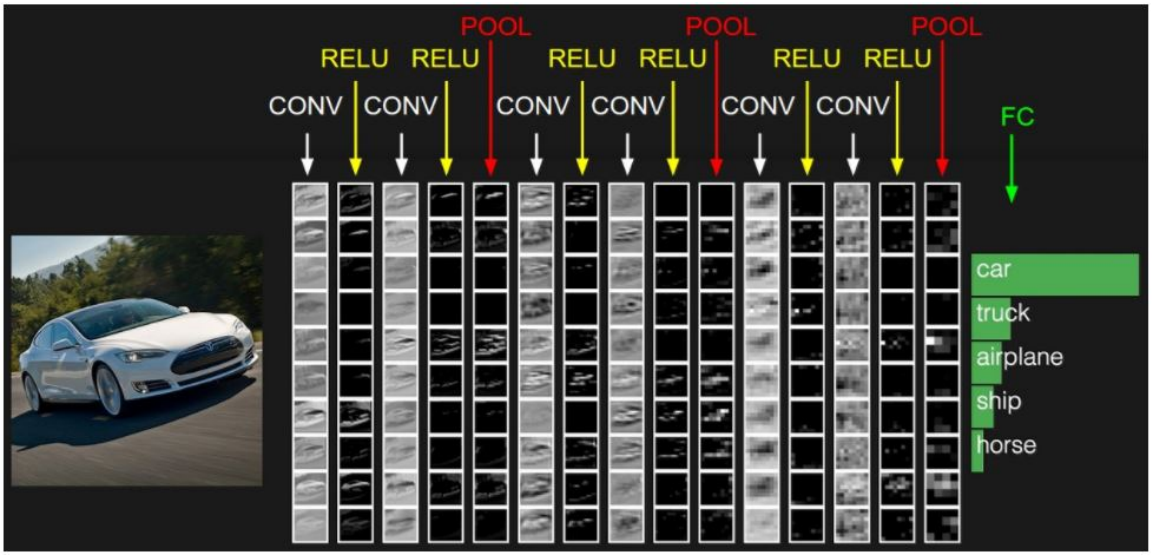

Figure 2.8: Example CNN architecture showing the layers and their effect on the image [3].

Figure 2.8 shows an example of a basic CNN architecture for image classification. The input (on the left) is an RGB image and the output is predictions with associated probabilities of what class the input image belongs to. Along the top of Figure 2.8 the types of layers are listed with CONV being convolution layers, RELU meaning the ReLU activation function, POOL being max pooling layers, and FC being a fully connected layer. This architecture follows the standard format of CONV, RELU, CONV, RELU, POOL with an FC layer at the end to perform the classification. The images in the center of Figure 2.8 show how each neuron in each layer is activated as the image is passed through the network. The CONV layers are activated by certain features or structures in the image, the RELU layers are simply passing the output of the CONV layer through the ReLU function (shown in (2.17)), and the POOL layers are downsizing the output of the RELU layers. The final FC layer has a node for each possible output class and is used to determine the output prediction of the model.

2.5 Deep Residual Networks

There are many popular CNN model architectures that have been developed, and proven to work for many different tasks, that researchers often use in lieu of developing a new architecture from scratch. Among the most popular is the Deep Residual Networks, or ResNet, developed by He et al. [6] in 2015. ResNet is a very powerful architecture that won first place in the ILSVRC 2015 image classification challenge as well as many other challenges.

The idea behind ResNet was to use much deeper networks than previously used. The problem with creating deeper networks is just stacking more layers does not always improve performance and, in fact, can have a significant negative impact on performance. He et al. [6] performed many experiments to prove this fact and Figure 2.9 shows the results of one of these experiments. It is shown that a normal CNN with 20 layers performs better than one with 56 layers in both training and testing accuracies.

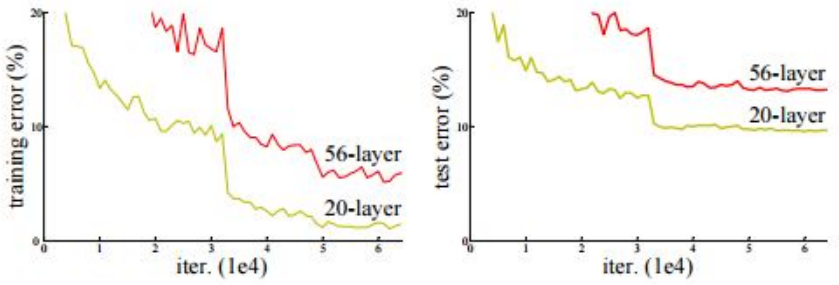

Figure 2.9: Training error (left) and test error (right) on CIFAR-10 with 20-layer and 50-layer networks [6].

The problem with deeper networks is that as the network gets deeper, the accuracy reaches a saturation point and then begins to rapidly degrade. To combat this, the ResNet architecture introduces a *deep residual learning* framework into the network. The residual framework utilizes residual learning blocks such as the one shown in Figure 2.10.

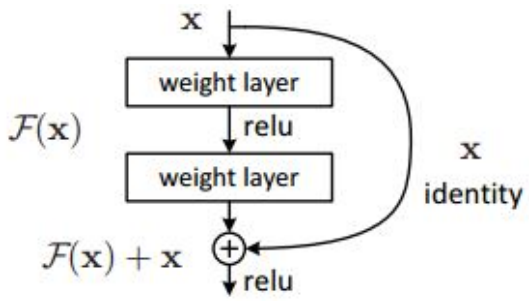

Figure 2.10: Residual learning block [6].

The residual blocks are designed based on the idea that a shallower network can be constructed into an equal deeper network by adding additional layers that are simply identity mappings. The residual blocks provide a means for the underlying mapping between layers to be explicitly fit or learned. This converts the mapping between layers, $\mathcal{F}(x)$, to $\mathcal{F}(x) + x$ where x, the residual, can be explicitly learned. Structured in this way, the network can collapse certain undesired extra layers by forcing the residual to zero if that is the optimal solution. This residual structure can be accomplished by using "shortcut connections" between stacks of layers as shown in Figure 2.10.

Using this idea, He et al. [6] constructed deeper networks by adapting "plain" networks with shortcut connections every few layers. Figure 2.11 illustrates the architecture of deep residual networks. On the left is VGG-19 [7], the deepest network previously used. In the middle a 34-layer "plain" network is shown that is similar to VGG-19 with many added layers. On the right a 34-layer residual network is shown that is identical to the "plain" network with the exception of the added shortcut connections.

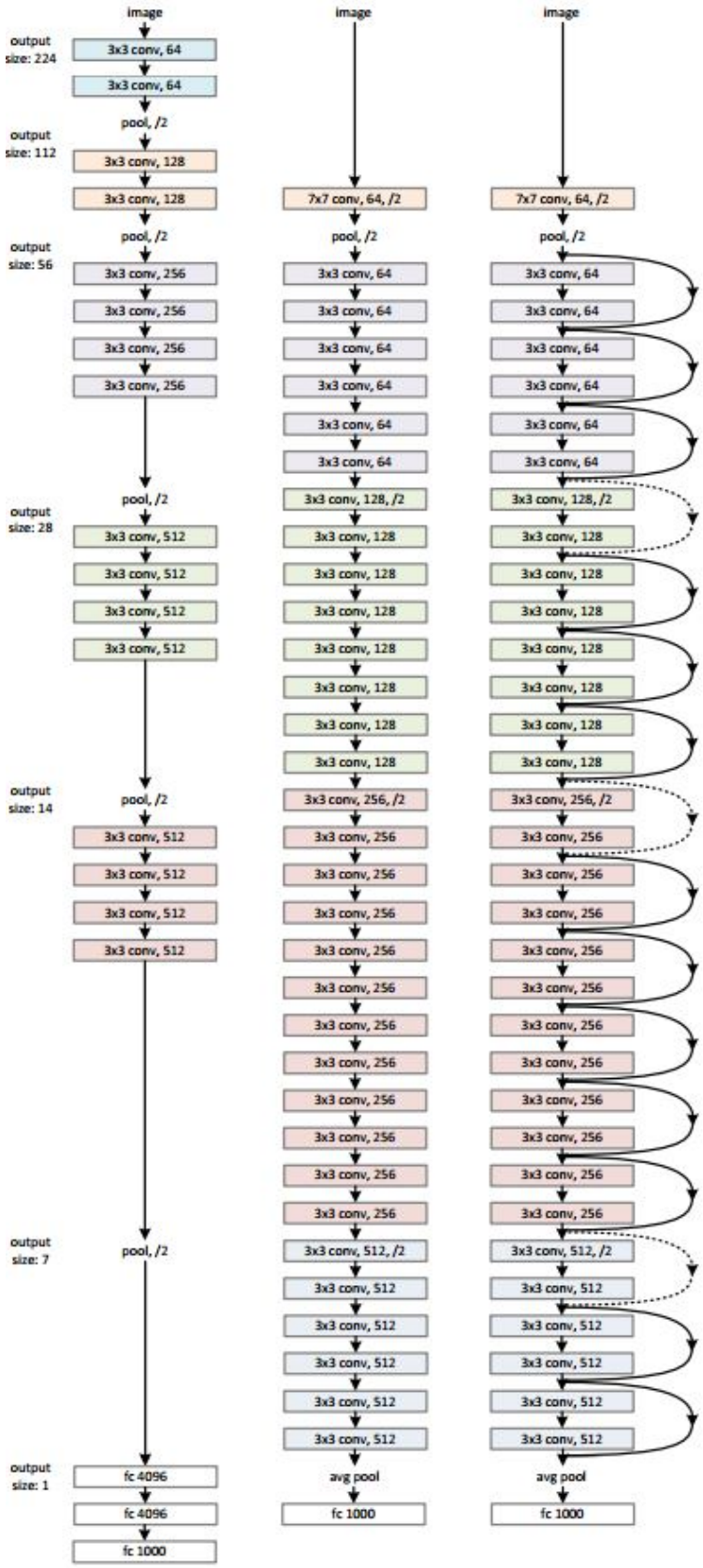

Figure 2.11: Example network architectures. (left) VGG-19 [7] (middle) 34-layer "plain" network (right) 34-layer residual network [6].

Using the same structure as in Figure 2.11 residual networks were created with 18, 34, 50, 101, and 152 layers, typically denoted as ResNet-N where N is the number of layers. For models with more than 34 layers He et al. [6] used a "bottleneck" design between the shortcut connections. Figure 2.12 shows the bottleneck residual architecture. The two 3×3 CONV layers between the shortcut connections were replaced with three CONV layers: a 1×1 followed by a 3×3 and another 1×1. The 1×1 CONV layers were responsible for reducing and increasing the dimensions to achieve a smaller dimension for the 3×3 layer. This drastically reduced the number of learned parameters for these larger networks.

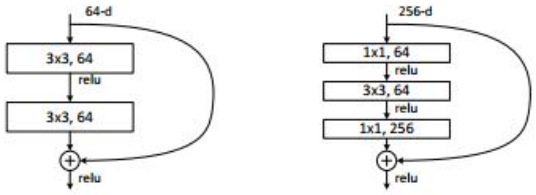

Figure 2.12: Bottleneck residual block used in ResNet-50, ResNet-101, and ResNet-152 [6].

Figure 2.13 shows a summary of the layers in each of the different depth ResNet architectures. All of the networks utilize similar building blocks, and the depth is determined by the number of blocks used. In every network downsizing is achieved by using a stride of 2 in conv3_1, conv_1, and conv5_1 rather than by using pooling layers. The last row in Figure 2.13 shows the Floating Point Operations (FLOPs) of each network. It is interesting to note that even ResNet-152 has far fewer FLOPs (11.3 billion) than VGG-19 (19.6 billion) due to the bottleneck architecture. The 50/101/152-layer ResNets outperform the 34-layer model by a considerable margin on the ImageNet and CIRFAR-10 datasets, and the more layers the model had, the better it performed. This shows that the residual networks are able to expand to much higher depths without suffering from any degradation and that the shortcut connections are able to learn the mapping between layers.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 2.13: Summary of ResNet architectures [6].

Chapter 3

Platform

Part of developing a new localization system for indoor agents was to perform real-world testing by implementing the system on an autonomous indoor agent. The agent, created as a research platform for research on autonomous navigation and other related fields, is an effective access wheelchair that was heavily modified to be able to navigate and travel autonomously. The agent, termed Machine Intelligence Lab Personal Electronic Transport (Milpet), is outfitted with many different sensors that help it observe it's environment including a Light Detection and Ranging (LiDAR), an omni-vision camera, ultrasonic range sensors, and wheel encoders. Milpet can be driven with a bluetooth joystick, by touch screen controls, or by speech commands and is able to autonomously navigate through an indoor environment toward a goal location. The platform was created from a mass-produced motorized wheelchair that was mostly dismantled to allow for new hardware and software to be added.

3.1 Hardware

Milpet started as a Quickie S-11 motorized wheelchair shown in Figure 3.1. All of the electronics besides the motors were removed so they could be replaced by new components and circuitry. The goal of the redesign was to create a hardware/software platform that could be used for many different aspects of autonomous navigation research. Part of this goal was to make the hardware easily expandable if a new sensor

or component needed to be added. This influenced many of the design decisions of the hardware platform.

Figure 3.1: Stock Quickie S-11 wheelchair.

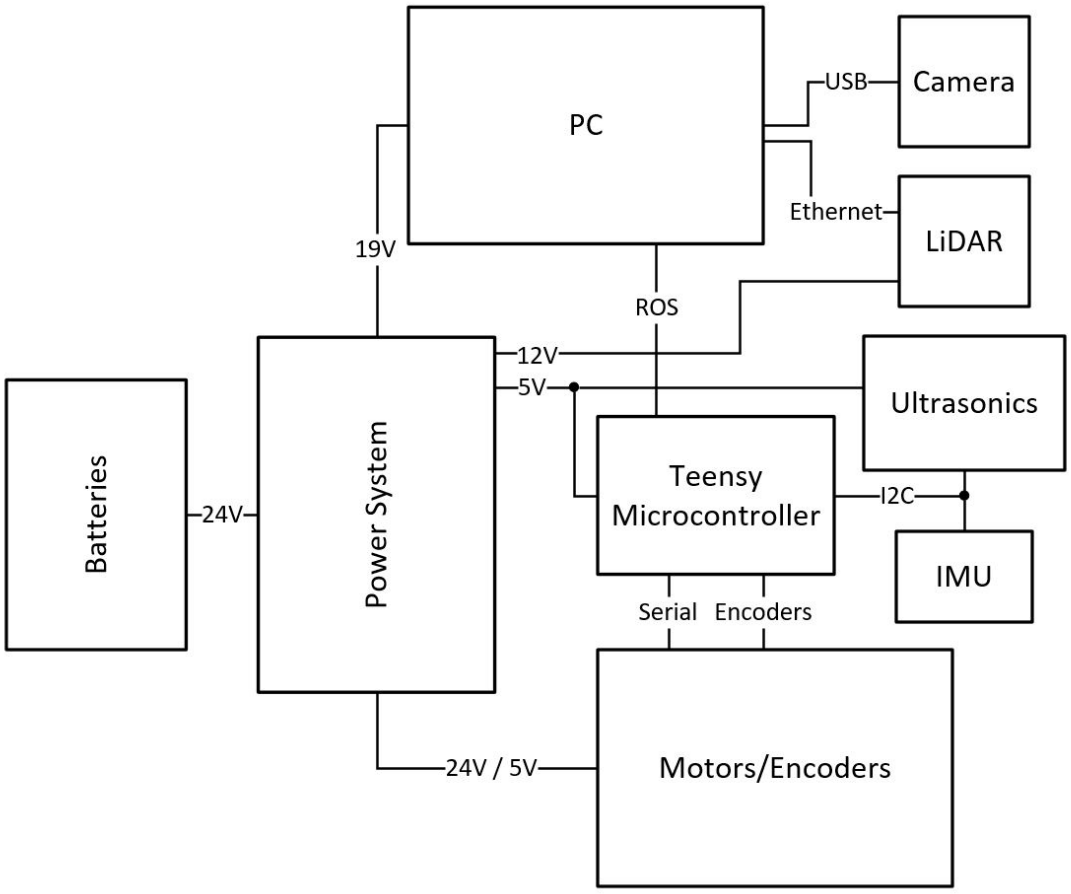

Figure 3.2: Top-level hardware system diagram of Milpet. Shows how each of the main components interact with each other.

Figure 3.2 shows the top-level hardware system diagram of Milpet. Here, each box

represents a subsystem of the hardware system, and the lines between them represent how the systems interact. Most interactions stem from the power system as this is the subsystem that takes in the 24V from the batteries and converts it to the voltages required by the other subsystems. This system also provides many safety features for both the other components and the users. Other subsystems include the embedded Personal Computer (PC), which functions as the computation center for the entire system, a microcontroller to handle the low-level hardware control, motors, and the various sensors. Figure 3.3 shows the circuit diagram for the battery system.

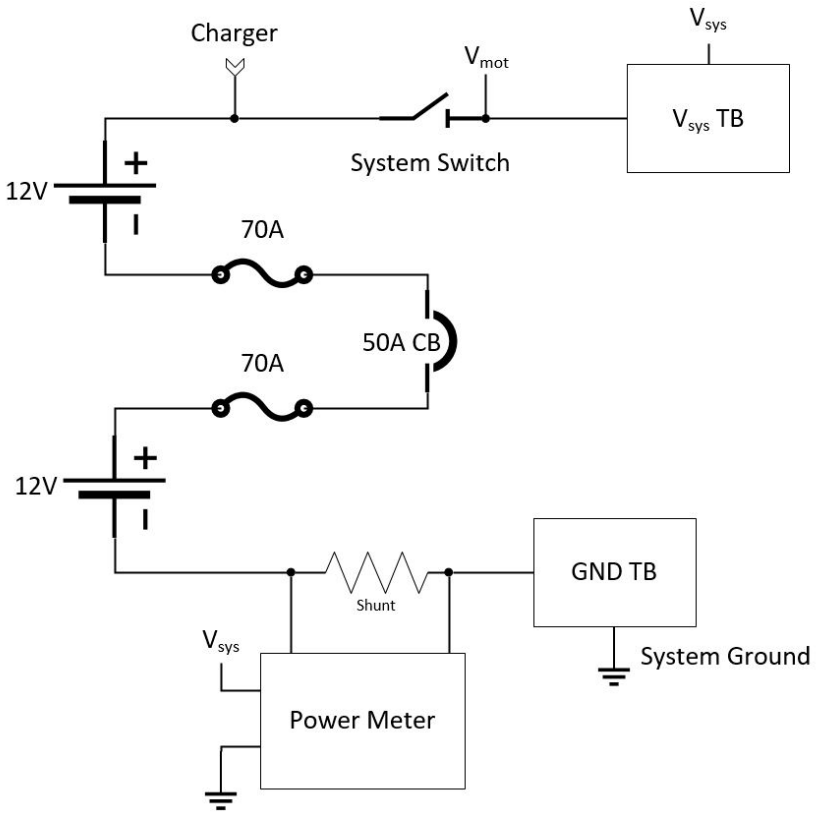

Figure 3.3: Circuit diagram for the battery system of Milpet.

The battery system circuitry, shown in Figure 3.3, shows how the two 12V batteries are wired together and some of the protection circuitry. The two batteries are wired in series to achieve the necessary 24V system voltage. Each battery has an in-line 70-amp fuse in case of catastrophic failures such as a short circuit of one of the batteries.

The batteries together have a 50-amp circuit breaker as over-current protection. The system voltage is wired through a main switch to allow the entire system to switched on and off. Also included is a power meter that continuously displays system voltage and the current draw, measured via a low-value shunt resistor. Figure 3.3 also shows where the battery charger is wired, where the motor driver is wired, and that terminal blocks (TB) were used for system voltage and system ground. This system mainly feeds the power system seen in Figure 3.4.

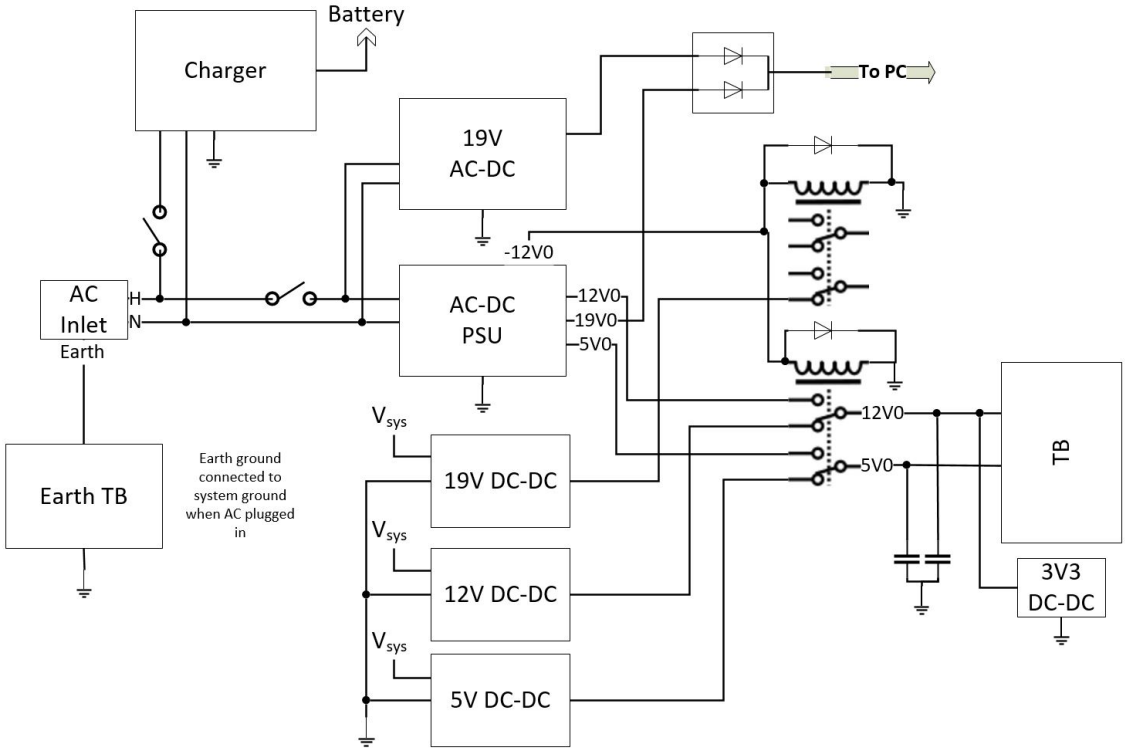

Figure 3.4: Circuit diagram for the power system of Milpet.

The power subsystem, seen in Figure 3.4, handles converting the 24V Direct Current (DC) from the batteries to all of the voltages required by other components. This system is also responsible for charging the batteries and allowing for the agent to be powered entirely from a standard 120V Alternating Current (AC) source. The entire system requires four different voltages: 19V, 12V, 5V, and 3.3V for various components. Since the agent can be powered from the on board batteries or from a

120-V AC source, each of the required voltages, with the exception of 3.3V, can be generated from battery power or a standard wall outlet. To generate these voltages from the battery three separate DC to DC regulators are used for 19V, 12V, and 5V (shown in Figure 3.4 as "DC-DC"). To generate the required voltages from the 120V AC source two separate power supplies are used; one to generate 19V DC (labeled "19V AC-DC") and one to generate 12V and 5V DC (labeled "AC-DC PSU").

Milpet also has the ability to switch between battery power and AC while running without interruption. For the 12V and 5V rails this is handled by a Double Pull, Double Throw (DPDT) relay. The coil of the relay is powered by a third rail (-12V) of the "AC-DC PSU." This was done so that when the AC-DC power supply is powered on, the relay will switch the 12V and 5V rails of the system to the ones generated by the AC-DC supply rather than the ones generated from the DC-DC supplies. Large capacitors were also added to the two rails to facilitate switching times. The 19V rail switching is handled slightly differently due to the fact that it is generated by a separate supply. Both the AC and battery generated 19V rails are wired through diodes into a single line that powers the PC. This means both supplies are connected at all times so there is no need to physically switch between them. The diodes are used to prevent current from one supply from flowing back into the other supply if both supplies are active. This system works by allowing current to flow from either supply depending on which is at a higher voltage. So, current will flow from whichever supply is powered or, if both supplies are powered, current will flow from the AC-DC supply as it outputs a slightly higher voltage. The 3.3V rail is generated from the 12V rail so it does not require switching as the 12V rail is already properly switched.

Figure 3.4 also shows how the AC input to the agent is handled and how the charger is wired. On the wheelchair there is a standard C13 inlet for 120V AC that is Earth grounded. The Earth ground is tied directly to system ground to ensure that

all grounds in the system are unified. From there, the neutral line is wired directly to the charger and the two AC supplies while the hot line is wired through two Single Pole, Single Throw (SPST) switches. These switches are wired to the charger and the two AC supplies allowing for them to switched on and off. The output of the charger is then wired back to the batteries (shown in Figure 3.3), and the AC supplies are wired into the switching circuitry previously mentioned.

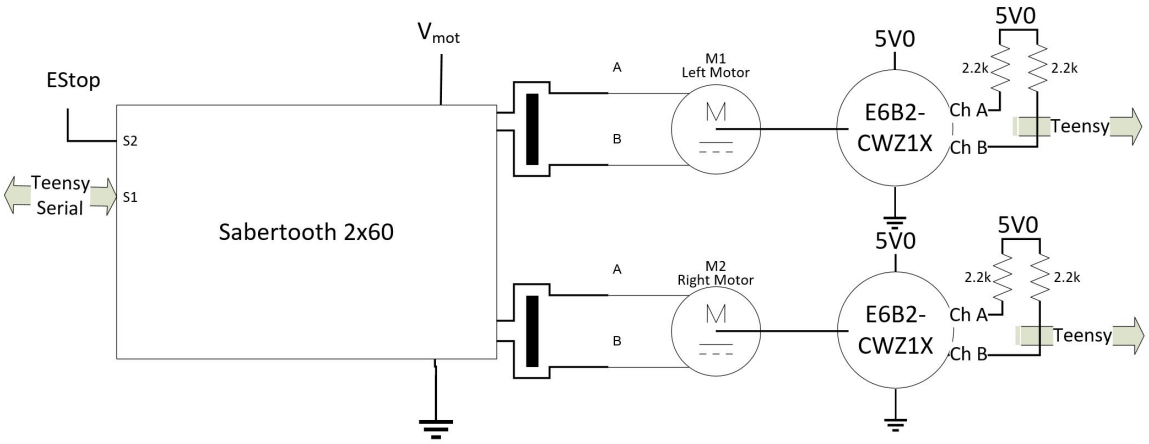

Figure 3.5: Circuit diagram for the motor and encoder system of Milpet.

Figure 3.5 shows the circuit diagram for the motor subsystem. The system includes the motors themselves, wheel encoders, and the motor driver. The motors are the same motors that were equipped to the wheelchair from the factory. Not much information is provided besides that they are standard brushed DC motors. Attached to the motor shaft are a pair of rotary encoders, specifically Omron E6B2-CWZ1X 1000 P/R. A diagram of some of the internals of the encoder is shown in Appendix A. These encoders are industrial quadrature encoders that have 1000 pulses per revolution of the encoder shaft. Since the encoders are located on the motor shaft, rather than the wheels themselves, the 20:1 gear ratio results in 20,000 pulses per wheel revolution. The encoders allow for the agent to accurately measure the change in distance for shorter distances. Over time however, small errors, caused by wheel slippage and other factors, cause the encoder readings to become erroneous, eliminating

them as a valid option for long-term localization. The two output channels of each encoder are wired to the microcontroller which keeps track of the count. Since the encoders have an open-collector output, pull-up resistors are needed for each output channel.

The motor driver is a Sabertooth 2x60 by Dimension Engineering. The Sabertooth 2x60 is a dual output motor driver capable of 60 amps continuous current on each channel which is beyond adequate for the provided motors. This particular motor driver can be controlled using analog voltages, servo commands, or serial commands. For this application, serial commands are used and are generated by the microcontroller. Serial commands provide the most flexible solution and allow for the use of the emergency stop functionality of the Sabertooth. The outputs of the motor driver are wired directly to the motors. The large black bars in Figure 3.5 are ferrite cores that prevent any high frequency noise from reaching the motors. The motor driver and motors are powered directly from the batteries as shown in Figure 3.3. The Sabertooth contains a separate input for emergency stop functionality which is wired to the emergency stop system of the agent shown in Figure 3.6.

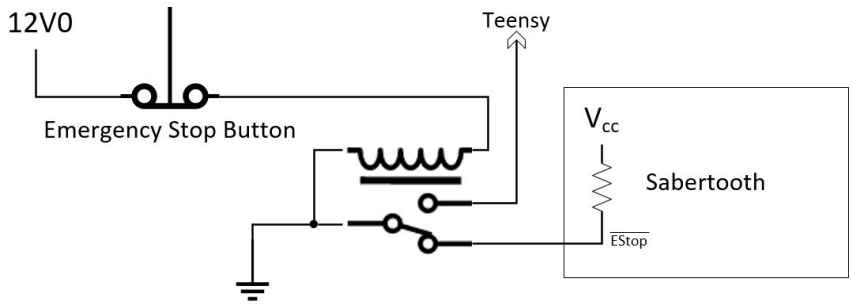

Figure 3.6: Circuit diagram for the emergency stop system of Milpet.

Figure 3.6 shows the emergency stop circuitry used on the agent. The Sabertooth motor controller has built-in emergency stop functionality (when running in serial mode) that provides an easy way to emergency stop the motors. The stop signal on the Sabertooth is an active-low signal, meaning it stops the motors when the signal

is pulled to ground. The signal is pulled up internally within the motor controller so that if the pin is unconnected the signal is high and the motors can spin. To make use of the active-low signal a relay was used to pull the signal to ground. Ground was wired to the common pin of the relay and the normally connected output of the relay was wired to the Sabertooth. The coil of the relay was wired to 12V through the pushbutton emergency stop. This configuration of the relay and button allows for the emergency stop to be activated when the emergency stop button is pressed or if any connection in this system breaks since the activation occurs when the relay coil is not receiving power. In addition, this system provides an easy way to add additional emergency stop mechanisms by simply adding the circuity in line with the existing button. The normally open output of the relay was also wired to a digital pin of the microcontroller, that is internally pulled up, so that the microcontroller was aware when the emergency stop was triggered.

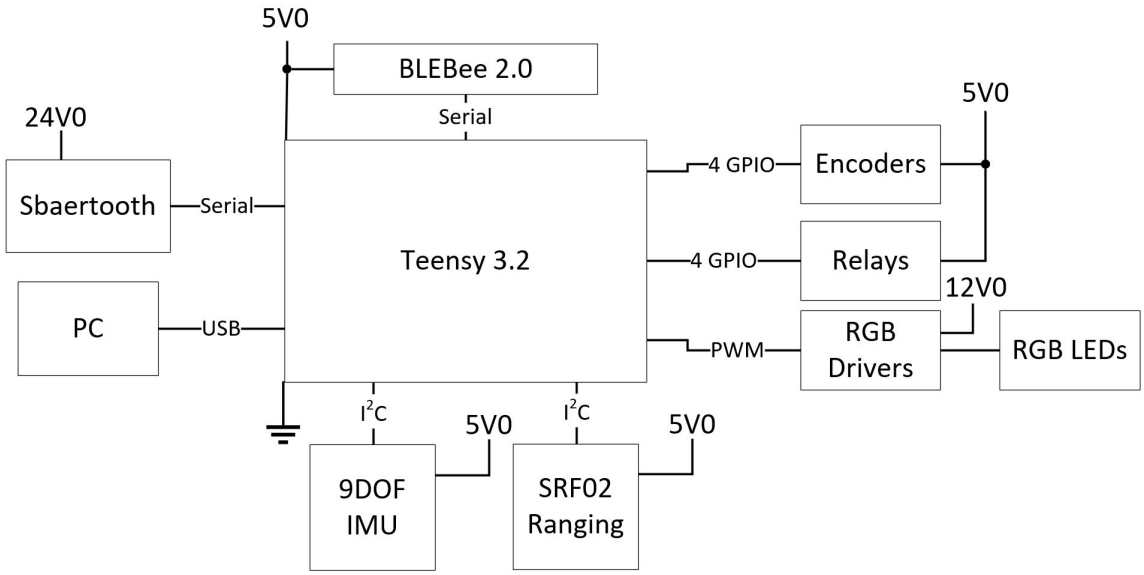

Figure 3.7: Circuit diagram for the microcontroller system of Milpet.

A very important component in this design is the microcontroller which performs most of the low-level hardware control. The diagram for the microcontroller subsystem is shown in Figure 3.7. The Teensy 3.2 by PJRC was chosen as the microcontroller

for this project. The Teensy is a powerful, yet small microcontroller that is compatible with Arduino code. Some of the features of the Teensy that were important to this platform are the 96-MHz ARM processor, 34 General Purpose Input Output (GPIO) pins, 12 timers, three serial buses, and two I^2C buses. In addition, the Teensy features two hardware quadrature decoders that are independent of the main processor. This is very beneficial as it allows for the two wheel encoders to be counted without interrupting the processor many times a second and potentially slowing down other operations.

Figure 3.7 shows the many components that the microcontroller controls. While the wheelchair is mainly driven autonomously it can also be driven by an mobile application that works through Bluetooth. The Teensy is connected to a Bluetooth (BLEBee 2.0) module that receives commands from the application. The module and the Teensy communicate through serial commands. The Teensy also communicates with the Sabertooth motor driver through serial. Also connected are the wheel encoders, which are wired to specific GPIO pins of the Teensy for the hardware quadrature decoders. There is also a four-channel relay board that is driven by GPIO pins from the Teensy. The relays are used to switch the LEDs and a 12V horn on and off. The system also contains some RGB LEDs that are used for status indication that are driven from three Pulse Width Modulation (PWM) pins from the Teensy. The LED driver circuit, shown in Appendix A, is required to drive the LEDs from PWM signals. The driver allows for the color of LEDS to be changed by varying the duty cycle of each PWM signal. The Teensy also controls the ultrasonic ranging sensors, specifically the SRF02s by Devantech, through one of the I^2C buses. The other I^2C bus is used to communicate with an Adafruit 9 DOF IMU. The IMU contains a magnetometer, accelerometer, and gyroscope on a single chip, each with 3 measurement axes. Finally, the Teensy is connected to the PC via USB which allows for the PC to send commands to the Teensy and the Teensy to send status and sensor

information back to the PC.

Table 3.1 shows a breakdown of which pins of the Teensy are being used and for what purpose. The "Function" column describes the hardware function of each pin, the "Description" states which device is wired to that particular pin, the "Direction" column denotes whether the pin is being used as an input or an output, and the "Active Level" column denotes whether the pin is active-low or active-high.

Table 3.1: Pin Usage Diagram for Teensy 3.2 on Milpet.

Pin	Function	Description	Direction	Active Level
0	Serial1 RX	Bluetooth TX	Input	n/a
1	Serial1 TX	Bluetooth RX	Output	n/a
3	Quad Dec 1A	Left Encoder - Ch. A	Input	Low
4	Quad Dec 1B	Left Encoder - Ch. B	Input	Low
10	Serial2 TX	Sabertooth Serial	Output	n/a
13	LED	Onboard LED	Output	High
14	Digital I/O	Relay 1	Output	Low
15	Digital I/O	Relay 2	Output	Low
16	Digital I/O	Relay 3	Output	Low
17	Digital I/O	Relay 4	Output	Low
18	I^2C-0 SDA	IMU I^2C Data		n/a
19	I^2C-0 SCL	IMU I^2C Clock		n/a
20	Digital I/O	Emergency Stop Status (Interrupt)	Input	Low
21	PWM	RGB LEDs - Red Channel	Output	n/a
22	PWM	RGB LEDs - Green Channel	Output	n/a
23	PWM	RGB LEDs - Blue Channel	Output	n/a
25	Quad Dec 2B	Right Encoder - Ch. B	Input	Low
29	I^2C-1 SDA	Ultrasonic I^2C Data		n/a
30	I^2C-1 SCL	Ultrasonic I^2C Clock		n/a
32	Quad Dec 2A	Right Encoder - Ch. A	Input	Low

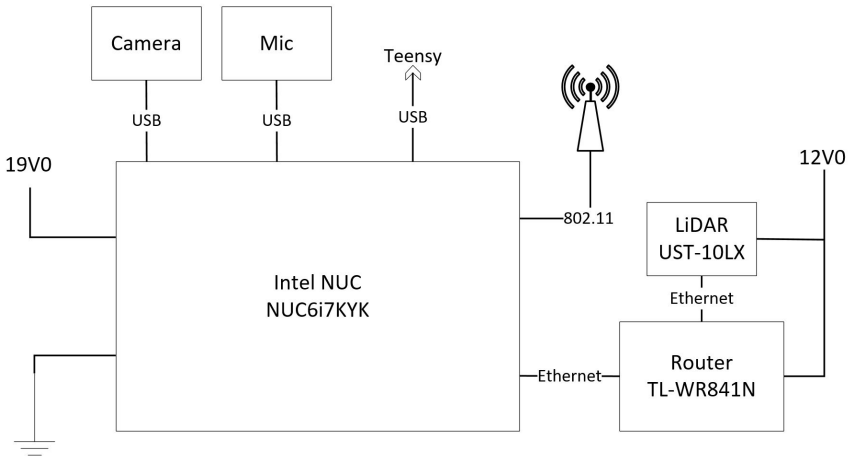

Figure 3.8: Circuit diagram for the PC system of Milpet.

The main computation center for the entire system is the embedded PC. The computer that was chosen for this platform is an Intel NUC (NUC6i7KYK) mini PC that is equipped with a quad-core x86 processor, 32 GB of RAM, and 256 GB of solid state storage. This hardware is sufficient for all of the computation and communication the PC is required to perform. The PC runs the Linux distribution Ubuntu (version 16.04) and the newest version of Robot Operating System (ROS), Kinetic Kame. This system handles communication between all of the other subsystems and performs most of the computationally intensive algorithms required for navigation.

Figure 3.8 shows the diagram of components that are connected to the Intel NUC. The computer is the only device that is powered from the 19V rail. The camera and microphone are simply connected via USB which provides a data connection as well as power to these devices. The Teensy microcontroller is also connected via USB but this connection is for data only. The PC has a built-in wireless card to allow connection to any wireless network for an Internet connection. The computer is also connected to a separate wireless router (the TL-WR841N by TP Link) via Ethernet. The router serves two main purposes: the first is to connect the LiDAR to the PC, as the LiDAR is also connected to the router via Ethernet, and the second is to broadcast a local wireless network that can be used to remotely connect to the PC

when the PC does not have an Internet connection.

Figure 3.9 shows Milpet after all of the hardware modifications were completed.

Figure 3.9: Milpet - Machine Intelligence Lab Personal Electronic Transport.

3.2 Software

Milpet includes a significant amount of software for control, navigation, and integration. This software includes low-level algorithms for motor control and sensor reading and high-level algorithms for path planning, localization, and obstacle avoidance. The low-level software is run on the microcontroller to allow for direct interaction with the hardware, and the high level algorithms are run on the embedded PC as it is a significantly more powerful system which is required for the complex algorithms. The embedded PC also handles integrating each of the software subsystems together to form a coherent system.

3.2.1 Embedded PC

The embedded PC installed on Milpet runs all of the high-level navigation algorithms and handles the communication of all the software subsystems. The powerful processor and large amount of memory allows for many complex algorithms, that include image processing and heavy matrix math, to run fluidly at the same time. The PC is running the latest long-term support version of the Linux distribution Ubuntu version 16.04. Ubuntu is a free and open-source operating system that is very popular for use in robotics and other hardware applications. Also installed on the PC is the newest version of ROS called Kinetic Kame. ROS is a popular software package that provides libraries and packages for easily integrating hardware and software subsystems together. Another benefit to ROS is, since it is a popular development package, there is a large amount of open-source code to accomplish common tasks such as reading sensors or performing basic algorithms.

3.2.1.1 Speech

One of the main forms of interaction with Milpet is through speech. The user can speak into a microphone fixed to the wheelchair and command the wheelchair to move. The user can use basic commands such as "Go Forward" and "Turn Left" or more complicated commands such as "Take me to room 1100" or "Go to the end of the hall and take a right." The basic commands simply control the motors whereas the complex commands trigger autonomous navigation.

For the speech to text conversion, the software package PocketSphinx [62] was used along with a customized dictionary, an acoustic model, and a language model. The custom dictionary contained only words that were needed for the various commands. Using this small dictionary helped to improve recognition as the model had a small subset of words to choose from. PocketSphinx supports both online and offline recognition, but, to eliminate a dependency on the Internet, offline recognition was

used. The software package (added to a ROS node) works by listening to any input noise that is above a predetermined noise level and constantly outputting the recognized text after each pause in speech. The latency from speech to text was initially around 2.3 s but was improved to 0.64 s by making improvements to the source code.

Milpet is also capable of talking back to a user by using text to speech. Milpet will speak to the user when their destination is reached, when it is stuck, or when in various other situations. The desired strings are converted to speech and played to the user via a speaker installed on the wheelchair. For text to speech, the software package sound_play [63] was used. Sound_play is an easy-to-use package that can convert strings to sounds and play them through any audio device, and it is compatible with ROS. A custom wrapper was added to provide a ROS topic where any node could publish a plain-text message to be played to the user.

3.2.1.2 Environment Map

The software system of Milpet was built assuming that a map of the environment is known before entering that environment. Most public buildings, including the buildings where the system was tested, provide PDF versions of each section of the building. These PDFs can be used to create a map for Milpet by converting the PDF to an occupancy grid. An occupancy grid is simply a matrix where the value 0 denotes a space where the agent can travel and the value 1 denotes an obstacle, such as a wall. The resolution of the grid, what real size is represented by a pixel, is a parameter and was set at 4 inches per pixel for this research.

3.2.1.3 Localization

The localization algorithm used on Milpet is the topic of this research. To see the details on the algorithm, see Chapter 4. The algorithm reads data from multiple sensors and produces a single XY coordinate on the map as the estimate of the

location of the agent.

3.2.1.4 Path Planning

Milpet uses an adapted D* Lite [64] algorithm for path planning. D* Lite is a popular form of path planning that uses incremental heuristic search methods to find the shortest path from a start point to a specified goal point. A goal is specified by the system as a single XY coordinate within the map, at which time the path planning algorithm calculates the shortest path from the current location to the goal location.

Once the path is known, it is divided into a specified number of intermediate waypoints. The waypoints act as intermediate goals that the agent must travel through to reach the destination. Having intermediate goals allows control to be passed to the obstacle avoidance algorithm in between these waypoints. If Milpet was solely dependent on following the path to get to the goal, every time a new obstacle (not present in the map) obstructs Milpet, re-planning would be needed, which can be expensive. With waypoints, the obstacle avoidance can work at a more local and reactive level while the path planning handles overall progress. The waypoints are chosen as single pixel values along the path but are then dilated to include many pixels in the surrounding area as well. This provides some leeway for localization by creating a group of pixels that represent the goal rather than relying on the localization to estimate a single pixel value. The path planning algorithm oversees the agent's progress along the path and provides a new heading direction to the obstacle avoidance algorithm each time a waypoint is reached. This algorithm is also responsible for stopping the agent once it reaches the destination. Figure 3.10 shows an example of the path planning algorithm and the intermediate waypoints. The path, shown as the red line, is planned from the green start point, on the left, to the yellow destination around the corner. The blue squares along the path represent the dilated waypoints that are evenly spaced throughout the path.

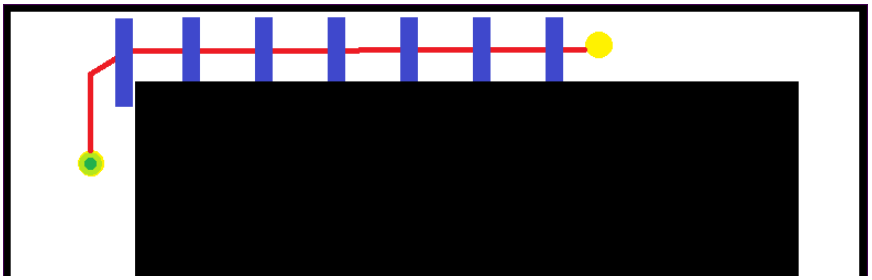

Figure 3.10: Example of path planning with intermediate waypoints.

3.2.1.5 Obstacle Avoidance

For obstacle avoidance, Milpet uses data from the LiDAR sensor and an algorithm inspired by Vector Field Histograms [65] and the Dynamic Window Approach [66]. The algorithm receives goal headings from the path planner which can change at each waypoint. The algorithm tries to have the agent travel in the general direction of that heading while also avoiding obstacles along the way. For example, the agent may be traveling in the correct direction but then encounter an obstacle. It will veer safely around the obstacle and then try to return to the goal heading. The goal heading at each iteration is calculated as a weighted linear combination of the new goal heading and the previous heading to allow for smooth changes and to prevent the algorithm from getting stuck between two free zones, or areas with no obstacles present.

The LiDAR data used for the obstacle avoidance algorithm is a single channel of 1081 depth measurements that cover a 270° field of view. The data are first passed through a median filter with the three previous scans and filtered to remove any large spikes. An obstacle is determined by values in the scan below a certain distance threshold. The scan is then divided into "free blocks," where a block is a section of the scan that contains no obstacles. Each block contains a range of free heading values that the algorithm can choose from. The algorithm then chooses the heading based on the free heading values, the goal heading, and the previous heading.

3.2.1.6 Robot Operating System

Robot Operating System (ROS) [67] is the software package that integrates all the high-level algorithms, the low-level algorithms, and hardware control. ROS allows users to create a hierarchical structure of sections of code that perform certain tasks and provides a means for these sections of code to communicate with each other. Figure 3.11 shows a simple example of the basic ROS architecture and how each part communicates.

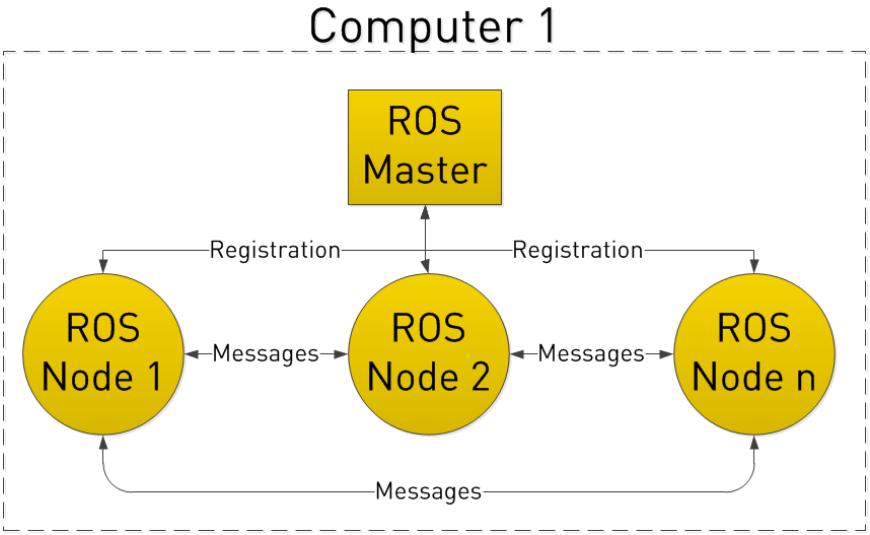

Figure 3.11: Diagram depicting the basic ROS architecture [8].

At the top level of the hierarchy are *packages*. Packages are simply an organizational structure for the lower objects. Inside packages are *nodes* which are sections of code that perform, usually a single, specific task. For example, there could be a node that handles reading data from a camera and another node that processes that data and performs face detection. Each node contains some specific code to declare to ROS that it is a node and the data it needs and provides. There are a few special nodes that are always running, including "ROS Master" which handles registration of nodes and topics. The communication between nodes is handled through a series of *messages*. Messages are structured data, analogous to a "struct" in some program-

ming languages, that can be passed around between nodes. Messages, referred to as *topics*, are passed around via a publisher/subscriber pattern. Any node can *publish* a message (send out data to other nodes) or *subscribe* to a message (receive data from another node). Any node can publish a message and, once published, the message is available to any node that subscribes to the message. ROS provides functions to both publish and subscribe to messages and provides many default message types. The message types include simple messages like String or Integer that contain a single value, and more complicated messages such as a LaserScan which is a pre-built structure for data from a LiDAR sensor. If the desired message is not available, custom messages can be created by the user. ROS also handles all of the threading and data storage associated with messages automatically in the background.

For Milpet, a simple ROS structure was created that includes nodes for each of the software subsystems. Much of the code was written from scratch, but some basic nodes and algorithms were reused from the open-source community. Code for ROS can be written in many different programming languages, but Python was chosen for all coding tasks within this platform. A typical way to display a system created in ROS is to create a *ROS diagram*. A ROS diagram is a diagram that shows all of the nodes that exist in the system as well as messages each node publishes and subscribes to. The diagram for Milpet is shown in Figure 3.12.

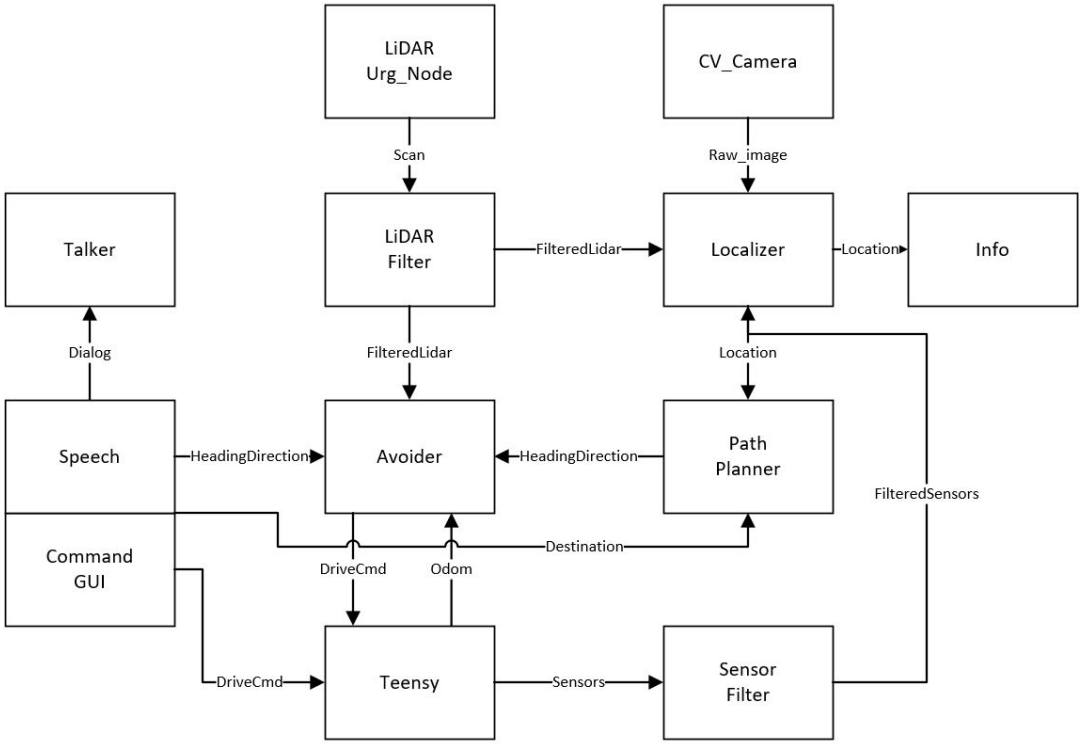

Figure 3.12: ROS diagram showing the ROS architecture for Milpet. The boxes represent nodes and the lines represent messages passed between nodes.

With respect to Figure 3.12, the "CV_Camera" and "Urg_Node" nodes handle reading input from the omni-vision camera and LiDAR respectively. These nodes were developed by the open-source community and convert the data from the respective sensor into ROS messages. The "LiDAR Filter" node constantly performs smoothing and median filtering on the LiDAR and publishes that filtered data for other nodes to use. Similarly, the "Sensor Filter" node filters the ultrasonic and IMU data from the microcontroller and publishes the filtered data. The "Localizer" node receives the camera, LiDAR, and sensor data, performs localization, and publishes the location within the occupancy grid where it predicts the wheelchair is located. The two main input mechanisms for Milpet, speech and a Graphical User Interface (GUI) are each implemented within a node, the "Speech" node and the "Command GUI" node respectively. The "Path Planner" node receives the location and a destination

from either the speech input or the GUI, and plans a path to the destination using the algorithm described in Section 3.2.1.4. After the path has been planned, the node publishes a direction for the wheelchair to travel to get to the next intermediate waypoint. The "Avoider" node then receives this direction and sends drive commands to the "Teensy" node to move the wheelchair. This node is responsible for performing obstacle avoidance as described in Section 3.2.1.5 and sending the appropriate drive commands. In addition, the "Speech" and "Command GUI" nodes can publish drive commands directly if a specific destination is not defined. Milpet also has the ability to speak back to the user. The text to speech and audio output is handled in the "Talker" node. Finally, there is the "Info" node which shows an image of the map and the predicted location on the screen for visualization purposes.

3.2.2 Microcontroller

The wheelchair is equipped with a microcontroller that is responsible for low-level hardware control. This includes controlling the wheelchair's motors, reading the wheel encoders to track the wheelchair's movement, and reading the various sensors. All of the code for the microcontroller was written in C++. Since the Teensy 3.2 is compatible with Arduino, the Arduino development environment was used to compile and upload the code. The environment is installed on the embedded PC allowing for code to easily be modified and reuploaded whenever needed.

3.2.2.1 Wheel Encoders

The wheelchair is equipped with two quadrature rotary encoders; one on each wheel. Encoders precisely monitor the rotation of the wheels to track velocity, and consequently distance traveled. Quadrature encoders are also able to determine the direction in which the wheels are spinning. Together, these measurements allow for the wheelchair's position to be tracked over time. Inside a quadrature encoder there are

two slotted disks, often made of glass. The encoders work by shining a light through the disks to a sensor on the other side. The disks rotate as the shaft rotates creating pulses of light. The light sensor outputs a digital square wave from these pulses of light that can be used to track wheel movement. The disks are carefully designed so that the pulses occur at a very specific interval and so that the two disks create waves that are 90° out of phase. Figure 3.13 shows an example of output from a quadrature encoder.

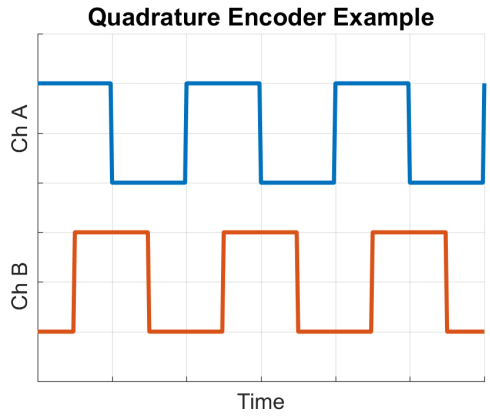

Figure 3.13: Example output waves from a quadrature encoder. The two signals, Ch A and Ch B, are 90° out of phase.

Figure 3.13 shows an example of the waves generated by a quadrature encoder. These are two output signals, usually called "Channel A" and "Channel B," one of which is out of phase from the other. To determine odometry information, the pulses of one signal can be counted. By counting pulses over a fixed interval of time, the velocity of the wheel can be determined because as the wheel spins faster, the pulses occur more often. Also, by keeping track of the total count, the distance traveled by a wheel can be determined. The second channel can help observe the direction of rotation by observing which wave is occurring first, since the two are out of phase. For example, if Channel B is high on the falling edge of Channel A, the agent is traveling forward; if Channel B is low on the the falling edge of Channel A, the agent is traveling in reverse. To utilize the pulse count information for odometry the "resolution" of the

encoder must be known. The resolution is how many slots there are around the edge of the disk. The more slots, the more pulses that occur per rotation of the encoder shaft. Knowing the resolution of the encoder and the circumference of the wheel the encoder is attached to allows for the velocity and distances to be calculated.

The Teensy 3.2 microcontroller is equipped with two hardware quadrature decoders that are designed to count signals output by quadrature encoders. The Teensy keeps track of the count using a hardware timer and stores the current count inside a register. The register can be read at any time the pulse count value is desired. In addition, the decoders automatically detect direction and decrement the count if the wheel is rotating in reverse. The Teensy is responsible for periodically using the count value to calculate the current velocity of each wheel, update the current pose of the wheelchair, and update the current XY position relative to the start point. This can be accomplished through a series of equations that are based on the pulse count and known parameters of the agent.

$$\lambda = \frac{C}{R \times G} \tag{3.1}$$

$$V_{r|l} = \lambda \frac{P_t - P_{t-1}}{t} \tag{3.2}$$

The first step is to calculate the wheel parameter, λ using (3.1). The wheel parameter describes the distance traveled by the wheel during one encoder pulse period and is specific to the agent. It requires that the circumference of the wheel, C and the the resolution, or pulses per revolution of the encoder shaft, R to be known. In some cases, such as with Milpet, it also requires the gear ratio of the motor, G to be known. This is required if the encoder is connected to the motor shaft rather than directly to the wheel. Equation (3.2) then shows how to calculate the right and left wheel velocities, V_r and V_l, using the difference in pulse count $(P_t - P_{t-1})$ over a

known time interval, t. Since, Milpet is a differential drive robot, the right and left wheel velocities are all that is required to calculate odometry information.

$$V = \frac{V_r + V_l}{2} \tag{3.3}$$

$$\omega = \frac{V_r - V_l}{b} \tag{3.4}$$

Equations (3.3) and (3.4) show how to use the individual wheel velocities to obtain the linear velocity, V, and the angular velocity, ω, of the wheelchair at any point in time. In (3.4), b is the distance from the center of mass of the wheelchair to the wheel. These equations only work because the wheelchair is uses a differential drive system. In addition to velocities, the distance traveled in XY space from a starting position and the current pose angle can be estimated using the pulse count using (3.5), (3.6), (3.7), and (3.8).

$$\Delta\Theta = \frac{\Delta r - \Delta l}{b} \tag{3.5}$$

$$\Delta d = \frac{\Delta r - \Delta l}{2} \tag{3.6}$$

$$\Delta x = \Delta d \cos\Theta + \frac{\Delta\Theta}{2} \tag{3.7}$$

$$\Delta y = \Delta d \sin\Theta + \frac{\Delta\Theta}{2} \tag{3.8}$$

Equation (3.5) shows how to calculate the change in angle, $\Delta\theta$, assuming the change was small, given the change in distance for the right and left wheel, Δr and Δl respectively. Equation (3.6) shows the calculation for the combined change in distance, Δd. Using Δd and $\Delta\theta$, the change in X, Δx, and the change in Y, Δy, can be calculated using (3.7) and (3.8). These equations are run by the Teensy at a small, fixed time interval to ensure the changes are small and the time change is

constant between iterations. The overall pose information that is kept by the Teensy is updated each iteration and published to the rest of the system.

3.2.2.2 Motor Control

The Teensy is also responsible for controlling the motors of the wheelchair. Milpet is equipped with a Sabertooth 2x60 dual motor driver that is responsible for sending current to the motors. The motor controller is driven via serial commands issued by the Teensy. The motor controller has a specific set of commands to control both of the motors' speeds independently. The software library provided with the Sabertooth provides an interface that makes issuing commands very simple. On the Teensy it is as easy as calling a single function that specifies which motor to drive and a value between -128 and 128 that specifies the relative speed of that motor with -128 being full reverse and 128 being full forward. The Sabertooth converts these commands to PWM signals that provide current to the motors.

Any node that needs to drive the motors can publish a ROS message to the "drive_cmd" topic which the Teensy is subscribed to. The messages contain a desired linear velocity, V in m/s and a desired angular velocity, ω in rads/s. The Teensy receives these commands and first calculates the required right and left wheel velocities to achieve the desired linear and angular velocity using (3.9) and (3.10).

$$V_r = V + \omega\frac{b}{2} \tag{3.9}$$

$$V_l = V - \omega\frac{b}{2} \tag{3.10}$$

Equations (3.9) and (3.10) show how to calculate required right and left wheel velocities, V_r and V_l from the desired linear velocity, V, the desired angular velocity, ω, and the base width, b. To command the motors to drive each wheel at the required

velocity a closed-loop Proportion Integral Derivative (PID) algorithm is used. The PID loops are used to maintain a desired velocity on each wheel based on feedback from the encoders. In general, a PID loop is a powerful control theory algorithm that uses an error value, how much the actual value differs from the desired value, to calculate the new output value. The general equation for a PID controller is shown in Equation (3.11).

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{3.11}$$

Equation (3.11) shows that the output at time t, $u(t)$, is calculated using the error at time t, $e(t)$, the integral of the error from time 0 to time t, and the derivate of the error at time t. Each term in this calculation has a multiplier, K_p, K_i, or K_d that determines how much of each term is used. These values are tunable parameters and are different for every application of a PID. Tuning these values changes the behavior of the output, and choosing the correct values for the application is a difficult but important task. Usually values are chosen by arbitrarily choosing an initial value and tuning until the desired behavior is achieved. A PID loop is a iterative process that can be used to calculate an output value of any scale based on an input value of a different scale. For Milpet, a PID loop was used on each wheel to calculate the necessary motor value (0-128) based on the desired velocity and current velocity in m/s. Equations (3.12), (3.13), (3.14), and (3.15) show specifically how the PID was implemented on the Teensy.

$$S = \frac{V \times M}{V_{max}} \tag{3.12}$$

$$e(t) = S_t - S_{act}(t) \tag{3.13}$$

$$i(t) = i(t-1) + K_i e(t); \quad 0 < i(t) < M \tag{3.14}$$

$$o(t) = K_p e(t) + i(t) - K_d(S_{act}(t) - S_{act}(t-1)) \tag{3.15}$$

Since the velocity input and feedback used was on a significantly different scale than the motor output, (3.12) was used to scale velocity values to motor outputs. To calculate the scaled value, S, the velocity, V, was multiplied by the maximum motor output, M (127), and divided by the maximum velocity, V_{max} (3 m/s). Equation (3.13) shows how the error at time t, $e(t)$, was calculated by taking the difference in the target scaled velocity, S_t, and the actual scaled velocity reported by the encoders at time t, $S_{act}(t)$. Equation (3.14) shows the calculation for the integral term at time t, $i(t)$. The integral term is an accumulation of the error over time but is constrained between 0 and M to avoid it growing to large. Finally, (3.15) shows how the output value at time t, $o(t)$, is calculated. This calculation includes the proportional term, the integral term, and the derivative term. The derivative term is the difference in the measured, scaled velocity at time t, $S_{act}(t)$, and the velocity from the previous iteration, $S_{act}(t-1)$. The tuning parameters that achieved the best behavior were found to be $K_p = 0.3$, $K_i = 3$, and $K_d = 0.001$.

Each PID loop was updated many times a second with the velocity values that were measured by the encoders. Using this method of control, the Teensy was able to maintain wheel velocities to +/- 0.1 m/s even under changing conditions, such as an incline or changes in weight. Since each wheel was controlled individually, the wheelchair was able to drive straight regardless of minor differences in the right and left wheel.

3.2.2.3 Sensors

The Teensy is also responsible for reading data from both the IMU and the ultrasonic range sensors. Both devices communicate over I^2C, and each uses one of the two individual I^2C buses the Teensy has. The IMU contains a magnetometer, an accelerometer, and a gyroscope; however only the magnetometer is used currently. The magnetometer is used to determine the compass heading of the wheelchair. The Teensy reads the data from the magnetometer using I^2C commands. The data, is a measure of magnetic field, is measured in gauss. The Adafruit library provided with the specific IMU provides a means to convert this measurement to a heading. The heading measurement is then put through a 1D Kalman filter to eliminate some of the variance caused by noise.

The eight ultrasonic range sensors surround the wheelchair with two in each cardinal direction. Each sensor must be explicitly told to take a reading, and readings can take up to 70 ms. Also, care must be taken not to have sensors close to each other reading at the same time as they may interfere with each other. The Teensy tells the sensors to read, via an I^2C command, in two groups of four, one in each cardinal direction per group. Only after one group has finished reading is the second group commanded to read. After all sensors have taken a reading the Teensy loops through and reads the distance from each sensor. Figure 3.14 shows the location and direction of the ultrasonic sensors and each group differentiated by color. The Teensy uses timers to know when readings are complete to ensure the code is not halted while waiting for measurements. The distances, along with the heading from the compass, are all then published in a ROS message for use in the navigation algorithms.

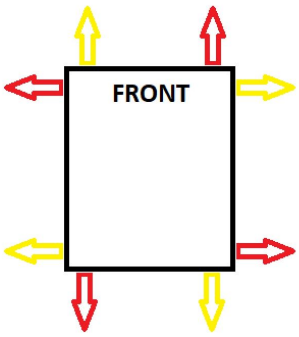

Figure 3.14: Depiction of ultrasonic range sensor locations and ranging groups.

3.2.3 Bluetooth Phone Applications

To achieve manual control over Milpet, since the original joystick was removed, a phone application for iOS and Android was specially designed. The application, called BlueBot Controller, is available to the public on both the Google Play Store and the iOS App Store. Both applications feature a digital joystick for motor control as well as buttons for additional functionality. Also, the application supports receiving information from the agent and displaying it to the user. The application communicates with the wheelchair via Bluetooth Low Energy (BLE) to the BLEBee Bluetooth module, shown in Figure 3.7, which in turn communicates the information to the Teensy via serial. The applications both feature a similar simple, intuitive UI that is easy for any user to learn. Figure 3.15 shows an image of the main screen of the Android app (left) and the iOS app (right).

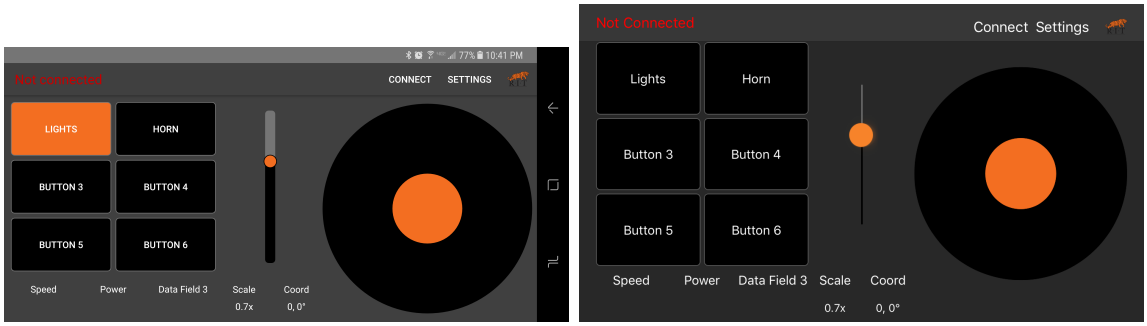

Figure 3.15: (left) The main screen of the Android application. (right) The main screen of the iOS application.

On the main screen, seen in Figure 3.15, the digital joystick is on the right side, the buttons are on the left, and along the bottom are the different fields for displaying data from the agent and fields that display the state of the joystick. The joystick outputs polar coordinates with the angle value ranging from 0° to 360° and the magnitude ranging from 0 to 100. The slider in the center of the application can be used to control the scale of the magnitude in 0.1 increments. Along the top of the screen is a connection status indicator, a button to show the connection screen, and a button to open the settings menu. The connection screen will scan for nearby Bluetooth devices and allow the user to connect to or disconnect from any compatible Bluetooth device. The settings menu is a comprehensive menu where the button and data field labels, the button type (between toggle and momentary), the data transmission rate, and other aspects can be changed.

For Milpet, the application is used to manually drive the wheelchair around when needed, and the buttons are used to control various hardware components, such as the LEDs, the horn, and the additional relays. While the application was designed for use with Milpet, it was also made generic enough to work with other agents or platforms. The app is compatible with any Bluetooth LE module, and the commands themselves are not specific to Milpet and therefore can be used to control other systems.

Chapter 4

Methods

4.1 Overview

The goal of this research was to develop an indoor localization algorithm that met the following requirements:

- Use only sensors mounted to the agent to gather data
- Not rely on external signals to assist with the location estimate
- Be accurate to within half the size of the agent
- Be robust to handle changing conditions within the environment
- Be fast enough to run in real time

At a high level, the algorithm that was developed uses data from an omni-vision camera and ultrasonic sensors along with machine learning and a filtering algorithm to perform real-time localization. The data from the camera are used to train a classifier with the ability to predict agent location. An adapted particle filtering algorithm is then used to increase the overall accuracy. Data from the ultrasonic sensors are added into the filtering algorithm to further fine tune the accuracy.

4.2 Testing Environment

The localization algorithm was developed to focus on indoor environments where the primary environment of an agent could be a building or a series of buildings.

Training a single classification model to predict the location within such a large space is not feasible; therefore the environments needed to be split into separate local environments. For this research, these local environments were taken to be two independent sections of a floor of an academic building. In addition, the focus was on localizing within hallways rather than inside individual rooms as rooms would need to be treated as a separate local environment as well. Two different hallways were chosen, the maps of which can be seen in Figure 4.1.

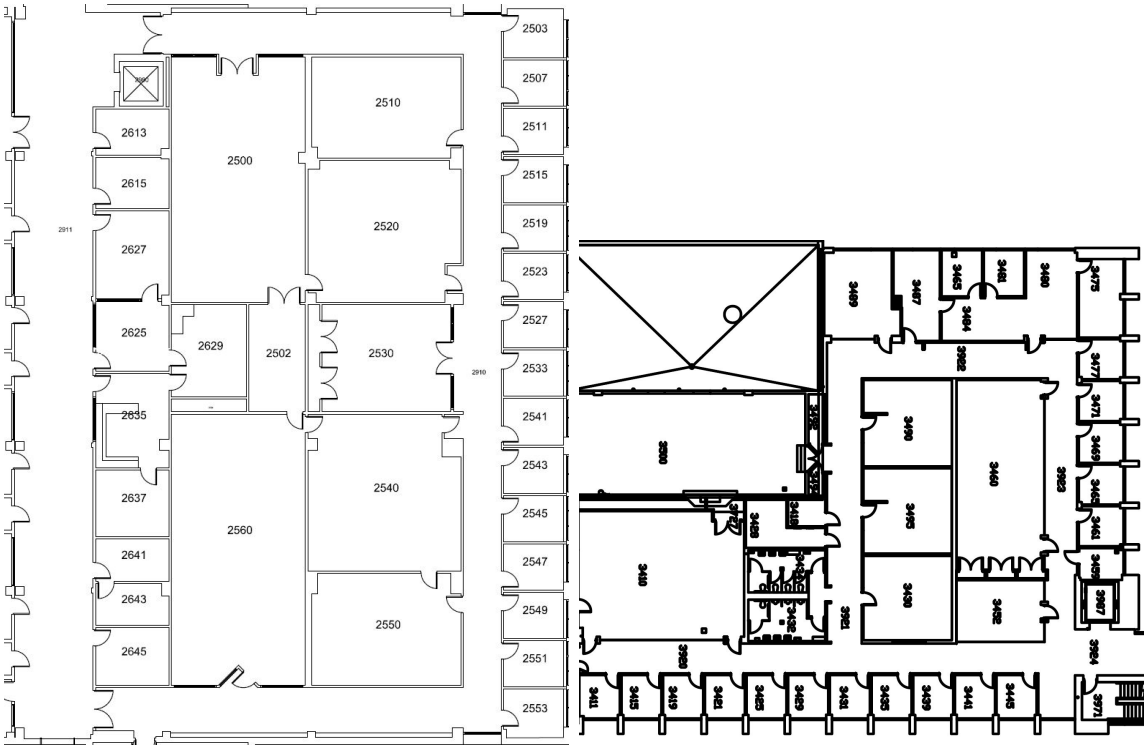

Figure 4.1: (left) Floor plan map of first hallway tested, Hallway A. (right) Floor plan map of second hallway tested, Hallway B.

Figure 4.1 shows the floor plan of the two local environments, denoted Hallway A (left) and Hallway B (right), that were used for testing. The floor plans were provided, in PDF form, by the institution as is typical with most public spaces. For scale, the image of Hallway A represents a height and width of about 135 ft by 90 ft, and the image of Hallway B represents a height of about 100 ft and a width of about 130 ft. Each hallway was treated a separate local environment, and therefore each

hallway required a separate classification model to be trained.

The floor plans needed to be converted into a form that the system could use to perform localization and path planning. The form chosen was an occupancy grid. An occupancy grid is a matrix where the value zero is free space and the value one is an obstacle. The grids were created on a scale where one "pixel" in the grid was equal to a 4×4 inch section. Figure 4.2 shows the occupancy grids generated for each hallway.

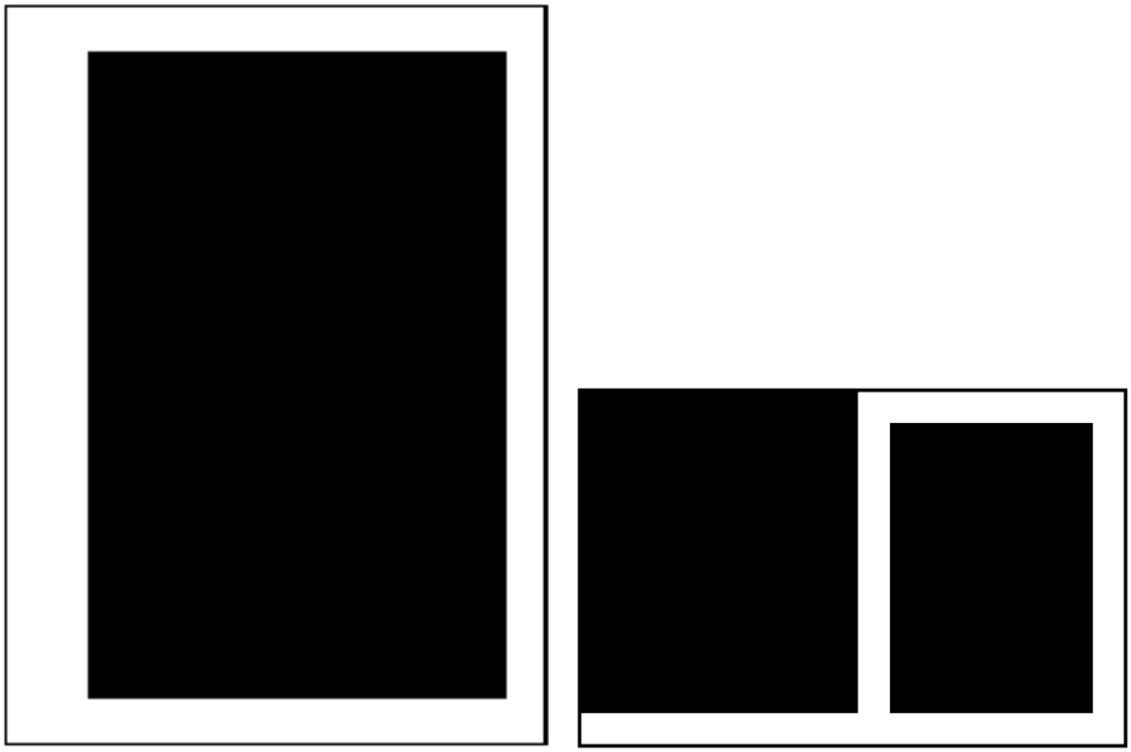

Figure 4.2: Hallway A (left) and Hallway B (right) shown in occupancy grid form where each pixel is 4×4 inches.

4.3 Data Collection

The first step to building the algorithm was to collect sample data from the omni-vision camera in order to train a classification model. To collect the data, the agent was driven through the hallway and images were periodically captured from the omni-vision camera. The particular classification models that were used required labeled training data, meaning that each training sample is required to have an associated

label. The label corresponds to what class the sample is from: in this case, where the image was taken within the map. This presents a challenge as, without accurate localization, there is no way to accurately know where the agent is located at any given time. To solve this problem, a basic form of dead reckoning localization, using data from only the wheel encoders, was implemented as described in Section 3.2.2.1. The dead reckoning provided a means to specify a label with each image that was captured while the agent was moving through the map. However, over time, dead reckoning quickly losses accuracy especially when the agent is rotating to move around corners. This meant that dead reckoning was not reliable after a certain distance and therefore could be used for only short straight distances.

Since both hallways were too large for dead reckoning to remain accurate, each hallway was divided into smaller, straight sections that were small enough to ensure dead reckoning would remain reliable. How each hallway was divided is shown in Figure 4.3.

Figure 4.3: Hallway A (left) and Hallway B (right) divided into smaller sections to prevent dead reckoning errors.

The two hallways were divided in such a way that each section remained short and did not include any corners. In between these divisions the odometry information is reset, based on physical measurements of the hallway marked by tape, and the agent is exactly positioned to mitigate any labeling errors during data collection.

Once the hallways were divided, the locations of each class to be learned by the classification model, referred to as waypoints, needed to be determined. It is not possible to set a waypoint at each pixel in the map as this would result in too many classes for the classifier to have to distinguish. Instead, waypoints were set every 3 ft, or nine pixels in the occupancy grid, and only in the center of each hallway. The value of 3 ft was chosen as this was the approximate length of the test agent, Milpet. Choosing this distance means the classifier is able to predict location only within three feet and in the center of the hallway. To improve the accuracy and allow for a lateral position within the hallway to be predicted, the particle filtering algorithm

and ultrasonic data were added as discussed later. The labels for each class were chosen as integers that started at an arbitrary start point and increased by one for each waypoint. Figure 4.4 shows an example of the class labeling for Hallway A.

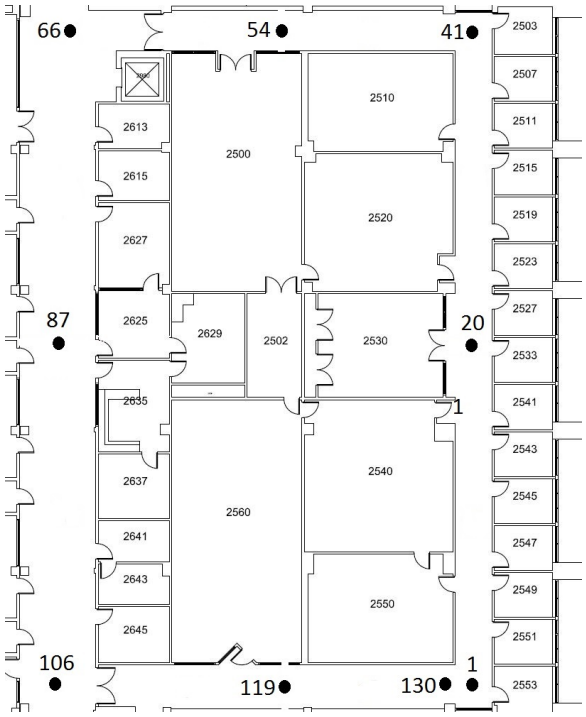

Figure 4.4: Example of waypoint labeling in Hallway A.

In Figure 4.4, various waypoint locations, or classes, are shown as black dots with their associated label written next to them. In total, Hallway A had 130 waypoints and Hallway B had 101.

Once the divisions were set the process of actually collecting the data was as follows:

1. Measure hallway and place markings at waypoints located at section boundaries
2. Start the agent positioned at a waypoint on section boundary
3. Specify how many waypoints to collect before stopping
4. Command agent to drive straight forward at a fixed velocity
5. Every 3 ft, as reported by odometry, save an image from the camera and the LiDAR scan.

6. After agent stops, reposition at next waypoint (may involve rotating)
7. Reset odometry information
8. Repeat for each section

In the process above, the images were saved with a file naming convention based on section and class number within that section that allowed for class label to be extracted at a later point. This process could be repeated any number of times, in either hallway, to collect many trials of data with differing conditions, such as lighting, direction of travel, or with or without people around.

4.3.1 Preprocessing

The omni-vision camera being used for testing is actually a low-cost, low-resolution webcam pointed up into a small 360° mirror designed for a smartphone. This camera has a lower cost than a purpose-built omni-vision camera but also produces lower quality images. Figure 4.5 shows an example of an image captured from the omni-vision camera.

Figure 4.5: Raw image output from omni-vision camera.

The image in Figure 4.5 is of size $960 \times 720 \times 3$, but most of the image does not contain usable information. Since the camera is pointed into a small mirror, the mirror housing takes up most of the image. The useful data lie within the ring towards the center of the image, with the data outside the ring and in the center of the ring being wasted. Even on the outer edge of the ring, parts of the mount for the camera can be seen. Some preprocessing needs to be done to extract only the useful information from these images. The preprocessing steps are as follows:

- Determine center and inner and outer radius of ring
- Mask all of image that is outside the outer ring
- Mask center of image inside inner ring
- Crop image to the outer ring

The center and radius of the ring are fixed values as position and size of the ring are consistent from image to image. Those values are used to create a mask that replaces all pixels outside the ring and at the center of the ring with zeros. Once the image is masked, it is cropped to include just the ring. An example of the image after preprocessing is shown in Figure 4.6.

Figure 4.6: Omni-vision image after preprocessing.

The preprocessing image in Figure 4.6 went from the original size of $960 \times 720 \times 3$ to $321 \times 321 \times 3$ due to cropping. From this point, more preprocessing steps are applied

that are specific to each model being used. Both models require an additional resize and the SVM requires the image to be converted to grayscale and converted into a single vector.

4.3.2 Datasets

One dataset was created for each hallway. The datasets were created by completing many "laps" of each hallway, where a lap consisted of capturing an image at each waypoint. Groups of laps were captured at different times of the day to try and capture variations in daylight. Laps were also captured moving forward, following the direction of the class labels, and moving in reverse (the opposite direction). The class labels were kept consistent in both directions in an effort to train a classifier to be invariant to direction. Some laps were taken with people walking through the hallways while with others, people were avoided. Within each group of laps, the lateral positioning of the agent was varied slightly as well the starting point of each section of the hallway to introduce slight variations to the datasets.

Table 4.1: Hallway A dataset breakdown.

Number of Laps	Direction	Time of Day	Total Images
13	Forward	Night	1,742
5	Forward	Day	670
5	Reverse	Night	670
5	Reverse	Day	670
3	Forward	Active Day	402
31			**4,154**

Table 4.1 shows a breakdown of the dataset created for Hallway A. Hallway A consisted of a total of 31 laps resulting in 4154 images. These images provided data at different times of day and in different directions. In addition, some laps consisted of people walking through, especially the "active day" laps as these were taken during a busy time of day. The breakdown of the dataset for Hallway B can be seen in Table

4.2.

Table 4.2: Hallway B dataset breakdown.

Number of Laps	Direction	Time of Day	Total Images
5	Forward	Night	505
5	Forward	Day	505
5	Reverse	Night	488
5	Reverse	Day	505
20			**2,003**

The dataset for Hallway B consisted of 20 total laps resulting in 2003 images. The dataset captured variations in daylight and direction.

To increase the size of the datasets considerably and in an attempt to create a model that was rotationally invariant, software rotation was applied to each training image. Since the images are from an omni-vision camera, rotating an image simulates rotating the agent. This was helpful, as it meant that data for different rotations did not need to be physically gathered. For each image in each dataset, rotations of ±2, ±4, and ±6 degrees were applied along with every 30 degree increment. This totaled 18 different rotations applied to each image (2°, 4°, 6°, 30°, ... , 330°, 354°, 356°, 358°) including the original image. The final dataset sizes can be seen in Table 4.3.

Table 4.3: Dataset sizes after augmentation.

Dataset	Original Size	Size after Augmentation
Hallway A	4,154	74,772
Hallway B	2,003	36,054

4.4 Models

Two different classification models were tested on the data collected. The first was a Support Vector Machine (SVM) and the second was a Convolutional Neural Network

(CNN). Each classification model was trained to predict which waypoint the agent was at from a single omni-vision image.

4.4.1 Support Vector Machine

The first classification model that was tested was a multi-class, linear SVM. A linear kernel function was used and the cost value. C, was set to 1. SVMs require the input to be a single 1D vector so the preprocessed image could not be passed in directly. In addition, SVMs do not work well with very large dimensionality so some additional preprocessing was required. In addition to the preprocessing described in Section 4.3.1, the image was converted to grayscale and scaled to 15% of the original size using cubic interpolation to preserve as much information as possible. No additional processing or formal dimensionality reduction was performed. Figure 4.7 shows an example of the final preprocessed image for the SVM, enlarged for visibility.

Figure 4.7: Omni-vision image after additional preprocessing for SVM.

The final image size for images passed to the SVM is $48 \times 48 \times 1$, reduced from the original $321 \times 321 \times 3$. While the image is much smaller and therefore less sharp, some of the prominent features can still be seen. After the image has been rescaled, it is converted to a 1D vector, of size 2304, by flattening the image.

4.4.2 Convolution Neural Network

The second model that was tested was a CNN. Specifically, an unmodified version of ResNet-50 [6] was used. The architecture was exactly that of ResNet-50 shown in Figure 2.13, with the exception of the last layer. The versions of ResNet with more layers were not considered as the algorithm needs to run in real-time and more layers results in an increase in testing time. The developers of ResNet provide the source code written for the deep learning framework Caffe [68]. Due to restrictions of the testing agent, the Python deep learning framework TensorFlow [69] was used. The developers of TensorFlow include a version of the ResNet code, ported from Caffe, built into the TensorFlow framework.

The only additional preprocessing required for the ResNet model was to resize the images from $321 \times 321 \times 3$ to $224 \times 224 \times 3$ to match the input size of the ResNet model. Again, a cubic interpolation resize operation was used to preserve as much of the image as possible.

4.4.2.1 Training

A typical pattern to follow when training a deep convolution neural network is to use a pre-trained network and "fine-tune" this network with additional data specific to the task. This task is called *transfer learning* and is used to save training time and to reduce the number of images required to accurately train the model. Models are usually pre-trained on a large dataset similar to the data for the new task. The idea is that the first layers of the network will be similar among similar data so they may not need to be changed from task to task. ResNet was originally trained on ImageNet [70], a dataset of 1.2 million natural images of objects that fall into 1 of 1000 classes. The network was trained to predict which class an image belonged to. After training, the final weights for this model were saved and provided with the model. These weights were used to build a pre-trained version of ResNet for this

algorithm. The intuition was some of the features of the images in ImageNet would be similar to those found in the hallway environments.

Once the pre-trained weights were applied, the model was fine-tuned on the hallway datasets. To do this, the last fully-connected (FC) layer of ResNet had to be replaced, as the original last layer was made to predict from 1000 classes. For Hallway A it was replaced with an FC layer of size 130, for the 130 classes, and for Hallway B it was replaced with an FC layer of size 101. The new FC layer was initialized using the Xavier uniform initialization [71]. After replacing the layers and loading the pre-trained weights for all other layers, the models were trained as normal. During training, the weights for every layer were allowed to change with the same learning rate.

During training, to add additional variation to the datasets, a random brightness offset between $\pm 5\%$ was applied to each image. While adding variation, this also ensured that the model did not see the exact same image more than once even though many iterations of the dataset were used for training. Mean subtraction was also performed on each image at train time to help to normalize the images.

4.5 Localization Algorithm

The localization algorithm that was developed, being termed *Waypoint Classification with Particle Filters*, uses the trained classification model along with an adapted particle filter algorithm to determine the location of the agent. There are three types of movement that can estimated in a hallway environment: longitudinal, lateral, and rotational. Longitudinal refers to movement down the hallway, lateral refers to the side to side offset from the center of the hallway, and rotational refers to which direction the agent is facing. For this algorithm, the classification model is used to determine longitudinal position by determining the weights of the particles in the particle filter algorithm, and the ultrasonic data are used to manipulate the particle

weights to aid in the lateral positioning. The rotation of the agent is determined by the odometry information from the wheel encoders. Algorithm 3 shows the high-level description of the Waypoint Classification with Particle Filters algorithm.

Algorithm 3 Waypoint Classification with Particle Filters.

```
Initialize particles with a random location within map
while True do
    Read in sensor data and perform preprocessing
    Pass image through classifier to obtain probability of each waypoint
    Adjust waypoints laterally with ultrasonic data
    Update particles based on robot movement
    Weight particles based on waypoint probabilities
    Estimate location from weighted particles
    Resample particles based on weight
end while
```

Particle Initialization

A "particle" is is a single XY coordinate within the occupancy grid for the current map and represents a possible location for the agent to be located. Particles can only be in valid places within the map (i.e., where the occupancy grid is equal to zero). The particles are initialized by choosing K random, valid coordinates within the grid without replacement. K has to be high enough to ensure particles are well dispersed throughout the map but not so high that every coordinate within the map is chosen. Empirical testing showed that a value of about 3 particles per square foot of valid area performed well. Particles are stored as a list of XY coordinate pairs. The initial batch of particles is saved throughout the duration of the algorithm to be reused during resampling. Figure 4.8 shows an example of Hallway A with $K = 1500$ particles randomly placed within the valid spaces.

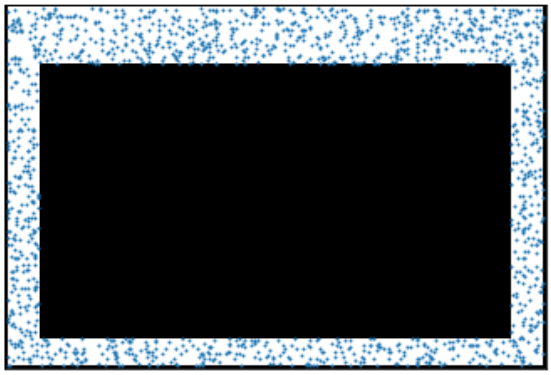

Figure 4.8: Hallway A initialized with 1500 particles.

Reading Sensors and Preprocessing

Data have to be read from three different sensors. First, a single image is read from the omni-vision camera's live feed. The same preprocessing that is done during the training of the classification model, shown in Section 4.3.1, is done to the captured image. This includes the masking, cropping, resizing, and mean subtraction. Next, the data from the ultrasonics are read. These data are in the form of a vector of ranges for each ultrasonic sensor equipped to the agent. From these ranges, the distance to the nearest object on both the right and left side can be determined. A rolling average of the right and left ranges is kept to help smooth any sudden changes due to an obstacle quickly passing by the sensors. The length of the average filter, L, is a parameter that can be altered to change the behavior of the sensors. Finally, data from the wheel encoders is read to determine how much the agent has moved since the last iteration and the current rotation. These data are in the form of odometry information that is calculated at the microcontroller level, as shown in Section 3.2.2.1. Additionally, the LiDAR scan is also read but currently not used.

Classification

The preprocessed image is passed through a classification model that predicts which waypoint the agent is closest to. The model can be any classification model that is trained to predict waypoints. For this research, ResNet-50 was the primary classification model (Section 2.5). In addition to predicting the closest waypoint, the model is able to generate a probability for all waypoints. For ResNet, this is done by applying a softmax to the final fully connected layer of the model.

With any classification model, there is a chance of a misprediction and, in some cases, the model can predict a waypoint very far away from the current location. This is especially a problem if two distant waypoints look very similar. To avoid mispredictions drastically affecting the location estimate, a mask is applied to the probabilities that emphasizes the region around the previous prediction. The mask weights the waypoints closest to the waypoint predicted on the previous iteration higher and other waypoints lower as they get further away from the previous prediction. An example plot of the mask is shown in Figure 4.9.

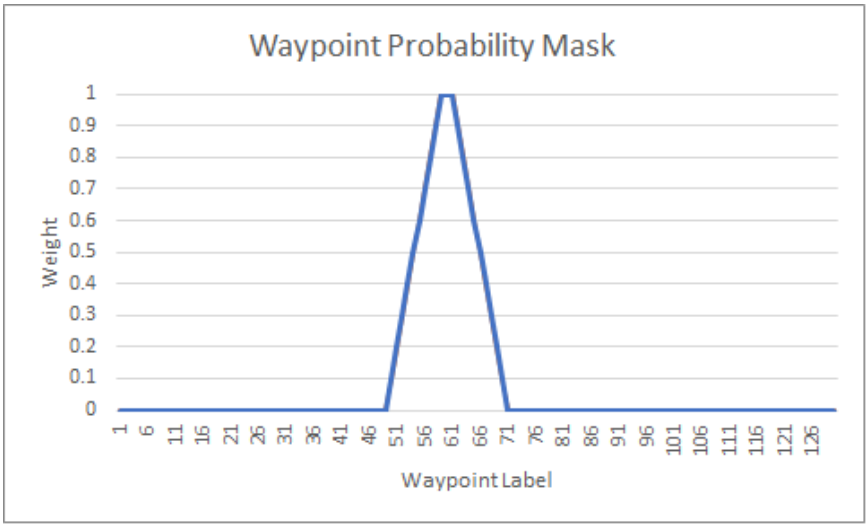

Figure 4.9: Example of waypoint probability mask centered at waypoint 60.

In Figure 4.9 the previous prediction was waypoint 60 so waypoints 59, 60, and

61 receive a weight of 1. The weight then decreases by 0.1 for each waypoint on both sides until it reaches 0, at which point all other waypoints have no weight. This is an example of a hallway with no 3-way or 4-way intersections, like Hallway A, but the same concept is applied for hallways with intersections, like Hallway B, where weights extend in each direction at an intersection. This mask is applied to the probabilities output by the classification model by element-wise multiplying the mask and the list of waypoint probabilities. It is also possible to use a very low weight instead of zero for the tails of the mask. A low weight would allow the algorithm to slowly recover from converging on a an incorrect location. The value of zero was used because testing showed that about only five waypoints received any non-zero probability from the classification model. Having this few waypoints meant that, if a low weight was used instead of zero for the mask, it would still be possible for the estimation to jump many waypoints.

Lateral Adjustment

Lateral adjustment refers trying to have the algorithm account for changes in lateral position within a hallway, meaning if the agent is on the right side or the left side of the hallway as opposed to the middle. This is done by adjusting the waypoint locations within the map based on data from the ultrasonics sensors. Each waypoint has a known coordinate within the map. Initially each waypoint is located in the center of the hallway, as this is where the data is gathered to train the classification model. These locations are a basis to determine particle weights. The ultrasonic data is used to move the waypoint coordinates right or left if the agent is determined to be right or left. Algorithm 4 shows how the ultrasonic data is used to determine if the agent is shifted right or left.

The ultrasonic sensors, after preprocessing, give a distance on the right and left side of the agent, D_r and D_l respectively. Subtracting the left distance from the

Algorithm 4 Lateral Position from Ultrasonic Data.

1: $\Delta = D_r - D_l$
2: **if** $\Delta < (\Delta_{center} - T)$ **then**
3: Shifted right
4: **else if** $\Delta > (\Delta_{center} + T)$ **then**
5: Shifted left
6: **else**
7: Centered
8: **end if**

right, yields a offset, Δ. The value of Δ can then be compared to the offset value that corresponds to the center to the hallway, $\Delta_{center} \pm$ a threshold value, T. The value of Δ_{center} is specific to each hallway and may not always be zero as some hallways contain objects along most of the edge which change the average offset value. The average offset can be determined by driving the agent through the center of a hallway and averaging the difference in right and left distances. T is a threshold value that determines when the agent is shifted enough to move the waypoint coordinates. To shift the coordinates, the current estimate of location and the direction of travel are used to determine in which direction X or Y to perform the shift. When a shift is needed, every waypoint coordinate within the hallway is shifted by a fixed amount. Figure 4.10 shows an an example of the centered waypoint coordinates (center) and the coordinates shifted left (left) and right (right).

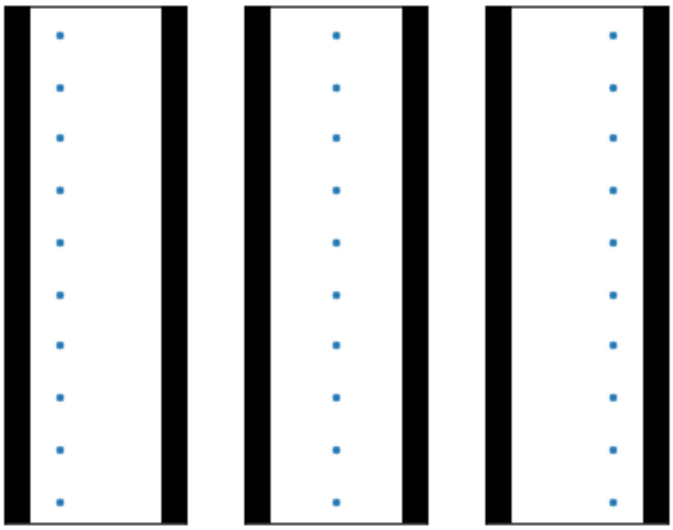

Figure 4.10: Example of normal centered waypoint coordinates (center) and the coordinates shifted to the left (left) and right (right).

Movement Update

Between iterations, the agent may move a certain distance. It is important to update the particles with this movement to ensure the group of highly weighted particles stays nears the agent. The movement from one iteration to the next is tracked via wheel encoders. The new odometry information is compared to the odometry from the previous iteration to determine how much the agent has moved. This movement is applied to every particle, unless the movement would force the particle into an invalid area in the map, in which case that particle is not moved.

Particle Weighting

Determining the new particle weights is the most important step in any particle filter algorithm. The weights determine how the particles will be resampled and ultimately th estimation of location. For this algorithm the particle weights are determined using the probabilities from the classification model, after being masked, and the location of each waypoint, after being adjusted for lateral changes. The weight of each particle

is determined using (4.1).

$$W_i = \sum_{i=1}^{n} \left(1 - \frac{d_i}{D}\right) \times P_i \tag{4.1}$$

Equation (4.1) shows how the weight of each individual particle W_i is determined. First, the n nearest waypoints to the particle, by Euclidean distance, are determined. Then, the weight is calculated as a summation of each of the nearest waypoint's probability, P_i, weighted by the ratio of the distance to that waypoint, d_i, over the total distance to all n nearest waypoints, D. The ratio, $\frac{d_i}{D}$ is subtracted from 1 to ensure closer waypoints are weighted more heavily.

Location Estimation

The location estimate is a prediction of agent location at the current time and is the final output from the localization algorithm. The estimate is in the form of a single XY coordinate within the map. The estimate is determined with the following steps:

1. Determine raw estimate from particle weights
2. Determine final estimate as a combination of raw estimate and previous estimate
3. If the estimate is invalid, clip it to valid space

The first step is to determine the raw estimate from the particle weights. This is done by taking a weighted average of the top $X\%$ of particles after sorting the particles by their weight. The value for the X and Y coordinate of the raw estimate, R_x and R_y respectively, is determined separately. The next step is to calculate the adjusted estimate by using (4.2) and (4.3).

$$E_{x_t} = \alpha E_{x_{t-1}} + \beta R_x \tag{4.2}$$

$$E_{y_t} = \alpha E_{y_{t-1}} + \beta R_y \tag{4.3}$$

The adjusted X and Y values for the estimate, E_{x_t} and E_{y_t}, are calculated as a weighted sum of the X and Y values of the previous estimate, $E_{x_{t-1}}$ and $E_{y_{t-1}}$, and the raw estimates, R_x and R_y, from the particle weights. The estimate is computed in this way to prevent unreasonable jumps that would be physically impossible for the agent to make. α and β are parameters that can be adjusted to control how fast the estimate can change. The final step for estimating location is to clip E_{x_t} and E_{y_t} to the nearest valid area if in an invalid occupancy grid location.

Particle Resampling

The final step of the algorithm is to resample the particles for the next iteration. To resample the particles, a weighted random sample is drawn from the current set of particles where the weights are the calculated particle weights. The samples are drawn with replacement, meaning that the same particle can be drawn multiple times. Most of the particles are drawn in this way; however, $Z\%$ of them are randomly drawn from the initial batch of randomly distributed particles. This is done, to prevent the algorithm from possibly getting stuck at an incorrect location. Another possibility for resampling particles is to add noise to all or some of the particles sampled from the existing particles. Adding a small amount of noise could help to disperse the particles focused around the agent to allow for more movement in the next iteration, both longitudinally and laterally.

There are many parameters to this algorithm that can change the behavior. The value of these parameters is dependent on the specific map being used, the movement speed of the agent, and the reliability of the sensors being used. A summary of all of the parameters is shown in Table 4.4.

Table 4.4: Tunable parameters for localization algorithm.

Parameter	Description
K	Number of particles
L	Length of the ultrasonic average filters
Δ_{center}	Center of the hallway in terms of ultrasonic ranges
T	Threshold for determining if agent is shifted in hallway
n	Number of nearest waypoints to consider when weighting particles
α	Weight of previous location estimate when calculating new estimate
β	Weight of raw location estimate when calculating new estimate
Z	Percentage of particles sampled from the initial batch of particles

Chapter 5

Results

5.1 Classification Models

The classification models were tested independently from the rest of the algorithm to observe their standalone performance on waypoint classification. Initially, both the linear SVM and ResNet-50 models were tested to compare the results from each. For the initial comparison, the testing dataset was creating by taking all of the data from Hallway A and applying only $\pm 2^{\circ}$ rotations. The dataset was randomly split into a training and validation set, where the training set would be used to train the model, and the validation set would be used as a held-out test set to report accuracies. Table 5.1 shows the final size of each dataset.

Table 5.1: Initial dataset size for Hallway A.

Original Size	Training Size	Validation Size
12,462	11,216	1,246

Both the SVM and Resnet-50 models were trained on the training set. For the SVM the additional preprocessing discussed in Section 4.3.1 was applied, and for ResNet no augmentation was applied during training in an attempt to create a close comparison between the models. The accuracies on the test sets are reported in Table 5.2.

Table 5.2: Initial dataset results for Hallway A.

Model	Correct/Total	Accuracy
SVM	1,246/1,246	100.0%
ResNet-50	1,245/1,246	99.9%

Both of the models performed very well on the initial dataset of Hallway A. The accuracies were most likely inflated due to overfitting and a lack of variance within the dataset, but these results showed that both models were capable of performing the task of waypoint classification. Even though both models performed similarly, with the SVM even outperforming ResNet slightly, the choice was made to continue with ResNet as the primary model for the algorithm. Deep learning has been outperforming conventional methods, such as SVMs, on most classification tasks. In addition, these experiments showed that more variation, such as more rotations and changes in brightness, needed to be added to the datasets, increasing their size significantly. SVMs become very difficult to train as the dataset size grows, while deep models are easily trained with very large datasets.

Following the initial testing, ResNet was tested on the datasets created for Hallway A and Hallway B, detailed in Section 4.3.2. Each dataset was randomly split into a training and validation set, the sizes of which are shown in Table 5.3.

Table 5.3: Train and validation size of hallway datasets.

Dataset	Training Size	Validation Size
Hallway A	59,818	14,954
Hallway B	28,844	7,210

During training, additional augmentation was added to each image with a random brightness offset between ±5% and mean subtraction was performed. The model for each dataset was trained using a mini-batch gradient descent with a batch size of 85 images. The Adaptive Moment Estimation (ADAM) optimization method [72] and

an exponentially decaying learning rate were also used. The model for Hallway A was trained for around 17k training steps, and the model for Hallway B was trained for around 12k training steps, each on a single GPU. After training, the validation set was tested on each model. The training and validation accuracies are shown in Table 5.4.

Table 5.4: Hallway dataset ResNet results.

Dataset	Training Accuracy	Validation Accuracy
Hallway A	98.8%	99.8%
Hallway B	99.3%	99.9%

5.2 Localization Algorithm

The developed localization algorithm was implemented on the test platform, Milpet to be tested. The algorithm ran inside a ROS node on the main PC and continuously updated the estimation of position using data from the omni-vision camera and other sensors. When combined with other algorithms for path planning, and obstacle avoidance, the agent was able to navigate autonomously from one point in Hallway A to another. While this does attest to the functionality of the localization algorithm, it was difficult to gather numeric results or examples of the algorithm working. For this, a testing scheme was created to show testing results from Milpet by capturing data while driving, and running the algorithm at a later time with the data collected.

5.2.1 Testing Methodology

To gather results of the localization algorithm running on Milpet, it was beneficial to be able to be able to gather data from the sensors and pass that data through the algorithm offline. This ensured the results could be observed and recorded easily while the algorithm was running. Because the data were gathered from the agent

while driving, testing offline would show the same results had the testing been done online.

The data were gathered in an identical fashion to how the data for training the models were gathered. The agent was commanded to move forward and a script was run that saved the data from the omni-vision camera, the ultrasonics, and the wheel encoders once per second. The odometry information from the encoders, again, acted as the "ground truth" for where the agent was located as it traveled. This meant that stopping and reseting the odometry information periodically was imperative. The agent was stopped and repositioned at exactly the same places as was done for the previous data collection, shown in Figure 4.3. At each stopping point, the odometry information was manually set to the correct, known values — ensuring that it could not drift too far. In between the stopping points the agent can also be manually driven to demonstrate lateral movement. For tests without lateral movement, to further mitigate errors caused by drift in the odometry information, the coordinates of each data point were manually forced to the center of the hallway as that is where the agent was actually located. How the algorithm handles changing conditions can be observed by simply gathering data under different conditions.

The data that are gathered from the "test drives" can then be used to run the localization algorithm. Each iteration, the next set of data is passed through just as it would be while running on Milpet. While the algorithm is being run, different images and metrics can be gathered about the performance. The same algorithm parameters, outlined in Table 4.4, were used for all tests and are listed in Table 5.5.

Table 5.5: Algorithm parameters for testing.

Parameter	Value
K	1500
L	5
Δ_{center}	Varied per hallway
T	45 cm
n	3
α	85%
β	15%
Z	10%
$Speed$	0.25 m/s

5.3 Localization Results

While the algorithm was running, the location of each particle, the actual location of the agent, and the predicted location of the agent could be recorded at each iteration. All of this information could be shown in a single image to visualize the progress and performance of the algorithm. Figure 5.1 shows an example of two of these images, one in Hallway A where the estimate is far from the actual location (top) and one in Hallway B where the estimate is close to the actual location (bottom).

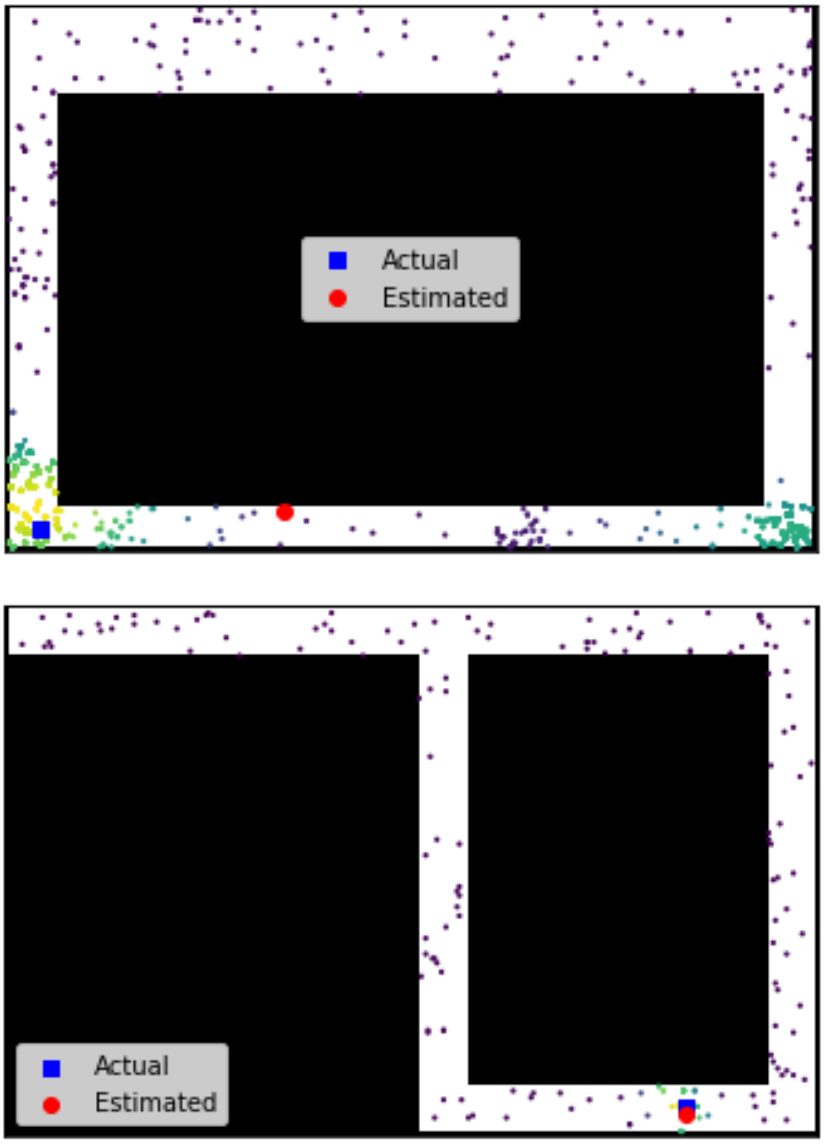

Figure 5.1: Example of visualization of the localization algorithm. (top) Hallway A with poor estimate and (bottom) Hallway B with good estimate.

In Figure 5.1 the small dots represent particles, with the color of the particles corresponding to how many are located in one pixel, ranging from dark purple for a single particle, to yellow for many. The larger blue square is the actual location, according to odometry, and the larger red circle is the estimate of location output by the localization algorithm. In Figure 5.1 (top) the estimate is far off from the actual location as the particles are split between the two corners. This split is because

the classifier has weighted each corner as possible, with the left corner being more probable. In Figure 5.1 (bottom) the estimate is almost the same as the actual, meaning the algorithm is performing well.

To fully test the algorithm and its robustness to changing conditions many "test drives" were collected under varying conditions. Test data were collected in both hallways, at varying times of day, in forward and reverse, with and without lateral movement, and under some other special conditions. This included images where the daylight was shining directly through the windows, such as in Figure 5.2.

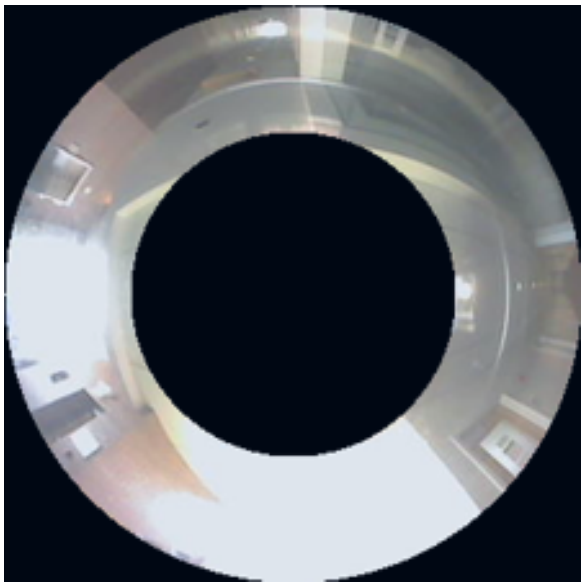

Figure 5.2: Example test image taken with intense daylight.

Figure 5.2 shows an example, taken from Hallway B, where daylight is coming through windows on two sides of the image. The light washes out most of that side of the image, which introduces a challenge for the classification model.

Table 5.6 lists each of the tests that were run and their resulting average error.

Table 5.6: Localization testing results.

Test	Hallway	Time of Day	Direction	Lateral Movement	Special Conditions	Average Error (px)
1	A	Day	Forward	None	None	4.073
2	A	Day	Reverse	None	None	3.641
3	A	Night	Forward	None	None	3.432
4	A	Night	Reverse	None	None	3.726
5	A	Night	Forward	None	0.65 m/s	4.659
6	A	Night	Forward	None	1.0 m/s	5.572
7	B	Day	Forward	None	None	2.000
8	B	Night	Forward	None	None	2.270
9	B	Day	Reverse	None	None	2.102
10	B	Night	Reverse	None	None	2.586
11	B	Day	Forward	None	People	2.283
12	A	Day	Forward	Yes	None	7.170*
13	A	Day	Forward	Yes	None	9.561*
14	B	Day	Forward	Yes	None	4.330*
15	B	Day	Forward	Yes	None	4.162*
16	A	Day	Forward	Yes	Autonomous	11.238**

Table 5.6 shows each of the tests that were run and the environmental conditions at the time of data collection. The hallway column specifies the hallway, the time of day column shows whether the data were captured in daylight or at night, the direction column specifies the direction of travel in the hallway, the lateral movement column specifies any lateral movement within the test, and the special conditions records if there were any additional abnormal conditions. The final column, the average error column, specifies the value of the metric used to evaluate the performance of each test. The average error is a measure of the average Euclidean distance from the ground truth location, provided by odometry, and the estimated location. The value is shown in pixels on the map, where a pixel is 4×4 inches.

Tests 1-4 are tests in Hallway A where the agent was centered for the duration of the test and there were no special conditions besides changing direction and daylight. The results show that the average error ranged from 3.4 to 3.7 pixels for all four tests.

When converted to inches, the average error ranges from 13.6 in to 14.8 in. The size of the agent used for testing is around 36 in by 36 in, meaning that, on average, the error for Hallway A under normal conditions was less than half the size of the agent. Tests 1-4 also show the algorithm is mostly invariant to direction of travel and lighting conditions as the average errors between the four tests are similar. Tests 7-10 show the same standard tests for Hallway B. These tests have a lower average error than Hallway A which may be attributed to Hallway B being easier for the classification model to learn. Tests 7-10 also reiterate the algorithm's invariance to direction and lighting.

Another form of visualization for the testing results is to plot the actual path and the estimated path on a single plot with the map as the background. This shows a direct comparison between the ground truth and the predictions. Figure 5.3 shows this visualization for Test 1 (top) and Test 7 (bottom) from Table 5.6.

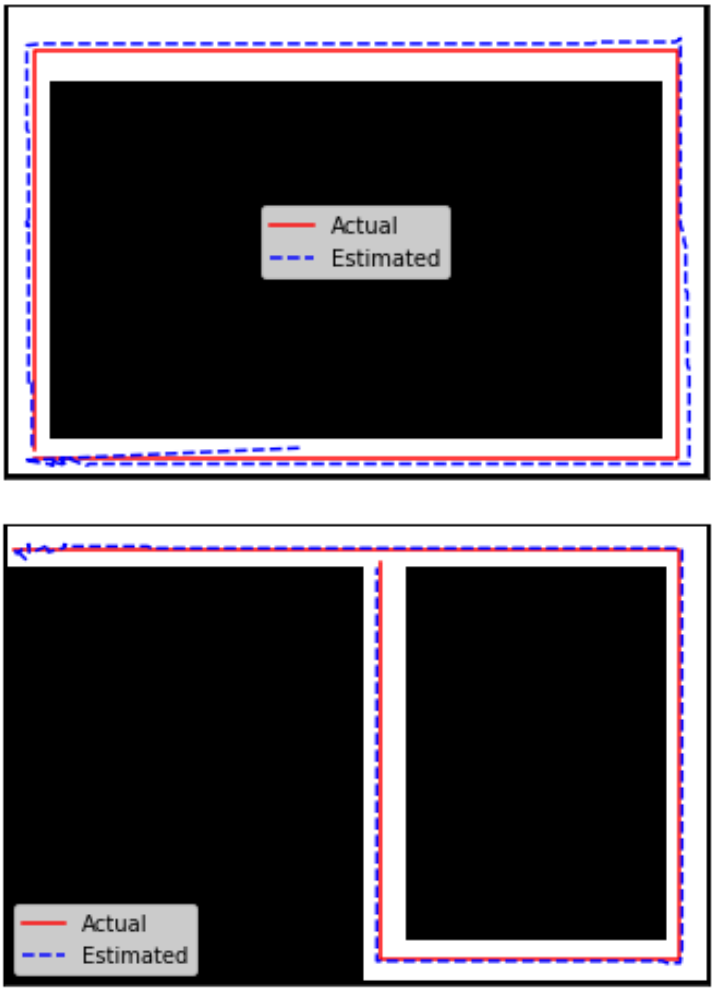

Figure 5.3: Path comparison results for Test 1 (top) and Test 7 (bottom).

In Figure 5.3 the red solid line represents the actual path taken by the agent, and the blue dashed line represents the estimated path taken. For both of these tests the agent remained centered within the hallway for the duration of the test. The estimated path, in both tests, follows the actual path very closely down the hallway but sometimes drifts off center for periods of time. This is most likely due to how the particles are distributed as there may not be many particles directly in the center at all times and therefore the estimate may be slightly off center. In Figure 5.3 (left) the estimated path starts somewhere in the middle of the lower hallway and then jumps to the corner. This is because the first image in this test was misclassified by the

classification model causing the estimate to be off. The next image was then classified correctly, and the estimate rapidly improved.

Tests 5 and 6 demonstrate how the algorithm reacts to changes in speed. All of the training data were collected at 0.25 m/s. Increasing the speed could have an effect on the performance as the images may be more blurry. The tests show that as the speed increases, the average error also increases. This increase in error is most likely due to degraded performance with the classification model, but the odometry information becoming inaccurate at higher speeds could also be a factor. Figure 5.4 shows a comparison between images taken at different speeds in similar locations in the hallway. It is difficult to see a difference in the images from 0.25 m/s (left) and 0.65 m/s (center), but at 1 m/s (right) the image is noticeably more blurry.

Figure 5.4: Comparison of images taken at different speeds. (left) 0.25 m/s, (center) at 0.65 m/s, and (right) taken at 1 m/s.

Figure 5.5 shows the visualization of the test results for Test 6 from Table 5.6. During Test 6 the agent was traveling much faster than the speed at which training data were collected. The training data were gathered at 0.25 m/s, and Test 6 was performed at 1 m/s, four times the speed.

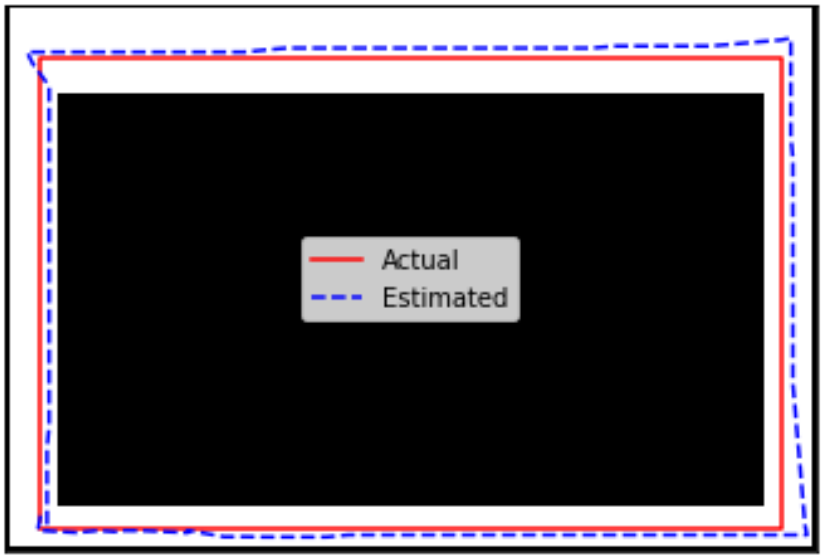

Figure 5.5: Path comparison results for Test 6.

Test 6 had a higher error rate when compared to Test 4 which had matching conditions with the exception of speed. Figure 5.5 shows that the error was higher because the path deviated laterally more than usual. This was due to the distance between iterations of the algorithm being much higher and causing the path to "overshoot" in the corners.

Test 11 demonstrates the algorithm's performance when many people are traveling near and around the agent. For this test, a heavy traffic environment was analyzed by having multiple people continuously walk past and stand near the agent while the data were being collected. The average error for this test shows that people had little to no effect on the algorithm's performance as the error was only slightly higher than that in Test 7, where the conditions were similar, but without the heavy traffic.

Tests 12 and 13 show how the algorithm performs when there is lateral movement within the hallway. These two tests were shorter, each consisting of just a single corner in Hallway A. The data for these tests required the agent to be driven manually as lateral movements needed to be added. Manual driving adds inconsistencies in speed and turning radius. The average error of Tests 12 and 13 are shown as 7.170 pixels

and 9.561 pixels respectively, higher than Test 1, which had similar conditions. The error is inflated due to two factors: manual driving and errors in the ground truth location from the odometry. When the agent is turning, the odometry data quickly become unreliable due to compounding errors. The errors are a combination of wheel slippage and the agent containing a complicated arrangement of four caster wheels that often make wheel movement unpredictable. As the agent is turning, the ground truth data from odometry is much less reliable which may inflate the average error of the test as the algorithm is being compared directly to the odometry. The results of Test 12 and 13 are better visualized by comparing the actual path to the estimated path as shown in Figure 5.6.

For Tests 14 and 15, the agent was driven straight with a constant offset from the center of the hallway: 9 inches to the right in Test 14 and 18 inches to the right in Test 15. Each test consisted of one full lap of Hallway B. The error for each test was higher than that of Test 7, which had similar conditions but was conducted in the center of the hallway. The increase in error is most likely caused by degraded performance of the classification model, the method used to shift the waypoint locations based on ultrasonic data, and errors in the odometry information. The classification model was trained on images from only the center of the hallway, meaning that the performance away from the center could be reduced. The waypoints are shifted a constant amount based on a single threshold of the ultrasonic data. This means that the agent must be shifted enough to move the waypoints. This could lead to increased error if the agent is shifted but not shifted enough to move the waypoint locations. While the agent was moving straight for Tests 14 and 15, error in the odometry information may have still played a roll in the increased average errors. Since the agent was not in the center, the odometry information could not be kept centered and therefore did drift slightly throughout the tests.

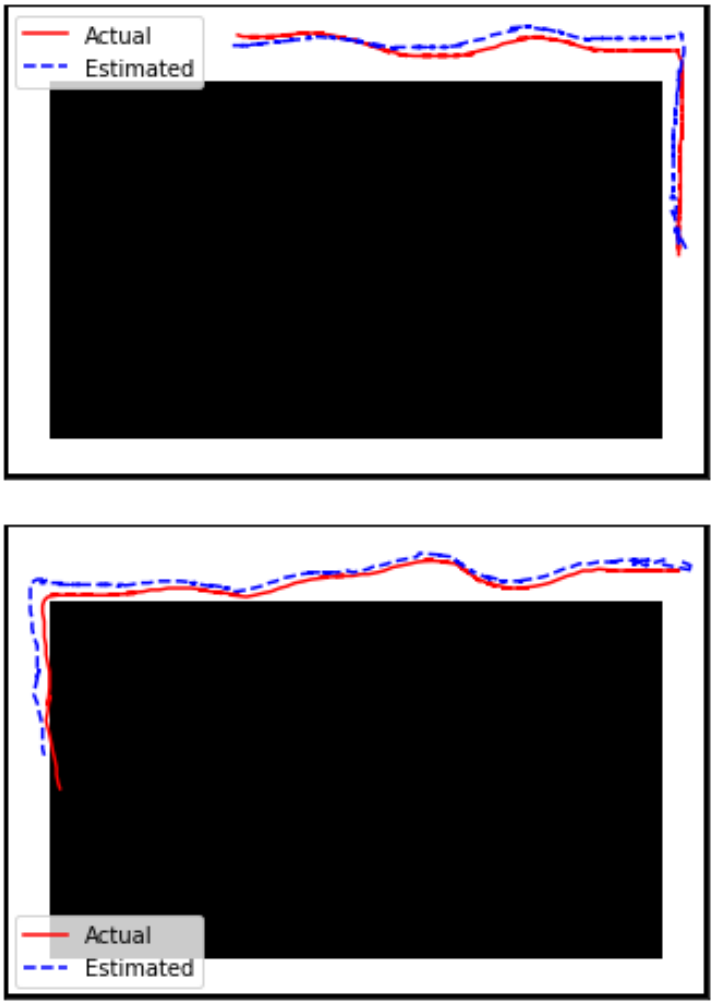

Figure 5.6: Path comparison results for Test 12 (top) and Test 13 (bottom).

Figure 5.6 shows the estimated path versus the path from the odometry information for Test 12 (top) and 13 (bottom). During each test, the agent was driven side to side within the hallway rather than remaining in the center. While the average errors for these tests were high, this visualization reveals that the algorithm was able to estimate the lateral movements of the agent. The algorithm follows local lateral movement even when the overall path may be off. This visualization also reveals that the odometry information can become unreliable after many turns. For example, in Figure 5.6 (bottom), the odometry shows the agent traveling into the invalid area in the center of the map which did not actually occur. In that case, the estimated

path was more accurate to how the agent moved than the odometry. These errors in odometry caused higher error rates in both of the tests.

The final test in Table 5.6, Test 16, was a test of the fully autonomous navigation of the agent, Milpet. During this test, the agent was commanded to move from one point in the hallway to another three separate times. This test required the combination of path planning, the localization algorithm, and obstacle avoidance. When the command was given, the path planning algorithm found the shortest path from the current location, provided by the localization algorithm, to the destination, and the obstacle avoidance algorithm moved the agent along the path while avoiding obstacles. While traveling, the localization algorithm was continuously updating the agent's location so the path planning algorithm could track progress to the goal. During the test, the odometry information and the estimated location were recorded to determine an error. The average error for this test was 11.238 pixels. However, as with Tests 12 and 13, the error is heavily inflated due to odometry errors. It is better to look at the visualization of results shown in Figure 5.7.

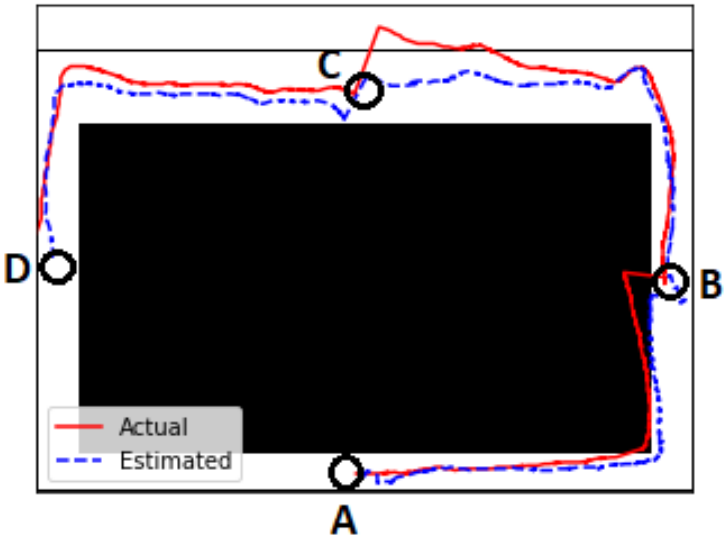

Figure 5.7: Path comparison results for Test 16.

Figure 5.7 shows the comparison of the estimated path versus the odometry in-

formation during the fully autonomous test, Test 16. This test consisted of the agent being commanded to go from point A, denoted in Figure 5.7, to point B, then from point B to C, and finally from point C to point D. In between each step, the agent was stopped, and the odometry was reset, resulting in the large jagged sections seen near each point. While the agent was traveling, it did not travel in perfectly straight lines or remain centered in the hallway as there were obstacles along the walls of the hallways that caused the agent to shift to avoid them. This visualization further shows that the odometry information is very unreliable as it often indicates travel into invalid areas of the map or even outside the map. While the odometry is generally incorrect globally, the local changes and turns are correct, with the exception of the big changes where the odometry was reset. In terms of the algorithm, this visualization shows good and bad performance. In some cases, such as along the top of the map, the estimated path follows local changes from the odometry while also correctly identifying the global position. In other cases, such as on the right side and after any odometry reset, it is shown that the algorithm follows the changes in the odometry too closely, causing errors. This happens as a result of updating the particles with local movement from the odometry each iteration.

Another visualization for this test is to compare the estimated path with the path generated by the path planning algorithm. The planned path is a global path that the agent does not necessarily follow as there may be obstacles along the path; however it can be used to compare with the estimated path. Figure 5.8 shows this comparison.

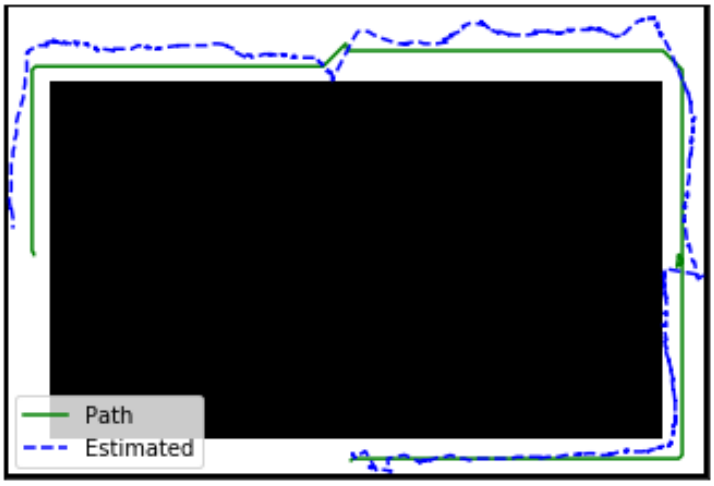

Figure 5.8: Path comparison results for Test 16 when compared to the path generated by the path planning algorithm.

In Figure 5.8 the green solid line is the planned path, and the blue dashed line is the estimated path from the localization algorithm. This visualization shows how closely the estimated path followed the planned path. The planned path is straight and centered within the hallway as it is the global path; however the agent did not follow this path exactly due to the obstacle avoidance algorithm. The one large shift in the planned path along the top of Figure 5.8 was caused by the estimated location being shifted from the center when the path was planned at that point. In general, the localization algorithm correctly tracks the agent along the path, with the exception of when the agent must deviate from the path or when the algorithm follows the incorrect odometry information too closely.

The fully autonomous test, Test 16, also shows that the localization algorithm is good enough to achieve fully autonomous navigation when combined with the path planning and obstacle avoidance algorithms implemented on Milpet. Even with some tracking error, fully autonomous operation is achieved thanks to the local waypoints utilized in the path planning algorithm, detailed in Section 3.2.1.4.

To show the algorithm was capable of running in real-time, the computation time was measured on the embedded PC of the test agent. Both a single iteration of the

whole localization algorithm and the time taken to pass an image through the ResNet model were measured individually. Table 5.7 shows the timing results. Each time shown in Table 5.7 is an average of ten iterations while ROS, including most nodes, was also running on the PC.

Table 5.7: Algorithm timing results.

Operation	Computation Time (s)
ResNet Test	0.1742
Full Algorithm	0.2113

The timing analysis results shown in Table 5.7 show that the full algorithm takes an average of 0.21 seconds to run. This means it is possible to run one iteration of the algorithm four times per second; however it may be better to run an iteration only two or three times per second. The algorithm is very CPU intensive, and running it four times per second would use most of the system resources that may be needed by other navigation algorithms. If the estimated location needs to be updated more often, the algorithm can be modified to produce faster estimates, from only odometry information, and periodically run the full algorithm to correct the estimate.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

The topic of agent localization remains, and will continue to remain, an active topic of research for many years to come. The growing desire for autonomous agents drives the demand for cheaper, more accurate, and more robust localization solutions. While there is no single solution for all agents and all environments, it is desirable to have a system that can generalize to as many different scenarios as possible. This research presents a novel approach to indoor localization that has been proven to be fast enough to run in real-time and robust to changing environmental conditions. The algorithm only utilizes data from sensors attached to the agent, does not rely on external signals, and is accurate to within one half of the test agent's size. While the algorithm was tested on an autonomous wheelchair platform, it can be adapted to many other agents and many different environments. Another significant contribution of this research was creating the autonomous wheelchair, Milpet. Milpet is an affective access wheelchair capable of fully autonomous navigation which demonstrates the possibility of autonomous wheelchairs being used to aid those confined to a wheelchair in the future. Autonomous wheelchairs, such as Milpet, could one day give privacy and independence back to those who rely on a caretaker every day of their lives.

6.2 Future Work

There are a few areas of the algorithm that could be improved in the future. The biggest improvement would be to develop an additional part of the algorithm that can predict the rotation, or pose, of the agent at any time. Currently, the algorithm relies on odometry information from the wheel encoders to know the pose, which is prone to error due to compounding error from the encoders. One method would be to train an additional classification model to predict only pose from the omni-vision images, ultrasonic data, or both. Another area for improvement is the lateral prediction. The ultrasonic data become unreliable for lateral positioning if too many obstacles are present in the hallway. It may be possible, again, to train a classification model to predict between discrete lateral classes.

An additional area for future work could be developing a more efficient way to collect data to train the classification model. The process now involves precise measurement and then driving the agent through the hallway many times while stopping periodically to reset the odometry. It would be ideal to be able to drive the agent through the environment just a few times, without specific measurements, and have it automatically label the data in an unsupervised fashion.

One final area of future work is to expand the algorithm to work seamlessly throughout an entire building. The first step to this would be testing and adapting the algorithm to work inside rooms rather than just hallways. Next, the individual maps of an entire building would need to be learned and some way to identify which room or hallway the agent is traveling through would need to be developed. This could be in the form of a global classification model that is trained to predict which local area of the building the agent is located in. Then, from there, each local map and classifier could take over.

Bibliography

[1] A. Wong, M. Yousefhussien, and R. Ptucha, "Localization using omnivision-based manifold particle filters," in *SPIE/IS&T Electronic Imaging*. International Society for Optics and Photonics, 2015, pp. 940 606–940 606.

[2] Itseez, "Introduction to support vector machines," http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html, 2014, accessed: 2017-04-24.

[3] F.-F. Li, A. Karpathy, and J. Johnson, "Cs231n: Convolutional neural networks for visual recognition," 2016. [Online]. Available: http://cs231n.stanford.edu/

[4] D. V. Boxel, "Deep learning with tensorflow," 2016.

[5] R. Ptucha, "Deep learning, supervised," 2015.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[8] C. Robotics, "Ros 101: Intro to the robot operating system," http://robohub.org/ros-101-intro-to-the-robot-operating-system/, 2014, accessed: 2017-05-08.

[9] S. He and S.-H. G. Chan, "Wi-fi fingerprint-based indoor positioning: Recent advances and comparisons," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 466–490, 2016.

[10] Y. Wang, Q. Ye, J. Cheng, and L. Wang, "Rssi-based bluetooth indoor localization," in *2015 11th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*. IEEE, 2015, Conference Proceedings, pp. 165–171.

[11] A. H. Salamah, M. Tamazin, M. A. Sharkas, and M. Khedr, "An enhanced wifi indoor localization system based on machine learning," in *Indoor Positioning and Indoor Navigation (IPIN), 2016 International Conference on*. IEEE, 2016, Conference Proceedings, pp. 1–8.

[12] B.-S. Choi and J.-J. Lee, "Sensor network based localization algorithm using fusion sensor-agent for indoor service robot," *IEEE Transactions on Consumer Electronics*, vol. 56, no. 3, pp. 1457–1465, 2010.

[13] C. Ramer, J. Sessner, M. Scholz, X. Zhang, and J. Franke, "Fusing low-cost sensor data for localization and mapping of automated guided vehicle fleets in

indoor applications," in *Multisensor Fusion and Integration for Intelligent Systems (MFI), 2015 IEEE International Conference on*. IEEE, 2015, Conference Proceedings, pp. 65–70.

[14] D. Jeon and H. Choi, "Multi-sensor fusion for vehicle localization in real environment," in *Control, Automation and Systems (ICCAS), 2015 15th International Conference on*. IEEE, 2016, Conference Proceedings, pp. 411–415.

[15] S. Kwon, K. Yang, and S. Park, "An effective kalman filter localization method for mobile robots," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2006, Conference Proceedings, pp. 1524–1529.

[16] S. Y. Kim, K. S. Yoon, D. H. Lee, and M. H. Lee, "The loosely coupled integration system using a pseudolite ultrasonic system and a dead-reckoning for the autonomous mobile robots localization," in *Control Automation and Systems (ICCAS), 2010 International Conference on*. IEEE, 2010, Conference Proceedings, pp. 1908–1911.

[17] D. Fontanelli, L. Ricciato, and S. Soatto, "A fast ransac-based registration algorithm for accurate localization in unknown environments using lidar measurements," in *2007 IEEE International Conference on Automation Science and Engineering*. IEEE, 2007, Conference Proceedings, pp. 597–602.

[18] K. Wan, L. Ma, and X. Tan, "An improvement algorithm on ransac for image-based indoor localization," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2016 International*. IEEE, 2016, Conference Proceedings, pp. 842–845.

[19] K. Guan, L. Ma, X. Tan, and S. Guo, "Vision-based indoor localization approach based on surf and landmark," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2016 International*. IEEE, 2016, Conference Proceedings, pp. 655–659.

[20] C. Piciarelli, "Visual indoor localization in known environments," *IEEE Signal Processing Letters*, vol. 23, no. 10, pp. 1330–1334, 2016.

[21] S. Bag, V. Venkatachalapathy, and R. W. Ptucha, "Motion estimation using visual odometry and deep learning localization," *Electronic Imaging*, vol. 2017, no. 19, pp. 62–69, 2017.

[22] L. Ojeda and J. Borenstein, "Non-gps navigation with the personal dead-reckoning system," in *Defense and Security Symposium*. International Society for Optics and Photonics, 2007, pp. 65 610C–65 610C.

[23] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust monte carlo localization for mobile robots," *Artificial intelligence*, vol. 128, no. 1-2, pp. 99–141, 2001.

[24] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE robotics & automation magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[25] S. Thrun and J. J. Leonard, "Simultaneous localization and mapping," in *Springer handbook of robotics*. Springer, 2008, pp. 871–889.

[26] M. Montemerlo, S. Thrun, D. Koller, B. Wegbreit *et al.*, "Fastslam: A factored solution to the simultaneous localization and mapping problem," in *Aaai/iaai*, 2002, pp. 593–598.

[27] A. Neves, H. C. Fonseca, and C. G. Ralha, "Location agent: a study using different wireless protocols for indoor localization," *International Journal of Wireless Communications and Mobile Computing*, vol. 1, pp. 1–6, 2013.

[28] D.-V. Nguyen, M. V. Recalde, and F. Nashashibi, "Low speed vehicle localization using wifi fingerprinting," in *International Conference on Control, Automation, Robotics and Vision, ICARCV 2016*, 2016.

[29] C. Chen, Y. Chen, Y. Han, H.-Q. Lai, and K. R. Liu, "Achieving centimeter-accuracy indoor localization on wifi platforms: A frequency hopping approach," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 111–121, 2017.

[30] T. Sanpechuda and L. Kovavisaruch, "A review of rfid localization: Applications and techniques," in *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2008. ECTI-CON 2008. 5th International Conference on*, vol. 2. IEEE, 2008, pp. 769–772.

[31] M. Sugano, T. Kawazoe, Y. Ohta, and M. Murata, "Indoor localization system using rssi measurement of wireless sensor network based on zigbee standard," *Target*, vol. 538, p. 050, 2006.

[32] M. N. Husen and S. Lee, "Design guideline of wi-fi fingerprinting in indoor localization using invariant received signal strength," in *Information and Communication Technology (ICICTM), International Conference on*. IEEE, 2016, pp. 260–265.

[33] L. Miao, "Calibration-free wireless indoor localization (cafloc)," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, March 2017, pp. 337–342.

[34] M. Liu, H. Wang, Y. Yang, Y. Zhang, L. Li, J. Xu, Y. Du, L. Ma, and N. Wang, "Rfid indoor localization system for tag and tagfree target based on interference," in *Advanced Communication Technology (ICACT), 2017 19th International Conference on*. IEEE, 2017, pp. 372–376.

[35] B. Bischoff, D. Nguyen-Tuong, F. Streichert, M. Ewert, and A. Knoll, "Fusing vision and odometry for accurate indoor robot localization," in *Control Automation Robotics & Vision (ICARCV), 2012 12th International Conference on*. IEEE, 2012, pp. 347–352.

[36] N. Ganganath and H. Leung, "Mobile robot localization using odometry and kinect sensor," in *Emerging Signal Processing Applications (ESPA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 91–94.

[37] J. Biswas and M. Veloso, "Depth camera based indoor mobile robot localization and navigation," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 1697–1702.

[38] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[39] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," *Computer vision–ECCV 2006*, pp. 404–417, 2006.

[40] N. Mostofi, M. Elhabiby, and N. El-Sheimy, "Indoor localization and mapping using camera and inertial measurement unit (imu)," in *Position, Location and Navigation Symposium-PLANS 2014, 2014 IEEE/ION*. IEEE, 2014, pp. 1329–1335.

[41] W. Sui and K. Wang, "An accurate indoor localization approach using cellphone camera," in *Natural Computation (ICNC), 2015 11th International Conference on*. IEEE, 2015, pp. 949–953.

[42] J.-M. Morel and G. Yu, "Asift: A new framework for fully affine invariant image comparison," *SIAM Journal on Imaging Sciences*, vol. 2, no. 2, pp. 438–469, 2009.

[43] J. Hesch and N. Trawny, "Simultaneous localization and mapping using an omnidirectional camera," *Available: www-users. cs. umn. edu/˜ joel/_files/BoS. pdf*, 2005.

[44] E. Menegatti, A. Pretto, A. Scarpa, and E. Pagello, "Omnidirectional vision scan matching for robot localization in dynamic environments," *IEEE transactions on robotics*, vol. 22, no. 3, pp. 523–535, 2006.

[45] Y.-b. LI, Z.-l. CAO, C.-j. LIU, and S.-h. YE, "Omni-directional visual sensor for mobile robot navigation based on particle filter [j]," *Chinese Journal of Sensors and Actuators*, vol. 5, p. 026, 2009.

[46] Z. Cao and S. Liu, "Dynamic omni-directional vision localization using a beacon tracker based on particle filter," in *Optics East 2007*. International Society for Optics and Photonics, 2007, pp. 67 640U–67 640U.

[47] B. Resch, J. Lang, and H. P. Lensch, "Local image feature matching improvements for omnidirectional camera systems," in *Pattern Recognition (ICPR), 2014 22nd International Conference on*. IEEE, 2014, pp. 918–923.

[48] R. E. Kalman *et al.*, "Contributions to the theory of optimal control," *Bol. Soc. Mat. Mexicana*, vol. 5, no. 2, pp. 102–119, 1960.

[49] S. J. Julier and J. K. Uhlmann, "New extension of the kalman filter to nonlinear systems," in *AeroSense'97*. International Society for Optics and Photonics, 1997, pp. 182–193.

[50] E. A. Wan and R. Van Der Merwe, "The unscented kalman filter for nonlinear estimation," in *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*. Ieee, 2000, pp. 153–158.

[51] J. Handschin, "Monte carlo techniques for prediction and filtering of non-linear stochastic processes," *Automatica*, vol. 6, no. 4, pp. 555–563, 1970.

[52] N. J. Gordon, D. J. Salmond, and A. F. Smith, "Novel approach to nonlinear/non-gaussian bayesian state estimation," in *IEE Proceedings F (Radar and Signal Processing)*, vol. 140, no. 2. IET, 1993, pp. 107–113.

[53] S. Thrun, "Particle filters in robotics," in *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 2002, pp. 511–518.

[54] A. H. Jazwinski, *Stochastic processes and filtering theory*. Courier Corporation, 2007.

[55] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[56] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[57] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," *Machine learning: ECML-98*, pp. 137–142, 1998.

[58] E. Osuna, R. Freund, and F. Girosit, "Training support vector machines: an application to face detection," in *Computer vision and pattern recognition, 1997. Proceedings., 1997 IEEE computer society conference on*. IEEE, 1997, pp. 130–136.

[59] B. Scholkopf, "The kernel trick for distances," *Advances in neural information processing systems*, pp. 301–307, 2001.

[60] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[61] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[62] D. Huggins-Daines, M. Kumar, A. Chan, A. W. Black, M. Ravishankar, and A. I. Rudnicky, "Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices," in *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, vol. 1. IEEE, 2006, pp. I–I.

[63] B. Gassend, "Ros audio_common package," https://github.com/ros-drivers/audio_common, 2017.

[64] S. Koenig and M. Likhachev, "Dˆ* lite," in *AAAI/IAAI*, 2002, pp. 476–483.

[65] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, 1991.

[66] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics & Automation Magazine*, vol. 4, no. 1, pp. 23–33, 1997.

[67] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[68] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[69] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[70] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[71] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks."

[72] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

Appendix A: Additional Hardware Diagrams

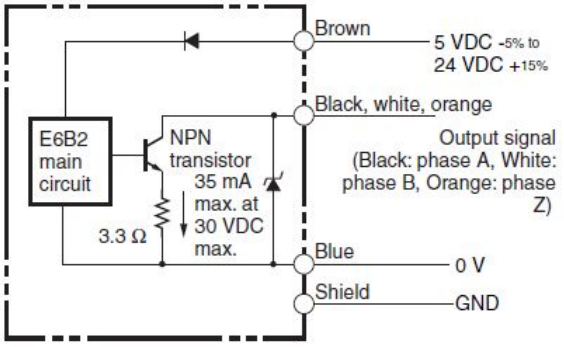

Figure A.1: Internal diagram of the E6B2-CWZ6C quadrature encoder.

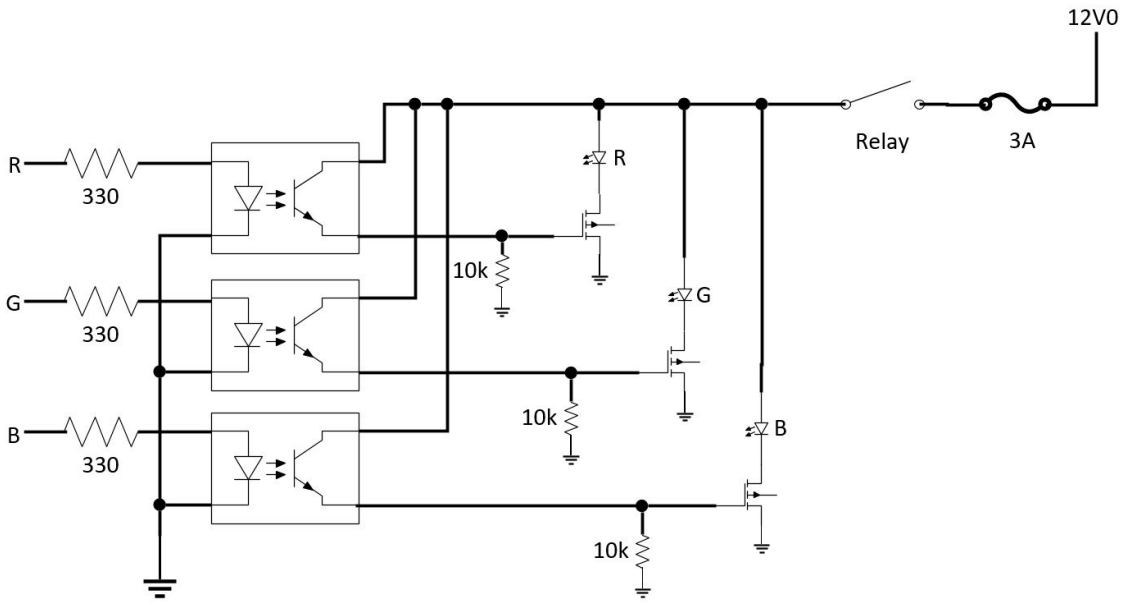

Figure A.2: Diagram of LED driver used on Milpet. The R,G,B inputs are PWM signals that control each of the R,G,B LEDs.