2011

Developing Cost-Effective Robot Navigation

Timothy J. Perkins
Lehigh University

Developing Cost-Effective Robot Navigation

by

Timothy J. Perkins

A Thesis
Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for Degree of
Master of Science

in

Computer Engineering

Lehigh University

May 2011

Contents

List of Figures

1 Abstract

This thesis discusses the partial development of a smart wheelchair robot. The purpose of the smart wheelchair is to provide transportation to wheelchair bound persons by use of an autonomous system. The user must only issue high level commands such as "take me to the bookstore," and the robot will adapt its planning and intelligence accordingly. The main function of the smart wheelchair is transportation, so it follows that the most important aspect of the wheelchair is its ability to navigate. Robot navigation is a difficult but well studied problem. However, the smart wheelchair is unique in that it has strict cost requirements. This constraint limits the effectiveness of the techniques currently used in the field.

The smart wheelchair project is still under development at the VADER Laboratory. The author's contributions to the project during his course of study are discussed in this thesis. Contributions include the initial work required to make the robot platform functional, the basic navigation software and improvements made to it, a low cost self-contained pose system, and terrain classification. Terrain classification is prerequisite for the future work of dynamic drive control profile selection for improved navigation. This thesis gives perspective to the problem of robot navigation for low cost platforms and small development teams.

2 Introduction and Contributions

The goal of the smart wheelchair project is to provide cost-effective autonomous transportation as required by wheelchair bound persons in every-day life situations. The project is ultimately intended to be a consumer product. Therefore, the wheelchair must be able to navigate in practically every environment, including indoors and outdoors. The wheelchair's navigation must be reliable, robust, and accurate. However, the wheelchair must not be prohibitively expensive to the target user, so the wheelchair must use low cost sensors and actuators. Essentially, this makes a difficult problem even more difficult.

Outdoor navigation is a notoriously difficult problem for robots. The successes and failures of the DARPA challenges demonstrate the complexity of the problem [5]. Robots limited to indoor environments have a safer and more predictable existence; they can take advantage of smooth floors and perpendicular walls. On the other hand, anything can go wrong for an outdoor robot. For example, moving along the wrong path could cause the robot to overturn. In general, outdoor robots have a very small margin of safety. For this reason, and because the primary function of the wheelchair is transportation, a special emphasis has been placed on the performance of the wheelchair's outdoor navigation.

All forms of robot navigation depend on the given robot's locomotion and the corresponding kinematic model. Dead reckoning, the most simple form of robot navigation, uses only these concepts. More sophisticated navigation techniques can use any number of concepts, but there will always be a need to predict the robot's motion based on a set of control signals. Therefore, it is necessary to accurately model a robot's locomotion in order to enable high performance navigation. For the smart wheelchair, this need is satisfied with a differential drive kinematic model and gyroscope corrected odometry.

While accurate odometry is useful, it is not sufficient to estimate position. The problem with odometry is that errors in the position estimate accumulate unbounded over time. A simple method of localization is proposed to correct odometry and bound the error. The localization builds upon the results of gyroscope corrected odometry and implements an extended Kalman filter with an odometry prediction phase and a GPS correction phase. The system is described as self-contained because it hides the details of localization from the robot's higher level intelligence. The results of localization can then appear as the output of an abstracted position sensor.

Navigation can be further improved by considering what type of terrain is being traversed. If a robot can detect the current terrain type, it can dynamically select a drive control profile for the best performance on the given terrain type. Additionally, terrain types inconsistent with the robot's position estimate could be used as a mechanism to identify localization failures. The smart wheelchair is able to detect three different types of terrain using several unique terrain classifiers. Two classifiers use tactile information and the third uses spatial information provided by a 3D LIDAR. The final result is given by a weighted vote combining each classifiers' result. The task of creating and managing drive control profiles remains as future work.

None of the preceding results would be possible without suitable device drivers. The work of this thesis includes the development of three substantial device drivers and some firmware. A device driver has been written for one of the lab's newest sensors, an IFM O3D200 3D LIDAR. This device is unique in its use of XML RPC. Another device driver has been written for the MicroStrain 3DM-GX1 IMU, which surprisingly lacked a Linux driver despite its age. The last device driver was written specifically for the smart wheelchair. This includes the firmware for the chair, which runs the wheelchair's motor controller.

3 Literature Survey

The wheelchair was originally developed for the Automated Transport and Retrieval System (ATRS) project by VADER Laboratory and Freedom Sciences [4]. This project allows a wheelchair bound person to transport their wheelchair in a vehicle without the need of a van conversion or an attendant to help load the chair. This is made possible by an automatic docking procedure. The smart wheelchair project is very different form the ATRS project. The ATRS project requires that the wheelchair be positioned within a designated area, and then sensors on the vehicle are used to control the path of wheelchair. The smart wheelchair project places all sensors and control on board the wheelchair. Therefore, the wheelchair must be self sufficient and truly autonomous.

Outdoor navigation is a difficult problem for robots. The problem is exemplified by the DARPA challenges of 2004, 2005, and 2007. An article on the 2005 DARPA challenge gives a summary of the problems encountered by Caltech's Alice [5]. After a year of dedicated development by a team of programmers, the robot ultimately failed because of high voltage power lines interfering with the GPS signal. The challenges imposed by the complexity of the environment only get worse in urbanized areas. In the 2007 DARPA Urban Challenge, only 11 entries were selected to race out of 53 qualifying entries, and only 6 of the entries actually finished the race.

A GPS based Kalman filter has been developed for this thesis. GPS is particularly problematic in urban environments. GPS signals can be reflected off of buildings causing an error in the GPS fix. This is known as a multi-path error. Multi-path errors cannot be detected by examining the signal's dilution of precision (DOP). Various techniques have been proposed to filter and correct multi-path errors. Some methods take advantage of the internals of the Kalman filter, and reject GPS fixes based on the innovation and its covariance [11]. This technique requires precise tunning of the filter and rejection threshold based on the environment. The environment the wheelchair will vary greatly, so this did not seem appropriate.

More advanced techniques use additional sensors to detect occluded satellites. An omni-directional camera or 3D LIDAR is used to create a 3D map of the robot's surroundings which is compared to the estimated satellite geometry. The robot can then determine if there is a line-of-sight to the satellite [9][8]. Signals from the occluded satellites are discarded when calculating the GPS fix. Unfortunately, these techniques cannot be used because the wheelchair is unable to support the additional sensors.

4 Device Drivers

The overall software architecture developed in support of this thesis is shown in Figure 1. The higher level perception algorithms (upper blocks) are discussed in Sections 5 through 7. This section focuses on the lower level driver and firmware development (lower blocks). Device drivers were written when not available for a given device. A total of three drivers were written for this thesis; the wheelchair, the inertial measurement unit (IMU), and the flash LIDAR. All drivers were developed using a similar process. Generic driver libraries were written in C or C++ to implement the basic functionality. Additional driver software was then written for the two development platforms used by the lab, MATLAB and ROS.

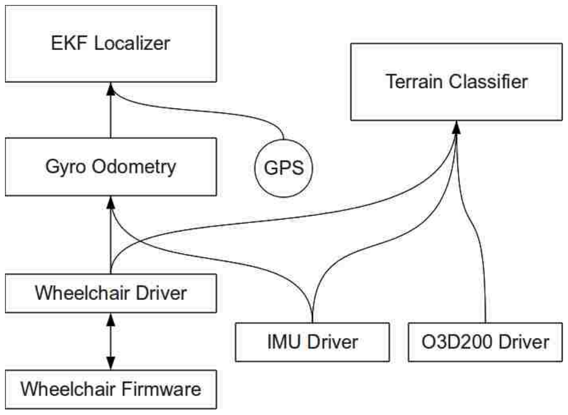

Figure 1: The software architecture. The driver are located at the bottom. The upper blocks will be discussed in the following sections.

4.1 MATLAB

MATLAB is an interpreted weakly-typed programming language. It was primarily used for prototyping and analysis. The MATLAB environment includes a C API and a MATLAB specific C or C++ compiler which is used to create shared libraries with the appropriate MATLAB specific linkages. The compiler is known as *mex* and the shared libraries are known as mex files. One compiled, mex files can then be interpreted as part of a MATLAB sentence.

The usage of a mex file is the same as a MATLAB function. The mex file is called like a function, with the file name as the function name. The mex file can accept a variable number of arguments, and can return a variable number of values. Typically the state of the driver is stored internally in the mex file and hidden from the user. It is common to use only one mex file to interface with the device driver. Various device driver operations can be selected by specifying a "command" string as the first argument when calling the mex file. The number of additional arguments and return values depends on the command string, and both will be constant for a given command string.

MATLAB is somewhat unique in the way it represents data. The language has only one primitive type, the matrix. Other types such as cells and structures can contain many matrices. This must be considered in mex files when moving data from C/C++ to MATLAB, as all C/C++ primitives must be converted into matrices. Structures are also useful in representing data. Structures can contain many named matrices.

4.2 ROS

ROS, the Robotic Operating System, is a meta-operating system designed for automatons. In general, ROS is much faster than MATLAB, so it was used during the robot's actual runtime. A ROS system is organized into node programs that communicate with the aid of a master node. A device driver program for a ROS system must be able to perform all of the utilities of a ROS node and interface with a device. In the case of the device being a sensor, a ROS node will typically poll the device for data, and publish the data to a sensor topic.

ROS supports publisher/subscriber and service/caller semantics.

ROS nodes are standard binary executables or Python scripts, but typically rely on ROS specific code which is organized into packages. For example, nearly all programs will depend on the *roscpp* or *rospy* packages for core utilities such as inter-node communication. Programs which need to perform transforms between different coordinate frames will depend on the *tf* package. For a C++ program, this means that the build process must link to these packages. This is accomplished by the ROS build system, *rosmake*.

ROS nodes are typically described by their parameters, data types, topics, services, and associated direction of data flow. Parameters are set during initialization or during runtime by the use of the *rosparam* utility. This is the method used to configure all aspects of a node, for example device paths, baud rates, flags, etc. Messages are essentially named data structures. Messages can subscribed or published to topics. A topic names a channel for messages of a certain type and purpose. Nodes can also call or handle RPCs known as services. Services are parameterized by messages and return messages.

4.3 Smart Wheelchair

The smart wheelchair development platform used for this effort is at Figure 2. The platform is based upon a Pronto M91 power chair with significant enhancements. The two drive wheels are equipped with 12 bit encoders and an embedded Linux computer has been installed under the seat. Freedom Sciences provided the embedded Linux distribution and cross-compiler, as well as a device driver and C API for controlling the motors and reading measurements from the encoders. Creating a robotic wheelchair required expanding on their work. The task was split into two sections: software on the embedded computer to manage the motor control and encoder measurements, from now on referred to as firmware, and client software which abstracts the wheelchair to a single device.

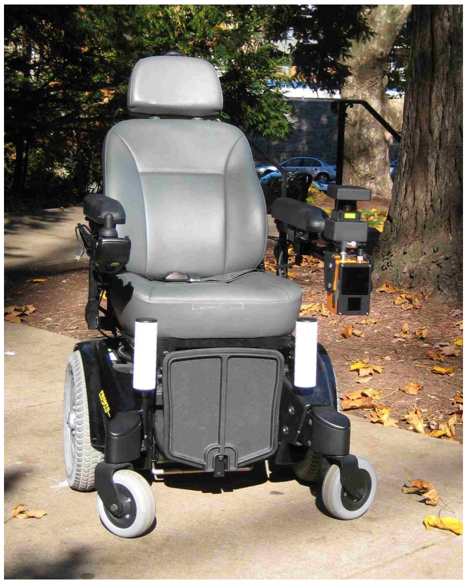

Figure 2: The smart wheelchair development platform integrates encoders, an 3DM-GX1 IMU, IFM O3D200 flash LIDAR, and WAAS enabled GPS.

The wheelchair firmware utilizes a PID controller developed by Mr. Samuel Kirkpatrick [7]. In his thesis, Mr. Kirkpatrick describes the difficulty of the control signals not being directly correlated to what is being controlled, and the process of finding the wheelchair's plant model. The motor control API provided by Freedom sciences does not use v and $omega$, but rather forward/reverse and left/right control signals. An affine transformation is used to map v and $omega$ to the control signals. The plant model is found experimentally by recording the steady state response over all possible control signals. The wheelchair is put on a lift and the wheels are replaced by weights to simulate the expected torque on each motor during normal operation on even ground.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} FR \\ LR \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{1}$$

Mr. Kirkpatrick also describes the challenges of using automatic PID tunning to produce high accuracy results. This is accomplished using a Ziegler-Nichols tuning process. The process starts by setting the PID gains, K_p, K_i, and K_d, to zero. K_p is then increased until the system reaches steady state oscillation, this is recorded as the critical gain, K_u. The period of the system's oscillation is also recorded as T_u. The system gains are then calculated according to the Ziegler-Nichols equations.

$$K_p = \frac{K_u}{5}, \qquad K_i = \frac{2K_p}{T_u}, \qquad K_d = \frac{K_p T_u}{3} \tag{2}$$

Mr. Kirkpatrick wrote his controller in the C language. This code was written to tune and test the controller. It was not intended to actually drive the chair, so the controller was rewritten. This rewrite has become the wheelchair firmware. Besides hosting the PID controller, the firmware also manages the input of control signals and the output of encoder measurements. Safe and predictable operation is ensured by extensive testing.

The embedded Linux computer can not be used to host the robot's intelligence for two reasons: most algorithms would be intractable on the limited hardware, and no ports are available for the attachment of sensors. The embedded computer is designed to be weather proof, so it lacks a keyboard, screen, etc. The only interface is an ethernet connector, which provides the facility for a remote terminal. Using a client computer has many benefits, including scalable hardware and an accessible user interface. The only requirement is that the embedded computer and client computer must be able to communicate control signals and encoder measurements.

Communication to and from the wheelchair is achieved by use of the Spread Toolkit [1]. This technology is chosen for a number of reasons including performance, reliability, and portability. The Spread Toolkit has also been used with some other past projects of the VADER Lab, such as Little Ben [2]. Two Spread mailboxes are used. A mailbox named “encoder” is used as a channel for encoder measurements. These messages include time and the left and right wheel encoder values as colon separated separated text. A mailbox named “control” is used as a channel for control signals. These messages include time, forward velocity, and angular velocity, and are also colon separated text. The exact format of the messages is listed in Appendix A. Plain text is used because it is human readable and cross platform. It also provides easy encoding and decoding of messages. The wheelchair’s relatively low frequency of communication implies that the overhead of working with text is acceptable.

The wheelchair firmware is written to be extremely reliable and safe. A configurable watchdog timer inhibits the wheelchair’s motion if the interval between control signals is too long. Functional communication channels are asserted on each communication. The firmware also validates every message sent or received. In the event of an error, the wheelchair must fail gracefully in a safe and predictable manner. Safe operation of the wheelchair is made certain by extensive unit testing of all aspects of the firmware. A comprehensive list of unit tests can be found in Appendix A.

The wheelchair device driver, as used by a client computer, is trivial because of the simple interface provided by the Spread Toolkit. The driver needs only to open the appropriate Spread mailboxes and convert the various Spread messages to usable data. It is therefore up to the client to manage the frequency of control signals to avoid the watchdog timer arresting motion. This is a desired behavior because it encourages the client-side programmer to write more stable client code.

The MATLAB driver is a mex file wrapping Spread C function calls and some supporting m files. The mex file is comprised of only four functions; open, close, send, and receive. The send and receive functions are parameterized by a character sting, so data must be encoded and decoded appropriately. The supporting m files provide this functionality.

The ROS driver was somewhat more involved. ROS typically represents control signals as *geometry_msgs/Twist* messages. A controller node listens for twist messages, and attempts to convert them into Spread messages and send them to the "control" topic. ROS does not supply a standard message for encoder measurements, therefore a new message type was created, *atrs/Ticks*. This message type is used by other odometry node, which subscribe to ticks messages and publish odometry messages. Separating the encoder measurements from the odometry calculations allows for greater modularity and configurability of the odometry. Odometry and gyroscope corrected odometry will be discussed in Sections 5 and 6.

4.4 MicroStrain 3DM-GX1

The MicroStrain 3DM-GX1 is an inertial measurement unit (IMU). It contains three angular gyroscopes, three accelerometers, and three magnetometers, a 16 bit analog to digital converter, and a microcontroller. It uses this hardware to provide temperature compensated, calibrated and bias corrected movement and orientation measurements. The device can also calculate various representations of orientation including a rotation matrix, quaternion, and Euler angles. The 3DM-GX1 as mounted on the smart wheelchair frame is shown at Figure 3.

The device transports data by RS-232 serial communication; a RS-232 to USB convert can be used in the absence of a serial port. This communication

Figure 3: The MicroStrain 3DM-GX1 AHRS/IMU.

used fixed length binary command and response packets. Two modes of operation are supported, polling and streaming. In polling mode the programmer request a measurement by sending the appropriate command. In streaming mode, measurements are sent repeatedly after only one command is sent. Polling mode provides flexibility by allowing the programmer to send any command at any time, but stream offers a significant increase in speed. The maximum streaming frequency is 100Hz and there is one communication per transaction. The maximum polling frequency is only 50Hz because there are two communications per transaction. Polling and streaming modes illustrate are illustrated in the pseudo code of Figure 4.

```
while data is needed do
    send(channel, command)
    // wait a few milliseconds
    response ← receive(channel)
    data ← parse(response)
    process(data)
end while
```

```
send(channel, streamcommand)
while data is needed do
    response ← receive(channel)
    data ← parse(response)
    process(data)
end while
```

Figure 4: Polling is shown on the left, streaming is shown on the right.

The generic device driver is written in C++ for Ubuntu or any other Unix-like OS supporting standard terminal I/O facilities. The code for the driver is split into two parts, with each part organized into a class. A generic raw serial I/O class hides the low level details of opening a file descriptor, selecting, read, writing, etc. This greatly simplifies the use of serial communication in the other 3DM-GX1 specific class. Interacting with the IMU typically involves sending a command, waiting for a response, and parsing. The 3DM-GX1 specific class hides this process from the programmer. For every particular data transaction, there is a corresponding method which encapsulates that transaction. Special consideration is taken to differentiate between polling and streaming modes. By default the driver assumes polling mode until a special method to start streaming is called. Likewise, another special method to stop streaming will return the driver to polling mode. Methods which poll the IMU are not available in streaming mode, and methods which acquire streaming data are not available in polling mode. An exception is thrown if a method is called in the wrong mode.

The robustness of the generic device driver affords a very straight forward write of the MATLAB interface. The mex file simply wraps the object initialization and method calls. The mex file must also transform the data received from the IMU into a MATLAB data type. The mex file parses the response buffers and places the data into MATLAB structures with clearly identifiable fields. The interface is described in Appendix B.

The design of the ROS node was modeled after an existing IMU node in the main ROS distribution. The node basically reads streaming data from the device and publishes the data as ROS messages, with a few extra features. The node includes a calibration service, which allows the caller to temporarily stop the device and find the gyroscope bias. The node also performs various diagnostics and publishes diagnostic messages. Self tests are also available to validate correct operation.

The 3DM-GX1 node publishes the typical IMU message, *sensor_msgs/IMU*. However, this message only includes the orientation as a quaternion, the linear accelerations, and the angular velocities; it does not include magnetometer readings. For the sake of completion, a new message was created to separately represent magnetometer readings, *m3dmgx1_imu/Magfield*. Ideally, this new message should be contained in the general IMU message, but this would require ROS updating the distribution or maintaining a branch of *sensor_msgs*.

4.5 IFM O3D200

The smart wheelchair integrates an IFM O3D200 3D Flash LIDAR underneath the left armrest, as shown in Figure 5. In effect, a flash LIDAR is a camera which measures distances. It features a 64 by 50 pixel PMD matrix and a large infrared illuminator. The sensor measures light intensity and distance using the principle of time-of-flight. Besides supplying intensity and distance images, the device is also capable of calculating the 3D coordinates and supplying x, y, and z value images. The O3D200 is originally intended for indoor use, but it has several qualities which make it the best choice for outdoor use.

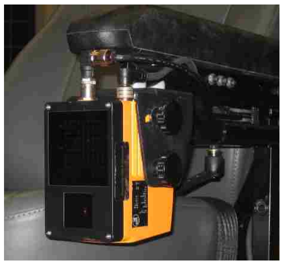
Figure 5: The IFM O3D200 3D Flash LIDAR.

Recently the Kinect has gained popularity as a 3D sensor [12]. The Kinect does not work outside because its emitted signal is relatively weak compared to the expected noise. The O3D200 works outside with only a slight drop in performance. Additionally, Kinect is a toy; it is constructed of plastic and has moving parts. The O3D200 is designed for industrial use and it is design to be resilient and compact. It has an all metal body and no moving parts.

LIDARs measure distance by using the principle of time-of-flight. A LIDAR will emit light and then wait for the light to bounce back. The time delay is divided by the speed of light to find the distance traveled. However, it is not feasible to measure the actual time delay because it is so small. In practice, the time delay is found by measuring the light's change in phase, but this creates an aliasing problem. The O3D200 can only measure distances in some unambiguous range, which depends on the frequency of the emitted light. Distances outside of this range are measured as the distance modulo the unambiguous range. The O3D200 provides two frequency modes. Single frequency mode has a shorter unambiguous range of 7.5 m, but a higher FPS. Dual frequency mode has a much larger unambiguous range or 45.0 m, but half the FPS. Is is important to note that the unambiguous range does not change the effective range of the sensor, which is based on the intensity of the emitted light. The effective range of the sensor was experimentally determined to be about 12.0 m.

The device provides two channels of communication over an ethernet cable. XML RPC is used for the configuration of the device. A TCP/IP socket is used for triggering exposures and the transfer of images. The device is able to return multiple images for each exposure, depending on the imaging mode. The O3D200 is also capable of some simple image processing including median and mean filtering. The 3D LIDARS's FPS depends on the desired exposure time, but the maximum FPS is 20 Hz in single frequency mode, and 10 Hz in dual frequency mode.

While some software preexisted the driver written for this thesis, it was of limited use. IFM supplied operator software that visualized the data, but provided no way of getting the data for further processing. IFM also provided a SDK, but it was primarily written for Windows. A small Linux demo existed, but was not robust enough for actual use on a robot.

The generic device driver for the O3D200 was written in C for Ubuntu and any other Unix-like OS with libxmlrpc. The driver API is organized into three sets of functions: a set of XML RPC functions for configuration, a set of TCP functions for requesting and receiving images, and a set of high-level functions combining both. The first two sets are intended for specific usages, the third set provides an easy interface for the most common usage. Any mixture of functions are supported. This design allows for an API which both powerful and easy to use. A C structure is used to hold information about the device and any other data required for communication. The generic driver is packaged as a shared library so it can be used easily by other programs.

The MATLAB interface provides the basic functionality required by the typical user. It allows the user to initialize the device, set the imaging mode, and get images. It also allows the user to change basic camera setting such as the frequency mode and the exposure time. The mex file was written using the high-level functions of the API, making it very easy to implement and maintain. A complete listing of the mex file interface can be found in Appendix C.

The O3D200 node was modeled after other camera nodes. Camera nodes typically publish images and distortion calibration information as *sensor_msgs/Image* and *sensor_msgs/CameraInfo* messages respectively. For the 3D LIDAR node, image messages are replaced by *sensor_msgs/PointCloud2* messages. Calibration information is not published because the O3D200 cannot be calibrated. Camera nodes typically expose the camera settings as node parameters. For each LIDAR setting there is a corresponding node parameter. On execution the O3D200 node initializes the LIDAR, configures it according to node paramaters, and begins streaming $< x, y, z >$ coordinates and light intensity i. The O3D200 node also performs various diagnostics and publishes diagnostic messages.

5 Odometry

Given robot and its kinematic model, and a set of motion measurements sampled at some frequency, a simple open-loop controller can be designed with an estimate of the robot's position as the state. This estimation of position is known as odometry. Odometry is described by Equation 3, where $\mathbf{p}$ is the robot's position and $\dot{\mathbf{p}}$ is the robot's motion which can be measured. If the motion measurements are sampled at a frequency f_s, then $\Delta t = 1/f_s$ is the time between measurements. This illustrates the fact that odometry is simply the integration of motion over time.

$$\mathbf{p}_t = \dot{\mathbf{p}}_t \cdot \Delta t + \mathbf{p}_{t-1} = \sum_{q=0}^{t} \dot{\mathbf{p}}_q \cdot \Delta t \tag{3}$$

It is important to note that no feedback is provided, and errors in the state estimate are not corrected. In other words, the errors of the motion measurements are accumulated in the position estimate, and therefore the error in position grows unbounded over time. Despite this, odometry is very useful in practice, if only to supplement more sophisticated localization techniques.

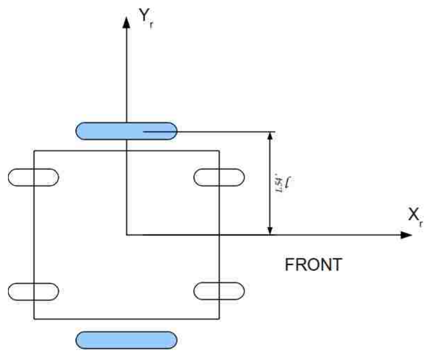

Figure 6: The robot's local frame.

The wheelchair has differential drive kinematics. The wheel chair has pneumatic tires on the two drive wheels (shaded blue) and four rubber caster wheels, as seen in Figure 2. The two drive wheels are each equipped with an encoder which provides measurements of angular displacement. The robots local frame will be defined as the following. The wheelchair is facing the in the positive X direction, and the left side of the wheelchair is in the positive Y direction. This is shown in Figure 6. Let the rotational displacements of the wheels be denoted as $\Delta\phi_l$ and $\Delta\phi_r$, for the left and right wheel respectively. It is also useful to know the angular velocities of each wheel, $\dot{\phi}_l$ and $\dot{\phi}_l$, shown in Equation 4. Let the wheel radii be r_r and r_l, and the wheels separated by length $2l$.

$$\begin{bmatrix} \dot{\phi}_l \\ \dot{\phi}_r \end{bmatrix} = \frac{1}{\Delta t} \begin{bmatrix} \Delta\phi_l \\ \Delta\phi_r \end{bmatrix} \tag{4}$$

Various assumptions must be made to derive the kinematic model. It is assumed that the wheels are always vertical to the ground, each wheel has a single point of contact, there is no wheel slippage, the rotation of each wheel is around the vertical axis, and the wheels are connected by a rigid frame. The constraints of each wheel must also be considered. The wheelchair's two pneumatic wheels are fixed wheels and will enforce constrains on the motion of the robot. Fixed wheels can only move in the plane of the wheel, motion orthogonal to the plane of the wheel must be zero. The four caster wheels will not constrain the movement of the robot. These assumptions and constraints allow for an idealized yet realistic model of the wheelchairs kinematics.

We will represent the position and orientation of the wheelchair in the global frame with the vector $\mathbf{p} = [\begin{array}{ccc} x & y & \theta \end{array}]^T$. A rotation matrix $\mathbf{R}(\theta) \in \mathbf{SO(3)}$ about the vertical Z axis is used to perform a coordinate transform between the global frame to the robot's local frame. For each fixed wheel, there will be two constraints. The rolling constraint is shown in Equation 5 and the sliding constraint is shown in Equation 6. The rolling constraint represents the rolling

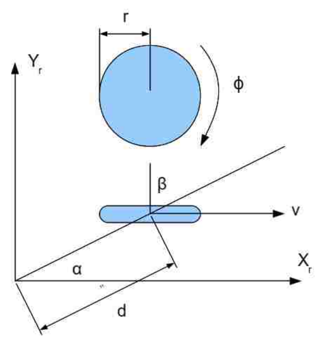

Figure 7: A fixed wheel in the robot's local frame.

motion of the wheel, while the sliding constraint restricts all motion to the plane of the wheel [10]. The values of α and β define the geometry of the wheel, as shown in Figure 7. The length d is the distance from the wheel to the robot's axis of rotation.

$$\begin{bmatrix} \sin(\alpha+\beta) & -\cos(\alpha+\beta) & (-d)\cos(\beta) \end{bmatrix} \mathbf{R}(\theta)\ \dot{\mathbf{p}} - r\dot{\phi} = 0 \tag{5}$$

$$\begin{bmatrix} \cos(\alpha+\beta) & \sin(\alpha+\beta) & d\sin(\beta) \end{bmatrix} \mathbf{R}(\theta)\ \dot{\mathbf{p}} = 0 \tag{6}$$

The geometry of each wheel is known so the constraints can be found immediately. The left wheel will have $\alpha = \pi/2$ and $\beta = 0$, and the right wheel will have $\alpha = -\pi/2$ and $\beta = \pi$. The value of d is simply l. The constraints for each wheel are then combined to give the constraints for the entire robot, as shown in Equation 7. This can be inverted to find the forward kinematic model, as shown in Equation 8 and 9. Notice that the sliding constraint is the same for each wheel and only shown once.

$$\begin{bmatrix} 1 & 0 & l \\ 1 & 0 & -l \\ 0 & 1 & 0 \end{bmatrix} \mathbf{R}(\theta)\ \dot{\mathbf{p}} = \begin{bmatrix} r_r\dot{\phi}_r \\ r_l\dot{\phi}_l \\ 0 \end{bmatrix} \tag{7}$$

$$\dot{\mathbf{p}} = \mathbf{R}(\theta)^{-1} \begin{bmatrix} 1 & 0 & l \\ 1 & 0 & -l \\ 0 & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} r_r\dot{\phi}_r \\ r_l\dot{\phi}_l \\ 0 \end{bmatrix} \tag{8}$$

$$\dot{\mathbf{p}} = \mathbf{R}(\theta)^{-1} \begin{bmatrix} 1/2 & 1/2 \\ 0 & 0 \\ 1/(2l) & -1/(2l) \end{bmatrix} \begin{bmatrix} r_r\dot{\phi}_r \\ r_l\dot{\phi}_l \end{bmatrix} \tag{9}$$

Knowing the kinematic model allows the robot to estimate its change in position given $\dot{\phi}_r$ and $\dot{\phi}_l$. However, there is a slight implementation issue. Notice that the encoders actually provide $\Delta\phi_r$ and $\Delta\phi_l$. Also notice that odometry eventually multiplies $\dot{\mathbf{p}}$ by time. The kinematic equation can be easily modified to take advantage of this observation by multiplying both sides by Δt. The $\dot{\phi}$ will become $\Delta\phi$, and $\dot{\mathbf{p}}$ will become $\Delta\mathbf{p}$. This is shown in Equation 10. The result is a cleaner implementation of odometry which is not dependent on time. It is also more accurate, since errors in the measurement of time no longer have an effect on the system.

$$\Delta\mathbf{p} = \mathbf{R}(\theta)^{-1} \begin{bmatrix} 1/2 & 1/2 \\ 0 & 0 \\ 1/(2l) & -1/(2l) \end{bmatrix} \begin{bmatrix} r_r\Delta\phi_r \\ r_l\Delta\phi_l \end{bmatrix} \tag{10}$$

It is desirable to describe the robot's motion in the most intuitive way possible. This is accomplished by describing the robot's motion using two parameters: v for the robot's linear forward velocity and ω for the robot's angular velocity. The values of v and ω can be found by combining the rotational velocities of the wheels, as shown in Equation 11. The (v, ω) motion pair can then be used to simplify the kinematic model to Equation 12, which can be further simplified to Equation 13. This last simplification matches our intuitive understanding the robot's motion, where change in position is dependent on the speed and heading of the robot, and change in heading depends on the turning speed.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \\ 1/(2l) & -1/(2l) \end{bmatrix} \begin{bmatrix} r_r \Delta\phi_r \\ r_l \Delta\phi_l \end{bmatrix} \tag{11}$$

$$\dot{\mathbf{p}} = \mathbf{R}(\theta)^{-1} \begin{bmatrix} v \\ 0 \\ \omega \end{bmatrix} \tag{12}$$

$$\begin{aligned} x_t &= x_{t-1} + v \ \cos\theta_{t-1} \ \Delta t \\ y_t &= y_{t-1} + v \ \sin\theta_{t-1} \ \Delta t \\ \theta_t &= \theta_{t-1} + \omega \ \Delta t \end{aligned} \tag{13}$$

The error in odometry can not be measured directly. However, the error in the encoder measurements can be determined experimentally. This error can be propagated through the kinematic model to find the error in the (v, ω) motion pair. This in turn can be used to find the error in the position estimate provided by odometry. The error in the encoder measurements is represented as a covariance matrix $\mathbf{C}_{\Delta\phi} = diag(\sigma_{\Delta\phi_r}, \sigma_{\Delta\phi_l})$. The relation defined in Equation 11 is used to find the new covariance matrix representing the error in in v and ω. This is shown in Equation 14 and 15. Notice that the covariance matrix representing the motion error will be constant, since $\mathbf{C}_{\Delta\phi}$, Δt, and the wheel radii are constant.

$$\mathbf{J}_{\Delta\phi} = \frac{1}{\Delta t} \begin{bmatrix} r_r/2 & r_l/2 \\ r_r/(2l) & -r_l/(2l) \end{bmatrix} \tag{14}$$

$$\mathbf{C}_m = \mathbf{J}_{\Delta\phi} \ \mathbf{C}_{\Delta\phi} \ \mathbf{J}_{\Delta\phi}{}^T \tag{15}$$

The same strategy can be used to find the covariance matrix representing the error in the position estimate provided by odometry. Two Jacobians are required. The first Jacobian propagates the error in the motion to the error in the position. The second Jacobian propagates the past error in position to the current error in position. The final result is Equation 18. This means that both estimates of position and error in position can be updated for every pair of encoder measurements.

$$\mathbf{J}_m = \begin{bmatrix} \cos\theta\ \Delta t & 0 \\ \sin\theta\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \tag{16}$$

$$\mathbf{J}_p = \begin{bmatrix} 1 & 0 & -\sin\theta\ v\Delta t \\ 0 & 1 & \cos\theta\ v\Delta t \\ 0 & 0 & 1 \end{bmatrix} \tag{17}$$

$$\mathbf{C}_{p|t} = \mathbf{J}_p\ \mathbf{C}_{p|t-1}\ \mathbf{J}_p{}^T + \mathbf{J}_m\ \mathbf{C}_m\ \mathbf{J}_m{}^T \tag{18}$$

6 Gyroscope Corrected Odometry

6.1 Poblem Statement

The accuracy of odometry is limited by the rapid accumulation of error in the position, and more importantly, the orientation. This is result of the robot's non-linear kinematic model. It was shown in the last section, in Equation 18, that error in ω accumulates in θ, and error in θ compounds the error in x and y. The largest contributor to odometry error is wheel slip, which effectively causes a bias in orientation. It can be determined from the kinematic model that a bias in orientation causes errors in position which scale quadratically with time. Mitigating wheel slip errors would cause the overall error in position would be greatly improved.

Gyroscope corrected odometry takes advantage of 3DM-GX1 IMU to reduce the error in ω and θ. The IMU measures the the wheelchairs rotational velocity ω_{IMU} using gyroscopes. The covariance matrix $\mathbf{C}_{\Delta\phi}$ is replaced by a new covariance matrix combining encoder error and IMU error, $\mathbf{C}_s = diag(\sigma_{\Delta\phi_r}, \sigma_{\Delta\phi_l}, \sigma_{\omega_{IMU}})$. The covariance matrix representing the error in motion must be updated to reflect this change as shown below. Notice that all other odometry equations, such as those for $\mathbf{C}_p$ and $\mathbf{J}_p$, will remain unchanged.

$$\begin{aligned} x_t &= x_{t-1} + v\ \cos\theta_{t-1}\ \Delta t \\ y_t &= y_{t-1} + v\ \sin\theta_{t-1}\ \Delta t \\ \theta_t &= \theta_{t-1} + \omega_{IMU}\ \Delta t \end{aligned} \tag{19}$$

$$\mathbf{J}_s = \frac{1}{\Delta t}\begin{bmatrix} r_r/2 & r_l/2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{20}$$

$$\mathbf{C}_m = \mathbf{J}_s\ \mathbf{C}_s\ \mathbf{J}_s{}^T \tag{21}$$

Wheel slip is defined as the difference between the measured wheel velocity and the actual wheel velocity, as shown in Equation 22. The wheel slips for the robot's left and right wheels are denoted as S_l and S_r respectively. When known, these wheel slips can be used to correct the robot's v and ω values, as shown in Equations 23 and 24. The problem of actually measuring S_l and S_r remains. In actuality, the wheelchair is not capable of distinguishing between S_l and S_r, so a heuristic is used to find an approximate solution.

$$s = r\dot{\phi} - v \tag{22}$$

$$v^* = \frac{(r_r \cdot \dot{\phi}_r - s_r) + (r_l \cdot \dot{\phi}_l - s_l)}{2} = v - \frac{s_r}{2} - \frac{s_l}{2} \tag{23}$$

$$\omega^* = \frac{(r_r \cdot \dot{\phi}_r - s_r) - (r_l \cdot \dot{\phi}_l - s_l)}{2l} = \omega - \frac{s_r}{2l} + \frac{s_l}{2l} = \omega_{IMU} \tag{24}$$

The smart wheelchair cannot directly measure wheel slip, it attempts to infer wheel slip by comparing the wheelchair's angular velocity as calculated by uncorrected odometry, ω, to the angular velocity as measured by the gyroscope, ω_{IMU}. It is assumed that the gyroscope is very accurate, and ω_{IMU} is approximately equal to the actual rotational velocity, ω^*. The wheelchair can quantify the effects of wheel slip, but it still cannot determine which wheel slipped or to what extent. That quantity is $(s_l - s_r)/(2l)$. This is enough to correct rotational velocity, since ω^* is simply equal to ω_{IMU}, as shown in Equation 24. Further assumptions must be made to find v^*, which may not hold in all situations. It is assumed that the wheels will always slip to resist motion; a wheel with a positive velocity will always have a positive slip. It will also be assumed that only one wheel slips at any given instant.

It is necessary to decide which wheel has slipped before a slip can be calculated. The error in rotational velocity is determined as $\omega_{err} = \omega_{IMU} - \omega$. The slipping wheel is identified by the sign of ω_{err} and the direction of the chair. The rotational velocity are described in the robot's local frame (Figure 6), so a positive rotation signifies a left turn. If the chair is moving forward and $sign(\omega_{err}) = -1$, then the uncorrected odometry is turning to the left faster then measured by the gyroscope. This implies that the left wheel is slipping, and over contributing to uncorrected odometry. Similarly, if $sign(\omega_{err}) = 1$, then the right wheel is slipping. When the wheelchair is traveling in reverse, the sign of the rotational velocity is effectively inverted so the opposite wheel is chosen. Once a wheel has been chosen, the rotational velocity of that wheel can be corrected. For example, Equation 25 shows the corrections for right wheel slip.

$$\dot{\phi}_r^* = \dot{\phi}_r - \frac{S_r}{r_r} = \frac{2l\ \omega_{IMU} + r_l\ \dot{\phi}_l}{r_r} \qquad \dot{\phi}_l^* = \dot{\phi}_l \tag{25}$$

The wheelchair does not attempt to calculate wheel slip if the chair is turning in place, or otherwise $sign(\dot{\phi}_l) \neq sign(\dot{\phi}_r)$. In this situation the wheelchair is in a tight turn which affects orientation more then displacement. Therefore, the wheelchair will benefit from corrections to rotational velocity, but corrections to linear velocity will be negligible.

6.2 Experimental Results

In practice, gyroscope corrected odometry performs much better than normal odometry. A test was constructed to compare the two methods of odometry. The wheelchair was driven down a 45.0 m long hallway at an average speed of 0.5 m/s in a serpentine path. This was done to induce wheel slip and other odometry errors. For each run, identical sensor data was processed by both methods. The results are shown in Figure 8. It can be observed that uncorrected odometry suffers greatly from drift. This is most likely caused by the deformable wheels having slightly different radii then found during the calibration. The gyroscope corrected odometry is unaffected by this inaccuracy. Overall, the path of gyroscope corrected odometry is more straight and representative of the true path. Each run seems only rotated by some random angle, and it would seem that the error is likely due to a slight misalignment of the starting position.

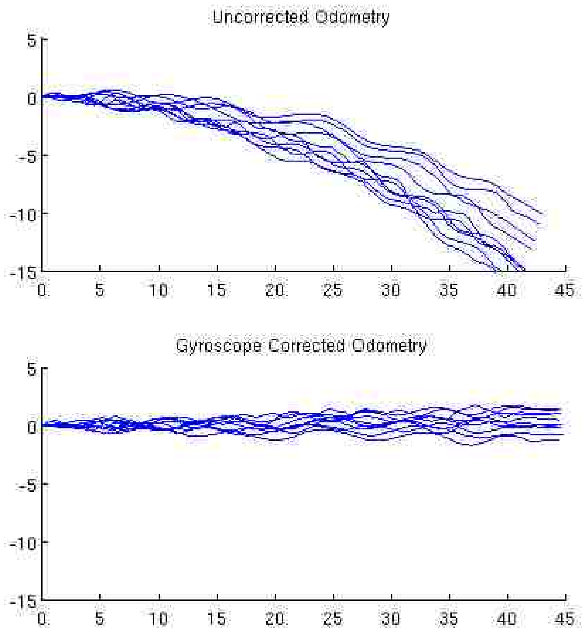

Figure 8: A comparison of odometry and gyroscope corrected odometry.

Gyroscope corrected odometry was also compared to uncorrected odometry on different terrains. Different terrains have different associated tractions which affect wheel slip. The results are shown in Figures 9, 10, and 11. The odometry was first tested on brick. There was very little wheel slip on this terrain, so the uncorrected odometry performed particularly well. The wheelchair was then tested on pavement. The result of brick and pavement were expected to be similar, but much more wheel slip occurred on the pavement. This was caused by the pavement being slightly sloped. Driving on a sloped surface will cause wheel slip to the down slope wheel. Finally, the wheelchair was tested on grass. The most wheel slip was was measured for grass, as expected. The results of gyroscope corrected odometry were very good. The position estimate was accurate even for high amount of wheel slip. The position estimate stayed very close to the ground truth provided by GPS.

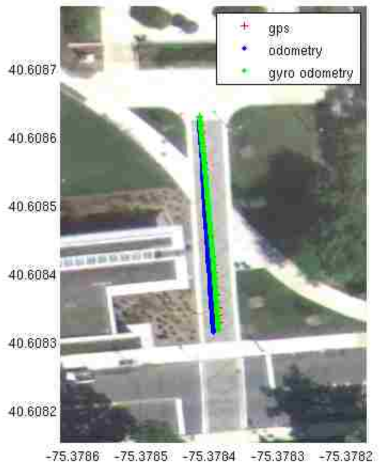

Figure 9: A comparison of odometry on brick.

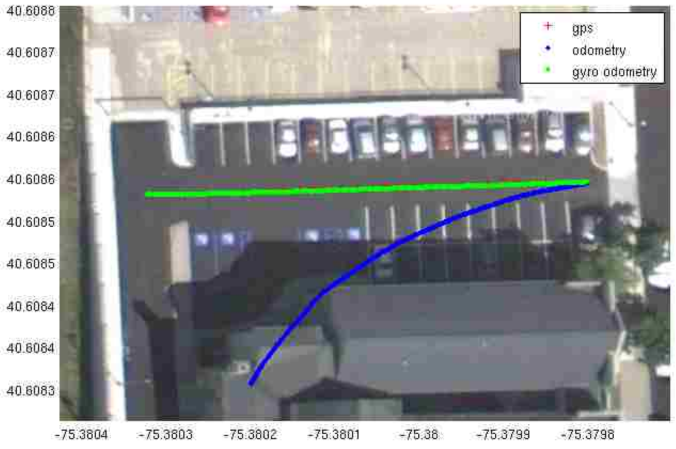

Figure 10: A comparison of odometry on pavement.

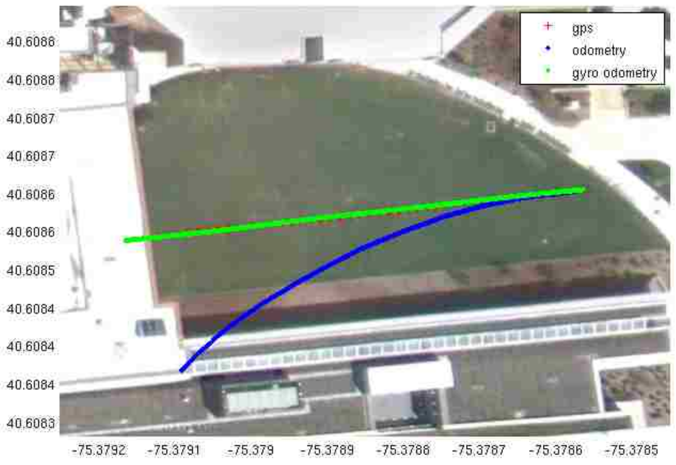

Figure 11: A comparison of odometry on grass.

7 Self Contained Pose System

7.1 Problem Statement

Little Ben used a OxTS RT-3050 pose system for high bandwidth pose estimation in the DARPA Urban Challenge [2]. Little Ben used to this pose system to completely determine its position and orientation, all other sensor information was used for obstacle detection and not for localization. The RT-3050 used a Kalman filter-based algorithm to combine inertial sensors, GPS updates with differential corrections from the OmniStar VBS service, and vehicle odometry information from the native embedded vehicle systems. Pose estimates where accurate to within half a meter circular error probable (CEP) and provided at a high update rate of 100 Hz. The RT-3050 was specifically designed for ground vehicles, and was resilient to GPS outages and GPS multi-path errors. Unfortunately, the RT-3050 was also expensive, costing about $40,000. For a consumer product such as a robotic wheelchair, the cost of a similar pose system would be prohibitively expensive.

The smart wheelchair has a similar yet more cost-effective pose system. The pose system uses a Kalman filter-based algorithm to combine the results of gyroscope corrected odometry with GPS measurements from a low cost sensor. Unfortunately, the wheelchair's pose system is not very accurate in urban environments because it suffers greatly from GPS multi-path errors, as described below. Unsuccessful attempts were made to correct for these errors. Further improvements to the pose system are left as future work.

The extended Kalman filter (EKF) is a nonlinear version of the Kalman filter. Specifically, the Kalman filter can be implemented by an EKF. However, an EKF is not provably optimal. An EKF may have nonlinear state transition or observation models, described by equations f and h respectively. The functions f and h can be used to predict the state and measurements, but they cannot be used directly for the propagation of error. The functions must be linearized about the current state and control vectors. This is done by finding the corresponding Jacobians **F** and **H**, shown below.

$$\mathbf{F}_k = \frac{\partial f}{\partial \mathbf{p}}\Big|_{\mathbf{p}_k,\mathbf{u}_k} \qquad \mathbf{H}_k = \frac{\partial h}{\partial \mathbf{p}}\Big|_{\mathbf{p}_k^-} \tag{26}$$

The state is the wheelchair's estimated position and orientation, $\mathbf{p} = [\begin{array}{ccc} x & y & \theta \end{array}]^T$, with error represented by covariance matrix **C**. The state transition function is used to predict the next state based on the current state and the control vector. The transition function f is taken directly from the kinematic model shown in Equation 13. The control vector is simply $[v, \omega]^T$. The noise associated with the controls is **Q**. The new error is calculated using a combination of the current error and the control error. The matrix **F** is the Jacobian of f with respect to the state vector **p**, and **W** is the Jacobian of f with respect to the control vector **u**.

$$\mathbf{p}_k^- = f(\mathbf{p}_k, \mathbf{u}_k) \tag{27}$$

$$\mathbf{C}_k^- = \mathbf{F}_k \mathbf{C}_{k-1} {\mathbf{F}_k}^T + \mathbf{W}_k \mathbf{Q}_k {\mathbf{W}_k}^T \tag{28}$$

The Kalman gain is found using the error prediction and the error of the observation. The matrix **H** is the Jacobian of the observation function h with respect to the state vector **p**, and **V** is the Jacobian of h with respect to the observation vector **z**. The observation model is trivial because the GPS directly measures position and orientation. Therefore $\mathbf{z} = [\begin{array}{ccc} x & y & \theta \end{array}]^T$, like the state vector. This causes the matrices **H** and **V** to be the identity matrix.

$$\mathbf{K}_k = \mathbf{C}_k^- {\mathbf{H}_k}^T (\mathbf{H}_k \mathbf{C}_k^- {\mathbf{H}_k}^T + \mathbf{V}_k \mathbf{R} {\mathbf{V}_k}^T)^{-1} \tag{29}$$

$$\mathbf{p}_{k+1} = \mathbf{p}_k^- + \mathbf{K}_k(\mathbf{z}_k - h(\mathbf{p}_k^-)) \tag{30}$$

$$\mathbf{C}_{k+1} = \mathbf{C}_k^- - \mathbf{K}_k\mathbf{H}_k\mathbf{C}_k^- \tag{31}$$

It became apparent that the pose system accuracy was limited by the accuracy of the GPS. Multiple signal reflections, or multi-path errors, are common in urban environments where GPS signals can bounce off of buildings. The horizontal dilution of precision (HDOP) does not model multi-path errors, so confidence based on HDOP problematic. Multi-path errors cannot be filtered based on confidence. The performance of the EKF is degraded due to the correction phase using incorrect but overconfident GPS measurements. The smart wheelchair robot is intended for use in urban environments, so it must be able to remove multi-path errors in order to effectively utilize GPS.

Research has been conducted to reduce multi-path errors in urban environments by detecting occluded satellites. These methods rely on additional sensing to perceive buildings, such as an omni-directional infrared camera [9] or 3D LIDAR [8]. The wheelchair is equipped with a 3D LIDAR, but it has a narrow field of view and not capable of perceiving buildings. Given the state of the art, mounting either an omni-directional camera or suitable 3D LIDAR is not feasible because of cost and space requirements. Multi-path errors could also be improved by equipping another antenna or a better GPS sensor. This was not feasible because of cost.

A system was developed to mitigate multi-path errors and their adverse effect on urban navigation. This technique was derived from observation. In the absence of multi-path errors the GPS works as expected, and changes in position correspond the the chairs motion. When a multi-path error occurs, the GPS fix "jumps" to a new position not corresponding to the chairs motion. This sudden change in the GPS fix is known as a GPS jump. By detecting GPS

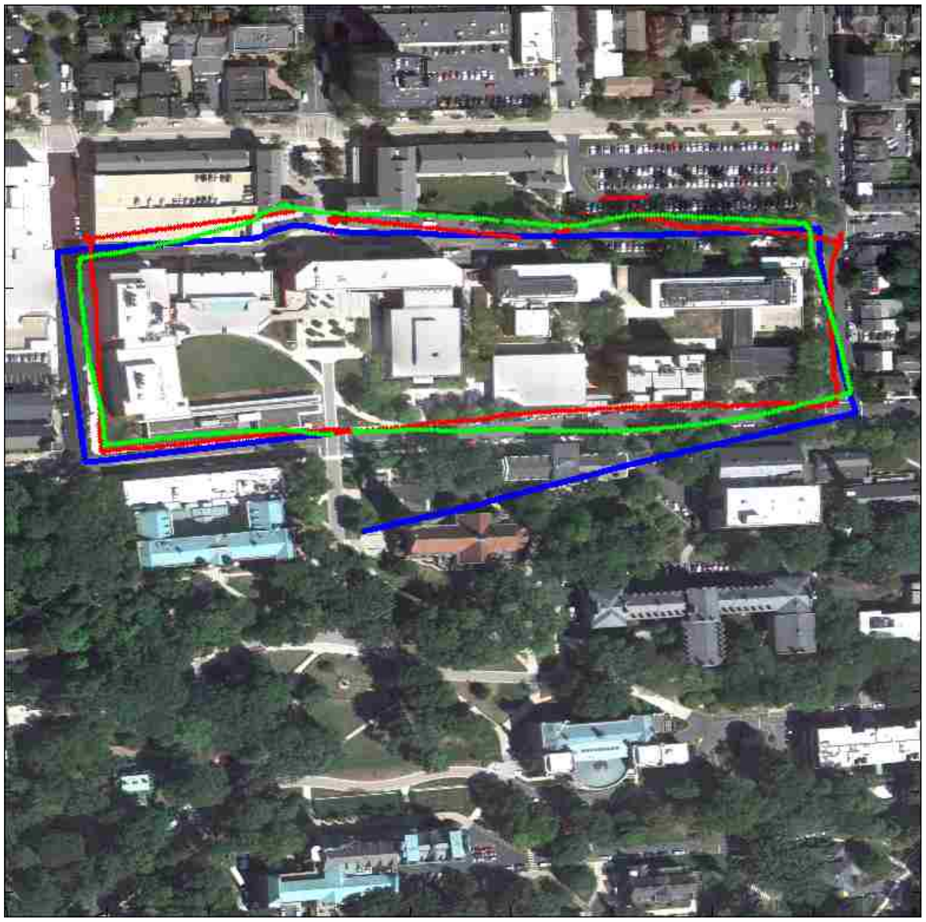

Figure 12: The initial results of the EKF. Odometry is shown in blue, GPS fix is shown in red, the result of the EKF is shown in green. The wheelchair starts near the middle Southern area heading East. It travels around the block and returns the the starting point.

jumps it is possible to detect when a multi-path error is occurring. Discarding GPS fixes suffering from multi-path errors should improve the accuracy of the EKF.

The detection of GPS jumps is based on the difference between GPS fixes and the state estimated by the EKF. Both the position and angular velocity of the chair are considered. First, the GPS fix is compared to the position predicted by the EKF. A GPS jump is detected if the difference in position exceeds a threshold derived from HDOP. Second, the angular velocity calculated from the last three GPS fixes is compared to the angular velocity predicted by the EKF (as measured by the gyroscope). A GPS jump is detected when the difference between angular velocity exceeds a threshold derived from the HDOP. When a GPS jump is detected, the GPS fix is discarded by the EKF and the correction phase is postponed.

7.2 Experimental Results

The results of GPS multi-path error filtering were not favorable. An example of the filter result is shown in Figure 13. The figure shows two concurrent multi-path errors. The first multi-path error is rejected based on angular velocity. The second multi-path error is not rejected because angular velocity matches the filter. The filter failed to remove a number of similar multi-path errors. Multi-path errors which cause a persistent drift of the GPS fix are particularly problematic. This is illustrated in Figure 14. These error do not cause GPS jumps, so they are undetectable to the filter. It appears that the complexity of multi-path error cannot be reduced to errors in displacement and angular velocity. Future work could be done to improve the filter, but the accuracy will always be limited by the overly simple notion of GPS jumps. It would be more productive to adopt more realistic models of multi-path error.

The self contained pose system ultimately failed because of unacceptable noise in the GPS fix. While this problem could have been solved by upgrading the robot's sensor suite, this would have decreased the cost-effectiveness of the robot. Attempts were made to develop techniques to correct GPS multi-path errors by detecting and filtering GPS jumps, but they were not successful.

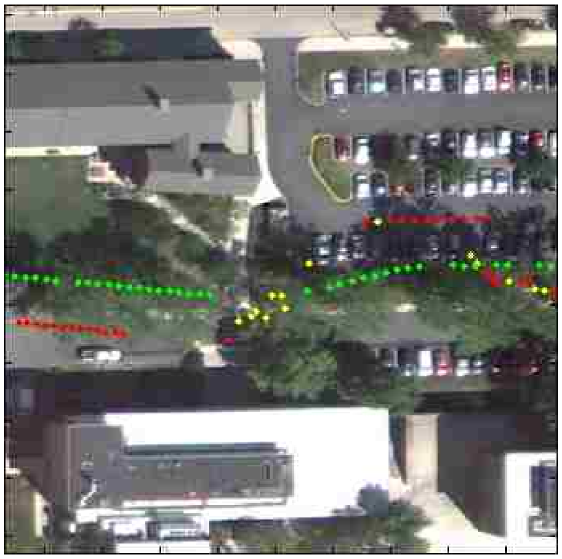

Figure 13: Particularly bad GPS data. The wheelchair is moving West to East on the sidewalk under trees. GPS fix is shown in red, rejected GPS fix is shown in yellow, he result of the EKF is shown in green. Notice that a multi-path error is not filtered because rotational velocity matches that of the wheelchair.

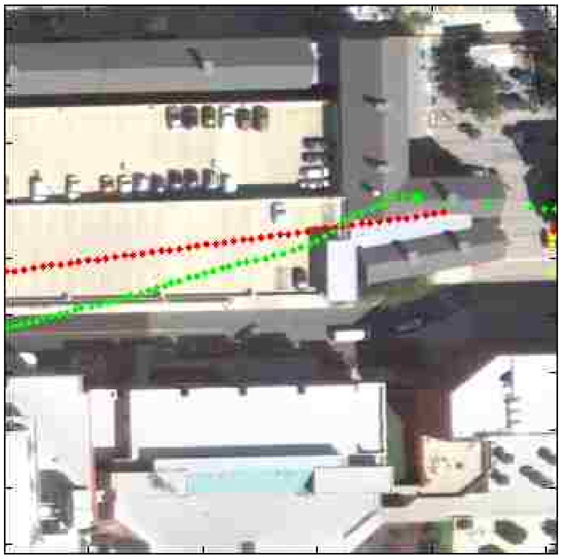

Figure 14: GPS multi-path causing a persistent error in GPS heading. The wheelchair is traveling West to East on the sidewalk South of the parking garage. The GPS has slowly drifted North and pulled the estimated position into a building. Such an error is almost indistinguishable from good data.

8 Terrain Classification

The wheelchair may need to navigate hazardous terrains such as ice, mud, and loose gravel. These terrains can cause the wheelchair's wheels to slip, adversely effecting the navigation performance. In the worst case, these terrains could cause navigation errors resulting in mission failure. Navigation performance could be increased for a given terrain by using a drive control profile tunned for that specific terrain. Terrain information could also be used to identify localization failures. For example, a localization error could be identified if the robot detected grass while expecting to be on a sidewalk. Of course, both of these ideas require knowing the terrain type. This thesis solves the terrain detection problem by implementing a terrain classifier utilizing traditional pattern recognition techniques.

In general, their are two approaches to terrain detection: visual and tactile. Visual approaches use cameras and LIDARs, tactile approaches use accelerometers and wheel slip measurements. Both approaches have been studied previously, and each has complementary advantages and disadvantages. This thesis discusses both approaches, and reasons that a combination of the two would increase terrain classification performance.

Visual approaches have been developed for various robots including those competing in the DARPA Urban Challenge, and the Mars Rover [6]. However, the performance of visual approaches is diminished under obscuring rain or snow conditions, superficial ground coverings, and shadows. Visual systems may also confuse terrains which look similar but have different tactile properties, for example dirt and mud.

Tactile approaches which measure vibration are naturally able to sense terrain roughness which directly effects drive control. Vibration methods also have the benefit of being robust in respect to weather conditions and ground coverings. Tactile approaches which measure wheel slip can distinguish between sets of terrains with similar expected slip values, but slip measurements are more dependent on weather conditions. The disadvantage shared by both methods is that tactile information for a path can only be collected while the robot is traversing that path, thus it is impossible to predict changes in terrain type. Visual information can collected for paths not yet traversed, and can therefore predict terrain type.

It was decided that three terrains would be considered for this thesis; grass, brick, and pavement. It was hypothesized that grass would be distinct from brick and pavement, and that brick and pavement should be similar and therefore more difficult to differentiate. Three different classifiers, each using a different sensor, were created to detect terrain type. A vibration classifier processing the vibration of the chassis caused by the road surface, a slip classifier processing the wheel slips found by gyroscope corrected odometry, and a texture classifier processing the variations in the ground surface.

The training sets were collected by driving the wheelchair over the three terrains while recording all relevant sensor outputs. The wheelchair was driven on flat sections of terrain, in a straight path, and at the nominal operational speed. Nearly flat sections of terrain were used to reduce the influence of slope on tactile measurements. The wheelchair's speed would also effect tactile measurements so data was recorded for two typical speeds, 0.3 m/s and 0.9 m/s, or the maximum speed of the first two drive modes. This implies that tactile classifiers will only provide accurate results when the wheelchair is moving at the nominal speed, or somewhere near 0.3 m/s to 0.9 m/s. It should also be noted that transient movement at the beginning and end of each recording was removed by truncation; accelerating from rest and braking to stop were not considered by the classifier.

8.1 Vibration Classifier

The vibration classifier uses tactile information about the driving surface to determine the terrain type. As the wheelchair moves the chassis is vibrated at different frequencies by various elements of the ground structure. Therefore, vibration measurements directly provides information on how the surface effects the robot chassis for a given motion, and indirectly provides information on the texture of the surface. The classifier does not attempt to distinguish between these concepts, and assumes they are identical for practical purposes.

The vibration classifier is a nearest neighbor classifier [3]. Nearest neighbor classifiers are are fast and surprising accurate, effectively approximating complex decision boundaries without any additional cost. There is also no classifier training, the training set is used directly for classification. The classifier works by comparing the test data point to the data points of the training set. The most frequent label of the nearest data points is selected. The nearest points are those with the least Euclidean distance. This classifier uses the L2 norm.

Data points are created by recording the 3D accelerometer vector for a set duration, and then finding an FFT for each dimension of the vector. For this thesis the duration of the recording was set to three seconds. With the IMU reporting accelerometer vectors at 100 Hz, the recording was 900 elements total. Half of each FFT is ignored because of symmetry, making the size of each data point 450 elements. Additionally, the first element of the FFT representing the average value was removed. This was done because the average values of the accelerations do not provide information on the texture of the surface; essentially the average values of the accelerations represent the g vector relative to the robot, or the orientation of the robot.

The remaining recording was then split into three second sections. The three second sections are not contiguous, but staggered at one second intervals. This was done to hopefully smooth the decision boundary intrinsic to the nearest neighbor classifier. A robot operating in real time could run the classifier on a three second moving window, allowing the robot to react instantly to changes in terrain. This was not done, and left for future work.

The results of the classifier were favorable. The overall accuracy was good. There was some confusion between pavement and brick as expected, but somewhat surprisingly the brick was classified as pavement much more than pavement was classified as brick. Grass was usually not confused, and had both a low type I and type II error. The success of the classifier can be contributed to the nicely separated class spaces. All of the classes are relatively distinct which allows for easier classification and high classifier performance. Examples of the data can be seen in Figures 15, 16, and 17.

	Classifier Label			
Actual Label	Brick	Grass	Pavement	Type I Error
Brick	99	17	62	79
Grass	0	146	13	13
Pavement	8	0	158	8
Type II Error	8	17	75	100
Confidence	0.925	0.896	0.678	

Table 1: Confusion matrix for the vibration classifier.

8.2 Slip Classifier

The slip classifier attempts to classify terrains by the amount of wheel slip detected during movement. Wheel slip is measured from a combination of gyroscope and wheel encoder measurements, as described in the section on gyroscope corrected odometry. Different terrains have an associated traction, effecting the ability of a wheel to grip the surface. The traction for a surface is inferred from wheel slip, which allows the classifier to decide a terrain type based on traction.

As described in Section 6, the wheelchair calculates wheel slip based on the

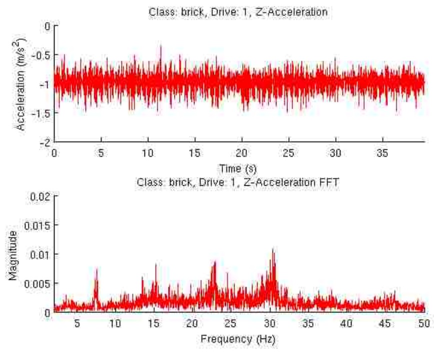

Figure 15: The vibration FFT for brick.

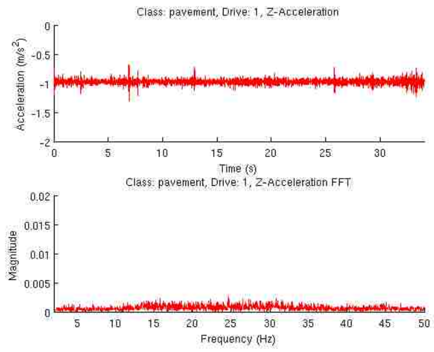

Figure 16: The vibration FFT for pavement.

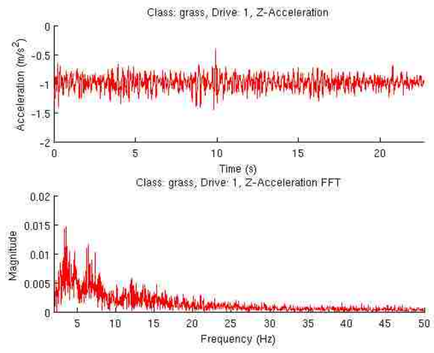

Figure 17: The vibration FFT for grass.

difference between the angular velocity calculated by odometry and the angular velocity measured by the gyroscope. The wheelchair is only able to approximate wheel slip, by assuming that only one wheel slips at a given instant. The classifier considers a span of slip measurements, so the slip measurements must be recorded over time. Slip measurements for each wheel are combined into single dimension vector by considering s_l as positive and s_r as negative. This will be referred to as the slip signal.

The slip classifier uses simple thresholding on the mean square (RMS) of the slip signal. The RMS provides the average magnitude of slip. For each terrain type, all of the slip vectors are concatenated and the RMS is calculated. Thresholds are calculated from these values. During classification, the RMS of a recording is thresholded and a terrain type is selected. This classifier is simple but effective. More complex classifiers are not appropriate given the nature of the data. More complex classifiers were attempted, but they did not perform well because they were overtrained on the training set.

The results of the slip classifier were not very accurate. The classifier was able to classify grass, but often confused brick and pavement as predicted. Slip was determined to be highly dependent on terrain slope. For example, if the wheelchair were to traverse sloped pavement, the slip would be high enough that it would be confused with grass. This causes the classifier to incorrectly match slope rather then terrain. Slip was also found to be very dependent on weather conditions. Trials run on wet grass had much higher slip than trials run on dry grass. As a consequence, the accuracy of the slip classifier varies greatly depending on the current conditions. In conclusion, the slip classifier is not a good terrain classifier. It is unable to discern between similar surfaces. Instead, slip measurements may be better suited to score the drivability of a known terrain.

	Classifier Label			
Actual Label	Brick	Grass	Pavement	Type I Error
Brick	81	27	70	97
Grass	4	146	9	13
Pavement	92	9	65	101
Type II Error	96	36	79	211
Confidence	0.458	0.802	0.451	

Table 2: Confusion matrix for the slip classifier.

8.3 Texture Classifier

The texture classifier is the only visual approach used in this thesis. It is assumed that each terrain type will have a unique texture, and that a terrain will be identifiable by that texture. For example, pavement will be relatively smooth with only a few edges, on the other hand grass will be uneven and bumpy. Traditionally, image processing techniques have been used to detect textures. This thesis presents a method to detect textures directly using spatial information obtained from 3D imaging.

The texture classifier uses an IFM O3D200 3D LIDAR in place of traditional cameras or LIDARs. The 3D LIDAR takes distance images of the area immediately in front of the robot, and these images are converted into point clouds. From each point cloud a ground plane is segmented and extracted. The ground plane is then processed by the texture classifier. It should be noted that the O3D200 is not pointed at the ground, and is not assigned specifically to the task of terrain detection. The O3D200 is mounted facing forward, as was required for other purposes such as mapping, localization, and obstacle avoidance. Despite this, a ground plane is usually visible and usable by the texture classifier.

The texture classifier is a nearest neighbor classifier, like the vibration classifier. Again this means that the classifier is not trained; classification depends directly on the training data. This classifier also uses an L2 norm. Similarly to the vibration classifier, frequency information is used to construct the data points. A spatial FFT of the ground plane is used to represent the texture of a given terrain. High frequencies correspond to sharp features such as loose rocks, low frequencies correspond to more gradual features such as small peaks or troughs. Special considerations are made to calculate the spatial FFT as described below.

A common requirement of classifiers is that the number of features remains constant, or in the case of the nearest neighbor classifiers, each point has the same dimension. This implies that each spatial FFT must have the same number of elements, and therefore each patch of ground plane must be exactly the same size. This is a difficult requirement for the O3D200, which provides relatively sparse point clouds. There is also the issue of actually calculating the spatial FFT. Calculating the spatial FFT requires a regular grid of data. The O3D200 provides point clouds which are highly irregular. In order to solve both problems, the ground plane is interpolated and formatted to a grid.

Data points are constructed as follows. For each point cloud, a patch of the ground plane is extracted; this patch must be roughly the required size of 1 m^2. The path is rotated and translated so it resides in the XY plane. The Z value of each point then represents the distance from the ground plane. The patch is then interpolated and formatted to a 1 m^2 grid which is 20 by 20 elements. The 2D spatial FFT is then performed on the grid. The DC component is removed, redundant symmetric data is discarded.

	Classifier Label			
Actual Label	Brick	Grass	Pavement	Type I Error
Brick	491	271	28	299
Grass	119	552	26	225
Pavement	200	170	386	370
Type II Error	399	441	54	894
Confidence	0.606	0.556	0.878	

Table 3: Confusion matrix for the texture classifier.

The results of the texture classifier were somewhat disappointing. The classifier often mislabeled brick and grass. The classifier had high confidence when classifying pavement, but it rarely used this label. The inaccuracy of the texture classifier is likely due to the processing of the data and the low resolution of the 3D LIDAR. In combination these factors remove nearly all high frequency information, making all data points more similar. Examples of the data can be seen in Figures 18, 19, and 20.

Inaccuracies could also be introduced during the ground plane extraction phase. The ground plane extractor was not tuned to any given terrain, and it was not tested if the ground plane extractor performs differently on various terrains. Errors in the ground plane extraction phase would cause biases in the data points leading to bad classifications. As future work, the performance of the ground plane extractor should verified. Ideally, the classifier should detect bad points, and filter them accordingly.

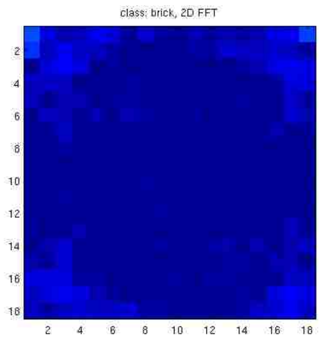

Figure 18: The spatial FFT for brick.

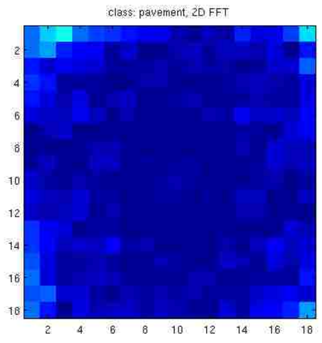

Figure 19: The spatial FFT for pavement.

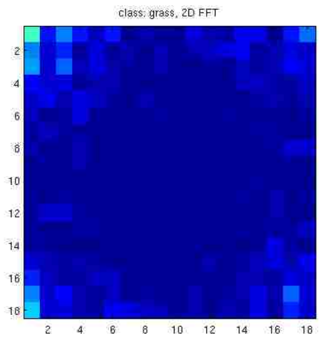

Figure 20: The spatial FFT for grass.

9 Conclusions and Future Work

This thesis makes four major contributions. Low level software, including device drivers and firmware, was created to enable perception and control. Odometry and gyroscope corrected odometry were developed to support future localization techniques. A self-contained pose system was developed and the correction of multi-path errors was attempted. Finally, three classifiers were created to detect terrain type.

Gyroscope corrected odometry presented a heuristic to detect wheel slip. This heuristic is very simple, and makes assumptions that are not always true. This technique could be improved upon. For example, the heuristic could be expanded to consider the wheelchair's control signal. It may then be able to differentiate between accelerating slip and breaking slip. The wheelchair can measure ω_{act} but it cannot measure v_{act}. Knowing both v_{act} and ω_{act} would allow the robot to determine the slip for both wheels. As future work, some other sensor could be used to accurately measure the wheelchair's linear velocity.

The future work for the pose system is somewhat open ended. State of the art techniques require sensing obstructed GPS signals. At present, the required sensors are not compatible with the wheelchair. In the future, these sensors may become smaller and more cost-effective, and these techniques can be implemented on the wheelchair. The filter could also be improved. The EKF used only gyroscope corrected odometry and GPS. Future work could attempt to integrate additional sensors, such as WiFi, RFID, LIDAR based localization, etc, to improve the filter results.

Three classifiers for terrain detection are presented. The vibration classifier, slip classifier, and texture classifier use tactile and visual methods to determine terrain type. The vibration classifier performed the best. The slip classifier was able to distinguish between some terrains, but its accuracy was limited by the variance of the slip measurement. The texture classifier attempted to extract texture information from 3D point clouds, but the low densities limited the results. A combinational classifier could combine the individual results by a weighted vote to give a final terrain type. Often combinations of classifiers can outperform all of the individual classifiers.

The slip classifier was prone to misclassification resulting from variations in the terrain. The slip measure is affected by a number of variables including slope. As future work, a more complex system would be able to detect slope using the IMU and adjust the classifier accordingly. This new classifier would be able to better classify terrains on any slope. Other future work remains, as slip measurements are still affected by weather, ground covering, etc.

This thesis proposes methods of detecting terrain type, with the goal of selecting terrain specific drive control profiles. The actual creation of terrain specific drive control profiles remains as future work. Research would need to be done on how to actually tune the drive control for a given terrain. While this is out of the scope of this thesis, it would no doubt be extremely useful to the wheelchair and other mobile robots.

References

[1] Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Johnathan Stanton. The spread toolkit: Architecture and performance. Technical report, John Hopkins University, 2004.

[2] Jonathan Bohren, Tully Foote, Jim Keller, Alex Kushleyev, Daniel D. Lee, Alex Stewart, Paul Vernaza, Jason C. Derenick, John R. Spletzer, and Brian Satterfield. Little ben: The ben franklin racing team's entry in the 2007 darpa urban challenge. In *The DARPA Urban Challenge'09*, pages 231–255, 2009.

[3] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification.* John Wiley & Sons, Inc., 2001.

[4] C. Gao, I. Hoffman, T. Miller, T. Panzarella, and J. Spletzer. Autonomous docking of a smart wheelchair for the automated transport and retrieval system (atrs). *Journal of Field Robotics*, 25(4-5):203–222, 2008.

[5] J. Gillula and J. Leibs. How to teach a van to drive: an undergraduate perspective on the 2005 darpa grand challenge. *Control Systems, IEEE*, 26(3):19–26, June 2006.

[6] I. Halatci, C.A. Brooks, and K. Iagnemma. Terrain classification and classifier fusion for planetary exploration rovers. In *Aerospace Conference, 2007 IEEE*, pages 1–11, March 2007.

[7] Samuel Kirkpatrick. Perception, planning, and control for mobile robotics systems. diploma thesis, Lehigh University, May 2010.

[8] D. Maier and A. Kleiner. Improved gps sensor model for mobile robots in urban terrain. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 4385–4390, May 2010.

[9] J.-i. Meguro, T. Murata, J.-i. Takiguchi, Y. Amano, and T. Hashizume. Gps accuracy improvement by satellite selection using omnidirectional infrared camera. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 1804–1810, September 2008.

[10] R. Siegwart and I. Nourbakhsh. *Introduction to Autonomous Mobile Robots.* MIT Press, 2004.

[11] S. Sukkarieh, E.M. Nebot, and H.F. Durrant-Whyte. Achieving integrity in an ins/gps navigation loop for autonomous land vehicle applications. In *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, volume 4, pages 3437–3442, May 1998.

[12] Jenna Wortham. With kinect controller, hackers take liberties. `http://www.nytimes.com/2010/11/22/technology/22hack.html`, November 21, 2010.

A Wheelchair Firmware and Device Driver Details

A.1 Firmware Unit Tests

The following unit tests were performed on the wheelchair firmware.

- Bad command line option causes graceful exit.
- Bad Spread address causes graceful exit.
- Spread connect error causes graceful exit.
- Spread receive error causes graceful exit.
- Spread send error causes graceful exit.
- Malformed control packet during decoding causes graceful exit.
- Malformed encoder packet during decoding causes graceful exit.
- Malformed control packet during encoding causes graceful exit.
- Malformed encoder packet during encoding causes graceful exit.
- Control v exceeds safety limit causes graceful exit.
- Control ω exceeds safety limit causes graceful exit.
- DriverHasControl fails, loss of driver control causes graceful exit.
- SetMotion fails, cannot set (v, ω) causes graceful exit.
- Watchdog timer expires, no control signals causes arrested motion.

A.2 Spread Message Format

Mailbox	Structure
encoder	dddddd.dddddd:+d.dd:+d.dd:dddddd:dddddd
	time, control v, control ω, left encoder, right encoder
control	dddddd.dddddd:+d.dd:+d.dd
	time, control v, control ω

Table 4: Formatting of the wheelchair Spread messages.

B 3DM-GX1 Device Driver Details

B.1 MATLAB API

- `m3dmgx1_mex('open', device_name)`
- `m3dmgx1_mex('close')`

  Initializes and closes the device.

- `m3dmgx1_mex('startstreaming')`
- `m3dmgx1_mex('stopstreaming')`

  Starts and stops streaming data mode.

- `data_structure = m3dmgx1_mex('streamdata')`

  Gets IMU data in streaming mode.

  `data_structure`

  `q` – The quaternion representing orientation
  `magfield` – The magnetic field 3D vector
  `accel` – The linear acclerations 3D vector
  `angrate` – The angular velocities 3D vector
  `time` – The time of the measurement

- `data_structure = m3dmgx1_mex('data')`

  Gets IMU data in polling mode.

- `info_structure = m3dmgx1_mex('info')`

  Gets the device information in polling mode.

  `info_structure`

  `serial` – The serial number
  `firmware` – The firmware version
  `loop_time` – The time duration of the internal IMU loop

- `euler_structure = m3dmgx1_mex('euler')`

  Gets Euler angles in polling mode.

  `euler_structure`

  `pitch` – The pitch angle
  `roll` – The roll angle

  `yaw` – The yaw angle
  `time` – The time of the measurement

- `matrix_structure = m3dmgx1_mex('matrix')`

  Gets orientation matrix in polling mode.

  `matrix_structure`

    `matrix` – The 3 by 3 orientation matrix
    `time` – The time of the measurement

- `m3dmgx1_mex('bias')`

  Gets orientation matrix in polling mode.

- `value = m3dmgx1_mex('eepromr', address)`
- `m3dmgx1_mex('eepromw', address, value)`

  Reads and writes values to the EEPROM memory.

C O3D200 Device Driver Details

C.1 MATLAB API

- `o3d200_mex('startup', xml_url)`
- `o3d200_mex('shutdown')`

  Initializes and closes the device.

- `o3d200_mex('mode', mode_string)`

  Sets the imaging mode using a mode string. Each letter of the mode string signifies an image to be created. The letters are defined as follows.

d/D	distance
i/I	intensity
x/X	x coordinate
y/Y	y coordinate
z/Z	z coordinate

  Capital letters tell the LIDAR to do a new capture before responding with the image. For example, the string 'xyz' would return images from an old capture, the string 'Xyz' would return images from a new capture, the string 'XYZ' would return images from three new captures.

- `[i_1, i_2, ..., i_n] = o3d200_mex('image')`

  Receives images according to the imaging mode. An image will be returned for each letter of the mode string. The images will be in the same left-to-right order as the mode string.

- `o3d200_mex('exposure', INT1, INT2)`

  Sets the image integration times, also known as the exposure times. INT1 is the first exposure time. INT2 is the second exposure time.

- `o3d200_mex('frequency', f)`

  Sets the frequency mode. The mode number can be 0 through 4. The various frequency modes are documented in the programmer's guide.

- `o3d200_mex('mideanfilter', on_off)`
- `o3d200_mex('meanfilter', on_off)`

  Turns a filter on or off.

- `frontend_structure = o3d200_mex('frontend')`

  Gets the LIDAR's frontend configuration.

  `frontend_structure`

  `f_mode` – The frequency mode (0-4)
  `double_sample` – The double sample mode (0=off, 1=on)
  `first_integral` – The first exposure time (ms)
  `second_integral` – The second exposure time (ms)
  `inter_frame_mute_time` – The inter-frame mute time (ms)

- `network_structure = o3d200_mex('network')`

  Get the LIDAR's network configuration.

  `network_structure`

  `ip` – The IP address
  `subnet` – The subnet address
  `gateway` – The gateway address
  `xml_port` – The XML port number
  `tcp_port` – The TCP port number
  `dhcp_mode` – The DHCP mode (0=off, 1=on)

Vita

Timothy J. Perkins grew up in Topsfield, Massachusetts. His parents are Timothy H. and Marie Perkins. He has three younger brothers. He graduated from Masconomet RHS in 2005. He received by B. S. in electrical engineering from Lehigh University in 2009.

From 2009 to 2011, he was a research assistant in the Vision, Assistive Devices, and Experimental Robotics (VADER) Laboratory under Professor John R. Spletzer at Lehigh University. He received his Masters degree in Computer Engineering in May 2011.