Obstacle Avoidance in Intelligent Assisted Walking Devices for Improving Mobility Among Seniors with Cognitive and Visual Impairments

by

Alexander Tettenborn

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Mechanical Engineering

Waterloo, Ontario Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Acknowledgements

I would like to express my sincere thanks to my supervisors, Dr. Jan Huissoon and Dr. James Tung for their constant guidance, support and patience throughout my studies. I would like to thank my review readers Dr. Armaghan Salehian and Dr. Behrad Khamesee for taking time out of their schedules to read this thesis.

I am privileged to have wonderful parents who have been kind and supportive throughout my entire life, who taught me so much, and without whom I have no doubt I would not have made it to where I am today. This thesis is a result of their efforts in nurturing the love of science and technology they recognized in me as a child.

I would especially like to thank my incredible wife Audra. Her love, support and unending patience has allowed me to pursue my passions. I promise her that someday I will finally leave the life of a student behind and I know that she'll still be there with me, unwavering in her determination that I do what I love.

Finally, I would like to thank all the other friends and family who were with me along the way. My brother Garnet, my uncle Peter, my aunt Anne, my grandfather Tom who I miss dearly, my family over in Germany, the gang out in Halifax, the friends from school who are still a big part of my life. All of them have been instrumental to me becoming the person I am today.

Abstract

Current research in walkers and rollators with integrated intelligent computing and robotic components shows promise in treatment, management and rehabilitation of a variety of ailments and disorders such as stroke, Alzheimer disease and multiple sclerosis.

In this thesis a novel intelligent walker is designed, constructed and tested for the purpose of examining whether we can increase mobility among individuals with vision and cognitive impairments hindering their ability to move collision free about their environments, by detecting obstacles and using brakes to guide the user around them.

This walker consists of a support frame, front castor wheels and rear particle brakes. Obstacle detection and localization are sensed by an onboard 3D depth camera and RGB camera (The Microsoft Kinect) and encoders in the rear wheels. This data is processed by an onboard laptop, producing a 2-dimensional map of the environment. This map is inputted into the control algorithms to make braking decisions for obstacle avoidance.

Two control algorithms are presented. The first is an open loop proportional gain control which determines necessary braking torque directly from the distance to the nearest obstacle. The second is a closed loop control which uses the systems dynamics and velocity data from the wheel encoders to estimate the forces being applied by the user and calculates the braking torque necessary to avoid obstacles.

The walker moment of inertia and the viscous damping parameters of the system are estimated experimentally. The effect of varying three parameters in the closed loop algorithm and one parameter in the open loop algorithm are examined in a corner turning test. Observations support predictions made by the derived system dynamics.

Lastly, the efficiency of the system at real world obstacle avoidance is tested in a controlled indoor obstacle course using goggles to impair the vision of otherwise able bodied test subjects. The open loop control algorithm was found to reduce the occurrence of collisions by 44% as compared to trials with no braking. The closed loop control algorithm was found to greatly reduce collisions with the front of the walker, however shows a tendency for over steering the user, producing a higher number of collisions with the walker's side. Possible causes and solutions to this problem are discussed.

This thesis demonstrates promise in the approach of using braking to help walker users avoid collisions with their environments. Discussion is offered about necessary next steps towards testing with regular users of assisted walking devices, and eventually real world use.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In Canada, the population commonly referred to as the “baby boomers” (born 1946 to 1965) will reach retirement age in the next two decades. This will cause the proportion of Canadians over the age of 65 to increase from its current rate of around 13%, to a projected 23%, while at the same time, those between the ages of 15 and 65 (the so-called 'working age' population), will decrease from 68% to around 60% [1]. In such an aging society we are likely to experience a shortage of workers available to care for seniors. This disparity makes it increasingly necessary for the elderly to maintain a level of independence; however, many such people suffer from injuries, poor eyesight, cognitive impairment, and a general lack of muscle strength, which leads to increased need for the support of other people. One area of vital importance in maintaining an independent lifestyle in old age is the ability to safely and steadily navigate one's environment. For about 9% of the nearly five million seniors currently living in Canada, this is accomplished primarily through the use of a walker or rollator device [2].

Figure 1.1: A rollator walker (left) and a simple walker (right)

These assisted walking devices, though smaller than a wheelchair, do (by the necessity of having a wide, firm base) have a degree of spatial bulk which can make them challenging to navigate through an environment. This presents an issue, as aging people tend to see a change in their cognitive ability to acquire and use spatial information about their environment; in general, distance perception and spatial orientation ability tend to decrease in older adults [3]. One common cause of this difficulty is progressive neurodegenerative diseases such as Alzheimer's [4]. This change occurs in addition to a general increase in vision impairment among the elderly. Studies have found that age is the single best predictor of blindness and visual impairment [5].

Walkers are a valuable and relevant intervention into the problem of maintaining mobility and independence in old age; however they have been shown to be problematic for some users in navigating through environments, both in user satisfaction surveys [6] and in laboratory studies [7], where users have been observed having trouble with tasks such as navigating doorways without colliding with the door frame. This presents an issue as an estimated 2037 injuries occur per year as a result of a walker colliding or catching on an environmental hazard [8].

In Chapter 2 of this thesis a background is presented on the issues involved in walker use and the approach of integrating intelligent systems into walkers to mitigate these problems. In Chapter 3 a novel assisted-walking device is presented with integrated intelligent systems designed to assist the user in obstacle avoidance. This new walker uses passive control to help users navigate through their environment while avoiding potentially destabilizing obstacles. The hardware (described in Chapter 3) includes an onboard controller, which takes input information from wheel encoders, a depth camera, and an RGB camera, and a pair of particle brakes in the wheels. Environment mapping and localization, described in Chapter 4, is achieved with the depth camera and wheel encoders. The physical system of the walker and user is characterized in Chapter 5 and used to design two potential control schemes to guide the user with minimal braking torque. These schemes are then experimentally evaluated in Chapter 6 with parameter variation testing and a controlled obstacle avoidance test. Finally Chapter 7 presents discussion on the results of the experiments and a description of future work.

Chapter 2

Background

2.1 Walker Use and Issues

As mentioned in Chapter 1 the number of seniors in Canada using a walker or rollator as their primary mobility aid is nearly half a million, as of 2009 [1]. A walker is generally defined as a frame that is designed to support someone (such as a baby, or an injured or elderly person) who needs help walking. This thesis deals solely with walkers designed for use by adults and ignores its use as an infant walking aid. Rollator is a genericized trademark that refers to a type of walker with 4 wheels, handlebars and, usually, a built-in seat and carrying basket. They are equipped with hand brakes in the handlebars for manual braking. The rollator was invented in Sweden and is used there by upwards of 300,000 people, the highest percentage of any country measured [9]. No specific data could be found on the prevalence of rollators versus two-wheeled or no-wheeled walkers in Canada.

Walkers are used for support and mobility among the elderly or injured. Clinical study has found that the use of a four-wheeled walker use can reduce the risk of fractures due to falls despite a decline in general status of health [9]. The ability to walk independently contributes to strengthened muscles and skeleton, which leads to reduced fractures, even though falls may still occur [10].

One of the major issues with walkers is that they increase the volume and area that the user takes up as they try to move around their environment; although this provides a stable platform for balance, it can lead to collisions with obstacles, which can in turn cause falls. Walker-related injury can occur as a result of collision with, or catching on, external obstacles [8]. One study found that in the United States 42,000 injuries associated with walker use are treated annually, a rate seven times higher than injuries associated with cane use. In an estimated 2037 of these cases the circumstance of the injury is “caught or hit aid on object” [ibid]. While this is a small section of the elderly population, this

number of cases warrants investigation into possible interventions to reduce the rate of collision associated injuries. In addition, collisions which are not associated with injuries but which, user satisfaction surveys show, can cause frustration and annoyance to the user and reduce their ability to maintain mobility in their own home, should be considered [6].

2.2 Intelligent Walkers

Integration of intelligent systems in assisted walking devices has been attempted with a variety of approaches, including obstacle avoidance, wayfinding, fall detection and mobility evaluation. This review will focus primarily on obstacle avoidance, which is the process of moving through an immediate environment (eg. getting across a room, around furniture, through doorways etc.). This is distinguishable from wayfinding, which is the process of to moving from one area in a building to another (eg. finding your way from the bedroom to the kitchen).

Intelligent walkers intended to alter the movement of the user in some way belong to either the active or passive types. Active walkers are those which provide energy to the walker's motion. This usually involves servo motors in the wheels which drive the walker [11]. These systems tend to be heavy and complex due to the motors, reduction gears and batteries required. In particular, batteries pose an issue for long term use as the servo motors used in these systems have high power requirements. There is also the possibility that inappropriately-controlled servo motors may move unintentionally, causing danger to their user. Passive walkers are those which steer or guide the user without providing energy to their motion. Guidance is achieved by using servo motors to change the walker's direction of motion by turning the castor wheels [12] or by attaching a steering wheel to a servo motor to allow for joint user/computer steering [13]. Another possibility is the use of brakes in two of the walker wheels; by engaging only one brake at a time, or by engaging one brake with more torque than the other, the dynamics of the walker are altered in order to steer around obstacles. Passive guidance using braking has lower power requirements than active motor-driven systems and has the safety feature of being unable to move on its own without user intention.

Previously, Hirata, Hara and Kosuge published in *IEEE Transactions on Robotics* a paper on the design and testing of passive-controlled intelligent walker, called the RT Walker, which uses rear-wheel servo brakes for slowing and turning [14]. As the RT Walker represents the most similar research to that presented in this thesis, the following section will examine Hirata, Hara and Kosuge's paper in depth.

The RT Walker uses a laser range-finder (LRF) for obstacle detection. The LRF used was a

one-dimensional arrays of lasers, which use time-of-flight technology to find the distance to a two-dimensional slice of the environment. LRFs are generally quite accurate, but are incapable of imaging the 3D environment at once; as a result, obstacle detection depends on the height of the LRF and its inclination, it could miss potential obstacles. For example, an LRF mounted one foot off the ground and angled downwards, as on the RT Walker, would not see the obstacle posed by the lip of a table (**Fig 2.1**).

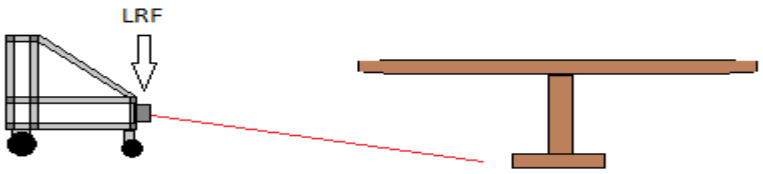


Figure 2.1: In this example the LRF can see the base of the table but misses the lip.

Detail is given on the RT Walker's control algorithm and the physical model behind it. The dynamics of the system are given, with respect to the force applied by the human user and the force applied by the servo brakes. Next, a method of altering the dynamics of the system by braking is outlined. This system uses closed loop control based on encoder readings of the velocity and acceleration of the system. The control scheme is then further modified to include sensory data of the environment. Two tilt-angle sensors provide information on the direction of gravity which is compensated for in the braking in order to render the dynamics of the system the same on a slope as on a level plane. The LRF detects both obstacles and drops. This information is used to produce an artificial potential field. The potential field is generated based on the distance between obstacles and a point in front of the walker. How this point is selected is not described. The brakes produce the necessary torque to simulate a force applied by the potential field on this point and this guides the walker around and away from these obstacles.

The first experiment tests the dynamic control of the walker without environmental sensing. The walker was placed on a slope facing downhill to simulate a user applying a constant force. Three desired dynamical coefficients were specified and the measured velocity response of the walker was compared to the theoretical velocity response. In all three cases the velocity reached a steady state at, or very near, the theoretical predicted value, validating the physical system.

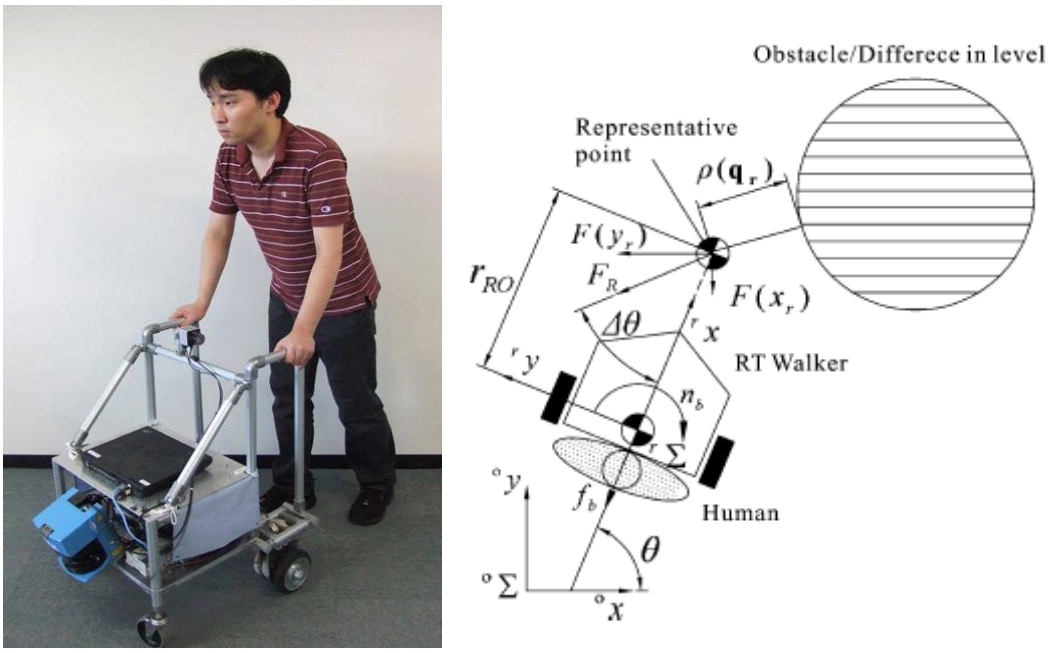


Figure 2.2: The RT Walker and its Obstacle Avoidance Method.

In the next experiment, the walker was again placed on a slope, now at an angle relative to the direction of the slope. The dynamical system was given five damping parameters relative to the rotational motion of the walker. It is shown that a larger damping coefficient produces a straighter arc of motion when the walker is released (as the gravity attempts to turn the walker to face downhill); however, in this case no comparison is made to the theoretical path. While this is the expected correlation, it does nothing to show the accuracy of the model. The theoretical paths should not have been difficult to calculate based on the already derived mathematical model and would have been useful in showing that the angular part of the model is as accurate as the translational part.

The paper ends with a short section on human testing. In the first trial, four university students were blindfolded to prevent visual feedback and guided the walker through a walled u-shaped path. All four students succeeded in avoiding collisions with the walls. In the second trial, five blindfolded university students were guided by the RT Walker along an s-shaped path between a start and end point. The authors state that "the differences between desired and actual paths were almost zero". In the final trial, a single blindfolded user was guided by the walker through an environment consisting of stairs, obstacles and a downward slope. The user successfully navigated this course. There are some issues with this methodology. A larger sample size is needed to improve the validity of these tests. In particular, the final test being done with only a single trial makes it difficult to draw any real conclusions on how well the walker works. With so few trials, the authors were unable to compile the data necessary to rule out chance as a major factor in the successful navigation. Furthermore, the trials

were not controlled; in order to confirm the improvement in user mobility, the authors could have had their subjects attempt to navigate the course with the walker's navigation algorithm disabled. It may seem obvious that a blindfolded user with both hands on the walker should not be able to navigate a course that he/she has no knowledge of, but it would be useful to show that the navigation software is an improvement over a user's ability to use his/her hands to help navigate by feeling around the space near the walker.

The intelligent walker presented in this thesis addresses some of the aforementioned issues in the methodology of Hirata, Hara and Kosuge. It avoids the problems inherent in the use of laser range-finders, by instead using a three-dimensional depth camera, which images a much larger portion of the environment at once. The passive control algorithms differ from those used by the RT-Walker and were tested more rigorously, using a controlled obstacle avoidance test with a larger number of trials. Unfortunately due to the lack of a control group in the RT-Walkers obstacle avoidance testing, it is not possible to compare the real world efficacy of the algorithms used by the two walkers.

Chapter 3

Hardware and Instrumentation

The key components of the novel intelligent walker (**Fig 3.1**) presented in this thesis are an onboard laptop, an Arduino® micro-controller, a pair of magnetic particle brakes, a pair of optical encoders and a Microsoft Kinect®. This chapter will first describe the general construction of the walker, and then will describe in more detail the composite parts: the Arduino, the particle brakes, the encoders, and the Kinect

Figure 3.1: Front and back views of the novel intelligent assisted-walking device.

3.1 Platform

The walker was constructed by fitting components on a standard rollator (a Dolomite Legacy

600). The walker is made of curved steel tubing. Its front two wheels are hard plastic castor-type wheels with rubber coating for friction. The fixed back wheels were removed and replaced with wheels attached to the particle brakes and encoders. A plastic seat was also removed to make room to mount the onboard computer. The steel handles are fitted with soft foam for user comfort.

Table 3.1: Walker Dimensions	
Handle Height	91 ± 1 cm
Handle Width	48 ± 1 cm
Back Wheel Diameter	204 ± 2 mm
Front Wheel Diameter	199 ± 2 mm
Back Wheel Track Width	68 ± 0.5 cm
Front Wheel Track Width	45 ± 0.5 cm
Total Length	74.5 ± 0.5 cm
Kinect Sensor Height	49.5 ± 0.5 cm
Kinect Sensor Distance From Back Wheels	41 ± 0.5 cm
Walker Mass	17.9 ± 0.1 kg

An Arduino microcontroller, a voltage regulator and a 12 volt lead acid rechargeable battery are affixed to a stable wooden platform which is held in a basket under the laptop platform. The battery is rated for 7.2 Amp-hours, so even at the maximum draw of 0.67A from each brake, and 1.08A from the Kinect, a full battery charge should last nearly three hours. In practice, the brakes will only be used for short intermittent periods so the charge should, speculatively, last longer. The Microsoft Kinect is hung upside down (to take advantage of the stand already attached to its bottom) from a crossbar on the front end of the walker (as shown in **Fig 3.7**).

The weight of the instrumented walker is significantly greater than that of an average rollator. The unmodified Dolomite Legacy 600 has a listed weight of 7.6kg [15]. Most of the additional weight comes from the particle brakes, each of which weigh 1.8kg and the battery, which weighs 2.47 kg. This will likely prevent the user from picking up the walker to move it, meaning it can only be rolled.

3.2 Controller

Input from encoders and output to wheel brakes is routed through an Arduino Uno (**Fig 3.2)**. The Uno is a micro-controller board based around the Atmega328, a micro-controller chip which has a 20MHz 8-bit CPU, 2 Kbytes of RAM, 32 Kbytes of Flash memory and 28 pins [16]. It is capable of

both digital and analog input and output. Analog output is accomplished through pulse width modulation (PWM). It can be powered by AC-to-DC adapter, battery or in this case, through a computer with a USB cable. Here it is used to keep track of the wheel revolution and maintain wheel torque while images from the Kinect are being processed. Image processing is beyond the computational capabilities of the Arduino and is done by an onboard computer (Asus Notebook PC Model TP500L) using the MATLAB computing environment.

The Arduino is given instructions by uploading "sketches", written in the Arduino Development Environment, to the board from a computer via USB. It can then run these independently of said computer. In order to interface with MATLAB, a support package called ArduinoIO is used. This provides MATLAB commands to set Arduino pin modes to input or output and then request input information, or set the output on a pin. ArduinoIO also contains functions for using servo motors and optical encoders. The optical encoder function uses interrupt pins (pins which interrupt the Arduino CPU during its normal processing to perform a different function) to keep track of the encoder count; however, the provided function requires two interrupt pins per encoder, while the Arduino Uno only contains two interrupt pins total and two encoders are required. Therefore, the sketch was modified to include a function to attach an encoder to one interrupt pin and another regular input pin. This halves the resolution of the encoders from 512 to 256 counts per rotation.

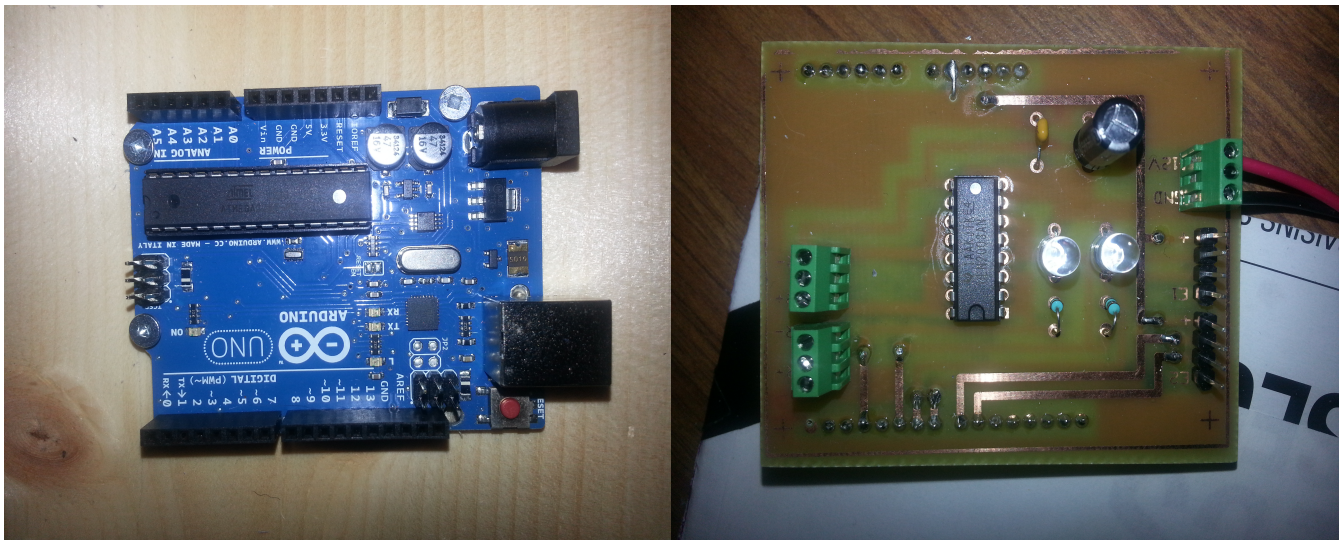


Figure 3.2: The Arduino Uno and the Electronic Shield Used to Control the Encoders and Particle Brakes

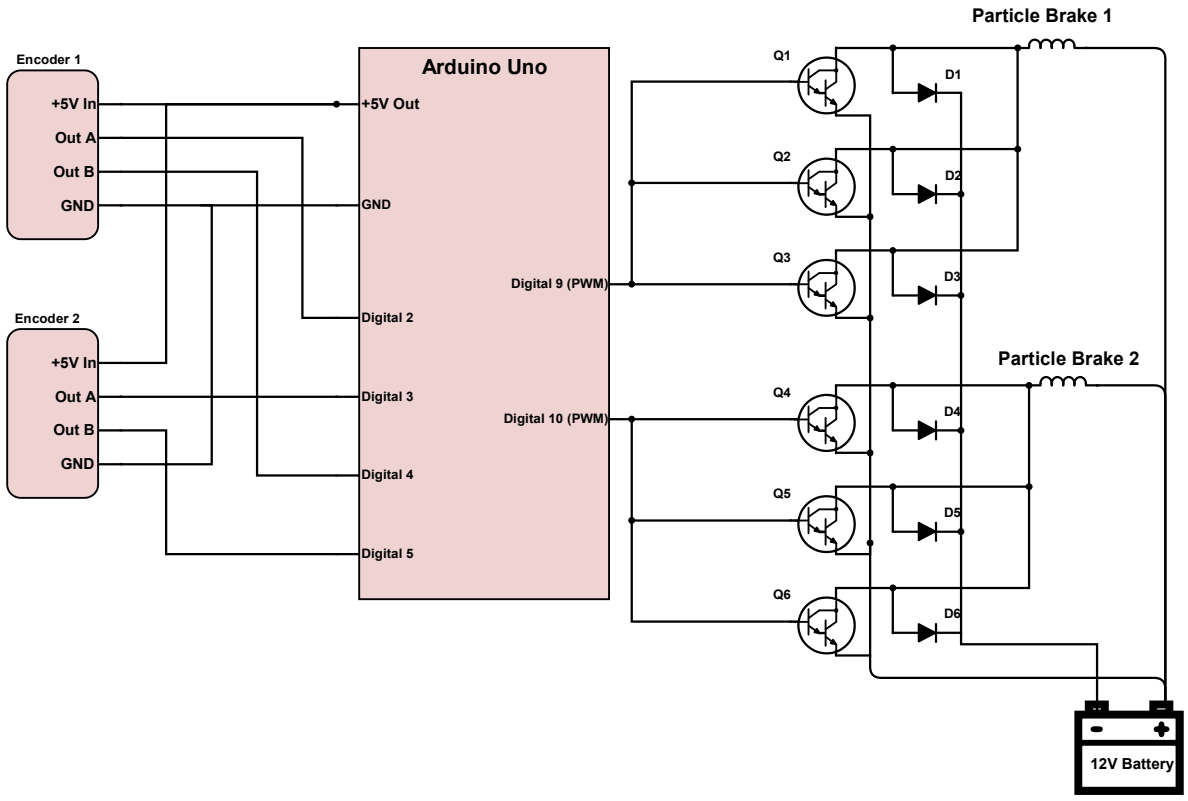


Figure 3.3: Schematic of the walker's circuitry, including the micro-controller, particle brakes and encoders.

3.3 Particle Brakes

Each of the back wheels of the walker was removed and replaced with a custom made wheel attached to a magnetic particle brake (MPB) made by Placid Industries, model number B35-12-H (Shown in **Fig 3.4)**. Current flowing through a coil in the MPB generates a magnetic field. A cavity within the brake holds a powder of magnetic particles. The brake rotor passes through this same cavity. When the magnetic field is generated, the magnetic particles bind, causing friction on the rotor. The friction applied is proportional to the magnetic field applied. MPB's are ideal for this sort of use because the torque applied to the brake can be controlled with a high degree of accuracy, and is near to linear with the input current [17].

The brakes uses a maximum of 12 volt input, with a 17 ohm coil resistance. This produces a 100% input current of 0.67 amps. At this current, the brake torque is 35 lb.-in. (3.95 Nm). At 0% current the brake torque is 0.6 lb.-in. (0.068 Nm). Between these two extremes the torque is nearly linear with the current, but resembles a hysteresis with a slightly lower torque when increasing from 0 amps than when decreasing from 0.67 amps. The brakes can dissipate 30 watts of heat and have a maximum rotation of 1800 RPM [17]. This should be more than sufficient for use on an assisted

walking device (provided walker users do not exceed 65km/h). The brake's response time to change in current is 35 mSec when unforced and 20 mSec when forced. This should also be sufficient, as the control algorithm requires on the order of 100 mSec per timestep.

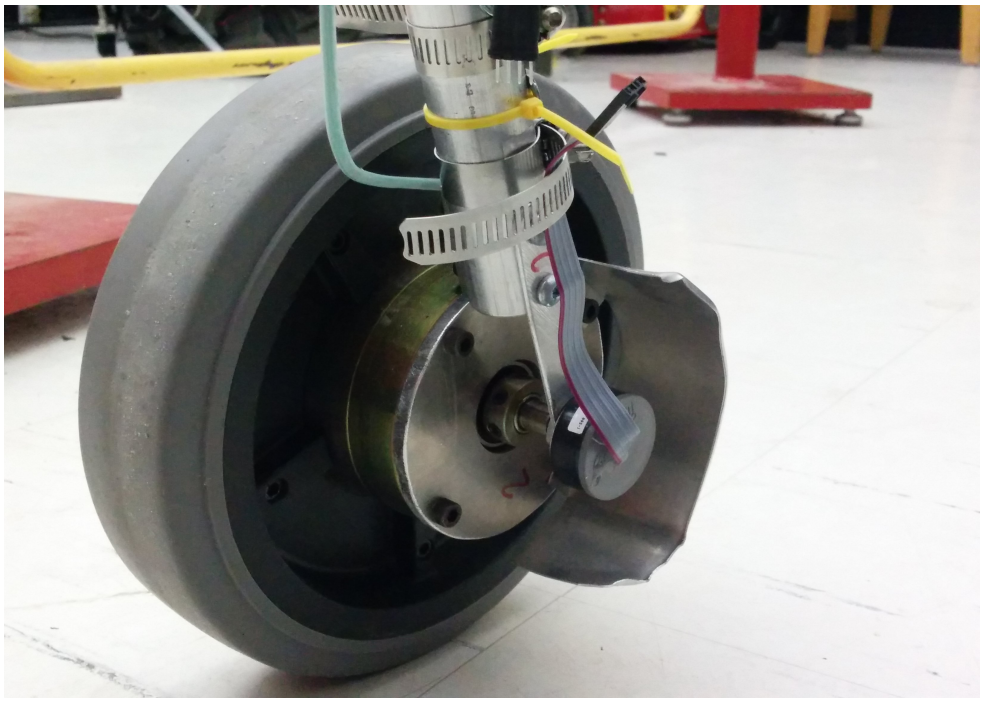


Figure 3.4: A Back Wheel with Encoder, Magnetic Particle Brake and Aluminum Shield.

3.4 Encoders

The walker's back wheels are fitted with a pair of optical rotary encoders (**Fig 3.5**), made by Grayhill Inc. model number B1K128-050. These encoders are incremental, reporting the motion of the encoder shaft, rather than the current position. When rotated, the encoders produce two square waveforms with a phase shift of a quarter cycle, shown in **Fig 3.6**. Upon detection of an edge in either waveform, the encoder's count is incremented (on the Arduino) positively or negatively, depending on whether the waveforms have the same state (both high or low) or different states. The encoders are mounted in line with the axis of the wheels, so they experience 1-to-1 rotation. With an encoder resolution of 256 counts per revolution and wheel diameter of 204 ± 2 mm, this corresponds to a distance resolution of 2.5 ± 0.02mm per count.

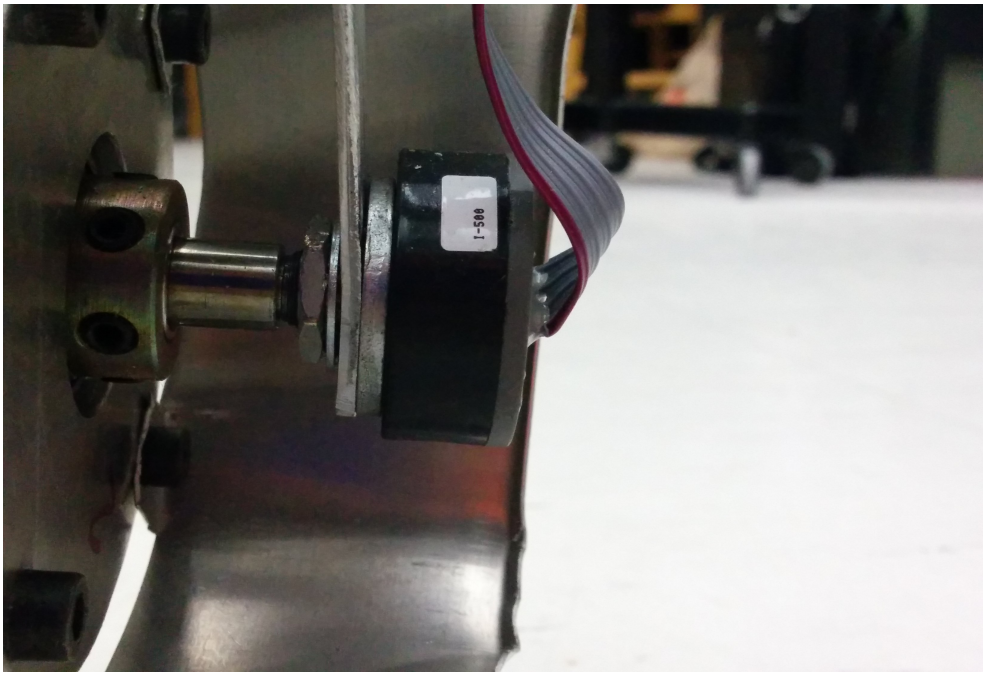


Figure 3.5: An Encoder Mounted to the Axis of a Back Wheel.

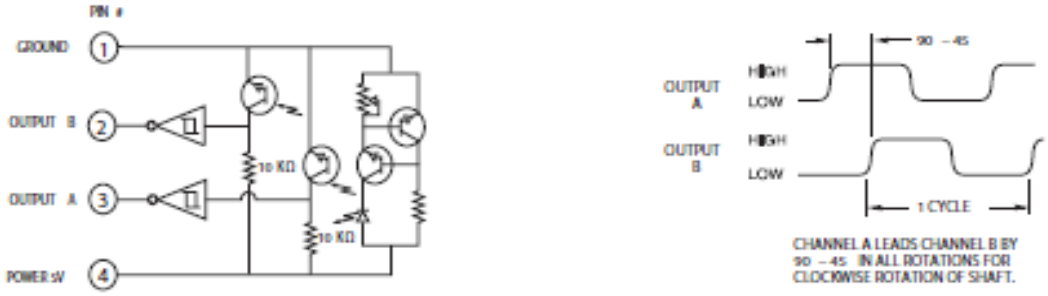


Figure 3.6: The circuitry and waveform of the optical encoder

3.5 Depth Camera

One aspect that differentiates the proposed walker from previous attempts discussed herein is in it's use of a full depth camera to sense the environment. It uses the Microsoft Kinect model 1473, a sensor suite which includes a 3D depth camera, an RGB camera, an accelerometer and a microphone. The Kinect is capable of full-body 3D motion capture, skeleton tracking, and facial recognition. The depth sensor uses an infrared laser projector and a monochrome CMOS sensor, emitting a 640 x 480 grid of laser points to capture depth data in the same resolution. Its angular field of view is 57 degrees horizontal and 43 degrees vertical with results of approximately 1.6 mm per pixel at a distance of one meter [19]. This allows the Kinect to accurately capture a much greater portion of the environment at

once than could a simpler time-of-flight laser rangefinder. The Kinect has a ranging limit of about 3.5 meters which is adequate for the speeds walker users are likely to achieve and the update frequency of the avoidance algorithm. The minimum viewing distance is 0.8m; this causes some technical issues, as will be discussed later. Kinect's coordinate system uses *x* as the horizontal component perpendicular to the direction of view, *y* as the vertical component (with the positive direction point downward with respect to the camera's stand) and *z* as the horizontal component parallel to the direction of view. In most literature, and in common usage, the *z* coordinate is the vertical component; to avoid confusion from this point on the coordinate system used will use *x* as the horizontal component parallel to the viewing angle, *y* as the horizontal component perpendicular to the viewing angle and *z* as the vertical component, with the positive direction pointing upwards.

Figure 3.7: The Microsoft Kinect mounted on the front of the walker.

Chapter 4

Obstacle Mapping and Localization

In this chapter an overview of the software is given followed by a description of the process of mapping obstacles and maintaining walker localization.

Figure 4.1 shows a single timestep in the obstacle mapping and avoidance process. The software initiates with preallocation of variables. An image is then taken by the depth camera and converted from depth image to a three-dimensional point cloud,which is then translated into a two-dimensional point cloud (flattened to the plane of the floor). Next, the software uses data from the encoders to perform localization (i.e. discerning the current position and attitude of the walker). At this step the translational and rotational velocities of the walker are also approximated. The point cloud is then translated and rotated according to this localization and added to a two-dimensional occupancy grid, which is persistent between timesteps. Next this grid is used to perform two control decisions; first, the software decides which direction the walker should turn if an obstacle is present; and second, with what torque the brakes should be engaged, based on the distance to the nearest obstacle and the walker's current position and velocity.

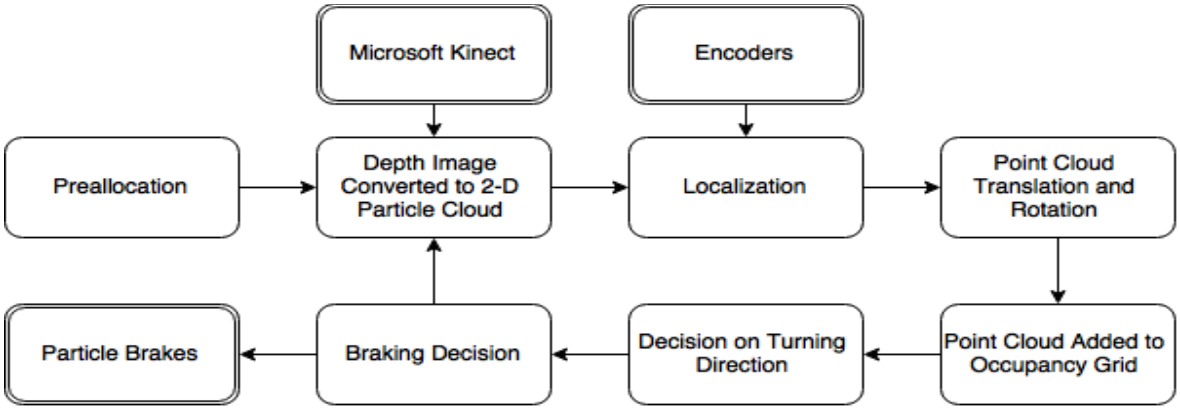


Figure 4.1: The general software structure used to control the walker

4.1 Point Cloud Generation

Depth data, as shown in **Fig 4.2**, from the Kinect is translated into a point cloud, as shown in **Fig 4.3**, using the MATLAB function depthToPointCloud. For computing efficiency every tenth point in this cloud is kept and the rest discarded. This gives an approximate resolution at maximum viewing distance of 5cm per pixel. Points are then removed whose z coordinate is less than -0.4m (the approximate level of the floor), or greater than 2m (points too high to be considered obstacles), the result is shown in **Fig 4.4**. The z coordinate is then discarded in order to produce a 2D scatter map of obstacle points, which are then rotated and translated according to the following equation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_{camera} \\ y_{camera} \end{bmatrix} \tag{4.1}$$

where φ, x_{camera} and y_{camera} are the viewing angle of the walker, and the x and y coordinates of the camera, as determined through dead reckoning using the encoders. This places the points in their correct location with relation to the walker's current position and heading.

Figure 4.2: An RGB image and its corresponding depth image taken by the Kinect

In this way, the walker can take a single snapshot and map a large portion of the environment, even finding obstacles that partially obscured by other obstacles, so long as some part of them is visible. It can also recognize obstacles which may not extend directly to the floor.

In order to produce a continuously updating map of the environment, the points are then added to a 2D histogram of horizontal plane, with a resolution of $5cm^2$. At the same time the section of the 2D plane which is currently being viewed is calculated and an array containing the number of times each pixel has been viewed by the Kinect is updated. This allows the algorithm to determine which obstacles are persistent (having been viewed consistently) and which are moving obstacles (such as a

person moving through the walker's field of view) or artifacts from the camera. Only pixels which have contained obstacle points for more than 50% of the timesteps in which they were viewed are considered to contain persistent obstacles. The result is an updating 2D occupancy grid showing which areas contain obstacles and which are safe for the walker to traverse, shown in **Fig 4.5**.

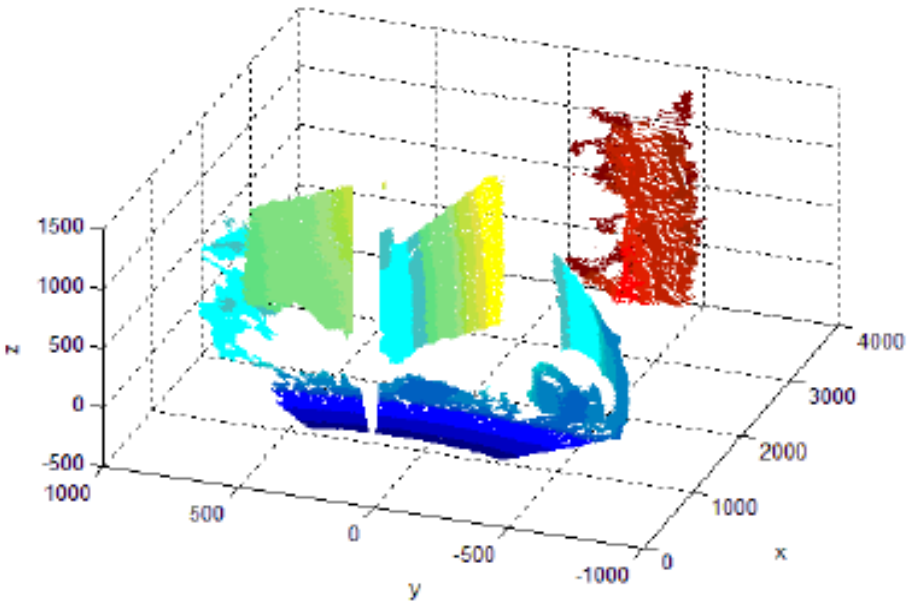


Figure 4.3: A point cloud generated from a Kinect depth image

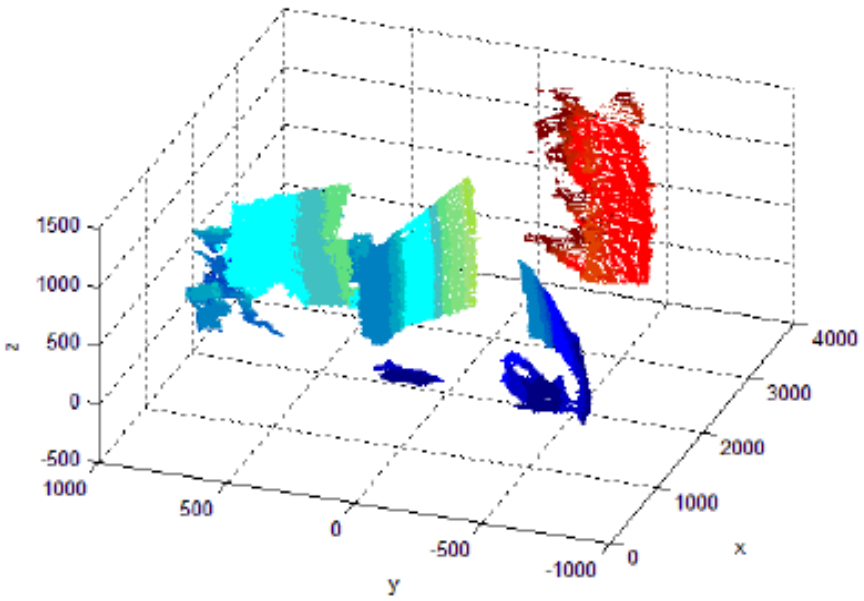


Figure 4.4: The point cloud from figure 4.2, with floor points removed.

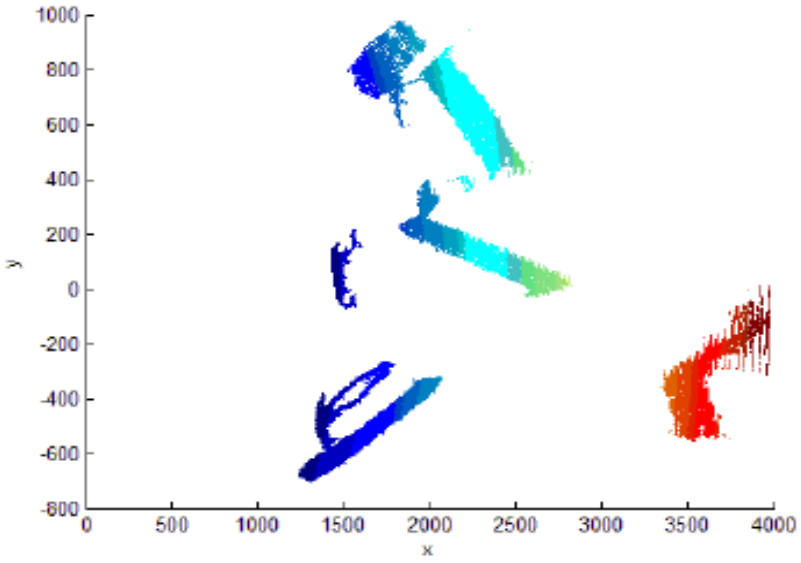


Figure 4.5: The point cloud from figure 4.3, transformed to a 2-D ground coordinate system.

The major drawback of this approach is in the technical limitations of the Kinect. On sharp turns it is possible for obstacles to pass unseen inside the minimum viewing distance of 0.8m. Such obstacles would never be added to the occupancy grid and so the brakes would not engage to prevent a collision. Depth cameras with closer minimum viewing distances could mitigate this issue.

4.2 Localization

A central problem in robots that map environments is keeping track of their own location within the environment as they sense it, known as the SLAM (Simultaneous Localization and Mapping) problem. Two successive measurements are only useful if the software knows how to overlay them, and this requires one to know how much they moved between the measurements.

The walker's tracks its own location through dead reckoning. For this only the wheel encoder readings are required. The Arduino micro-controller keeps a record of the total number of counts so in order to calculate the x and y change between timesteps, each encoder count is subtracted from the previous count. This gives a $\Delta count_{left}$ and $\Delta count_{right}$. The change in x and y is then given as:

$$\Delta x = \frac{(\Delta count_R + \Delta count_L)}{2} \frac{\pi r_{wheel}}{res_{enc}} \cos(\varphi)$$
$$\Delta y = \frac{(\Delta count_R + \Delta count_L)}{2} \frac{\pi r_{wheel}}{res_{enc}} \sin(\varphi) \qquad (4.9)$$

where res_{enc} is the count resolution of the encoders, in this case 256 counts per revolution. The heading,

φ, is given by:

$$\varphi = (count_R - count_L)\frac{\pi r_{wheel}}{r_{track} res_{enc}} \tag{4.10}$$

This is a very fast and simple way of keeping track of location, but has the drawback of not correcting for accumulating error. Because of the problem of accumulating encoder error, the map is cleared and reset after every 100 timesteps. This corresponds to roughly 10 seconds of computation time. Additionally, this ensures that out-of-date obstacle data does not interfere with user navigation.

Two alternate methods of localization, which may have produced more accurate locations using data from the Kinect's camera were attempted; however neither of these were successfully integrated with the rest of the walker's software and so were discarded in favor of dead reckoning. These methods require a significant amount of discussion that does not fit well within the structure of this thesis, so has been presented in Appendix A.

Chapter 5

System Characterization and Control Scheme

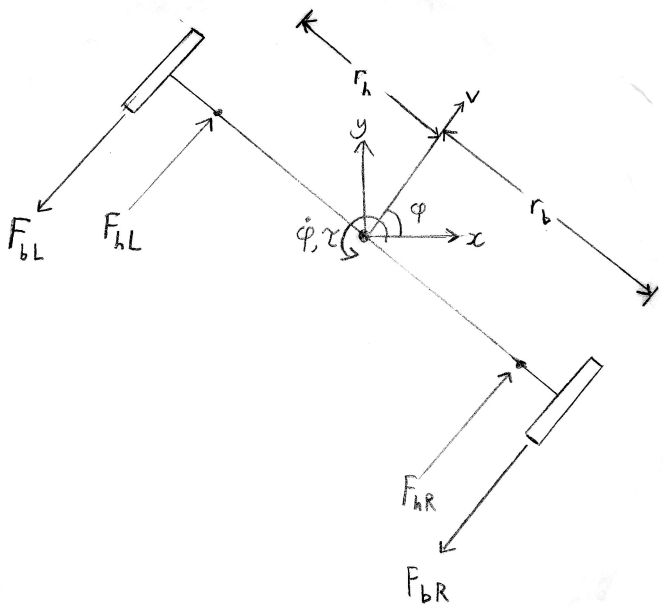


Figure 5.1: A diagram of the walker system and the forces applied to it.

5.1 Equations of Motion

For the purpose of characterizing the system the system state is given by $[x,y,\varphi]$ and its derivatives. The heading angle φ is taken as zero when the walker points in the x direction and increases in the counterclockwise direction.

For simplicity, the assumption is made that the center of mass is along the centerline of the walker and directly between the back wheels. The torque on the walker was then derived as:

$$\begin{aligned}\vec{\tau} &= \vec{r} \times \vec{F} \\ &= (-\vec{r}_h \times \vec{F}_{hL}) - (\vec{r}_b \times \vec{F}_{bL}) + (-\vec{r}_h \times \vec{F}_{hR}) - (\vec{r}_b \times \vec{F}_{bR}) - D_\varphi \dot{\varphi} \\ |\tau| &= r_b \left((\alpha F_{hR} - F_{bR}) - (\alpha F_{hL} - F_{bL}) \right) - D_\varphi \dot{\varphi}\end{aligned} \tag{5.1}$$

where F_{hL} and F_{hR} are the force applied to the left and right handle, r_h is the 2 dimensional radial distance from the center of mass to the handle, F_{bL} and F_{bR} are the force applied by the left and right brake and r_b is the two-dimensional radial distance from the center of mass to the back wheel. α is the ratio r_h/r_b and is introduced to simplify the expression. I is the moment of inertia and D_φ is the coefficient of rotational viscous damping of the system.

The second derivative of the heading is therefore:

$$\begin{aligned}\ddot{\varphi} &= \tau / I \\ &= \left((\alpha F_{hR} - F_{bR}) - (\alpha F_{hL} - F_{bL}) \right) \frac{r_b}{I} - \frac{D_\varphi \dot{\varphi}}{I}\end{aligned} \tag{5.2}$$

The second derivative of x and y were similarly derived from Newton's laws as:

$$\begin{aligned}\ddot{x} &= \left((F_{hL} - F_{bL}) + (F_{hR} - F_{bR}) \right) \frac{\cos(\varphi)}{m} - \frac{D\dot{x}}{m} \\ \ddot{y} &= \left((F_{hL} - F_{bL}) + (F_{hR} - F_{bR}) \right) \frac{\sin(\varphi)}{m} - \frac{D\dot{y}}{m}\end{aligned} \tag{5.3}$$

where D is the translational damping constant and m is the total mass of the system. In addition, the walker cannot have motion perpendicular to the direction of the back wheels. Therefore the controllable degrees of freedom are less than the total degrees of freedom in the walker's two-dimensional system, or nonholonomic, and has the following constraint:

$$\dot{x} \sin(\varphi) - \dot{y} \cos(\varphi) = 0 \tag{5.4}$$

For the purpose of obstacle avoidance, the walker's radius of curvature is of particular interest. This is given by:

$$R = \frac{ds}{d\varphi} = \frac{\partial s}{\partial t} \left(\frac{\partial \varphi}{\partial t} \right)^{-1} \tag{5.5}$$

where s is arc length. The first derivative of s with respect to time is the walker's speed so:

$$\frac{\partial s}{\partial t}=\sqrt{\dot{x}^{2}+\dot{y}^{2}} \tag{5.6}$$

For the purpose of finding an analytic expression for R with respect to the forces on the walker, the equilibrium state can be examined. If the second time derivatives of x, y and φ are set equal to zero an expression for the eventual equilibrium of the first time derivatives of x, y and φ as a function of the forces applied is found. This assumes that the walker has reached such velocities as to produce viscous forces that counterbalance the coulomb friction and applied forces, resulting in a net force of zero:

$$\begin{aligned}\dot{\varphi}&=\left((\alpha F_{hR}-F_{bR})-(\alpha F_{hL}-F_{bL})\right)\frac{r_b}{D_{\varphi}}\\ \dot{x}&=\left((F_{hL}-F_{bL})+(F_{hR}-F_{bR})\right)\frac{\cos(\varphi)}{D}\\ \dot{y}&=\left((F_{hL}-F_{bL})+(F_{hR}-F_{bR})\right)\frac{\sin(\varphi)}{D}\end{aligned} \tag{5.7}$$

If these equations are substituted into the nonholonomic constraint given above it is found that they already satisfy it. Substituting into the formula for the walker speed (5.6) gives:

$$\begin{aligned}\frac{\partial s}{\partial t}&=\sqrt{\left((F_{hL}-F_{bL})+(F_{hR}-F_{bR})\right)^{2}\left(\cos^{2}\varphi+\sin^{2}\varphi\right)\frac{1}{D^{2}}}\\ &=\left((F_{hL}-F_{bL})+(F_{hR}-F_{bR})\right)\frac{1}{D}\end{aligned} \tag{5.8}$$

Therefore the expression for the radius of curvature is:

$$R=\frac{\left((F_{hL}-F_{bL})+(F_{hR}-F_{bR})\right)}{\left((\alpha F_{hR}-F_{bR})-(\alpha F_{hL}-F_{bL})\right)}\frac{D_{\varphi}}{D r_b} \tag{5.9}$$

With data from the encoders the translational and rotational velocity of the walker can be directly measured. The force that the user is applying to either handle cannot be directly controlled. It would be useful to know these forces for the purpose of control. The formula for translational and rotational velocity can be rearranged to find:

$$
\begin{aligned}
(F_{hL}+F_{hR}) &= v*D+(F_{bL}+F_{bR}) \\
(F_{hR}-F_{hL}) &= \frac{\dot{\varphi} D_{\varphi}}{r_b \alpha}+(F_{bR}-F_{bL})/\alpha
\end{aligned}
\tag{5.10}
$$

Which combined give expressions for the handle forces:

$$
\begin{aligned}
F_{hR} &= \frac{1}{2}\left(v*D+(F_{bL}+F_{bR})+\frac{\dot{\varphi} D_{\varphi}}{r_b \alpha}+(F_{bR}-F_{bL})/\alpha \right) \\
F_{hL} &= \frac{1}{2}\left(v*D+(F_{bL}+F_{bR})-\frac{\dot{\varphi} D_{\varphi}}{r_b \alpha}-(F_{bR}-F_{bL})/\alpha \right)
\end{aligned}
\tag{5.11}
$$

All variables on the right side of these equations can be measured in advance or in real time, so the handle forces can be calculated in real time.

Likewise, if it is assumed that the handle forces are known, the braking force necessary to produce the desired movement can be derived.

$$
\begin{aligned}
F_{bR} &= \frac{1}{2}\left(-v*D+(F_{hL}+F_{hR})-\frac{\dot{\varphi} D_{\varphi}}{r_b}+(F_{hR}-F_{hL})\alpha \right) \\
F_{bL} &= \frac{1}{2}\left(-v*D+(F_{hL}+F_{hR})+\frac{\dot{\varphi} D_{\varphi}}{r_b}-(F_{hR}-F_{hL})\alpha \right)
\end{aligned}
\tag{5.12}
$$

5.2 Obstacle Detection and Turning Decision

In order to avoid obstacles the walker checks distance to the nearest obstacle directly in front of it, within a rectangle equal to the width of the walker. The walker then needs to decide in which direction to turn. In order to make this decision, a 90°arc in front of the walker, centered on the walker's heading, is divided into 30 3° subarcs. In each subarc the distance to the nearest obstacle is calculated. These are then used to perform two simple polar coordinate integrations, first in the angle interval [φ – 45°, φ] and then over [φ, φ + 45°]. The magenta circles on the obstacle map in **Fig 5.1** show the points used for this integration. These two integrations are compared to determine which turning direction has the most open space. An arc containing no points can represent one of two cases. Either that section of room has not yet been viewed yet, or the nearest obstacle in that arc is beyond the maximum viewing distance of the Kinect. Unless that section of the room is currently being viewed,

the safest option here is to assume that the first case is true and so that section may contain an obstacle; as a result, any arcs outside the current Kinect view with no points found is assigned an obstacle distance of zero and so does not affect the integration. Arcs that are currently in the Kinect's view, but contain no obstacle points, are set to the Kinect's maximum viewing distance of 3.5m.

5.3 Braking Algorithms

As the walker is maneuvered, the controller monitors it's velocity and trajectory. When an obstacle is in front of the walker, a pair of electromagnetic particle brakes engage in the rear wheels. With low torque braking the user should ideally feel as if the walker is simply guiding them to where they already want to go, rather than taking over their movement entirely. It is important that the user maintain a sense of autonomy in their movements so that they don't get frustrated with the walker moving in unwanted ways. This can be thought of as generating a haptic path for the user, causing motion in one direction to feel easier than motion in another. Two braking algorithms are presented here; both of them are purely reactive, with decisions based on the current state of the system. This approach was chosen because the human element makes it difficult to predict future states based on control. These approaches make no assumption about the user's intended destination or the state of the system in the future.

5.3.1 Open Loop Braking Algorithm

The first algorithm is an open loop proportional gain control, with braking torque directly proportional to obstacle distance with no dynamic (inertial) or friction/damping consideration. At 50cm (d_{min}) the brake is given maximum torque. This distance was chosen because it is the closest the walker can rotate around a single stationary wheel and still avoid collisions. The point at which the torque is set to zero (d_{max}) is more arbitrary. The effect of d_{max} is examined experimentally in chapter six. Thus the voltage sent to one of the brakes, depending on the desired turning direction, is given as:

$$\begin{aligned} V &= 0 && d > d_{max} \\ V &= 12\frac{d_{max}-d}{d_{max}-d_{min}} && d_{min} < d < d_{max} \\ V &= 12 && d < d_{min} \end{aligned} \qquad (5.13)$$

where d is the distance to the nearest obstacle in front of the walker.

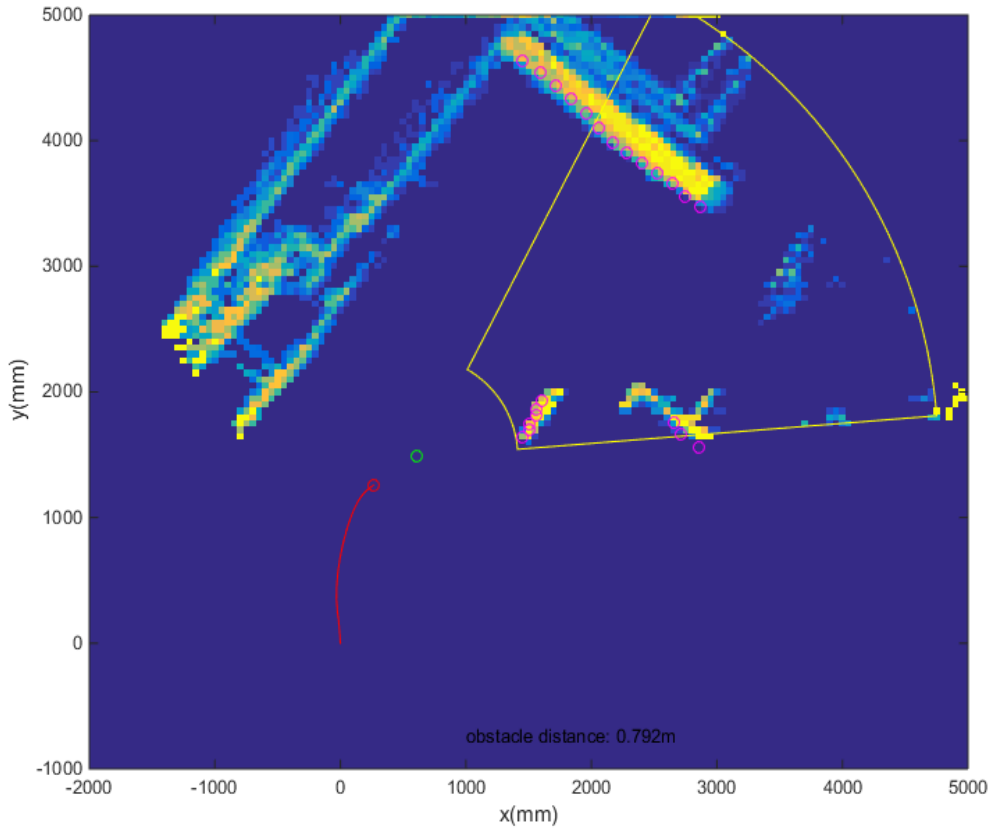


Figure 5.2: An example of the map generated by multiple depth images. The red dot and line represents the point between the back wheels and its path. The green dot shows the position of the Kinect. The yellow outline shows the section of room currently being viewing by the Kinect. The magenta dots represent points used in the algorithm to perform the integration.

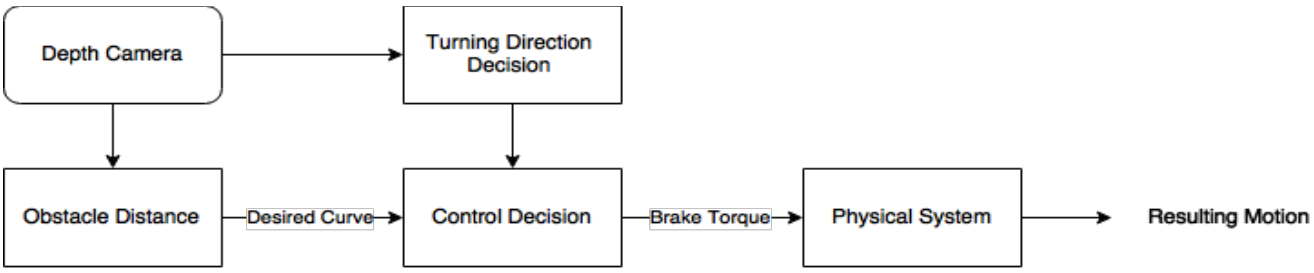


Figure 5.3: The open loop control algorithm for obstacle avoidance

5.3.2 Closed Loop Braking Algorithm

The second algorithm is a closed loop control taking advantage of the system characteristics previously derived. As shown in equation (5.11), the force being applied to either handle can be

estimated as long as the translation and rotational velocities, and braking forces are known. Braking forces are directly controlled here and the velocities are measured using the wheel encoders, the rest of the variables in equation (5.11) (damping constants and radii of the brakes and handles from the centerline) can be found experimentally (this will be detailed in chapter six).

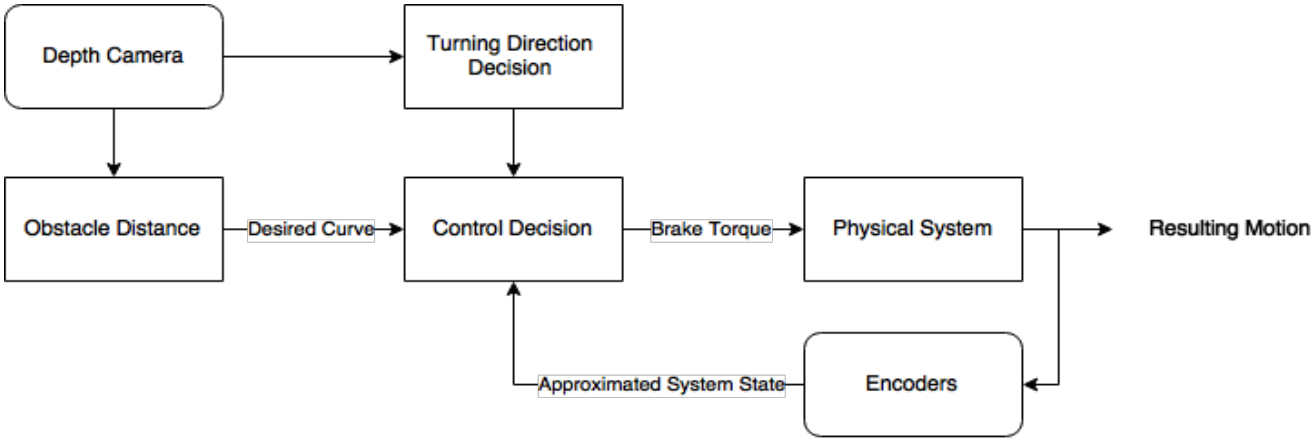


Figure 5.4: The closed loop control algorithm for obstacle avoidance

With the force on the handles estimated the feedback loop then uses equation (5.12):

$$F_{bR}=\frac{1}{2}\left(-v_d*D+(F_{hL}+F_{hR})-\frac{\dot{\varphi}_d D_\varphi}{r_b}+(F_{hR}-F_{hL})\alpha\right)$$
$$F_{bL}=\frac{1}{2}\left(-v_d*D+(F_{hL}+F_{hR})+\frac{\dot{\varphi}_d D_\varphi}{r_b}-(F_{hR}-F_{hL})\alpha\right) \quad (5.14)$$

to calculate the braking force needed to produce the desired translational and rotational velocities. With these equations it is possible to independently manipulate these velocities; however, for the purposes of avoiding obstacles, the equation for turning radius, which is a ratio of these two velocities, can be used. Therefore, for any given turning radius, there is a choice of where to set the desired translational and rotational velocities.

Because of inherent friction in the system, and because brakes cannot produce a force in the same direction as the motion, there is a positive lower bound to the braking force.

If equation (5.9) is manipulated, it shows that for a given desired turning radius the expression for the sum of the braking forces is:

$$F_{bR}+F_{bL}=\frac{R\, r_b D}{D_\varphi}\left(\alpha(F_{hR}-F_{hR})+(F_{bR}-F_{bL})\right)+F_{hL}+F_{hR} \quad (5.15)$$

It is apparent from this equation that in order to minimize the total braking force required to achieve the desired turning radius, the brake force on the side opposite the turning direction should be minimized. With a positive R, this means minimizing the right brake and with a negative R this means minimizing the left brake. Therefore in order to find the braking forces needed to both minimize total braking and achieve the desired R, first the brake on the side opposite the desired turning direction is set to its minimum (i.e. no current flow). Then equation (5.13) is rearranged to find the desired rotational velocity:

$$\begin{aligned}\dot{\varphi} &= -[2F_{bR\,min} - (F_{hL} + F_{hR}) - \alpha(F_{hR} - F_{hL})]\frac{1}{D_\varphi / r_b + RD} \quad \textit{for positive } R \\ \dot{\varphi} &= [2F_{bL\,min} - (F_{hL} + F_{hR}) + \alpha(F_{hR} - F_{hL})]\frac{1}{D_\varphi / r_b - RD} \quad \textit{for negative } R\end{aligned} \tag{5.16}$$

which are then substituted back into equation (5.13) to find the braking force required on the other wheel. If the result is greater than the maximum force the brake can apply then it is set to maximum.

By substituting equation (5.16) into (5.14), a final equation is found for the braking force required in whichever wheel is on the side towards which a turn is to be made, as a weighted sum of the forces from the opposite wheels minimum friction and the forces on the two handles:

$$\begin{aligned}F_{bL} &= \frac{RDr_b - D_\varphi}{RDr_b + D_\varphi} F_{bR\,min} + \frac{D_\varphi - \alpha\, RDr_b}{D_\varphi + RDr_b} F_{hR} + \frac{D_\varphi + \alpha\, RDr_b}{D_\varphi + RDr_b} F_{hL} \quad \textit{for positive } R \\ F_{bR} &= \frac{RDr_b + D_\varphi}{RDr_b - D_\varphi} F_{bL\,min} + \frac{D_\varphi - \alpha\, RDr_b}{D_\varphi - RDr_b} F_{hR} + \frac{D_\varphi + \alpha\, RDr_b}{D_\varphi - RDr_b} F_{hL} \quad \textit{for negative } R\end{aligned} \tag{5.17}$$

This allows the minimum possible braking torque required to produce the necessary forces to prevent collisions with nearby obstacles to be determined at each step.

Chapter 6

Experimental Testing

In this chapter a variety of experiments performed on the walker are described and their results discussed. First the moment of inertia of the walker is measured. Then the viscous and rotational damping coefficients are estimated. The effect of varying the control algorithm's coefficients is examined. Lastly the real world application of obstacle avoidance is tested on an obstacle course.

Early in testing it became apparent that slipping in the wheels was an issue. When the brakes on the walker engaged the wheels could slip along the ground, causing the encoders in that wheel to under-report. This is shown in Figure 6.1.

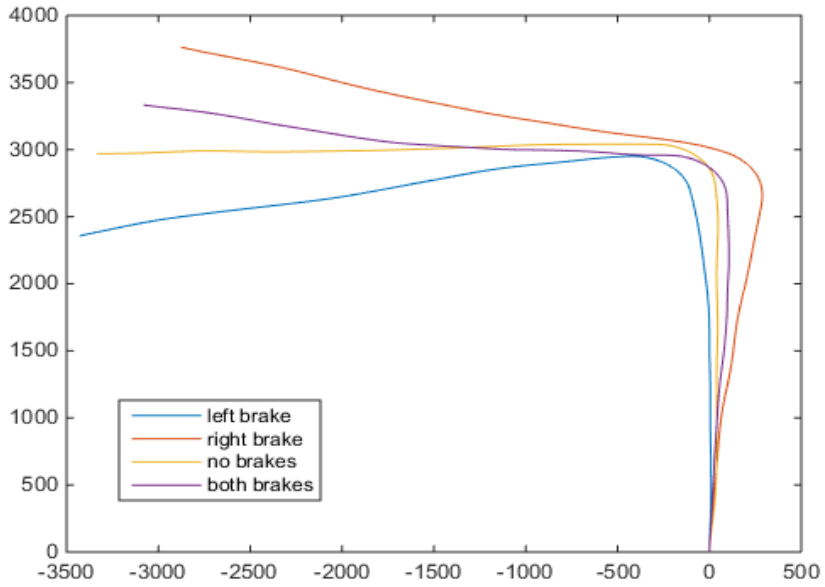


Figure 6.1: Four times the walker is forced along the same path, here shown by the orange line, with one, both or neither brakes engaged at 60% torque.

It is apparent that the path registered by the walker is curved in the direction of the braking wheel, despite the walker taking the same path in each trial. It was found that this error could be reduced by making sure that the floor surface was free of dust and by cleaning the outside of the wheels with alcohol swabs. Figure 6.2 shows the results of taking these measures.

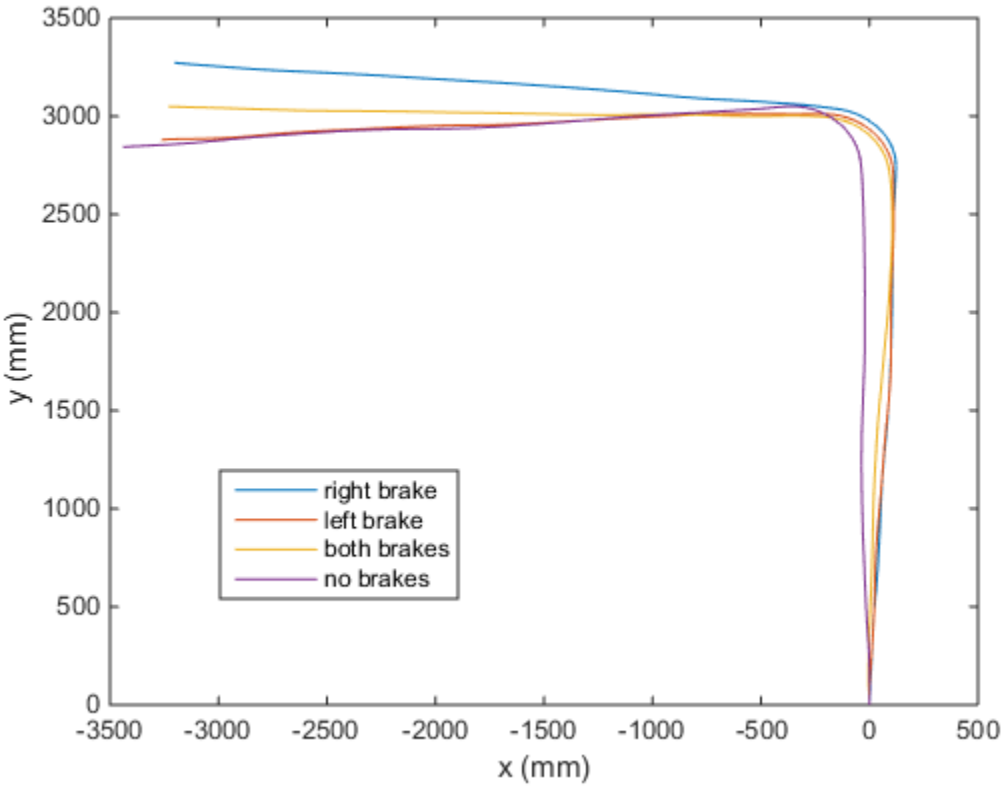


Figure 6.2: The same four trials as shown in Figure 6.4, performed after cleaning the floor and the outside of the wheel.

Subsequently, care was taken to keep the wheels and floor clean during experiments. In future iterations of this sort of intelligent walker this problem could be reduced by using wheels with softer rubber, or treads, and thus higher traction.

6.1 Mass Moment of Inertia

The mass moment of inertia was measured by hanging the walker by two cables to create a bifilar pendulum. Two fifteen pound weights were attached to each handle in order to simulate the downward force of the user. It was then given a small amount of angular displacement and allowed to rotate freely. By measuring the oscillation period the inertial moment can be calculated using the following equation**:**

$$I = \frac{mgT^2b^2}{4\pi^2 L} \tag{6.1}$$

where m is the mass, g is local gravity, T is the period of oscillation, b is the distance between the two cables and L is the total length of the cables [21]. The total mass of the walker, including the weight of the user on the handlebars, was measured at 31.5kg. The cables were 65cm long and spaced 44.5 cm apart.

The walker was given a small amount of angular displacement and then allowed to rotate freely for 20 seconds. The number of full rotations was then counted. This was done three times with the results given below.

Table 6.1: Moment of Inertia Measurements		
Trial	Measured Period	Calculated Mass Moment of Inertia
1	1.8s	7.88 kg m^2
2	2.0s	9.54 kg m^2
3	1.9s	8.65 kg m^2

This gives an average mass moment of inertia of 8.69 kg m^2.

6.2 Measuring the Viscous Damping Coefficients

To realize the control schemes described in chapter five, it is necessary to know the viscous damping coefficients for translational motion (D) and rotational motion (D_φ). In order to measure these, momentum was imparted on the walker and it was allowed to come to a complete stop with no current delivered to the brakes, while measuring the translational and rotational velocities using the onboard encoders. This was done six times with a simple forward push, in order to measure the translational coefficient and six times with a spin imparted, in order to measure the rotational coefficients.

By solving equations (5.2) and (5.3) it can be shown that without force applied to the handles, the resulting motion should have a decaying exponential part and a constant part.

$$\begin{aligned} \dot{\varphi} &= (F_{bLmin} - F_{bRmin})\frac{r_b}{D_\varphi}\left(1 - e^{\frac{-D_\varphi}{I}t}\right) + \dot{\varphi}_0 e^{\frac{-D_\varphi}{I}t} \\ v &= \frac{-(F_{bLmin} + F_{bRmin})}{D}\left(1 - e^{\frac{-D}{m}t}\right) + v_0 e^{\frac{-D}{m}t} \end{aligned} \tag{6.2}$$

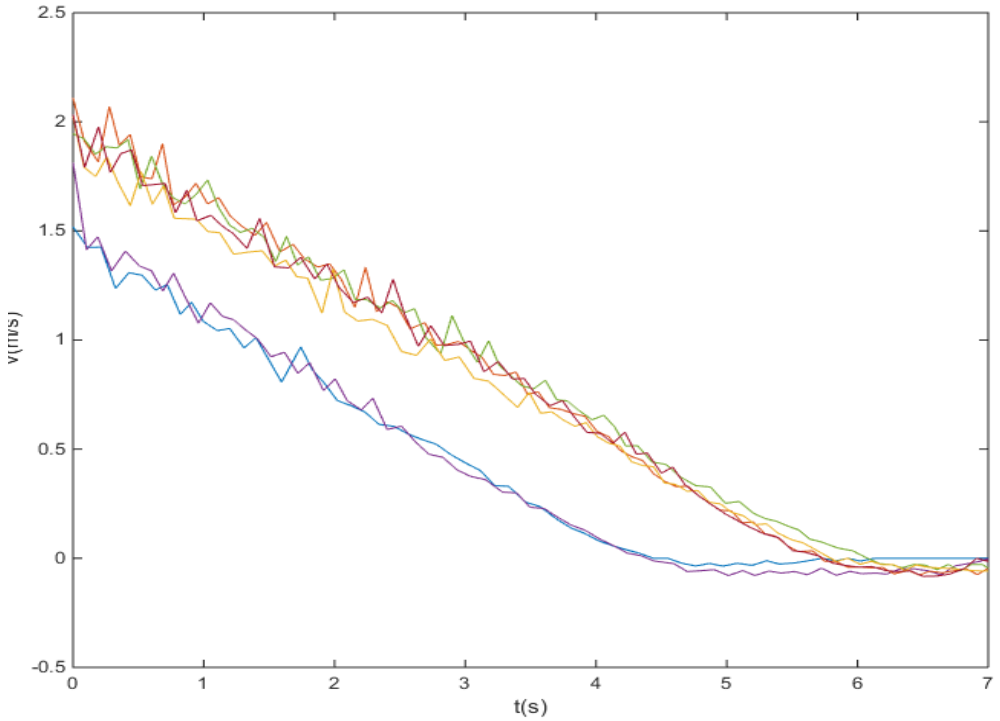


Figure 6.3:Recorded translational velocities over six trials with forward momentum and no spin imparted to walker

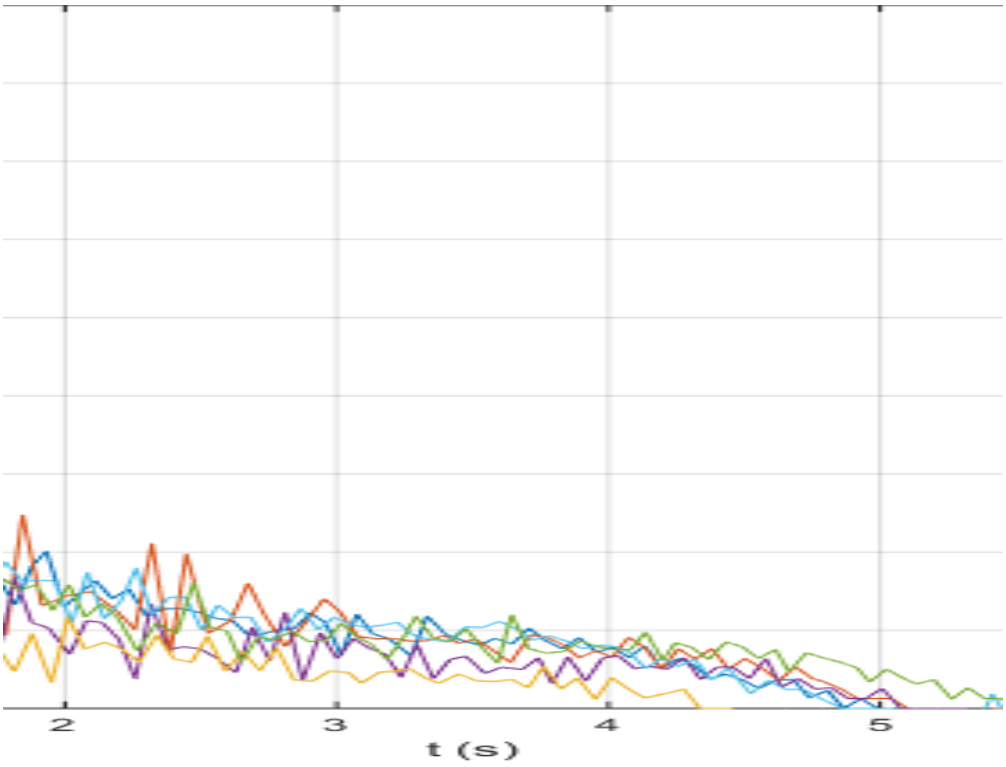


Figure 6.4:Recorded rotational velocities over six trials with spin imparted to the walker

These equations were then fit to each of the trials in order to estimate D and D_φ. The rotational velocity trials (shown in figure 6.4) yielded the values shown in the table below:

Table 6.2: Rotational Viscous Damping Measurements		
Trial	D_φ (Nms/rad)	95% Confidence Interval
1	10.91	[9.85, 11.96]
2	11.78	[10.58,12.99]
3	9.56	[7.88, 11.27]
4	11.15	[9.58, 12.72]
5	12.07	[10.57, 13.57]
6	10.15	[8.75, 11.55]
Mean	10.94	Std Dev: 0.95

The translational trials yielded anomalous results. As shown in Figure 6.3, they appear to decay linearly rather than exponentially. This resulted in the fits failing to converge for all of these trials. This may be explained if the series expansion of equation (6.2) is examined. Expanded around t=0 the series expansion is:

$$v = v_0 - \frac{D}{m} t \left(\frac{\left(F_{bLmin} + F_{bRmin} \right)}{D} + v_0 \right) + \frac{1}{2} \frac{D^2}{m^2} t^2 \left(\frac{\left(F_{bLmin} + F_{bRmin} \right)}{D} + v_0 \right) + Higher\ Order\ Terms \quad (6.3)$$

From this expansion it can be seen that for a small D/m term, the resulting equation would look approximately linear at small t. This may explain the difficulty in fitting to the curve.

Another way to determine viscous damping coefficients was to apply a consistent known force to the walker and record the resulting motion. A person pushing the walker would not fit these requirements as he/she would not be able to apply a consistent force, so a mechanical device was required. This device would need to apply a consistent force in the direction the walker faces, without adding any additional non-holonomic constraints to the system. For example, a large 4-wheeled robot would add additional resistance to turning that is not compensated for in the system equations. The chosen solution was to attach a single driving wheel between the two back wheels of the walker to simulate equal force being applied to each of the handles, while not restricting the walker's ability to turn. This wheel was attached to a 12V motor, mounted on a crossbar attached to the walker's back wheel tubing. It was powered from the onboard 12V battery through a MD10C enhanced 10A DC

motor driver, which received a 0-5V signal from the Arduino in order to control the voltage applied to the motor [22].

In order to best estimate the viscous damping coefficients the other variables inherent in the equations of motion derived in chapter five need to be accurately controlled or measured. The braking torque on each wheel is controlled electronically as described in Chapter 3. Forward force applied to the handles is controlled by measuring the current applied by the driving wheel's motor.

The braking torque was varied from 0 to 3.95 Newton-meters (Nm) in increments of 0.66 Nm. The voltages delivered to the motor were 7.1, 9.5 and 12 volts. For each combination of motor voltage and braking torque, the walker's path, and translational and rotational velocities were measured. The walker was allowed to move freely for at least twenty seconds in order to come to an equilibrium between braking forces (both viscous and coloumb) and applied force. This produced a circular path as anticipated based on equation (5.9).

The current applied to the motor was converted into motor torque based on the specifications of the motor used, which was then converted into the handle forces by multiplying torque by the driving wheel radius (0.105m) and assuming an even split between the two handles. With all relevant forces known and the encoder measurements of the translational and rotational velocities, equation (5.6) was then used to estimate the damping coefficients. These estimates had a very wide range. The rotational damping coefficient (D_φ) was estimated at 8.54±4.24 Nms/rad, while the translational damping coefficient (D) was estimated at 19.28±10.30 Ns/m. It should be noted that the result found here for the rotational damping coefficient is consistent with the results found in fitting. All six curve fitting trials found a rotational damping coefficient which falls within the uncertainty found in the motorized wheel-driven experiment.

6.3 Parameter Testing

With uncertainty in the value of D and D_φ it was determined that it would be useful to examine the effect that these parameters have on the motion of the walker. Additionally, the effect that d_{max} (the maximum distance at which the brakes will engage) has in both the open loop and closed loop control schemes was also of interest. In order to do this, the walker was pushed manually into a 90° corner of a room with the control algorithm active. This algorithm then steered the user around the corner (or failed to, in some cases). The walker's path, as measured by the wheel encoders, was saved for future examination. The velocity in these tests was kept to within 0.9-1.2 m/s for consistency.

The user's downward force on the walker changes its dynamics, as seen in chapter five.

Unfortunately, this walker did not have the instruments necessary to measure this force. In order to keep this force consistent during and between trials, 15lb weights were hung from each of the handlebar. The walker was then pushed forward using a force approximately parallel to the ground, so that the experimenter's weight did not add to the walker's total weight.

6.3.1 Translational Damping Coefficient (D):

Based on the results obtained from the motor-driven testing previously discussed, a range of four values of D between 9 Ns/m and 30 Ns/m were used for testing. For each value the test was run six times. Over all trials D_{φ} was held at 9.75 Nms/rad and d_{max} was held at 1250 mm.

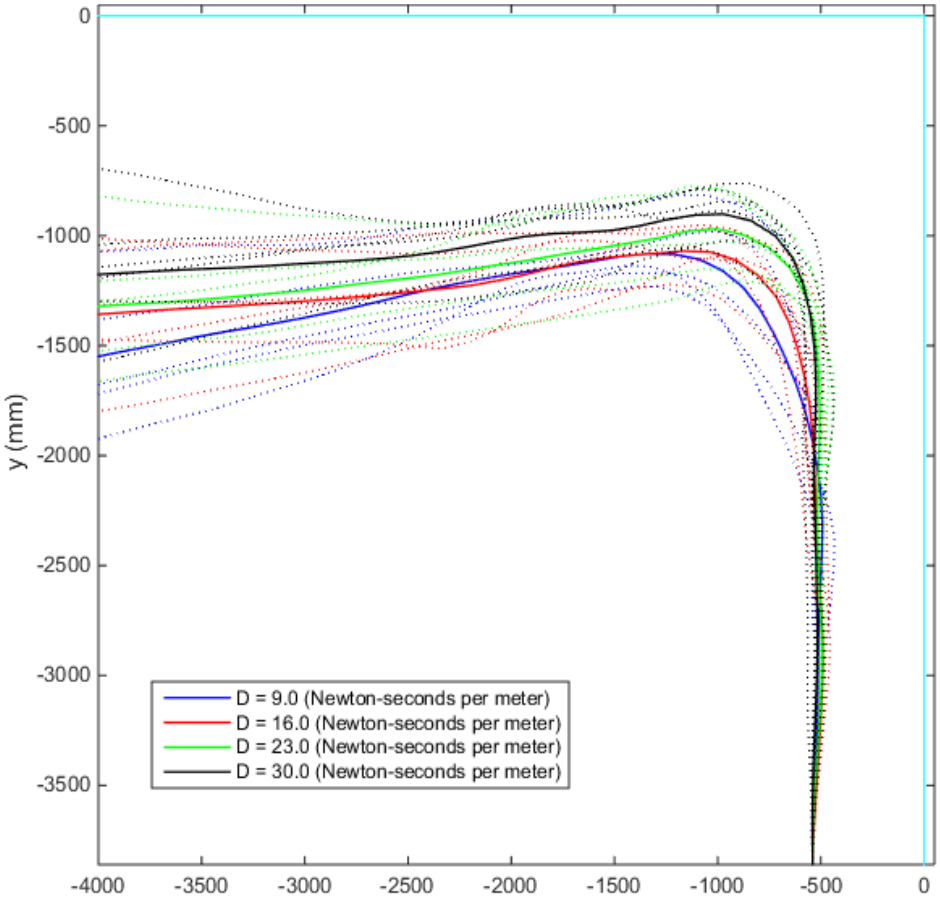


Figure 6.5: Corner trials for testing the effect of different values of D. This graph shows the position of the midpoint between the walker's back wheels. Individual trials are shown as dotted lines, while solid lines represent the average path for each value of D. The cyan lines represent the wall.

Figure 6.5 shows that the value used for D has only a small effect on the average path taken by the walker. The effect seems to be that higher values of D to produce tighter turns, while lower values produce wider turns. If equation (5.9) is examined, it can be seen that this is consistent with what the

system characterization predicts. The walker's radius of curvature is inversely proportional to the value of D, implying that a higher D should produce a tighter turn.

6.3.2 Rotational Damping Coefficient (D_φ):

Based on the results obtained from the motor-driven testing previously discussed, a range of four values of D_φ between 4.25 Nms/rad and 12.5 Nms/rad were used for testing. For each value the test was run six times. Over all trials D was held at 16 Ns/m and d_{max} was held at 1750 mm.

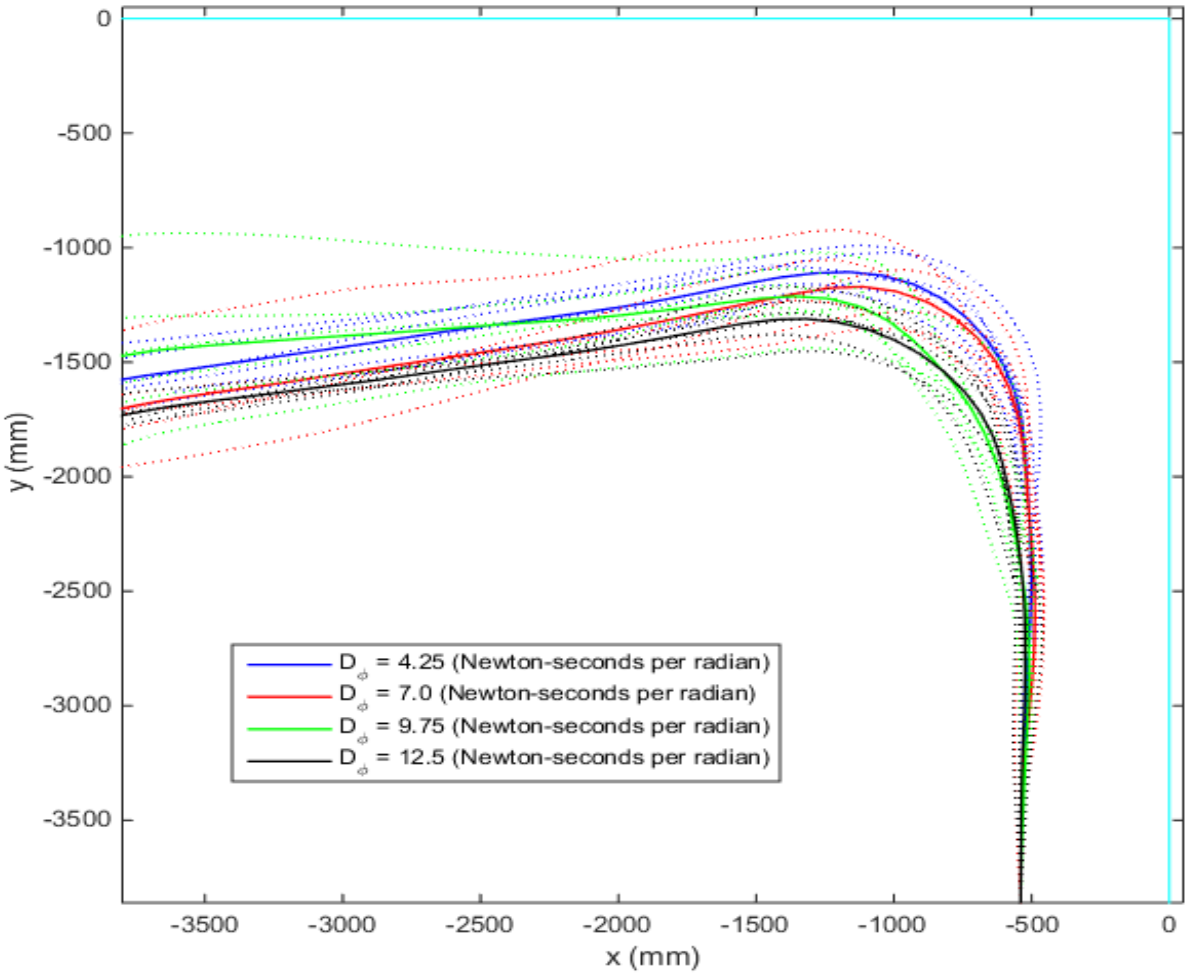


Figure 6.6: Corner trials for testing the effect of different values of D_φ. This graph shows the position of the midpoint between the walker's back wheels. Individual trials are shown as dotted lines, while solid lines represent the average path for each value of D_φ. The cyan lines represent the wall.

Again, figure 6.6 shows only a minor effect of large changes in the value of D_φ. The effect

observed is that higher values of D_φ produce wider turns, while lower values produce tighter turns. This is the inverse of what was observed for D and again, is consistent with what would be expected based on equation (5.9). Radius of curvature is directly proportional to D_φ, so a higher value of D_φ should produce a wider turn.

6.3.3 Maximum Distance (d_{max}) for the Closed Loop Algorithm:

The maximum distance to engage the braking algorithm was varied between 750mm and 2250mm. For each value the test was run six times. Over all trials D_φ was held at 9.75 Nms/rad and D was held at 16 Ns/m.

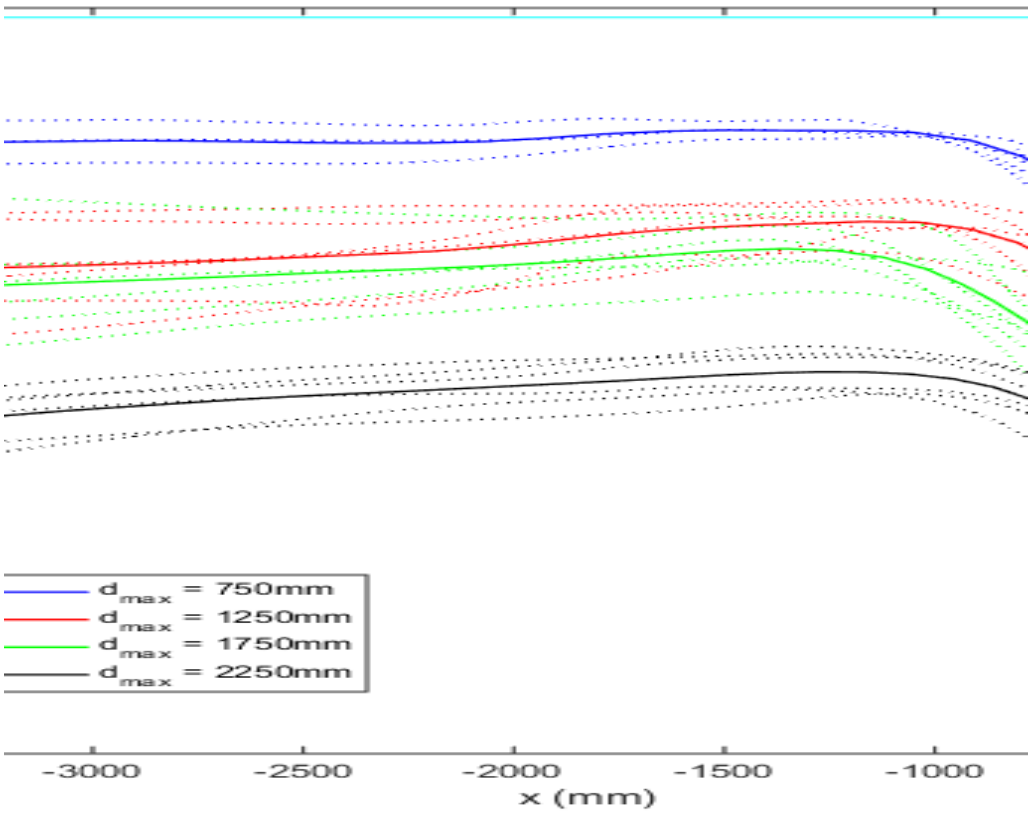


Figure 6.7: Corner trials for testing the effect of different values of d_{max} in the closed loop algorithm. This graph shows the position of the midpoint between the walker's back wheels. Individual trials are shown as dotted lines, while solid lines represent the average path for each value of d_{max}. The cyan lines represent the wall. Dashed lines represent the point where the algorithm begins braking.

In figure 6.7 it is apparent that changing d_{max} has a significant effect on the resulting path. A higher d_{max} results in the walker turning sooner and farther from the far wall. It appears that this

algorithm produces a similar path once engaged, regardless of obstacle distance. It is difficult to say with certainty why this is the case, but it is possible that this may indicate a problem with the simplification of using the equilibrium expression (equation (5.7)) in designing a feedback control system. This expression is only true when all forces (viscous friction, coloumb friction and applied force) are balanced; however this only occurs when the walker has accelerated or decelerated to produce the necessary viscous friction. The control algorithm implemented here assumes that equation (5.7) is true at every timestep, despite altering the braking force at every step. It may be that when the brakes are first applied the roughly tenth of a second between each timestep is not enough to reach this equilibrium again. This would cause higher velocities than predicted by equation (5.7) for the forces present. As a result, the handle force prediction would be artificially high, causing the control algorithm to apply more force to the brakes and thus a tighter turn would be achieved than if the simplification were more accurate.

Regardless of this, a d_{max} can be chosen which produces the desired behaviour. 750mm resulted in some collisions with the wall (as shown by the broken blue dotted lines in Figure 6.5), but 1250mm did not produce these collisions and resulted in a curve fairly close to the wall.

6.3.4 Maximum Distance (d_{max}) for the Open Loop Algorithm:

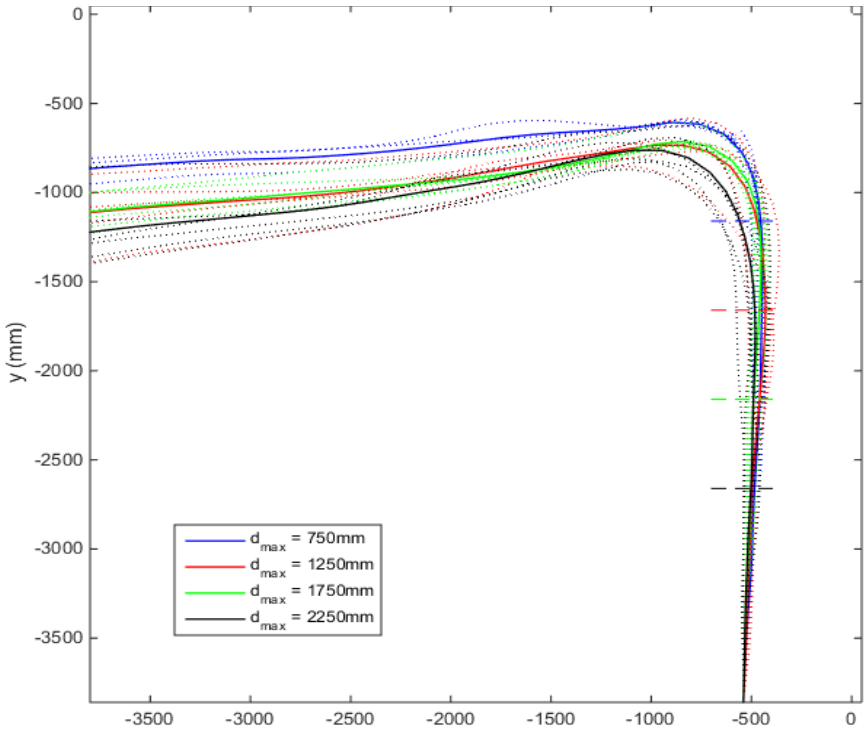


Figure 6.8: Corner trials for testing the effect of different values of d_{max} with the open loop braking algorithm. This graph shows the position of the midpoint between the walker's back wheels. Individual trials are shown as dotted lines, while solid lines represent the average path for each value of d_{max}. The cyan lines represent the wall. Dashed lines represent the point where the algorithm begins braking.

The maximum distance to engage the braking algorithm was again varied between 750mm and 2250mm, this time using the open loop guidance algorithm. For each value the test was run six times.

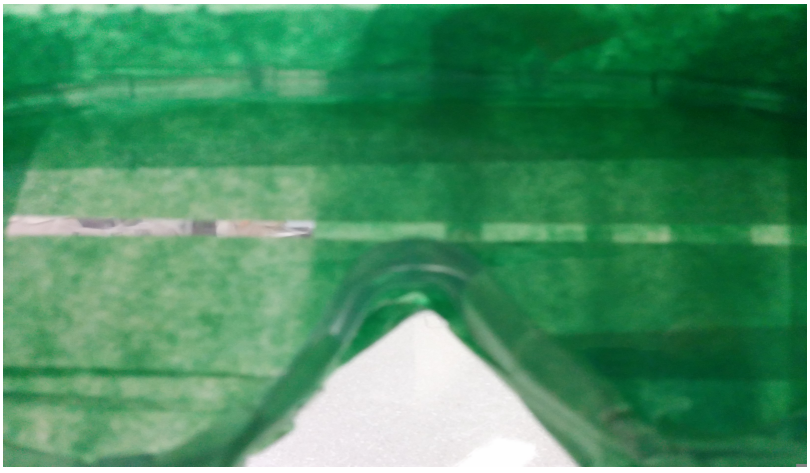
Figure 6.8 shows a much more subdued effect of varying d_{max}. Engaging the brakes farther from the wall cause the walker to turn sooner, but only by a very small amount, even between the highest and lowest d_{max}. This is as expected because, as equation (5.13) shows, with a larger d_{max} the brakes engage farther from the wall, but increase the torque more gradually as the walker approaches it, eventually leading up to the same point where the torque is at its maximum. This results in a roughly similar path, regardless of the value of d_{max}.

6.4 Obstacle Course Testing

In order to evaluate the walker's real world usefulness in collision avoidance an obstacle course (**Fig 6.10**) was constructed in a room roughly 10.5 m by 7 m. The course was constructed primarily of four-legged tables, desks, and office chairs. The course was designed to challenge the user, containing a high number of turns. However, obstacles were spaced so that movement between them was always possible unimpeded, with a path at least 90 cm wide, the width of a typical doorway.

Users were asked to begin at the door of the room and follow the course around the outside of the room. The course then spiraled in towards the center at which point the users were asked to turn around and make their way back to the beginning of the course.

During each trial the users wore a pair of plastic safety goggles, which had been altered with opaque tape, to restrict visual range and depth perception, and layers of clear tape, to blur the non-opaque areas (shown in **Fig 6.9**). These vision impairment goggles are not an accurate simulation of the kinds of vision impairment discussed in chapter one; however they did serve the purpose of artificially increasing the likelihood of collisions, which allowed accurate testing of the walker's effectiveness.



During these tests, users were carefully monitored for collisions with the environment. Collisions were defined as an event in which contact with the environment significantly altered or impeded the movement of the user. By this definition, minor scrapes against obstacles were not counted as collisions. Collisions were divided into three categories: collisions with the front of the walker; collisions with the side of the walker; and collisions in which an obstacle caught on the back wheel of the walker.

Three users were each asked to do ten trials, switching between navigating the course entirely manually, with no obstacle avoidance, navigating with the open loop control algorithm active, and navigating with the closed loop control algorithm active. This resulted in a total of ten trials each for manual navigation and the two algorithms. Before each trial the wheels were cleaned with alcohol swabs in order to improve traction and reduce encoder error.

Figure 6.9: View from inside the vision impairment goggles.

Figure 6.10: The obstacle course used for testing. Users were asked to start at the door of the room (approximately where this photo was taken), and navigate around the outside of the room, eventually spiraling into the center, where they were asked to turn around and then navigate back to the starting point.

In closed loop trials the parameters were set to D = 16 Ns/m, D_{φ} =7 Nms/rad and d_{max} = 1.25m. These parameters were chosen based on the corner turning experiments to turn the walker soon enough to avoid an obstacle in front of it, while not turning sooner or more widely than necessary, which would increase the number of collisions with obstacles to the walker's side.

In open loop trials the parameters were set to d_{min} = 0.50m and d_{max} = 1.25m. Again, these were chosen based on the corner turning experiments to turn the walker before obstacle collisions occurred and sharply enough to reduce the likelihood of the walker colliding with an obstacle to its side.

As with the cornering experiment, 15lb weights were hung from each of the handlebars and the user was asked to push the walker forward without putting downward force on it, in order to keep the

weight and dynamics of the walker consistent during and between all trials. In practice, it is impossible to assure that this was the case, and this may have had a minor effect on the results of the experiment. In future iterations of this walker force sensors in the handles to measure actual vertical forces applied will solve this issue.

The results of the obstacle course experiment are shown in the following table:

Table 6.3: Obstacle Course Experiment Results

Manual Trials			
User	Front Collisions	Side Collisions	Wheel Collisions
1	1	0	1
1	0	0	1
1	1	0	2
2	0	0	2
2	0	0	1
2	1	0	3
3	0	0	2
3	0	0	4
3	1	0	2
3	1	0	2
Total	**5**	**0**	**20**
Algorithm #1 (Open Loop Control) Trials			
User	Front Collisions	Side Collisions	Wheel Collisions
1	0	0	0
1	0	0	1
1	0	0	0
2	0	0	1
2	0	0	1
2	0	0	0
2	0	0	0
3	0	0	4
3	0	1	2
3	0	0	4
Total	**0**	**1**	**13**
Algorithm #2 (Closed Loop Control) Trials			

User	Front Collisions	Side Collisions	Wheel Collisions
1	0	1	3
1	0	0	1
1	0	1	2
1	0	0	1
2	0	0	2
2	0	0	1
2	0	1	2
3	0	1	2
3	0	1	3
3	0	0	5
Total	**0**	**5**	**22**

The results of the experiment show that the open loop control algorithm produced a decrease of 44% in overall collisions, while the closed loop algorithm produced an 8% increase.

When examined individually the different types of collisions illustrate the advantages and current issues of these algorithms. Front collisions occurred five times during the manual control trials, but not at all during trials with braking algorithms. This result was expected, as the front-facing depth camera makes this the easiest type of collision for the walker to avoid. Obstacles to the front are likely to have been mapped unless the walker performed a sharp turn that put obstacles closer than its minimum camera range. This did not seem to occur during the obstacle course trials.

The inverse case seems to be true for collisions with the side of the walker. These are the result of a turn occurring with an obstacle directly beside the walker, close enough that the turn cannot be completed. Side collisions occurred once in the open loop trials and five times in the closed loop trials. They appear to be a result of the walker's algorithms over-steering the walker (i.e. failing to disengage the brakes after a successful turn has been made and there are no more obstacles in front of it.) A possible explanation lies in the slipping problem described earlier. In testing the walker, it was observed that when the brake engages on one side to steer the walker away from an obstacle, slipping in that wheel can sometimes produce an effect where part of the obstacle being avoided appears to be dragged in front of the walker as it turns. This causes the algorithm to think that it has not yet turned enough to avoid the obstacle, which causes it to over-steer. In tight conditions, like those produced by the obstacle course, this can cause the walker to turn into other obstacles, causing side collision. This seems to be more common in the closed loop algorithm than in the open loop one.

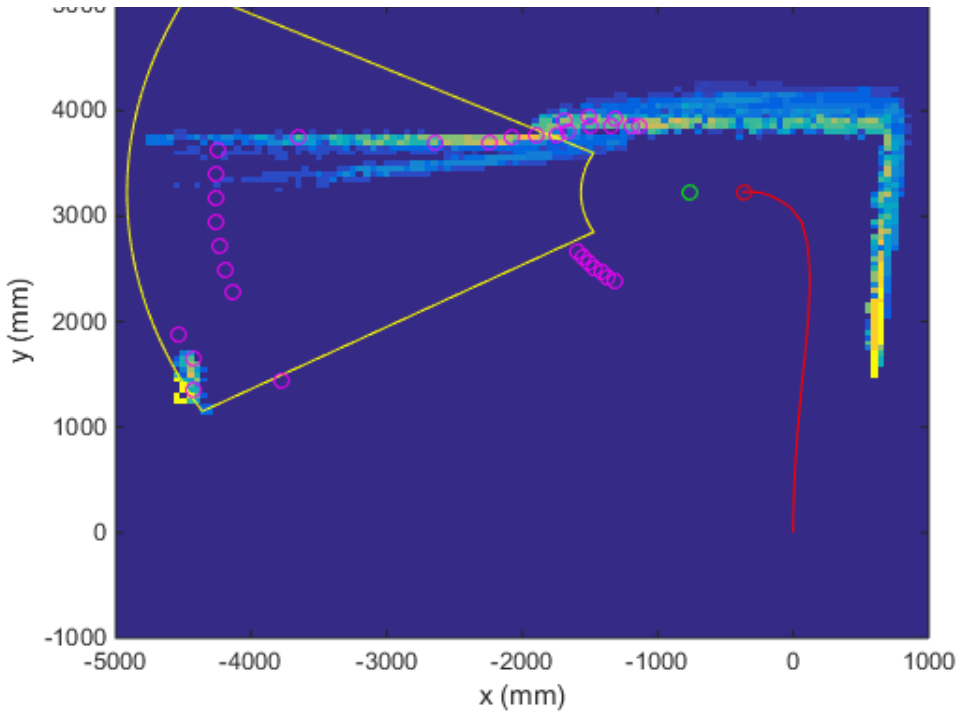


Figure 6.11: An example of encoder error causing an obstacle to appear to be dragged out in front of the walker on the obstacle map. In this example the walker has just made a turn in the corner of a room and is facing parallel to a wall.

Lastly, wheel collisions were by far the most prevalent collision, accounting for 83% of the collisions. This is likely a result of the rear wheels protruding 6.5cm from the walker's frame, due to the placement of the particle brakes. Were the walker redesigned to have the rear wheels directly below their support struts, as was the case with its original design before modifications, these collisions could be expected to drastically decrease in frequency across all trials. Many of these collisions seem to again be a product of the walker over-steering the user, not enough to cause a collision with the side, but enough to place obstacles in a position very close to the side where they cannot be seen by the depth camera, but are in the direct path of the rear wheel. As previously mentioned, this obstacle dragging behaviour is more prevalent in the closed loop algorithm tests, which could explain why rear wheel collisions were more common in the closed loop than the open loop trials (twenty-two collisions compared to thirteen). The closed loop algorithm seems to produce roughly the same number of rear wheel collisions as manual control (in this case a 9% increase was observed in the closed loop algorithm, which is not enough to draw a conclusion), however the open loop algorithm produced 35% fewer.

Chapter 7

Conclusion and Future Work

In this thesis a novel intelligent walker was designed and constructed, and two control algorithms were designed for the purpose of examining whether intelligent braking in an assisted walking device can help individuals who have trouble navigating their environment. The results of this effort show a clear promise to this application.

Parameter tests were performed by pushing the walker into a corner while varying the damping coefficients and maximum distances used by the obstacle avoidance algorithms. The damping coefficient tests showed that the closed loop algorithm is not significantly affected by varying these coefficients. The effect that was observed was consistent with the theoretical model proposed in Chapter 5. The closed loop maximum distance test showed a significant effect of varying the maximum distance for braking. This was not anticipated by the theoretical model, however may be the result of a simplification made in the model's derivation. This parameter should be set to produce desired turning behaviour. The open loop maximum distance test did not show a significant effect of varying the maximum distance parameter. This result was consistent with the open loop algorithm's proportional gain equation.

In obstacle course testing, the open loop control algorithm shows a 44% reduction in the occurrence of significant collisions. This is a considerable decrease. While the results of an obstacle course run by users with artificially simulated vision impairment cannot be directly applied to real world use by users with genuine disabilities, even a much smaller decrease in collisions with obstacles could feasibly prevent hundreds of falls per year [11], and, by helping alleviate anxiety about moving around one's own home, significantly improve the quality of life of their users. The closed loop algorithm did not show the same kind of improvement over unguided walker use.

Closed loop control yielded a slight (8%) increase in significant collisions. This is a null result,

but the placement of these collisions does yield valuable information about how the algorithms performance might be improved in future work. Collisions to the front of the walker were present in the manual control trials, but absent in the open and closed loop guidance trials. Inversely, collisions to the side of the walker were present in the open and closed loop guidance trials, but absent in the manual control trials. Collisions in which the rear wheel caught on an obstacle were prevalent in all three. This implies that the walker's guidance might perform better if its ability to avoid side collisions was improved and if collisions with the rear wheels were also reduced among all cases.

Unfortunately there does not currently exist a good benchmark to compare these results to. The RT-Walker build by Hirata, Hara and Kosuge [14] represents the closest comparable device to the one described here, but their testing was very brief and not controlled and so, cannot be compared to the test results obtained in this thesis. Speculatively, the walker presented here should perform as well as the RT-Walker in the tests involving turning to avoid colliding with walls, however it is not currently equipped to deal with transitioning onto and off of a slope and so would have trouble with this part of the RT-Walker's testing. If the RT-Walker were to be tested on the obstacle course described in this thesis it would likely have trouble detecting the tables used to make the course, as its laser rangefinder would only see the table legs.

The walker described in this thesis has several limitations which could be addressed in future iterations of the device. A reduction of wheel collisions should be simple to achieve; the frame of the walker would need to be redesigned in order to position the walker's struts directly above its rear wheels, while still accommodating the particle brakes. This would eliminate the 6.5cm protrusion that the wheels currently produce and likely prevent most such collisions from being a problem in the future.

The goal of reducing collisions with the side of the walker could also be achieved by altering the construction of the walker. The current system has only a forward-facing depth camera and relies on having recently viewed obstacles to the walker's sides. A more sophisticated system could employ other methods for detecting such obstacles, including two more depth sensing devices on either side (these would require a much smaller minimum depth-ranging distance than the depth camera currently being used in order to see obstacles nearby obstacles as they moves past) or small tactile devices sticking out from either side which would contact obstacles first and alert the walker to their presence. Additionally, exchanging the rear wheels for those with better traction (e.g. a softer rubber with better grip) would reduce error in the encoders and reduce the chances of the algorithms dragging obstacles across their maps as they turned. This in turn would reduce the likelihood of the walker steering more than necessary to avoid an obstacle and hitting others to its sides. Side collisions aside, this would have

the overall benefit of improving the accuracy of the walkers localization and thus improving the accuracy of the obstacle map.

The walker's localization might also be improved with methods of SLAM other than the currently implemented 'dead reckoning' system, including the use of the onboard camera for feedback filtering. Particle filtering and a hybrid particle and extended Kalman filter called FastSLAM were examined, but implementing them was not fully successful. Because of this they did not fit well within the structure of this thesis and are instead discussed in Appendix A.

With some or all of these improvements this assisted walking system will be ready to be tested experimentally among the individuals for whom it was designed, those with visual and cognitive disorders which produce difficulties in navigation. This would be accomplished first with tightly controlled trials in a planned course, similar to the obstacle course experiment presented here. This would then lead to a longer term study in which users with these same disabilities used an intelligent walker in their own home for a designated time period. They would then report back their level of satisfaction with the device and whether or not they experienced any falls during that time period. This would be compared to a control group of users of traditional walkers.

This crucial data would allow further improvements to the walkers design and hopefully would lead to a sophisticated assisted walking system which is ready to improve the mobility of everyday users. This thesis is the preliminary step towards this goal. The results presented here show a clear indication that the approach of passive braking-based obstacle avoidance merits further exploration. With additional research such an approach may prevent falls and improve the quality of life of thousands of people living with visual and cognitive disorders.

Appendix A

Alternate Localization Methods

A central problem in robots which map environments is keeping track of their own location within the environment as they sense it. Two successive measurements are only useful if one knows how to overlay them, and this requires one to know how much they moved between the measurements. The first attempt at solving this problem was by way of an algorithm called a "particle filter". A particle filter is a probabilistic method by which sequential monte carlo trials (random trials) are used to generate a particle cloud (set of discrete points) which is used to estimate the probability density of the unknown variable (in this case the position of the walker). At each successive timestep a new set of particles is generated from the previous set and a given command variable. These particles are then given weights based on some measurement and resampled. The basic particle filter algorithm for a set X_{t-1} of M particles, in psuedocode is:

Algorithm Particle Filter (X_{t-1}, u_t, z_t)

1: $X'_t = X_t = 0$

2: for m=1 to M

3: sample x_t^m from $p(x_t^m \mid u_t, x_{t-1}^m)$

4: $w_t^m = p(z_t \mid x_t^m)$

5: $X'_t = X'_t + < x_t^m, w_t^m >$

6 endfor

7: for m=1 to M do

8: draw i with probability proportional to w_t^i

9: add x_t^m to X_t

10: endfor

11: return X_t

This algorithm [23] first constructs two empty sets of particles, X'_t and X_t, the first of which is a temporary set while the second will become to output of the algorithm. New points are generated based on the points in set X_{t-1} and on the action parameter u_t. These new particles are then assigned an "importance factor", or weight, based on the new measurement z_t and added to X'_t. Finally, these new particles go through "importance sampling" in which they are redrawn from the set X'_t based on their weight. The resultant sampling becomes X_t. The set of new particles now represents the probability density of the property x.

Implementation of the Particle Filter

The above algorithm was implemented in MATLAB with X_t representing a set of 2-D positions within the environment. At each timestep the set X_{t-1} is updated based on the approximate distance and direction the Kinect had moved between timesteps. This is the action parameter u_t. In order to accomplish this points are picked from a normal distribution centered around the point $x_{t-1} + u_t$. The action parameter was meant to come from the wheel encoders, but at this point the encoders had not been added to the walker so measurements were taken by hand for testing.

In order to assign weights, a bivariate histogram of 2-D point cloud generated by the Kinect is made. For each particle this histogram is then translated based on the position and heading specified by the particle and compared to a similar histogram taken in the previous step. A weight is then assigned based on how close the two histograms are to each other. The weight chosen is the inverse of the total difference between the two histograms.

There were many issues with this implementation. The choice to simply give weights based on the inverse of the total difference between the two histograms should be proportional to $p(z_t \mid x_t^m)$, but doesn't necessarily represent it. Therefore the final set of particles doesn't necessarily represent the probability density of the walker's position. More importantly, this process of assigning weights was very computationally intensive. Upon implementation each timestep was found to take on the order of 10 seconds to complete even with fairly low M. After some consideration the simple particle filter was abandoned in favour of a more advanced algorithm called FastSLAM.

FastSLAM

FastSLAM is a powerful simultaneous localization and mapping algorithm, developed by researchers at

Carnegie Mellon University and Stanford University, which makes use of both the extended Kalman filter and particle filter techniques [24]. It operates recursively on a series of measurements containing noise and other inaccuracies, in order to maintain an accurate estimate of the walker's location within it's environment.

The goal of FastSLAM is to approximate the following posterior distribution:

$$p\left(\Theta, s^t \mid z^t, u^t, n^t\right)$$

Where $\Theta = \theta_1 \ldots \theta_N$ represent the position of a series of landmarks in the environment, $s^t = s_1 \ldots s_t$ represents the path of the vehicle (s_t representing the pose of the vehicle at timestep t), $z^t = z_1 \ldots z_t$ is a series of measurements of the range and bearing to nearby landmarks (only one landmark is observed per timestep t), $u^t = u_1 \ldots u_t$ is a sequence of robot command controls and $n^t = n_1 \ldots n_t$ are "data association variables" with n_t denoting which obstacle was observed at timestep t (n_t=k means that at timestep t, θ_k was observed). Thus the distribution gives the probability of a particular set of landmark locations and a specific vehicle path, given all prior observations on those landmarks and all prior robotic commands. In order to calculate this the algorithm requires the probability distribution $p(s_t \mid u_t\ s_{t-1})$ which describes how a control u_t affects the pose of the platform, and the distribution $p(z_t \mid s_t, \Theta, n_t)$ which describes how likely a measurement is, given the state (the position of the vehicle s_t the position of the particles Θ and the identity of the last observed landmark).

FastSLAM makes the important observation that the landmarks pose are independent given the vehicles pose, which means the above distribution can be factored:

$$p\left(\Theta, s^t \mid z^t, u^t, n^t\right) = p\left(s^t \mid z^t, u^t, n^t\right) \prod_k p\left(\theta_k \mid z^t, u^t, n^t, s^t\right)$$

FastSLAM then represents the distribution over trajectories, $p(s^t \mid z^t, u^t, n^t)$, as a particle filter with M particles, and within each particle the distribution over landmarks, $p(\theta_k \mid z^t, u^t, n^t, s^t)$, is represented by N extended Kalman filters (EKF). So at timestep t the m'th particle can be represented as:

$$S_t^{[m]} = s^{t,[m]}, \mu_{1,t}^{[m]}, \Sigma_{1,t}^{[m]}, \ldots, \mu_{N,t}^{[m]}, \Sigma_{N,t}^{[m]}$$

Where $\mu^{[m]}_{n,t}$ and $\Sigma^{[m]}_{n,t}$ are the mean and covariance respectively of the n'th EKF in the m'th particle at time t.

In a extended Kalman filter with N landmarks, each with two Cartesian coordinates, and a robot pose with two Cartesian coordinates and a heading, the mean would require 2N+3 parameters while the covariance matrix would need $(2N+3)^2$ parameters. FastSLAM's observation that the landmarks are independent allows for the covariance matrix to be replaced by N 2x2 covariance matrices, making the total number of parameters N(2+2x2)+3. A Kalman filter therefore requires $O(N^2)$ memory and $O(N^2)$ time to update while FastSLAM requires O(MN) memory and can be updated in O(MlogN) time. This

is a marked improvement when dealing with large numbers of landmarks.

Each update of the FastSLAM begins with a sampling of new poses based on the most recent motion command and observation.

$$s_t^{[m]} \sim p\left(s_t \mid s^{t-1,[m]}, u^t, z^t, n^t\right)$$

To obtain this distribution we find the predicted pose and landmark location:

$$\hat{s}_t^{\,[m]} = (x, y, \varphi) = s_{t-1}^{[m]} + u_t$$

$$\hat{\theta}_{n_t}^{\,[m]} = (u, v) = \mu_{n,t-1}^{[m]}$$

which are then used to find the predicted measurement and its Jacobians:

$$\hat{z}_t^{\,[m]} = g\left(\hat{\theta}_{n_t}^{\,[m]}, \hat{s}_t^{\,[m]}\right) = \begin{bmatrix} \sqrt{(u-x)^2 + (v-y)^2} \\ atan2(v-y.u-x) - \varphi \end{bmatrix}$$

$$G_\theta = \begin{bmatrix} (u-x)/r & (v-y)/r \\ -(v-y)/r^2 & (u-x)/r^2 \end{bmatrix}$$

$$G_s = \begin{bmatrix} (x-u)/r & (y-v)/r & 0 \\ -(y-v)/r^2 & (x-u)/r^2 & -1 \end{bmatrix}$$

Under the EKF approximation the above distribution is Gaussian with mean and variance:

$$\Sigma_{s_t}^{[m]} = \left[G_s^T Q_t^{[m]-1} G_s + P_t^{-1}\right]^{-1}$$

$$\mu_{s_t}^{[m]} = \Sigma_{s_t}^{[m]} G_s^T Q_t^{[m]-1} \left(z_t - \hat{z}_t^{\,[m]}\right) + \hat{s}_t^{\,[m]}$$

where:

$$Q_t^{[m]} = R_t + G_\theta \Sigma_{n_t,t-1}^{[m]} G_\theta^t$$

and R_t and P_t are the error covariances of the measurement and action respectively.

After new poses are sampled, the EKF corresponding to the observed landmark is updated using the standard Kalman gain:

$$K_t^{[m]} = \Sigma_{t-1}^{[m]} G_\theta^T Q_t^{[m]\,-1}$$

$$\mu_{n_t,t}^{[m]} = \mu_{n_t,t-1}^{[m]} + K_t^{[m]} \left(z_t - \hat{z}_t^{\,[m]}\right)$$

$$\Sigma_{n_t,t}^{[m]} = \left(I - K_t^{[m]} G_\theta\right) \Sigma_{n_t,t-1}^{[m]}$$

Lastly the particles must be resampled so that they match the desired posterior. In order to do this we assign each particle an importance factor:

$$w_t^{[m]} \propto p\left(z_t \mid s^{t-1,[m]}, u^t, z^{t-1}, n^t\right)$$

This can be approximated as a Gaussian function

$$w_t^{[m]} = f(z_t) = N\left(\hat{z}_t^{[m]},\ \ G_s P_t G_t^T + G_\theta \Sigma_{n_t,t-1}^{[m]} G_\theta^T + R_t\right)$$

The particles are then resampled with weights assigned based on this importance factor..

Implementation of FastSLAM

The FastSLAM algorithm was implemented in the MATLAB computing environment. Landmark measurement was accomplished by placing visual markers around the environment. These markers, called AprilTags, are simple images on paper, akin to a low resolution QR code. The Kinect's RGB camera images these markers and a simple algorithm can then estimate their distance [25]. AprilTags and their detection algorithm were developed by Professor Ed Olson of the APRIL Robotics Laboratory at the University of Michigan, in the Java programming language. It was later ported to C++ by Jeffrey Boyland and David Touretzky of Carnegie Mellon University. AprilTags are typically 6.5 inches per side with a coded sequence of white blocks which provides the algorithm with a tag id. If the camera is calibrated and the exact size of the tag known, the relative transform between the tag and the camera can be calculated. This allows for physical landmarks which can be easily and quickly associated between timesteps. The measured distances are then used as landmarks by FastSLAM to keep track of where the walker is and prevents error in the encoders from accumulating. The AprilTags are simply placed close to the ground on walls, cupboards etc. every couple of meters around the environment.

Five Examples of April Tags

At each step an the measurement z_t is [r, Φ] where r is the radial distance to the first apriltag viewed and Φ is its azimuthal angle. The action command u_t is [dx,dy,dφ] as read by the wheel encoders. The error on these is estimated at 5%.

April Tag detection was achieved through modifying a C++ port of the original Java code detection program. The C++ file was rewritten as a “MEX build script” which allows MATLAB to compile it (along with all necessary libraries and header files) as a binary MEX-file, callable from MATLAB. In this way a C++ function can be called from MATLAB and therefore can make use of C++ programs

outside of MATLAB such as Open Source Computer Vision (OpenCV). Because of this the complex AprilTag detection algorithm did not need to be rewritten entirely in MATLAB code.

This localization was found to converge well when implemented with m=100 particles. However a setback was experienced while trying to combine this with depth mapping. The AprilTag program was written to use linux libraries, however the Kinect, being a Microsoft product, was developed for Windows and while the onboard rgb camera was recognized by the linux distro Ubuntu, the depth camera was found, after much frustration, to be not compatible. Therefore the options left were to completely rewrite the AprilTag algorithm in MATLAB, or to leave FastSLAM out of the final implemented walker. Eventually due to time constraints it was decided to leave out FastSLAM in favor of a simpler, but more error prone, dead reckoning system, using only the wheel encoders. It would be a viable method of accurately tracking walker position while mapping the environment, if these technical issues could be solved.

Appendix B

Code

This appendix contains the code used by the novel walker for mapping, localization and obstacle avoidance. The following table describes the function of each of the scripts.

Script Name	Function
ObstacleAvoidanceSetup.m	Initial setup, connects MATLAB to the Microsoft Kinect and Arduino. Sets Arduino pins to receive data from encoders.
ObstacleAvoidanceOpen.m	Main code body for open loop obstacle detection and avoidance
ObstacleAvoidanceClosed.m	Main code body for closed loop obstacle detection and avoidance
KinectMapSimple.m	Function called by both open and closed loop main code in order to take a depth snapshot from the Kinect and turn it into an obstacle map
AprilFastSlam.m	Main code for FastSLAM using April Tags as described in Appendix A
FastSlamKalman.m	Called by FastSLAM code every time a Kalman Filter must be evaluated
RouletteSelection.m	Weighted random selection of members of input set

ObstacleAvoidanceSetup.m

```
clear
%connect to Arduino
a = arduino('COM4')

%Kinect Setup
colorVid = videoinput('kinect',1,'RGB_640x480'); %RGB camera
depthDevice = imaq.VideoDevice('kinect',2); %Depth Camera
start(colorVid);
```

```
step(depthDevice);
getsnapshot(colorVid);

%attach encoders
encoderAttach2(a,0,2,4)
encoderAttach2(a,1,3,5)
encoderReset(a,0)
encoderReset(a,1)
```

ObstacleAvoidanceOpen.m

```
%% Reset Encoders and Brakes


encoderReset(a,0)
encoderReset(a,1)
analogWrite(a,9,0)
analogWrite(a,10,0)

%% Constants

maxdist = 1000;%farthest distance to engage brake
mindist = 500;%distance for full brake
count = 2000; %number of timesteps to use
WalkerWidth = 690; %Total Width of Walker in mm
WalkerDepth = 740;  %Depth of Walker in mm
trackWidth = 640; %Distance between contact points of back wheels
wheelDiameter = 203; %Diameter of back wheels
countsPerRevolution = 256; %Number of encoder counts per full wheel/encoder
revolution
distancePerCount = pi * wheelDiameter / countsPerRevolution; %Wheel travel distance
in mm per encoder count
countsPerRotation = (trackWidth / wheelDiameter) * countsPerRevolution;%counts
required to rotate walker
radiansPerCount = pi * (wheelDiameter/trackWidth) / countsPerRevolution;%radians
rotated per encoder count

%% Preallocation
z = zeros(1,count); %z coordinate of point between walker's back wheels
x = zeros(1,count); %x coordinate of point between walker's back wheels
v = zeros(1,count); %forward velocity of walker
distance = zeros(1,count);

%stores previous timesteps encoder count to calculate coordinate deltas
leftCountsOld = 0;
rightCountsOld = 0;

%BinCenters contains coordinates of points for obstacle mapping
BinCenters = cell(1,2);
BinCenters{1} = -5000:50:5000;
BinCenters{2} = -5000:50:5000;
%Point grid for obstacle mapping
[Xgrid, Zgrid] = meshgrid(BinCenters{1}, BinCenters{2});
%Grid containing obstacle map
```

```
OccMap = zeros(length(BinCenters{1}),length(BinCenters{2}));

%nview and nview old contains a grid storing the number of times each point
%in the grid has been viewed
nview = zeros(201,201);
nviewold = zeros(201,201);

%Used to seperate the space in front of the walker into zones and find the
%distance to obstacles in each of those zones
nzones = 32;
zones = cell([nzones,1]);
zoner = zeros(1,nzones);
dist = zeros(1,nzones); %distance to nearest obstacle in each zone

%For graphing purposes to show obstacle distances
zmid = zeros(1,nzones);
xmid = zeros(1,nzones);

%Store time at each step.  Time is determined by the system clock
%not a prechosen timestep size.
t = zeros(1,count);

tic %start system clock

%% Mapping and Obstacle Avoidance Loop
for n=2:count
    %Reset every 100 frames
    if rem(n,20)==0
        %BinCenters contains coordinates of points for obstacle mapping
        BinCenters = cell(1,2);
        BinCenters{1} = -5000+x(n-1):50:5000+x(n-1);
        BinCenters{2} = -5000+z(n-1):50:5000+z(n-1);
        %Point grid for obstacle mapping
        [Xgrid, Zgrid] = meshgrid(BinCenters{1}, BinCenters{2});
        %Grid containing obstacle map
        OccMap = zeros(length(BinCenters{1}),length(BinCenters{2}));

        %nview and nview old contains a grid storing the number of times each point
        %in the grid has been viewed
        nview = zeros(201,201);
        nviewold = zeros(201,201);
    end
    %Get new count on each wheel.
    leftCounts = -encoderRead(a,0);
    rightCounts = encoderRead(a,1);

    t(n) = toc; %current system time

    %x and z coordinates of point cloud
    [Ox,Oz] = KinectMapSimple(depthDevice, colorVid);

    %number of encoder counts of each wheel since last timestep
    deltaLeft = leftCounts - leftCountsOld;
    deltaRight = rightCounts - rightCountsOld;

    %delta displacement of point between right and left wheel.
    deltaDistance = ((deltaLeft + deltaRight) / 2.0) * distancePerCount;
    %walker angle (heading of zero implies pointing in the z direction
    heading = (rightCounts - leftCounts) * radiansPerCount;
```

```
heading = wrapToPi(heading);

%delta x and z displacement of point between back wheels
deltaZ = deltaDistance * cos(heading);
deltaX = deltaDistance * sin(heading);

%current x and z displacement
z(n)=z(n-1)+deltaZ;
x(n)=x(n-1)+deltaX;
%current velocity
v(n) = sign(deltaDistance)*sqrt(deltaZ^2 + deltaX^2)/(1000*(t(n)-t(n-1)));
%x and z position of camera
camerax = x(n) + 410*sin(heading);
cameraz = z(n) + 410*cos(heading);

%store previous encoder counts
leftCountsOld = leftCounts;
rightCountsOld = rightCounts;

%rotate and translate new points based on current camera position
OxTemp = Ox;
OzTemp = Oz;
Ox = cos(heading)*OxTemp+sin(heading)*OzTemp;
Oz = cos(heading)*OzTemp-sin(heading)*OxTemp;
Ox = Ox+camerax;
Oz = Oz+cameraz;

%% View Stuff
%Determine which points have been viewed this timestep
phi = atan2(Xgrid-camerax,Zgrid-cameraz)-heading;
r = sqrt((Xgrid-camerax).^2 + ((Zgrid - cameraz).^2));
phihigh = phi < 0.50;
philow = phi > -0.51;
rhigh = r > 800;
rlow = r < 4150;
viewed = phihigh & philow & rhigh & rlow; %logical array of which
%points have been viewed

%for graphing the currently viewed area
xviewingareatop = [4150*sin(-0.51):50:4150*sin(0.5)];
xviewingareabottom=[800*sin(0.5):-50:800*sin(-0.51)];
zviewingareatop = sqrt(4150.^2-(xviewingareatop).^2);
zviewingareabottom = sqrt(800.^2-(xviewingareabottom).^2);
xviewingarea=[xviewingareabottom,xviewingareatop,xviewingareabottom(1)];
zviewingarea=[zviewingareabottom,zviewingareatop,zviewingareabottom(1)];
xviewingtemp = xviewingarea;
zviewingtemp = zviewingarea;
xviewingarea = cos(heading)*xviewingtemp + sin(heading)*zviewingtemp;
zviewingarea = cos(heading)*zviewingtemp - sin(heading)*xviewingtemp;
xviewingarea = xviewingarea + camerax;
zviewingarea = zviewingarea + cameraz;

%% Turn point cloud into obstacle
OccMap = OccMap.*nview;
nview(viewed) = nview(viewed)+1;
Oxz = [Ox,Oz];
%add new points to obstacle map
OccMap = OccMap + transpose(logical(hist3(Oxz, BinCenters)));
OccMap = (OccMap./nview);
```

```
OccMap(isnan(OccMap)) = 0;
OccMap(OccMap>1) = 1;

%% Distance to Obstacles

%Integration for turning decision
dtheta = pi/(2*nzones);
zoneedges = (-pi/4:dtheta:pi/4);
zonemid = zoneedges(1:nzones) + diff(zoneedges);
for i = 1:nzones
    zoners = r((phi>zoneedges(i)) & (phi < zoneedges(i+1)) & (OccMap > 0.5));
    if numel(zoners>0)
        zoner(i) = min(zoners);
    else
        zoner(i) = 450;
    end
end

%find obstacles in front of walker
slope = tan(pi/2-[heading-0.05, heading+0.05]);
slopeback = -1/tan(pi/2-heading);
xpoints = [-(WalkerWidth/2),(WalkerWidth/2)]*cos(-heading)+camerax;
zpoints = [-(WalkerWidth/2),(WalkerWidth/2)]*sin(-heading)+cameraz;
b = zpoints - xpoints.*slope;
camerab = cameraz - camerax*slopeback;
if cos(heading)>0
   if slope(1) > 0
       if slope(2) > 0
           zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
           (Zgrid > (slope(2)*Xgrid + b(2))) &...
           (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
       else
           if heading > 0
               zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
               (Zgrid > (slope(2)*Xgrid + b(2))) &...
               (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
           else
                zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
                (Zgrid < (slope(2)*Xgrid + b(2))) &...
                (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
           end
       end
   else
        if slope(2)<0
            zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
            (Zgrid < (slope(2)*Xgrid + b(2))) &...
            (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
        else
            zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
            (Zgrid > (slope(2)*Xgrid + b(2))) &...
            (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
        end
   end
else
    if slope(1) > 0
        if slope(2)>0
            zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
            (Zgrid < (slope(2)*Xgrid + b(2))) &...
            (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
```

```
            else
                if heading > 0
                    zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
                    (Zgrid > (slope(2)*Xgrid + b(2))) &...
                    (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
                else
                    zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
                    (Zgrid < (slope(2)*Xgrid + b(2))) &...
                    (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
                end
            end
        else
            if slope(2)<0
                zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
                (Zgrid > (slope(2)*Xgrid + b(2))) &...
                (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
            else
                zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
                (Zgrid < (slope(2)*Xgrid + b(2))) &...
                (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
            end
        end
    end

    %distance to nearest obstacle
    obstacledist = min(min(abs((slopeback)*Xgrid(zonefront)...
        -Zgrid(zonefront)+camerab)/(sqrt((slopeback)^2+1))));
    if numel(obstacledist)==0
        obstacledist = 1500;
    end

    %integrate right
    PolArea(1) = sum((1/2)*(zoner(1:nzones/2).^2)*dtheta);
    PolArea(2) = sum((1/2)*(zoner((nzones/2+1):end).^2)*dtheta);

    %fit1 = polyfit(xdist(zoner<5000),zdist(zoner<5000),1);
    xdist = camerax + zoner.*sin(zonemid+heading);
    zdist = cameraz + zoner.*cos(zonemid+heading);

    %% Braking
    if v(n) >0
    if PolArea(2)>PolArea(1)
        analogWrite(a,9,0)
        if (obstacledist > mindist) && (obstacledist < maxdist)
            analogWrite(a,10,round((-255/(maxdist -
mindist))*obstacledist+(255*maxdist/(maxdist-mindist))))
        elseif (obstacledist < mindist)
            analogWrite(a,10,255)
        else
            analogWrite(a,10,0)
        end
    else
        analogWrite(a,10,0)
        if (obstacledist > mindist) && (obstacledist < maxdist)
            analogWrite(a,9,round((-255/(maxdist -
mindist))*obstacledist+(255*maxdist/(maxdist-mindist))))
        elseif (obstacledist < mindist)
            analogWrite(a,9,255)
        else
```

```
            analogWrite(a,9,0)
        end
    end
    else
        analogWrite(a,9,0)
        analogWrite(a,10,0)
    end

    %% Plots

    imagesc(BinCenters{1},BinCenters{2},OccMap)
    hold on
    plot(x(n),z(n),'ro')
    plot(x(1:n),z(1:n),'r')
    plot(camerax,cameraz,'go')
    plot(xviewingarea,zviewingarea,'y')
    %%
    plot(xdist(zoner<5000),zdist(zoner<5000),'om')
    %%
    xlabel('x')
    ylabel('z')
    %axis([-5000 5000 -5000 5000])
    axis([min(BinCenters{1}) max(BinCenters{1}) min(BinCenters{2})
max(BinCenters{2})])
    set(gca,'Ydir','normal')
    str1 = ['obstacle distance: ', num2str(obstacledist/1000,3), 'm'];
    distance(i) = obstacledist/1000;
    text(min(BinCenters{1})+500,min(BinCenters{2})+500,str1)
    drawnow
    hold off
end

%stop current to brakes
analogWrite(a,9,0)
analogWrite(a,10,0)
%frequency count
freq = count/toc
beep
```

ObstacleAvoidanceClosed.m

```
%% Reset Encoders and Brakes

encoderReset(a,0)
encoderReset(a,1)
analogWrite(a,9,0)
analogWrite(a,10,0)

%% Constants

maxdist = 1250;%farthest distance to engage brake
mindist = 450;%distance for full brake
count = 200; %number of timesteps to use
```

```
WalkerWidth = 690; %Total Width of Walker in mm
WalkerDepth = 740;  %Depth of Walker in mm
trackWidth = 640; %Distance between contact points of back wheels
wheelDiameter = 203; %Diameter of back wheels
countsPerRevolution = 256; %Number of encoder counts per full wheel/encoder
revolution
distancePerCount = pi * wheelDiameter / countsPerRevolution; %Wheel travel distance
in mm per encoder count
countsPerRotation = (trackWidth / wheelDiameter) * countsPerRevolution;%counts
required to rotate walker
radiansPerCount = pi * (wheelDiameter/trackWidth) / countsPerRevolution;%radians
rotated per encoder count
D = 16;
Dphi= 7.0;
alpha = 46/64;
r_b = trackWidth/2000;

%% Preallocation
z = zeros(1,count); %z coordinate of point between walker's back wheels
x = zeros(1,count); %x coordinate of point between walker's back wheels
v = zeros(1,count); %forward velocity of walker
phidot = zeros(1,count); %rotational velocity of walker
distance = zeros(1,count);

%stores previous timesteps encoder count to calculate coordinate deltas
leftCountsOld = 0;
rightCountsOld = 0;

%Starting brake forces
Fblmin = 0.068;
Fbrmin = 0.068;
Fbr = Fbrmin;
Fbl = Fblmin;

%BinCenters contains coordinates of points for obstacle mapping
BinCenters = cell(1,2);
BinCenters{1} = -5000:50:5000;
BinCenters{2} = -5000:50:5000;
%Point grid for obstacle mapping
[Xgrid, Zgrid] = meshgrid(BinCenters{1}, BinCenters{2});
%Grid containing obstacle map
OccMap = zeros(length(BinCenters{1}),length(BinCenters{2}));

%nview and nview old contains a grid storing the number of times each point
%in the grid has been viewed
nview = zeros(201,201);
nviewold = zeros(201,201);

%Used to seperate the space in front of the walker into zones and find the
%distance to obstacles in each of those zones
nzones = 32;
zones = cell([nzones,1]);
zoner = zeros(1,nzones);
dist = zeros(1,nzones); %distance to nearest obstacle in each zone

%For graphing purposes to show obstacle distances
zmid = zeros(1,nzones);
xmid = zeros(1,nzones);
```

```
%Store time at each step.  Time is determined by the system clock
%not a prechosen timestep size.
t = zeros(1,count);

tic %start system clock

%% Mapping and Obstacle Avoidance Loop
for n=2:count
    %Reset every 100 frames
    if rem(n,201)==0
        %BinCenters contains coordinates of points for obstacle mapping
        BinCenters = cell(1,2);
        BinCenters{1} = -5000+x(n-1):50:5000+x(n-1);
        BinCenters{2} = -5000+z(n-1):50:5000+z(n-1);
        %Point grid for obstacle mapping
        [Xgrid, Zgrid] = meshgrid(BinCenters{1}, BinCenters{2});
        %Grid containing obstacle map
        OccMap = zeros(length(BinCenters{1}),length(BinCenters{2}));

        %nview and nview old contains a grid storing the number of times each point
        %in the grid has been viewed
        nview = zeros(201,201);
        nviewold = zeros(201,201);
    end
    %Get new count on each wheel.
    leftCounts = -encoderRead(a,0);
    rightCounts = encoderRead(a,1);

    t(n) = toc; %current system time

    %x and z coordinates of point cloud
    [Ox,Oz] = KinectMapSimple(depthDevice, colorVid);

    %number of encoder counts of each wheel since last timestep
    deltaLeft = leftCounts - leftCountsOld;
    deltaRight = rightCounts - rightCountsOld;

    %delta displacement of point between right and left wheel.
    deltaDistance = ((deltaLeft + deltaRight) / 2.0) * distancePerCount;
    %walker angle (heading of zero implies pointing in the z direction
    heading = (rightCounts - leftCounts) * radiansPerCount;
    heading = wrapToPi(heading);

    %delta x and z displacement of point between back wheels
    deltaZ = deltaDistance * cos(heading);
    deltaX = deltaDistance * sin(heading);

    %current x and z displacement
    z(n)=z(n-1)+deltaZ;
    x(n)=x(n-1)+deltaX;
    %current velocity
    v(n) = sign(deltaDistance)*sqrt(deltaZ^2 + deltaX^2)/(1000*(t(n)-t(n-1)));
    phidot(n) = ((rightCounts-rightCountsOld)-(leftCounts-
leftCountsOld))*radiansPerCount/(t(n)-t(n-1));
    %x and z position of camera
    camerax = x(n) + 410*sin(heading);
    cameraz = z(n) + 410*cos(heading);

    %store previous encoder counts
```

```
leftCountsOld = leftCounts;
rightCountsOld = rightCounts;

%rotate and translate new points based on current camera position
OxTemp = Ox;
OzTemp = Oz;
Ox = cos(heading)*OxTemp+sin(heading)*OzTemp;
Oz = cos(heading)*OzTemp-sin(heading)*OxTemp;
Ox = Ox+camerax;
Oz = Oz+cameraz;

%% View Stuff
%Determine which points have been viewed this timestep
phi = atan2(Xgrid-camerax,Zgrid-cameraz)-heading;
r = sqrt((Xgrid-camerax).^2 + ((Zgrid - cameraz).^2));
phihigh = phi < 0.50;
philow = phi > -0.51;
rhigh = r > 800;
rlow = r < 4150;
viewed = phihigh & philow & rhigh & rlow; %logical array of which
%points have been viewed

%for graphing the currently viewed area
xviewingareatop = [4150*sin(-0.51):50:4150*sin(0.5)];
xviewingareabottom=[800*sin(0.5):-50:800*sin(-0.51)];
zviewingareatop = sqrt(4150.^2-(xviewingareatop).^2);
zviewingareabottom = sqrt(800.^2-(xviewingareabottom).^2);
xviewingarea=[xviewingareabottom,xviewingareatop,xviewingareabottom(1)];
zviewingarea=[zviewingareabottom,zviewingareatop,zviewingareabottom(1)];
xviewingtemp = xviewingarea;
zviewingtemp = zviewingarea;
xviewingarea = cos(heading)*xviewingtemp + sin(heading)*zviewingtemp;
zviewingarea = cos(heading)*zviewingtemp - sin(heading)*xviewingtemp;
xviewingarea = xviewingarea + camerax;
zviewingarea = zviewingarea + cameraz;

%% Turn point cloud into obstacle
OccMap = OccMap.*nview;
nview(viewed) = nview(viewed)+1;
Oxz = [Ox,Oz];
%add new points to obstacle map
OccMap = OccMap + transpose(logical(hist3(Oxz, BinCenters)));
OccMap = (OccMap./nview);
OccMap(isnan(OccMap)) = 0;
OccMap(OccMap>1) = 1;

%% Distance to Obstacles

%Integration for turning decision
dtheta = 2*pi/(3*nzones);
zoneedges = (-pi/3:dtheta:pi/3);
zonemid = zoneedges(1:nzones) + diff(zoneedges);
for i = 1:nzones
    zoners = r((phi>zoneedges(i)) & (phi < zoneedges(i+1)) & (OccMap > 0.5));
    if numel(zoners>0)
        zoner(i) = min(zoners);
    else
        if i>7 & i<27
            zoner(i) = 3500;
```

```
        else
            zoner(i) = 1000;
        end
    end
end

%find obstacles in front of walker
slope = tan(pi/2-[heading-0.05, heading+0.05]);
slopeback = -1/tan(pi/2-heading);
xpoints = [-(WalkerWidth/2),(WalkerWidth/2)]*cos(-heading)+camerax;
zpoints = [-(WalkerWidth/2),(WalkerWidth/2)]*sin(-heading)+cameraz;
b = zpoints - xpoints.*slope;
camerab = cameraz - camerax*slopeback;
if cos(heading)>0
   if slope(1) > 0
       if slope(2) > 0
           zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
           (Zgrid > (slope(2)*Xgrid + b(2))) &...
           (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
       else
           if heading > 0
               zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
               (Zgrid > (slope(2)*Xgrid + b(2))) &...
               (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
           else
                zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
                (Zgrid < (slope(2)*Xgrid + b(2))) &...
                (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
           end
       end
   else
        if slope(2)<0
            zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
            (Zgrid < (slope(2)*Xgrid + b(2))) &...
            (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
        else
            zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
            (Zgrid > (slope(2)*Xgrid + b(2))) &...
            (Zgrid > (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
        end
   end
else
    if slope(1) > 0
        if slope(2)>0
            zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
            (Zgrid < (slope(2)*Xgrid + b(2))) &...
            (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
        else
            if heading > 0
                zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
                (Zgrid > (slope(2)*Xgrid + b(2))) &...
                (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
            else
                zonefront = (Zgrid > (slope(1)*Xgrid + b(1))) &...
                (Zgrid < (slope(2)*Xgrid + b(2))) &...
                (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
            end
        end
    else
```

```
            if slope(2)<0
                zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
                (Zgrid > (slope(2)*Xgrid + b(2))) &...
                (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
            else
                zonefront = (Zgrid < (slope(1)*Xgrid + b(1))) &...
                (Zgrid < (slope(2)*Xgrid + b(2))) &...
                (Zgrid < (slopeback*Xgrid + camerab)) & (OccMap > 0.5);
            end
        end
    end

    %distance to nearest obstacle
    obstacledist = min(min(abs((slopeback)*Xgrid(zonefront)...
        -Zgrid(zonefront)+camerab)/(sqrt((slopeback)^2+1))));
    if numel(obstacledist)==0
        obstacledist = 1500;
    end

    %integrate right
    PolArea(1) = sum((1/2)*(zoner(1:nzones/2).^2)*dtheta);
    PolArea(2) = sum((1/2)*(zoner((nzones/2+1):end).^2)*dtheta);


    %fit1 = polyfit(xdist(zoner<5000),zdist(zoner<5000),1);
    xdist = camerax + zoner.*sin(zonemid+heading);
    zdist = cameraz + zoner.*cos(zonemid+heading);

    %% Braking
    R_d = obstacledist/1000+0.5;
    currentv = mean([v(n), v(n-1)]);
    currentphidot = mean([phidot(n), phidot(n-1)]);
    Fhr = 0.5*(currentv*D + (Fbl + Fbr) + (currentphidot*Dphi)/(alpha*r_b) + (Fbr-
Fbl)/alpha);
    Fhl = 0.5*(currentv*D + (Fbl + Fbr) - (currentphidot*Dphi)/(alpha*r_b) - (Fbr-
Fbl)/alpha);

    if v(n) >0
    if PolArea(2)>PolArea(1)
        Fbl = Fblmin;
        phidotd = (2*Fblmin - (Fhl + Fhr)+alpha*(Fhr-Fhl))*1/((Dphi/r_b)-R_d*D);
        vd = abs(R_d*phidotd);
        analogWrite(a,9,0)
        if (obstacledist > mindist) && (obstacledist < maxdist)
            Fbr = 0.5*(-vd*D + (Fhl + Fhr) - phidotd*Dphi/r_b + alpha*(Fhr - Fhl));
            Torque = Fbr*r_b;
            brakenum = round((Torque-0.068)/0.0152);
            if brakenum > 255
                Fbr = 3.954;
                brakenum = 255;
            end
            if brakenum < 0
                Fbr = Fbrmin;
                brakenum = 0;
            end
            analogWrite(a,10,brakenum)
        elseif (obstacledist < mindist)
            analogWrite(a,10,255)
            Fbr = 3.954;
```

```
            brakenum = 255;
        else
            analogWrite(a,10,0)
            Fbr = Fbrmin;
            brakenum = 0;
        end
    else
        Fbr = Fbrmin;
        phidotd = -(2*Fbrmin - (Fhl + Fhr)-alpha*(Fhr-Fhl))*1/((Dphi/r_b)+R_d*D);
        vd = abs(R_d*phidotd);
        analogWrite(a,10,0)
        if (obstacledist > mindist) && (obstacledist < maxdist)
            Fbl = 0.5*(-vd*D + (Fhl + Fhr) + phidotd*Dphi/r_b - alpha*(Fhr - Fhl));
            Torque = Fbl*r_b;
            brakenum = round((Torque-0.068)/0.0152);
            if brakenum > 255
                brakenum = 255;
                Fbl = 3.954;
            end
            if brakenum < 0
                brakenum = 0;
                Fbl = Fblmin;
            end
            analogWrite(a,9,brakenum)
        elseif (obstacledist < mindist)
            brakenum = 255;
            analogWrite(a,9,255)
            Fbl = 3.954;
        else
            brakenum = 0;
            analogWrite(a,9,0)
            Fbr = Fbrmin;
        end
    end
    else
        brakenum = 0;
        analogWrite(a,9,0)
        analogWrite(a,10,0)
        Fbl = Fblmin;
        Fbr = Fbrmin;
    end

    %% Plots

    imagesc(BinCenters{1},BinCenters{2},OccMap)
    hold on
    plot(x(n),z(n),'ro')
    plot(x(1:n),z(1:n),'r')
    plot(camerax,cameraz,'go')
    plot(xviewingarea,zviewingarea,'y')
    %%
    plot(xdist(zoner<5000),zdist(zoner<5000),'om')
    %%
    xlabel('x')
    ylabel('z')
    axis([min(BinCenters{1}) max(BinCenters{1}) min(BinCenters{2})
max(BinCenters{2})])
    set(gca,'Ydir','normal')
    str1 = ['obstacle distance: ', num2str(obstacledist/1000,3), 'm'];
```

```
    distance(i) = obstacledist/1000;
    text(min(BinCenters{1})+500,min(BinCenters{2})+500,str1)
    str2 = ['brakenum: ', num2str(brakenum)];
    text(min(BinCenters{1})+500,min(BinCenters{2})+900,str2)
    drawnow
    hold off
end
analogWrite(a,9,0)
analogWrite(a,10,0)
freq = count/toc
beep
```

KinectMapSimple.m

```
function[Ox,Oz]= KinectMapSimple(depthDevice, colorVid)
%% Create system objects for the Kinect device
%colorImage = getsnapshot(colorVid);
depthImage = step(depthDevice);
%alignedColorImage = alignColorToDepth(depthImage,colorImage,depthDevice);

%% Create Point Cloud From Image
xyzPoints = depthToPointCloud(depthImage,depthDevice);
[xs, ys, ~] = size(xyzPoints);
xyzPoints = xyzPoints(1:10:xs,1:10:ys,:);
clear colorImage depthImage
xyzPoints(xyzPoints(:,:,2) < -0.4)=NaN;
xyzPoints(xyzPoints(:,:,2) > 1.5)=NaN;
%figure(1)
%showPointCloud(xyzPoints,alignedColorImage,'VerticalAxis','y','VerticalAxisDir','u
p');
%xlabel('X (m)');
%ylabel('Y (m)');
%zlabel('Z (m)');
Ox = reshape(xyzPoints(:,:,1),numel(xyzPoints(:,:,1)),1).*-1000;
Oz = reshape(xyzPoints(:,:,3),numel(xyzPoints(:,:,3)),1).*1000;
end
```

AprilFastSlam.m

```
clear
clc
close all
npart = 100; %number of particles
ntag = 1; %number of tags
nsteps = 10; %number of steps to take
R = [0.01^2,0;0,0.01^2]; %measurement error covariance
P = [0.001^2,0,0;0,0.001^2,0;0,0,0.001^2 ]; %action error covariance
weight = zeros(npart,1);
```

```
%% Initiate Particles

%PathArr = cell(npart,1); %each cell is the path of a particle with x, y,
% and theta components
for i = 1:npart
    PathArr{i} = zeros(3,nsteps);
end

%TagPos = cell(npart,1); %nth cell gives nth particle's estimated x,y mean
%of each of the tags
for i = 1:npart
   TagPos{i} = zeros(2,ntag);
end

%TagVar = cell(npart,1); %nth cell gives nth particle's estimated covariance
%matrix for each tag
 for i = 1:npart
   TagVar{i} = zeros(2,2,ntag);
   TagVar{i}(1,1,:) = 0.01;
   TagVar{i}(2,2,:) = 0.01;
 end

Action = zeros(3,1); %action from the encoders
TagIdent = 0; %array of the tag identity for each timestep

%% Step 1:

% tagdetector returns a 5x8 array containing information on each tag detected
% Each column contains 1) One or zero depending on whether this column
% contains information on a detected tag, 2) x position, 3) y position, 4)
% z position, 5) yaw, 6) pitch, 7) roll

tagarray = tagdetector;
%tagarray =
[1,0,0,0,0,;1,0,0,0,0,;1,0,0,0,0,;2,0,0,0,0,;3,0,0,0,0,;1,0,0,0,0,;1,0,0,0,0,;1,0,0
,0,0,;];
tagsdetected = sum(tagarray(1,1:5)); % # of tags detected
if tagsdetected == 0
    fprintf('err')
    return
end
TagIdent(1) = tagarray(2,1); %identity of 1st tag (only one is used per dt)
r = sqrt(tagarray(3,1)^2+tagarray(4,1)^2); %radial distance of tag
phi = atan2(tagarray(4,1),tagarray(3,1)); %azimuthal angle of tag
g = [r;phi];
z(:,1) = g;
for i = 1:npart
    TagPos{i}(:,TagIdent(1)) = [tagarray(3,1),tagarray(4,1)];
end

%% Further Steps:
for k = 2:nsteps
    pause
    tic
    Action(:,k) = [0.01;0;0];
    %for i = 1:npart
    %    MovErr = [0.02*randn;0.02*randn;0.02*randn];
    %    PathArr{i}(:,k) = PathArr{i}(:,k-1)+Action(:,k)+MovErr;
    %end
```

```
    tagarray = tagdetector;
    %tagarray =
[1,0,0,0,0,;1,0,0,0,0,;0.9,0,0,0,0,;2,0,0,0,0,;3,0,0,0,0,;1,0,0,0,0,;1,0,0,0,0,;1,0
,0,0,0,;];

    tagsdetected = sum(tagarray(1,1:5));
    if tagsdetected == 0
        fprintf('err')
        return
    end

    TagIdent(k) = tagarray(2,1);

    r = sqrt(tagarray(3,1)^2+tagarray(4,1)^2);
    phi = atan2(tagarray(4,1),tagarray(3,1));
    g = [r;phi];
    z(:,k) = g;

    %kalman filter
    for i = 1:npart

        x = PathArr{i}(1,k-1)+Action(1,k);
        y = PathArr{i}(2,k-1)+Action(2,k);
        phi = PathArr{i}(3,k-1)+Action(3,k);
        TagPosHat = [x;y;phi];
        u = TagPos{i}(1,TagIdent(k));
        v = TagPos{i}(2,TagIdent(k));

        zhat = [sqrt((u-x)^2 + (v-y)^2); atan2(v-y,u-x)-phi];
        Gs = [(x-u)/zhat(1), (y-v)/zhat(1), 0 ; (v-y)/(zhat(1)^2), (x-u)/
(zhat(1)^2), -1];
        Gtheta = [(u-x)/zhat(1), (v-y)/zhat(1); (v-y)/(zhat(1)^2), -(u-x)/
(zhat(1)^2)];

        Q = R + Gtheta*TagVar{i}(:,:,TagIdent(k))*transpose(Gtheta);
        samplesigma = inv(transpose(Gs)*inv(Q)*Gs + inv(P));
        samplemu = samplesigma*transpose(Gs)*inv(Q)*(z(:,k)-zhat)+TagPosHat;
        PathArr{i}(:,k) =
transpose(mvnrnd(transpose(samplemu),transpose(samplesigma)));

        x = PathArr{i}(1,k);
        y = PathArr{i}(2,k);
        phi = PathArr{i}(3,k);
        zhat = [sqrt((u-x)^2 + (v-y)^2); atan2(v-y,u-x)-phi];
        Gs = [(x-u)/zhat(1), (y-v)/zhat(1), 0 ; (v-y)/(zhat(1)^2), (x-u)/
(zhat(1)^2), -1];
        Gtheta = [(u-x)/zhat(1), (v-y)/zhat(1); (v-y)/(zhat(1)^2), -(u-x)/
(zhat(1)^2)];

        weightsigma = Gs*P*transpose(Gs) + Gtheta*TagVar{i}
(:,:,TagIdent(k))*transpose(Gtheta) + R;

        [TagPos{i}(:,TagIdent(k)), TagVar{i}
(:,:,TagIdent(k))]=FastSlamKalman(k,z(:,k),PathArr{i},TagIdent(k),TagPos{i},TagVar{
i});

        weight(i) = 1/(sqrt(4*(pi^2)*det(weightsigma)))*exp(-0.5*transpose(z(:,k)-
zhat)*inv(weightsigma)*(z(:,k)-zhat));
```

```
    end
    choice = rouletteselection(weight);
    for i = 1:npart
        PathArrNew{i} = PathArr{choice(i)};
        TagPosNew{i} = TagPos{choice(i)};
        TagVarNew{i} = TagVar{choice(i)};
    end
    PathArr = PathArrNew;
    TagPos = TagPosNew;
    TagVar = TagVarNew;
    toc
end
figure (1)
hold on
for n=1:100
    plot(PathArr{n}(1,:),PathArr{n}(2,:))
end
axis equal
```

FastSlamKalman.m

```
function [ munew, sigmanew ] = FastSlamKalman( step, z, PathArr, TagIdent, TagPos,
TagVar  )
%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here
R = [0.01^2,0;0,0.01^2];
x = PathArr(1,step);
y = PathArr(2,step);
phi = PathArr(3,step);
u = TagPos(1,TagIdent);
v = TagPos(2,TagIdent);
CoVar = TagVar(:,:,TagIdent);
zhat = [sqrt((u-x)^2 + (v-y)^2); atan2(v-y,u-x)-phi];
Gtheta = [(u-x)/zhat(1), (v-y)/zhat(1); (v-y)/(zhat(1)^2), -(u-x)/(zhat(1)^2)];
BigZ = Gtheta*CoVar*transpose(Gtheta) + R;
KalmanGain = CoVar*transpose(Gtheta)*inv(BigZ);
munew = transpose([u,v]) + KalmanGain*(z - zhat);
sigmanew = (eye(2) - KalmanGain*Gtheta)*CoVar;
end
```

RouletteSelection.m

```
function [ choice ] = rouletteselection(fitness)
%Roulette selection based on fitness input.
%choice is a vector of indices for the randomly drawn fitnesses
cp=cumsum(fitness);
choice = zeros(length(cp),1);
for i = 1:length(cp)
```

```
    selection = rand()*cp(end);
    choice(i) = find(cp>=selection,1);
end
end
```

References

1. "Population projections: Canada, the provinces and territories,2013 to 2063." *Statistics Canada*. Retrieved from http://www.statcan.gc.ca/daily-quotidien/140917/dq140917a-eng.htm, 2014-09-17 [2015-04-16]
2. "Canadian Seniors' Use & Views of Assistive Devices for Mobility." *Ipsos.* Retrieved from: http://www.ipsos-na.com/news-polls/pressrelease.aspx?id=4318, 2009-3-23 [2015-04-16]
3. Kirasic, K.C & Bernicki, M.R. "Acquisition of spatial knowledge under conditions of temporospatial discontinuity in young and elderly adults." *Psychological Research,* vol. 52, pp. 76-79, Feb 1990.
4. Liu, L et. al. "Spatial disorientation in persons with early senile dementia of the Alzheimer type." *American Journal of Occupational Therapy*, vol. 45, pp. 67-74, Jan 1991.
5. Loh, KY & Ogle, J. "Age Related Visual Impairment in the Elderly." *Medical Journal of Malaysia,* vol. 59, pp. 562-568, Oct 2004.
6. Brandt, A et. al. "Satisfaction with rollators among community living users: a follow-up study." *Disability and Rehabilitation,* vol. 25, pp. 343-353, 2003.
7. Tung, J et. al. "Combining ambulatory and laboratory assessment of rollator use for balance and mobility in neurological rehabilitation in-patients." *Disability and Rehabilitation: Assistive Technology,* pp. 1-8, April 2014.
8. Stevens et. al. "Unintentional Fall Injuries Associated with Walkers and Canes in Older Adults Treated in U.S. Emergency Departments." *Journal of the American Geriatrics Society,* vol. 57, issue 8, June 2009.
9. Laila Jonsson. "The Importance of the 4-Wheeled Walker for Elderly Women Living in their Home Environment – a three year study" *The Swedish Handicap Institute*
10. Johannson and Jarnlo. "Balance training in 70-year old women." *Physiotherapy Theory and Practice,* issue 7 pp. 121-125, 1991.
11. Sabatini et. al. "A Mobility Aid for the Support of Walking and Object Transportation of People with Motor Impairment". *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst,* pp. 1349-1354, 2002.
12. M. A. Peshkin, J. E. et. al. "Cobot architecture." *IEEE Trans. Robot. Autom*., vol. 17, no. 4, pp. 377–390, Aug 2001.
13. Rentschler et. al. "Intelligent walkers for the elderly: Performance and safety testing of VA-PAMAID robotic walker" *Journal of Rehabilitation, Research and Development,* vol. 40, no. 4, pp. 423-432. Oct 2003.
14. Hirata, Hara and Kosuge "Motion Control of Passive Intelligent Walker Using Servo Brakes" *IEEE Transactions on Robotics,* vol. 23, no. 5, pp. Oct 2007.
15. "Invacare® Canada Dolomite® Rollators" Retrieved from http://www.invacare.ca/doc_files/Dolomite%20Rollators_05-430C.pdf, 2010-04-11 [2015-04-16]

16. "Arduino UNO & Genuino UNO" *Arduino.* Retrieved from https://www.arduino.cc/en/Main/ArduinoBoardUno, n.d. [2015-04-16]
17. "Magnetic Particle BRAKE B35 Data Sheet" *Placid Industries.* Retrieved from http://www.placidindustries.com/spec.b35.pdf, n.d. [2015-04-16]
18. "Grayhill Optical Encoders Series 61K" *Grayhill Inc.* Retrieved from: http://lgrws01.grayhill.com/web1/images/ProductImages/Series%2061k%20Encoder.pdf, n.d. [2015-04-16]
19. "Kinect for Windows Sensor Components and Specifications" *Microsoft*. Retrieved from https://msdn.microsoft.com/en-us/library/jj131033.aspx, 2015 [2015-04-16]
20. John Mandel. "Use of the Singular Value Decomposition in Regular Analysis" *The American Statistician,* vol. 36, no. 1, pp. 15-24, Feb. 1982.
21. R. A. Mattey. "Bifilar Pendulum Technique for Determining Mass Properties of Discos Packages." *Natonal Technical Information Service,* Access Num. AD-787 506/5, 1974
22. "MD10C Enhanced 10Amp DC Motor Driver." *Cytron Technologies* Retrieved from: http://www.robotshop.com/media/files/PDF/user-manual-md10c-v2.pdf, Feb 2013. [2015-04-16]
23. S. Thrun, W. Burgard and D. Fox. "Probabilistic Robotics." *The MIT Press*, pp. 78, Aug. 2005
24. M. Montelmerlo et. al. "FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping That Provably Converges." *Proceedings of the Sixteenth IJCAI*, pp. 1151-1156, 2013
25. E. Olson. "AprilTag: A Robust and Flexible Visual Fiducial System." *Proceedings of the IEEE International Conference on Robotics and Automation,* pp. 3400-3407, May 2011