THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

Department of Mechanical and Nuclear Engineering

Autonomous Wheelchair Control

Matt Barnes
May 2013

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Mechanical Engineering
with honors in Mechanical Engineering

Reviewed and approved * by the following:

Dr. Sean Brennan
Assistant Professor of Mechanical Engineering
Thesis Supervisor

Dr. Hosam Fathy
Assistant Professor of Mechanical Engineering
Faculty Reader

Dr. HJ Sommer III
Professor of Mechanical Engineering
Honors Advisor

* Signatures are on file in the Schreyer Honors College

Abstract

People with disabilities are increasingly reliant on technology to provide freedom of mobility. However, current technology requires extensive direct user joystick input, which limits functionality and creates error between desired and actual system output. This project aims to create an autonomous system for an electric wheelchair, resulting in improved freedom of mobility for people with disabilities.

In an ideal system, the user's desired end location would be reached using advanced control algorithms, such that the device could implement all motor commands along the path. Conventional electric wheelchairs rely on users to manually direct the electric wheelchair through a series of cumbersome joystick maneuvers. In order to automate this process, the computer will need encoder feedback for motor control, LIDAR data for environment information, and path planning algorithms to achieve the desired end state. The project develops a robotic wheelchair testbed from the ground up with sensors, computers, and a power management system to conduct autonomous wheelchair experiments. Capabilities of the testbed are demonstrated with implementation of environment mapping and basic control algorithms, including closed-loop PID velocity control, and kinematic pose-following laws.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Research on mobile robotics in the last fifty years has contributed to widespread success in fields ranging from terrestrial planet exploration to advanced bomb disposal. Emerging areas – including autonomous vehicles – show significant promise both in academia and industry. Furthermore, Dr. Sean Brennan's lab needed a fully instrumented and easy to use platform useful for studying indoor robotics in future projects.

This project uses a previously donated electric wheelchair not only for cost and convenience reasons, but also in an attempt to improve mobility of disabled users. There are over 200,000 electric-powered wheelchairs users in the United States [1]. However, quadriplegics and other severely disabled users either have trouble or cannot control electric wheelchairs. Automating navigation processes using environment information and path planning algorithms would greatly improve the life-style of these disabled users.

1.2 Goals

The primary goal of this project is to create a platform for testing indoor mobile robotic applications. Significant hardware development is required to test and validate control algorithms. The construction of an autonomous wheelchair testbed requires designing and mounting sensors, computers, and power supply systems to a conventional electric wheelchair. Success of this objective is measured by demonstration of a robotic wheelchair controlled using both onboard and remote computers. Hardware development constitutes a majority of the time spent on the project.

Upon completing the test platform, several basic control algorithms were employed to demonstrate its capabilities. Potential challenges for autonomous wheelchairs include similar challenges in other mobile robotics applications, including localization, environment mapping, path planning, and closed-loop velocity or position control. Unique challenges exist regarding control algorithms specific for wheelchairs which will feel smooth or 'natural' to the human passenger.

Thus, a second major goal of this project is to demonstrate basic algorithms in the areas of localization, closed-loop velocity control, environment mapping and path planning which create 'natural' movements.

1.3 Outline of Remaining Chapters

Chapter 2 begins with a broad overview of autonomous wheelchair literature and related areas, including mobile robotics. After identifying topics of interest, the appropriate hardware is selected, interfaced, and installed in Chapter 3. Overall control architecture setup is devised in Chapter 4 and broken down into low-level control in Chapter 5 and high-level control in Chapter 6. Lastly, environment mapping in ROS is demonstrated in Chapter 7.

Chapter 2

Literature Review

2.1 Introduction

This chapter describes major topics of research on autonomous and semi-autonomous wheelchairs including navigation, velocity control, suspension control, stability control, and stair-climbing functionality [2]. Ultimately, navigation and velocity control are chosen as areas for further exploration with hardware development because affordable and high-performance navigation is a formidable challenge for commercial autonomous wheelchairs, and velocity control is an integral component of implementing navigation algorithms.

The chapter begins in Section 2.2 with an overview of existing platforms, including hardware specifications and experimental success of navigation algorithms on wheelchairs. Trends on the degree of autonomy are observed in Section 2.3 from the selected platforms and other publications. Section 2.4 conducts a more detailed analysis of path planning and obstacle avoidance algorithms used on wheelchairs and mobile robots.

2.2 Existing Autonomous Wheelchairs Platforms

Research on autonomous wheelchairs emerged during the 1980s and gained traction with computer advancements during the 1990s. Several platforms developed from the ground-up are chosen to demonstrate various approaches to tackle the aforementioned problems – and ultimately used to select an approach for this project. The following descriptions are listed in approximate chronological order.

2.2.1 NavChair

'NavChair,' developed at The University of Michigan, was one of the original semi-autonomous wheelchairs, with 'modes' for tackling commonly encountered problems.

- Hardware includes:
  - DOS-based 33 MHz 80486 computer
  - 12 ultrasonic sensors
- NavChair's three operating modes have varying levels of autonomy based on the users' needs, including general obstacle avoidance, door entry, and wall following.
- Vector field histogram (VFH) and vector force field (VFF) are employed for global navigation and obstacle avoidance. The wheelchair is 'pulled' towards the target and avoids obstacles by virtual forces, which vary inversely to object proximity. These algorithms are easily modified to include the irregular shape of the wheelchair.
- Results show algorithms are significantly slower than human users, and the VFH and VFF struggle with door passage due to the close proximity to the doorframe.

2.2.2 The Bremen Autonomous Wheelchair

The 'Bremen Autonomous Wheelchair' at the Universitt Bremen was a fully autonomous wheelchair with moderate success in environment mapping, path planning, and obstacle avoidance [3]. Important takeaways are:

- Hardware setup includes:
  - A SICK brand light detection and ranging (LIDAR) device on both the front and back of the wheelchair for environment mapping.
  - An omnidirectional camera, which is barely used compared to the LIDARs
  - Incremental encoders attached to the drive wheels. These readings drift because of variable wheel diameter and slippage.
- Global navigation uses A* graph search, where route segments are generated from Voronoi diagrams.
- Local obstacle avoidance extends the 'Dynamic Window Approach' (DWA) to construct safe paths from circular arcs and clothoids [4]. The DWA models the vehicle and obstacles as dynamic systems, searching admissible trajectories (in this case constructed from circular arcs and clothoids) under given velocity and acceleration actuator limits.
- Models environments using topological descriptions (e.g. lines), which requires significantly less memory than metric occupancy grids. Detects features from LIDAR data using angle histograms (i.e. assumes typical rectangular environments).
- Identifies incorporating irregular shaped wheelchair into path planning algorithms as a major challenge

2.2.3 Dr. Benjamin Kuipers' Research

Dr. Benjamin Kuipers' ongoing research at the University of Michigan uses a path-planning algorithms specifically developed for the unique 'natural movement' challenge with autonomous wheelchairs.

- Hardware information is limited, but the setup includes at least:
  - One Hokuyo LIDAR mounted on the front, recording laser scans at 20Hz
  - An inertial motion unit (IMU) to measure acceleration and infer jerk
  - 2.66 GHz laptop
- Environment maps are pre-generated from simultaneous localization and mapping (SLAM)
- A majority of the work focuses on generating smooth and 'graceful' paths for dynamic environments [5]. A Lyapunov-based kinematic control law derived via singular perturbation guarantees bounds on acceleration and jerk (thus 'graceful'). A singular perturbation approach essentially decomposes the control problem into a fast and slow subsystem for the steering and forward velocity, respectively. The control law generates a velocity manifold for a given target pose, which causes any wheelchair state to converge to the target.
- An equilibrium point model-predictive controller selects target poses to avoid obstacles and quickly reach the end location are generated using [6]. The MPC approach allows optimization over a finite horizon for multiple objectives. In this case, Kuipers uses two optimization iterations – one focused more on global navigation and the other for local obstacle avoidance.
- Results show the wheelchair is able to swiftly and smoothly navigate dynamic environments near its maximum velocity while not exceeding the acceleration and jerk bounds. The kinematic control law is updated at 20 Hz and paths are regenerated at 1 Hz, though the optimal path average convergence time is less than 200 ms.

2.3 Trends in Autonomy

As evidenced by the selected platforms, various degrees of autonomy exist to meet the needs of the user. Early research employed different 'modes' to complete common tasks, such as general obstacle avoidance, door entry, and wall following [7]. However, these manually-selected operation modes do not comprehensively cover all tasks. More recent semi-autonomous systems use basic voice control and obstacle avoidance; however, voice commands provide a slow flow of information and thus take significantly longer than manual operation [8]. Other interesting developments in semi-autonomous control include force-feedback joysticks for obstacle avoidance, forecasting user intentions, and hybrid electric/manual propulsion using pushrim-activated power-assisted wheelchairs [9, 10, 11].

Most current fully autonomous systems have similar goals and employ comparable sensors (e.g. LIDAR). Thus, the primary differentiating factors are the global path planning and local obstacle avoidance algorithms. For example, the Bremen Autonomous Wheelchair uses the Dynamic Window Approach and A* for the respective tasks, whereas the NavChair uses a Vector Field Histogram for both local and global navigation. [3, 7]

2.4 Path Planning Algorithms

Kinematic global path planning algorithms are generally divided into heuristic searches, sampling based methods, and potential fields. Heuristic searches such as A*, D*, and their variants provide optimal paths, but are computationally intensive in dynamic environments because they must recompute paths – including post-processing to create an admissibly smooth trajectory [6]. Incorporating the irregular wheelchair shape would be a formidable challenge for a heuristic approach. A* is most famous for its widespread success in the 2007 DARPA Urban Challenge.

Potential field methods (e.g. Vector Field Histogram) 'push' the robot from nearby obstacles and 'pull' the robot towards the target. The field is a simple function of the environment, and the control law computes the immediate action based on the location in the field. Unlike heuristic searches and sampling based methods, it does not explicitly determine a path to the target. Thus, it is remarkably efficient at handling dynamic environments because the potential field is updated in real-time. The downside of not using an explicit path is most metrics cannot be optimality guaranteed [12]. Furthermore, the control law may produce large acceleration and jerk when the field changes with the environment, which is not only uncomfortable but also potentially outside the motor capabilities. In practical mobile robotics, the Vector Field Histogram produces path results close to optimal A* searches.

Sampling based methods such as the Rapidly Exploring Random Tree are making significant progress, but thus far are not implemented on AWCs [13]. Though RRTs are an interesting area for future research, the author decided not to begin with this method due to limited use in the mobile robotics community.

This project chooses to use the novel approach proposed by Kuipers and Park specifically formulated to provide 'graceful' robot motion with human passengers. The aforementioned results show significant promise when quickly navigating dynamic environments. A more detailed analysis of the algorithms is presented in Chapter 6.

Chapter 3

Hardware Design

3.1 Wheelchair

The test-bed used for this project is an Invacare Ranger-X electric wheelchair, shown in Figure 3.1. The original electric wheelchair consists of two 12 V 55 Ah lead-acid batteries, individual rear-wheel drive 4-pole motors, and an on-board motor controller. Two large front caster wheels allow for a wide range of mobility. The on-board controller is open-loop, and thus only regulates the power to the motors – not the speed. User input is obtained through a joystick on the armrest.

An overview of the hardware setup, including the conventional electric wheelchair and expanded robotic system, is shown in Figure 3.2. The following sections explain the hardware component choices and interfaces.

Figure 3.1. Invacare Ranger-X Electric Wheelchair

3.2 Mounting Hardware

The test-bed was modified to support the additional sensor and computer requirements. Most hardware fit on a vertical rack constructed from 80/20 framing components, shown in Figure 3.3. In order to secure the rack to the frame, custom brackets were designed and water-jetted at the Penn State Learning Factory. Finally, shelves with mounting holes for each hardware component were laser-cut from acrylic. The Arduino, xPC and ROS computers, breakout boards, and power management system were secured to the shelving.

3.3 Sensors

In order to achieve closed-loop velocity control, accurate high frequency velocity estimates are required. Rotary encoders enable highly accurate measurement of wheel position, which can be differentiated to obtain velocity. Typical vehicle encoder mounting occurs on two or four of the wheels. However, the wheelchair poses several unique hardware challenges. First, the front

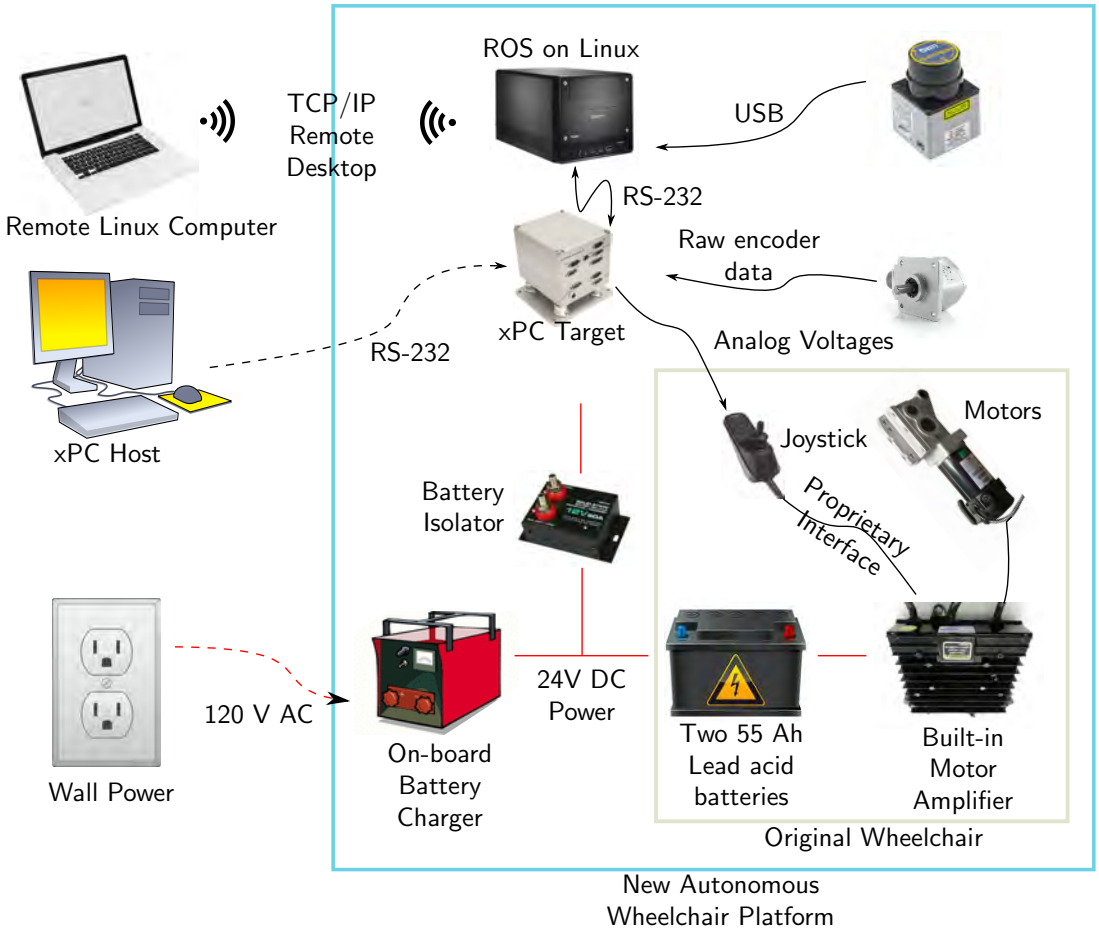

Figure 3.2. Hardware architecture overview. Dashed lines represent a temporary connection.

two caster wheels have rotational freedom in two directions – which would require four rotary encoders and two slip rings. The complexity of this setup creates additional undesirable sources of error. Secondly, the rear two wheels frequently slip or skid during rapid acceleration, as also evidenced in [3].

To avoid these problems, the encoders were attached to separate wheels directly below the drive axis. By assuming negligible sideslip of the wheelchair (a reasonable assumption as the wheelchair remains at speeds below 5 mph), one can quickly show no encoder wheel sideslip should occur since $r \times F = 0$ along the entire drive axle plane (i.e. the drive and encoder wheels are along the same axis, thus traveling forward or backward together). Further, quick-turn polyurethane wheels have minimal rotational slip and are designed for easy pivoting. Industrial grade HD25 encoders with an ample resolution of 2500 counts per revolution were used. An encoder mounting system – including spring suspension for accurate tracking over various terrains – was constructed at the Penn State Learning Factory, as seen in Figure 3.4.

In following with [5], a Hokuyo LIDAR was mounted in front of the passenger around knee-level height. Though this project does not yet incorporate environment mapping, the LIDAR will be required in future experiments.

3.4 xPC

xPC is a software environment for running Simulink models in real-time. A Simulink model is compiled into C code on an xPC host computer, and uploaded to an xPC target computer. Using the Embedded Option toolbox, the target computer can be separated from the host-computer and run in a stand-alone mode.

This project uses a fully modular PC/104 RTD Intelligent Data Acquisition Node (IDAN) system for the target computer. Two ISA boards, an Analog DAQ and an incremental encoder counter were used for all I/O needs.

Breakout boards were stacked between in-house laser-cut acrylic with engraved labels, shown in Figure 3.5. Multi-core wire, Molex Micro-Fit 3.0 connectors, and heat shrink were used to create reliable electrical connections.

3.5 Arduino

After the xPC Target computer underwent a critical hardware failure, a new system was required to read encoders, write analog voltages to the joystick, and communicate with the high-level ROS system. Arduino, a popular micro-controller, can operate as a ROS node – subscribing and publishing to topics just like other nodes. Thus, interfacing ROS with the Arduino's analog and digital I/O hardware is trivial.

Though the xPC Target computer operates extremely fast in hard real-time, the Arduino is sufficient for the needs of this project. An encoder breakout board developed by previous student Rich Mattes is employed to read the encoders.

3.6 ROS

The Ubuntu computer running Python in ROS was assembled in-house to meet design requirements. Due to the high processing demands, limited mounting space, and highly mobile environment, the computer used the following core hardware (accessories unmentioned):

- Small form factor motherboard with Intel i5 processor. The moderate amount of computing power in a small space allows the computer to fit on a single hardware shelf. A rack mount computer (comparable to a server) or laptop would have been more appropriate, but not chosen due to cost.
- 16 GB of RAM for processing the dense laser scan data.
- 128 GB solid-state hard-drive to prevent data loss during normal operating vibrations.

3.7 Power Management System

With two computers, several sensors, and motors – each with unique voltage and amperage requirements – providing safe and reliable power is essential to successful operation. An onboard Samplex 24V 25A charger connections to the wall with a Neutrik PowerCon True1 connector, which allows connection and disconnection under live load. Though the charger is capable of safely charging the battery and powering the computers at the same time, a 24V Solid-State Manson Battery Isolator is used to additionally protect the battery from overcharging while under load.

At any time, the charger can be disconnected and the system will automatically switch to battery operation without a break in computer operation. The xPC Target, motors, and onboard controller are directly powered by the 24V source. An inverter provides up to 400W of 120V AC power to the ROS computer, with ports for additional AC accessories in future upgrades.

Figure 3.3. Hardware rack shown with xPC Target computer, ROS, and power architecture

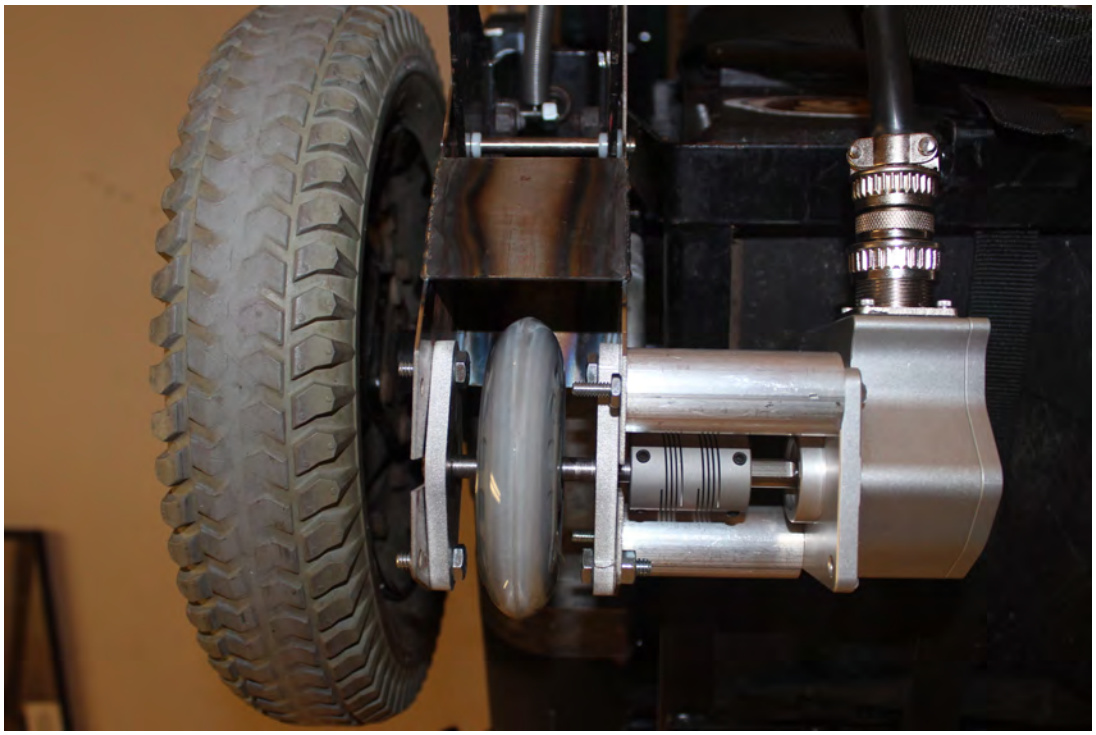

Figure 3.4. Encoder mounting system (only left wheel shown

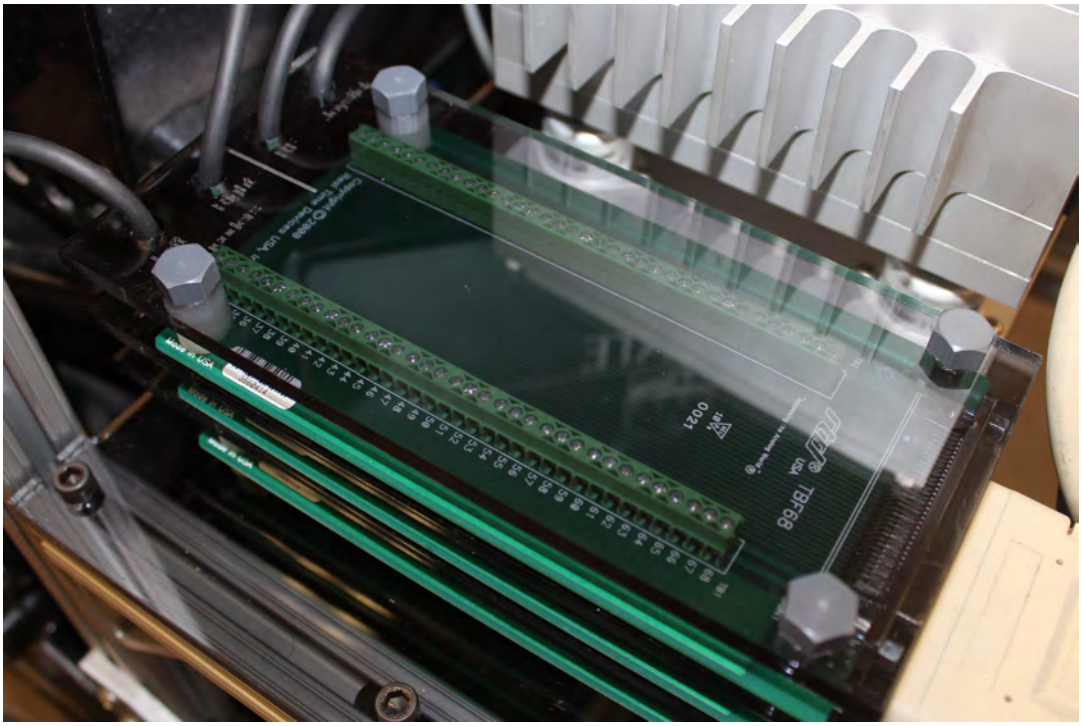

Figure 3.5. Breakout boards for the xPC Target computer

Chapter 4

Control Architecture

4.1 Introduction

The control architecture used on the robotic wheelchair is designed to handle the intended tasks while maintaing a significant degree of flexibility for future projects. Previous literature shows each control aspect demands different levels of speed and computational power [6]. Thus, the architecture is divided by the fundamental trade-off between fast and computationally intensive tasks, as seen in Figure 4.1.

On a very basic level, the motor speeds need to be quickly controlled – both to follow desired trajectories and stop in an emergency. Though the task is relatively simple, it must be executed in hard real-time and at very fast speeds. xPC Target is specifically designed for hard real-time control tasks, and the author chose an embedded xPC Target capable of real-time performance of at least 100 Hz for motor speed control, which is further discussed in Chapter 5. xPC is an extension of the MATLAB environment for implementing Simulink code on physical systems, and is widely used in industry for control applications.

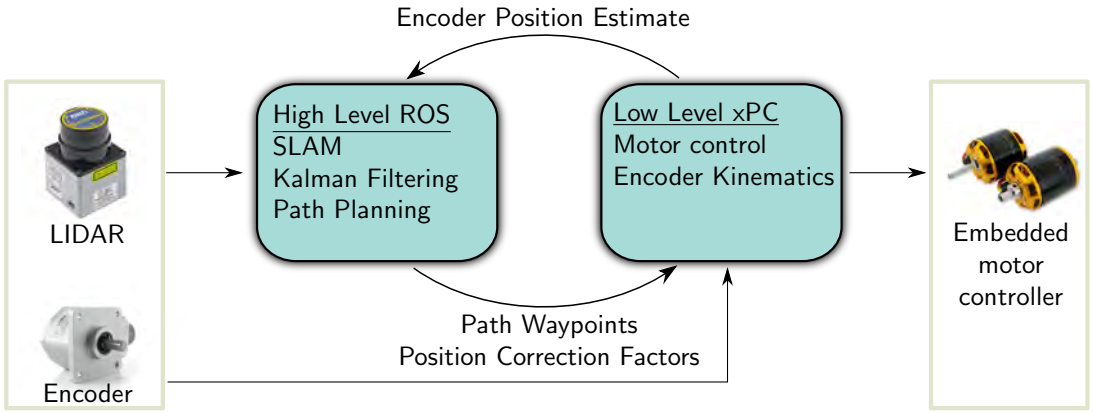

Figure 4.1. Control Architecture Overview

After a critical hardware failure onboard the xPC Target computer, the system was replaced with an Arduino running a ROS node. Though xPC is noticeably faster, the lower cost and ease of integration with ROS made Arduino a smart replacement.

On the other hand, path planning and environment mapping tasks require significantly more processing power and memory, but can operate as slow as 1 Hz [6]. Intensive computing tasks – including processing LIDAR data, environment mapping, and path planning – are designated to the high-level system. A Linux computer running the Robot Operating System (ROS) software framework was chosen for the high level systems. Widespread popularity in the robotics community, availability of a variety of libraries, and extreme flexibility make ROS a perfect choice for this platform.

ROS to xPC communication occurs via a standard RS-232 connection (a preferable Ethernet connection was not possible due to hardware limitations of the xPC) and the Arduino replacement uses standard USB protocol. Desired longitudinal and angular vehicle velocities from the kinematic control law are sent from the ROS computer to the xPC Target or Arduino. The xPC or Arduino executes the desired velocities and sends the resulting global position and heading updates back to ROS for the next kinematic control command. In this project, the xPC Target runs at 100 Hz with a connecting RS-232 baud rate of 115200 and the Arduino node operates at a maximum of approximately 40 Hz.

Chapter 5

Low Level Closed-Loop Velocity Control

5.1 Introduction

The low-level system is responsible for moving the wheelchair along the planned path by controlling motor speeds. Since the joystick accepts inputs for angular and longitudinal motor efforts, our model and controller are designed to control these metrics. Global position and heading are determined from encoders and the derived kinematic model in Section 5.2. Angular and longitudinal velocities of the wheelchair are easily determined by taking the derivative of position measurements.

Upon receiving desired velocities from the high-level system, closed-loop feedback is achieved by comparing desired angular and longitudinal velocity to measured angular and longitudinal velocities, respectively. Then, the error is translated into a motor control effort using two PID controllers, which are tuned in Section 5.3. Control effort is sent to the built-in motor controller by mimicking joystick commands using raw analog voltages in Section 5.4. The resulting estimated global position is reported back to ROS to update the robot's state. Figure 5.1 shows the overall low-level control architecture.

All low-level controls are performed on an embedded xPC system or Arduino. Therefore, the

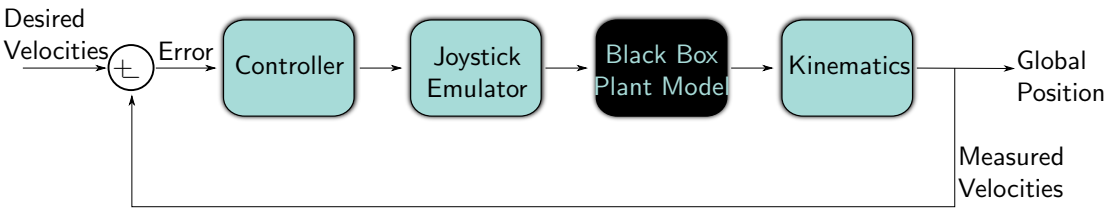

Figure 5.1. Low level control overview

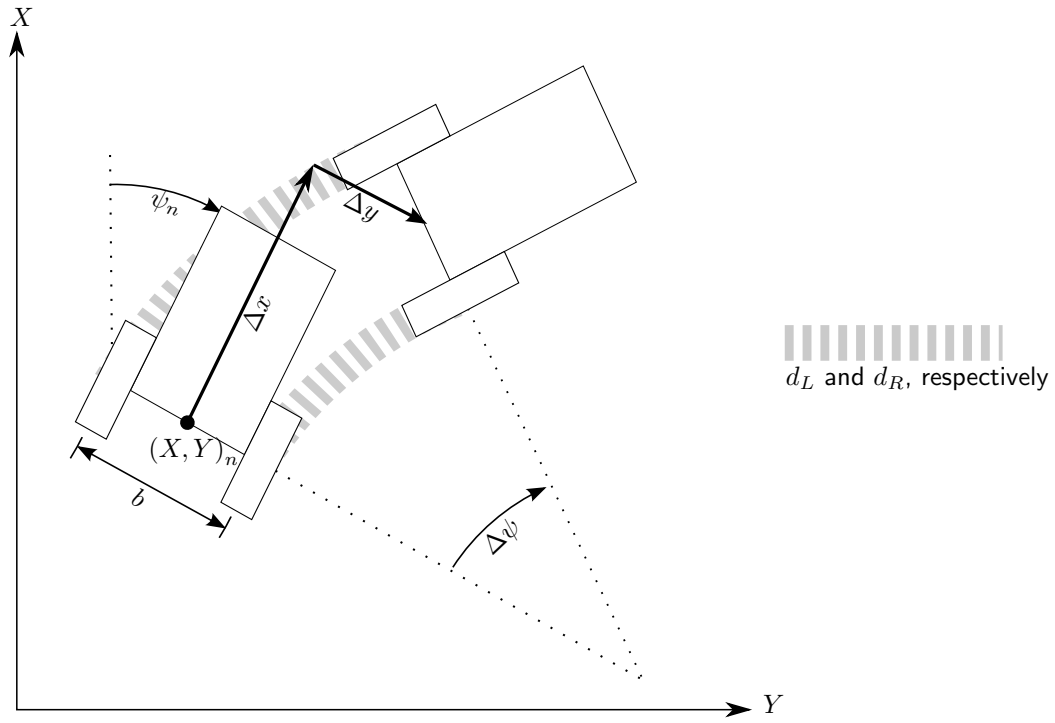

Figure 5.2. Kinematic Model

model is built entirely in the Simulink environment. Since both the xPC Target and Arduino operate in real-time, the model must be solved at fixed-step discrete time intervals, with rates of 100 and 40 Hz, respectively.

5.2 Kinematics

Since the wheelchair travels at low speeds, we assume there is no side-slip in our kinematics derivation. Any minor errors due to this assumption would eventually be corrected by Kalman filtering on the high-level ROS system. All coordinates follow the SAE convention.

We will derive the kinematics between arbitrary fixed time-step samples t_n and t_{n+1}. Figure 5.2 shows the measurements at each time-step. The variables are defined as:

α_l, α_r Angular distance the encoder shafts have rotated, respectively (radians)

b Wheelbase (m)

d_L, d_R Ground distance traveled by the left and right encoder wheels, respectively (m)

$\Delta x, \Delta y$ Change in 'local' position relative to previous position and heading (m)

ω Angular velocity of the wheelchair (rad/s)

ψ Global heading (rad)

R Turn radius (m)

$r_{enc} = 0.049$ Encoder wheel radius (m)

V Wheelchair longitudinal velocity (m/s)

X, Y Global position (m)

where $\Delta\psi \neq 0$. We will solve the case $\Delta\psi = 0$ (when the wheelchair is driving perfectly straight or not moving) later. From basic trigonometry:

$$d = \alpha r_{enc} \tag{5.1}$$

Further, one can calculate R and $\Delta\psi$ by simultaneously solving the system of equations:

$$\begin{cases} \left(R - {}^{b}/_{2}\right)\Delta\psi = d_R \\ \left(R + {}^{b}/_{2}\right)\Delta\psi = d_L \end{cases} \tag{5.2}$$

Resulting:

$$R = {}^{b}/_{2}\left(\frac{d_L + d_R}{d_L - d_R}\right) \tag{5.3}$$

$$\Delta\psi = \frac{d_L - d_R}{b} \tag{5.4}$$

At this point, calculating the angular and longitudinal velocities for closed-loop control is trivial:

$$\omega = \frac{\Delta\psi}{t_{n+1} - t_n} \tag{5.5}$$

$$V = \frac{d_L + d_R}{2} \tag{5.6}$$

Equations (5.5) and (6.7) are sufficient for velocity control, but ROS still needs the final global position for determining the next action. The incremental changes in local position relative to the previous heading are calculated as:

$$\Delta x = R\sin(\Delta\psi) \tag{5.7}$$

$$\Delta y = R\left(1 - \cos(\Delta\psi)\right) \tag{5.8}$$

Further, the small angle approximations can be applied since the model will be running at 100 Hz.

$$\Delta x = R\Delta\psi \tag{5.9}$$

$$\Delta y = 0.5R\Delta\psi^2 \tag{5.10}$$

For the case when $\Delta\psi = 0$, the wheelchair must be stationary or traveling straight. Thus,

$$\Delta x = d_L = d_R \tag{5.11}$$

$$\Delta y = 0 \tag{5.12}$$

when $\Delta\psi = 0$. Incremental changes in local position can be transformed to global coordinates by:

$$\psi_{n+1} = \psi_n + \Delta\psi = \sum_{i=1}^{n+1} \psi_i \tag{5.13}$$

$$\begin{bmatrix} \Delta X \\ \Delta Y \end{bmatrix} = \begin{bmatrix} \cos(\psi_n) & \sin(\psi_n) \\ -\sin(\psi_n) & \cos(\psi_n) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{5.14}$$

By combining (5.9), (5.10), and (5.14), the final kinematics are:

$$\psi_{n+1} = \sum_{i=1}^{n+1} \psi_i \tag{5.15}$$

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \begin{bmatrix} \cos(\psi_n) & \sin(\psi_n) \\ -\sin(\psi_n) & \cos(\psi_n) \end{bmatrix} \begin{bmatrix} R\Delta\psi^* \\ 0.5R\Delta\psi^{2**} \end{bmatrix} + \begin{bmatrix} X_n \\ Y_n \end{bmatrix} \tag{5.16}$$

where R and $\Delta\psi$ are defined in (5.3) and (5.4), respectively, and $R\Delta\psi^* = d_L = d_R$ and $0.5R\Delta\psi^{2**} = 0$ when $\Delta\psi = 0$, as shown in (5.11) and (5.12).

Complete Simulink diagrams of the kinematics are in Appendix A. Several other subsystems are required for practical application, such as the 16-bit overflow protection subsystem to correctly calculate $\Delta\alpha$ when the unsigned 16-bit encoder count integer overflows. The wrapping subsystem is a modification of the modulo operator to keep psi bounded by $(-\pi, \pi]$. Implementation of the kinematics for Arduino are in Appendix B.

5.3 PID Control

Advanced controllers (e.g. linear quadratic regulators, model predictive controllers) require an accurate model of the 'plant' (i.e. the motors) to tune parameters. Without motor models, we must use a simpler controller and tune parameters online. Thus, the most accurate choice is undoubtedly a classic PID (proportional integral derivative) controller [14]. We implement two PID controllers – one for angular velocity and one for longitudinal velocity – by comparing the desired velocities to the desired velocities and translating this into a motor control effort. The output effort is further saturated at the limits of the joystick and integration windup is prevented using the standard clamping anti-windup method in Simulink.

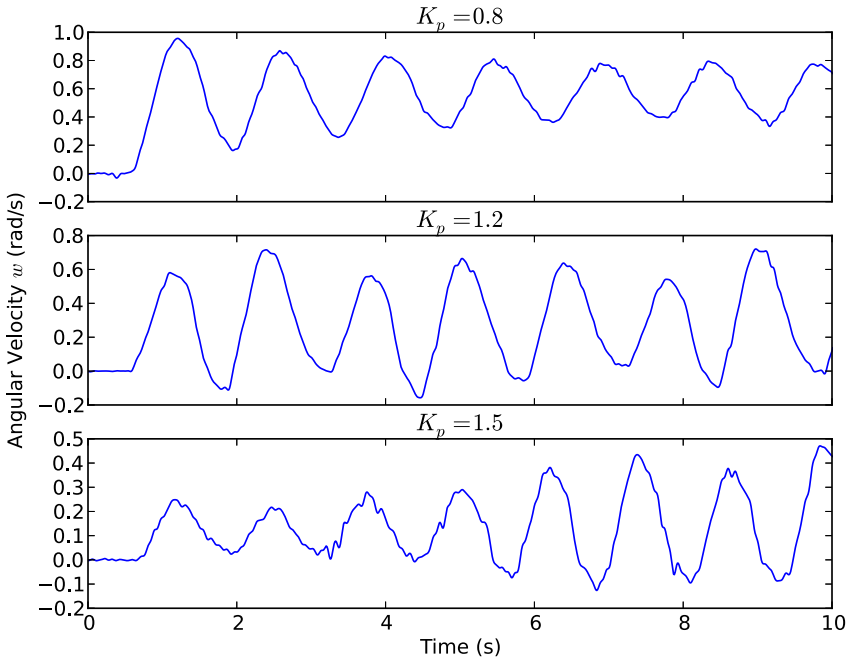

Figure 5.3. Ziegler-Nichols tuning of the angular PID controller

5.3.1 Gain tuning

The accurate yet aggressive online Ziegler-Nichols tuning method is used to determine proportional gain K_d, integral gain K_i, and derivative gain K_d values. The process can be summarized as:

1. Set $K_i = K_d = 0$ and vary K_d, observing the oscillating system response.
2. Adjust K_d until oscillation magnitudes reach an equilibrium. If oscillations decay increase K_d and vice versa.
3. Record the current proportional gain value as K_u and period as T_u. Plug these values into the Ziegler-Nichols equations to determine K_p, K_i, and K_d.

Ziegler-Nichols Gain	Angular	Longitudinal
$K_p = 0.6K_u$	0.72	0.90
$K_i = 2K_p/K_u$	$1.08(1/s)$	$1.03(1/s)$
$K_d = K_pT_u/8$	$0.12(s)$	$0.20(s)$

Table 5.1. PID gain values from Ziegler-Nichols tuning

The results of tuning the angular PID controller are shown in Figure 5.3. Note $K_p = 0.8$ is too low, as evidenced by the decaying oscillations. On the other hand, $K_p = 1.5$ is too high, but a value of 1.2 is almost perfect. The values from the angular and longitudinal (not shown) PID tests were plugged into the Ziegler-Nichols equations resulting in the values shown in Table 5.3.

Upon implementing the Ziegler-Nichols gain values, basic tests were conducted to confirm adequate system response. Results are provided in Chapter 8.

5.4 Joystick Emulator

To interface with the built-in motor controller and amplifier, the closed-loop PID control effort must be translated into joystick commands. For the Invacare Ranger-X wheelchair, joystick commands are analog voltages as seen in Figure 5.4. Voltages scale linearly such that the center stationary position is $2.5V$.

Control effort C can easily be translated into raw analog voltages V with the equation:

$$V = 1.5C + 2.5 \tag{5.17}$$

Simulink diagrams of the joystick emulator are in Appendix A.

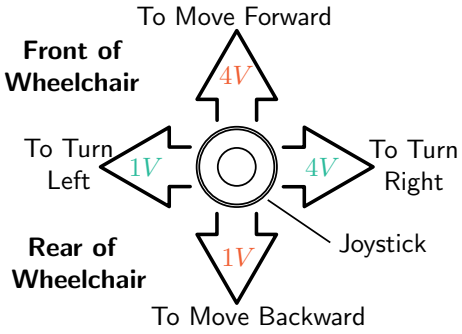

Figure 5.4. Joystick Control Emulator

Chapter 6

High Level Planning

6.1 Introduction

At a higher level, the autonomous system must decide which motor velocities will optimally achieve goals of fast, safe, and natural navigation to an end location. In following with [5] and [6], this projects breaks the problem into two parts: (1) the development of a smooth control law to for local navigation and (2) the development of a model-predictive controller (MPC) for globally optimal pose selection. For this thesis, only the local control law is implemented and the MPC algorithm is left for a future project.

6.2 Smooth Local Control Law

Unlike most field robotic applications, autonomous wheelchairs must execute comfortable and smooth motions for the human passenger, quantified by acceleration and jerk. A heuristic search (e.g. A*) does not hold guarantees on these values, and smoothing algorithms only further intensify the computationally intensive algorithm in dynamic environments. Potential field methods, though more appropriate for dynamic environments, are not designed to provide movements 'natural' to a human passenger. Thus, a kinematic control law developed by Park and Kuipers specifically for autonomous wheelchairs is used [5].

The Lyapunov-based feedback control law generates a velocity manifold for a given target pose. A significant advantage of planning a manifold instead of a path is no re-planning is required when the wheelchair deviates from the original trajectory. The velocity manifold covers the entire pose space and is only a function of the target pose – not the current wheelchair state. Updating the velocity manifold as the target pose changes requires minimal computational power.

The results of the kinematic control law can be summarized by the egocentric coordinate system depicted in Figure 6.1 and corresponding control law for angular velocity w and longitudinal velocity V in (6.4) - (6.7). The coordinate system completely describes the relation between the

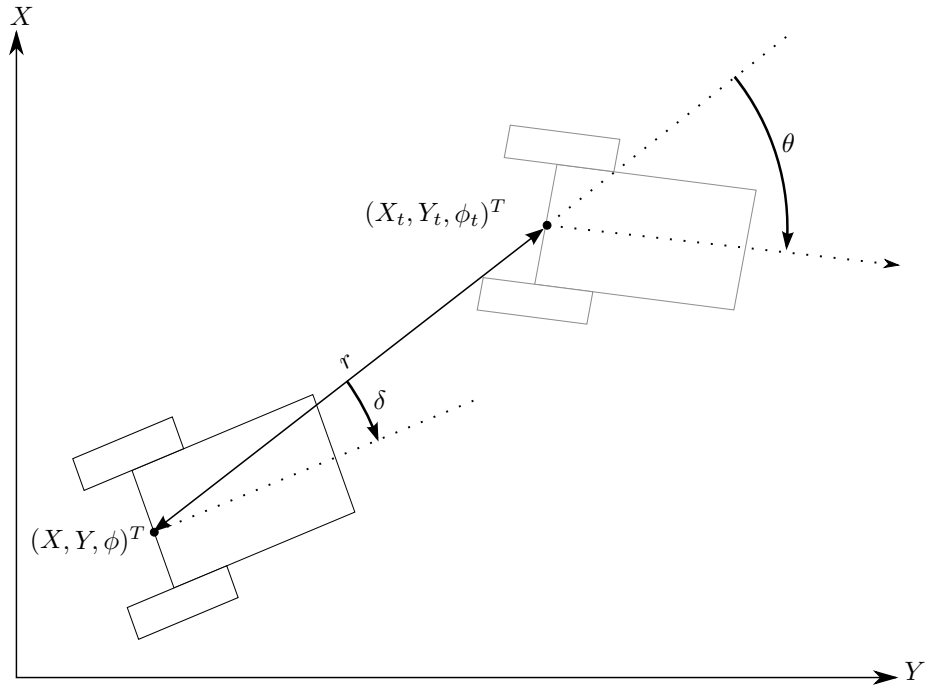

Figure 6.1. Low level control overview

current wheelchair pose $(X, Y, \phi)^T$ and target pose $(X_t, Y_t, \phi_t)^T$ by $(r, \theta, \delta)^T$ where:

$$r = \sqrt{(X - X_t)^2 + (Y - Y_t)^2} \tag{6.1}$$

$$\theta = \phi_t - \arctan\left(\frac{Y_t - Y}{X_t - X}\right) \tag{6.2}$$

$$\delta = \phi - \arctan\left(\frac{Y_t - Y}{X_t - X}\right) \tag{6.3}$$

For the sake of brevity, the derivation of the kinematic control law is not repeated as it is readily available in [5]. Important results to note are the path curvature κ is described for any target pose by:

$$\kappa = -\frac{1}{r}\left[k_2(\delta - \arctan(-k_1\theta)) + (1 + \frac{k_1}{1 + (k_1\theta)^2 \sin\delta}\right] \tag{6.4}$$

where parameters k_1 and k_2 are gain values shaping the manifold. The angular and longitudinal velocities are related by the path curvature such that:

$$\omega = \kappa V \tag{6.5}$$

Interestingly, this means the longitudinal velocity V is free, and any positive value will cause the wheelchair to converge to the target. The velocity is chosen by:

$$V(\kappa) = \frac{V_{max}}{1 + \beta|\kappa|^\lambda} \tag{6.6}$$

where V_{max} is the maximum velocity and parameters β and λ slow the wheelchair during tight turns. Further, the wheelchair must slow down and ultimately stop at the final target, so a slowdown rule is imposed:

$$V = \min\left(\frac{V_{max}}{r_{thresh}} r, V(\kappa)\right) \tag{6.7}$$

where r_{thresh} is the distance away when the wheelchair begins slowing down. The same gain values, maximum velocity, and thresholds are used as [5]. These values and complete Python code for implementing the control algorithm are in Appendix B

Chapter 7

Environment Mapping and Remote Operation in ROS

7.1 Introduction

Autonomous systems require a representation of the surrounding environment (e.g. a map) to make an informed decision regarding path planning. Most commonly, the map is preprogrammed or generated in real-time using a LIDAR, time-of-flight camera, or product similar to the Microsoft Kinect. For this project, a small Hokuyo LIDAR was already available and easily mounted to the front of the wheelchair.

To test the mapping capability, the wheelchair was setup for manual operation over a local wireless network. By connecting a laptop to the wireless network, the mapping can be observed in real-time. Furthermore, a video-game controller connected to the laptop can be used to manually remote control the wheelchair for mapping.

7.2 Integration with ROS

Several ROS libraries are available for interfacing with LIDAR and joystick hardware, transforming coordinate spaces, and visualizing results. For demonstration purposes, the mapping simply uses the LIDAR point cloud without any filtering, so coding requirements beyond the libraries is minimal.

Figure 7.1. Topic graph

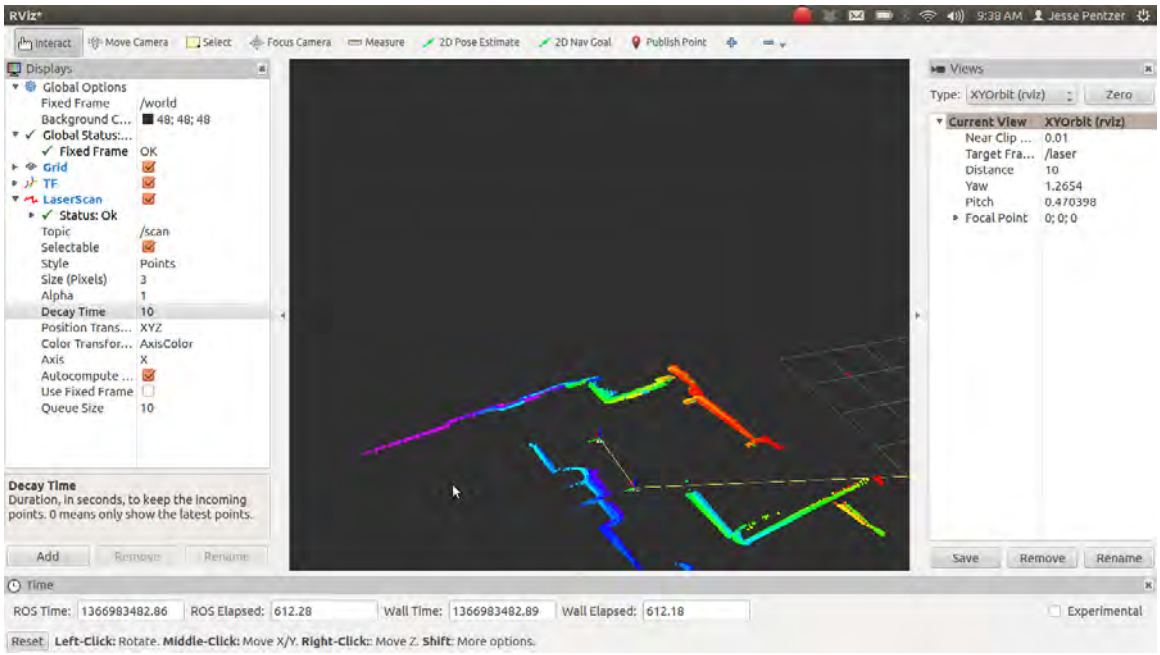

Figure 7.2. Environment mapping visualization in rviz

The 3D ROS visualization tool rviz subscribes to the laser scan, wheelchair pose, and transform topics. Basically, the transforms relate the wheelchair pose to a global 'zero' and the LIDAR frame of reference to the wheelchair pose. Thus, each laser scan can be visualized in the global frame, as seen in 7.2. Joystick commands are easily integrated using the joy_node library for ROS.

The code can be summarized by graphing the flow of topics, subscribers, and publishers using #rosrun rqt_graph rqt_graph, as seen in Figure 7.1. Thorough code documentation of the joystick mapping and transforms is provided in Appendix B.

Chapter 8

Results and Conclusions

This project successfully creates a testbed for indoor robotic experiments – including full sensor instrumentation, computing hardware, and a power-management system. Basic control algorithms are applied to achieve closed-loop velocity control with adequate system response performance.

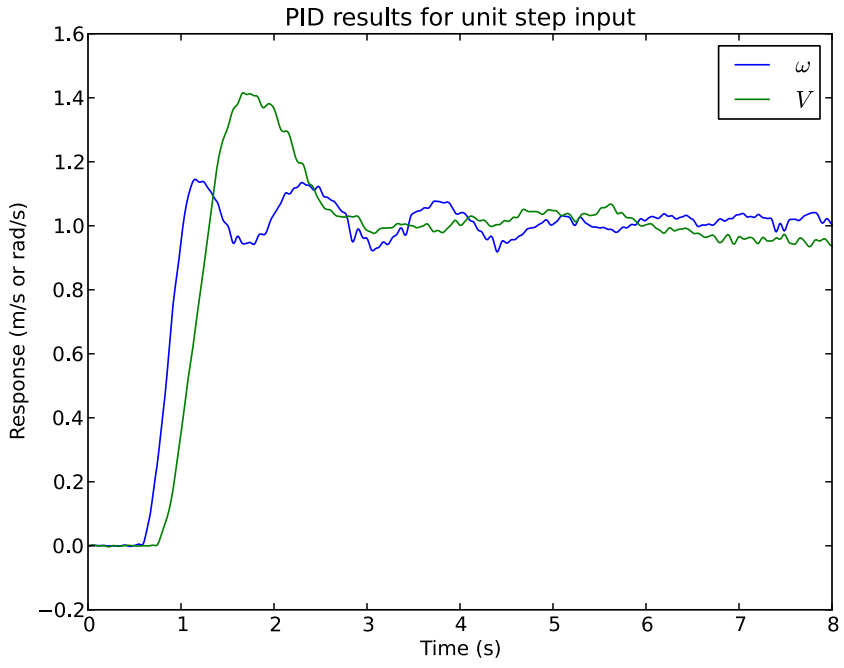

Figure 8.1. Step response of the angular and longitudinal velocity PID controllers

8.1 PID Results

Upon calculating the closed-loop velocity PID gain values determined in Chapter 5, basic tests were conducted to observe the controlled system response. Velocity and longitudinal velocity step impulses were chosen to quickly determine whether the Ziegler-Nichols method was correctly implemented. Figure 8.1 shows rise time is very fast, though overshoot is somewhat large – confirming the aggressiveness of the Ziegler-Nichols method.

8.2 Kinematic Control Law Results

Initial qualitative tests of the kinematic control law showed natural motion to target poses. Some problems arose regarding oscillations about the theoretical trajectory through the velocity manifold. Unfortunately, parameter tuning and data collection were not finished at the time of this submission. In the future, the author intends to finish implementing the smooth kinematic control law. Python code used for demonstrating the kinematic control law is available in Appendix B.

8.3 Environment Mapping

Environment mapping capabilities are demonstrated on an office hallway, as shown in Figure 8.2. Drift from encoders is extremely small, as evident by the straightness of the hallway walls and strong overlap during multiple traversals of dead-end areas. The Hokuyo LIDAR does not provide intensity values, so colors simply reflect the distance in front of the wheelchair.

8.4 Future Research

As previously mentioned in Chapter 2, several novel developments in semi-autonomous control have emerged in recent years. Wheelchair users are, like any human, a 'creature of habit' and

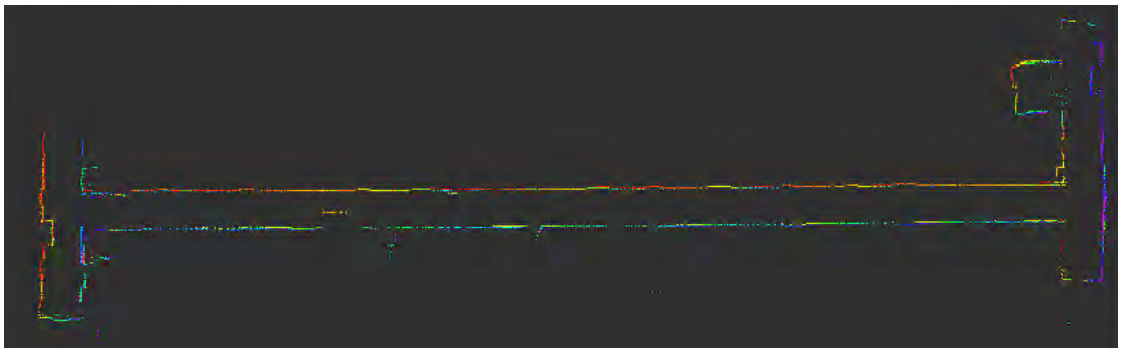

Figure 8.2. Laser point cloud from mapping an office hallway

have a finite set of goals (e.g. the bathroom, refrigerator, bedroom). To increase autonomy, these intentions can be predicted using a Partially Observable Markov Decision Process, which gradually learns users' behavior. Only broad gestures (e.g. a couple joystick sweeps or head motions) may be necessary to move a user to their destination [10].

Second, pushrim-activated power-assisted wheelchairs measure users' manual torque to the wheel and magnify the force using electric hub motors [11]. This enables any combination of electric and manual operation, thus gradually and safely transitioning users from a sedentary to active lifestyle. Interesting control problems exist with push-rim wheelchairs regarding mimicking natural movements.

Appendix A

Simulink Diagrams

The following Simulink diagrams are used for low-level velocity control on the xPC Target computer. Using the embedded option toolbox, the xPC Target can be disconnected from the host computer and run in standalone mode, thus enabling these Simulink programs to be run in real-time in a completely mobile environment.

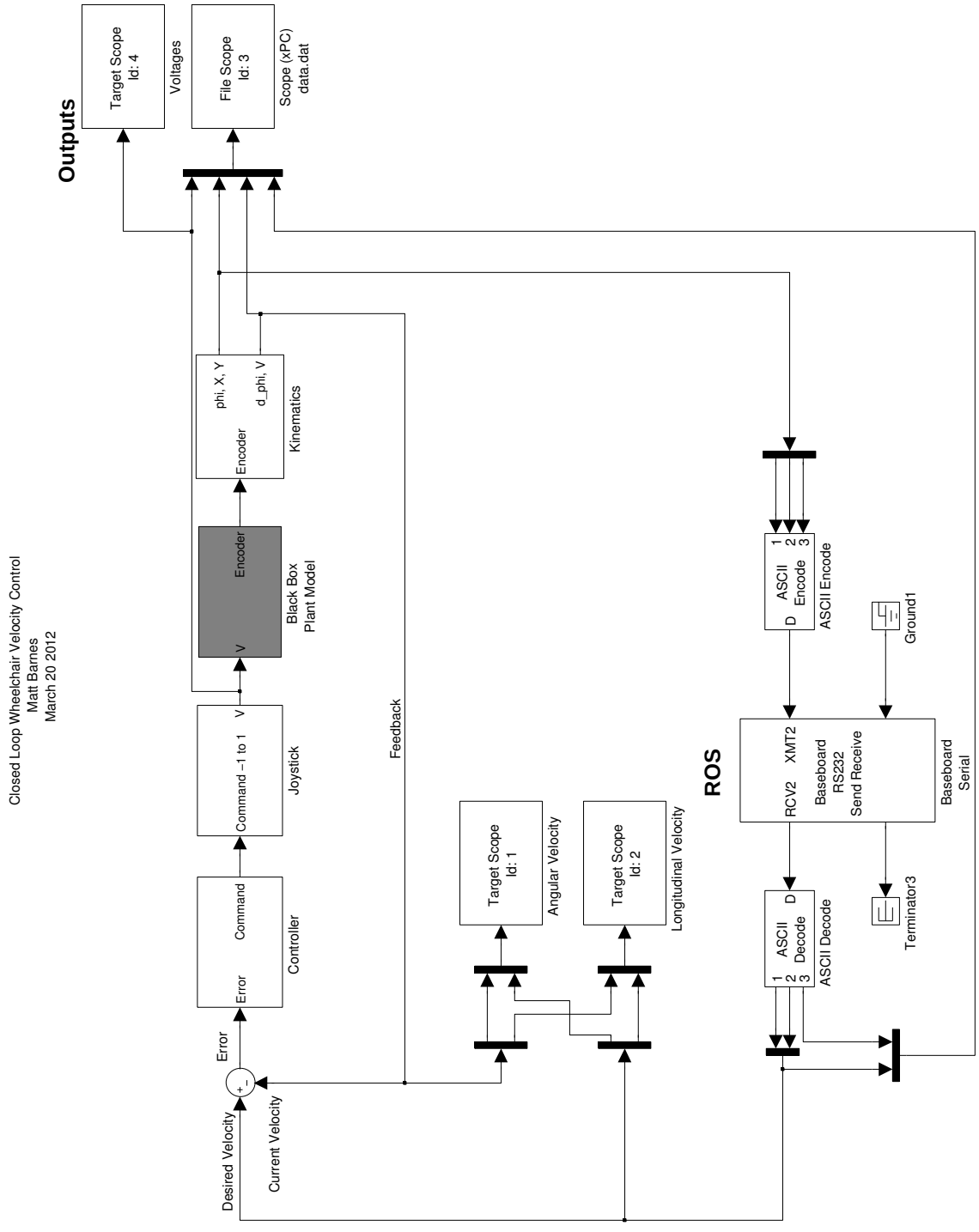

Figure A.1. Closed Loop Velocity Control Overview

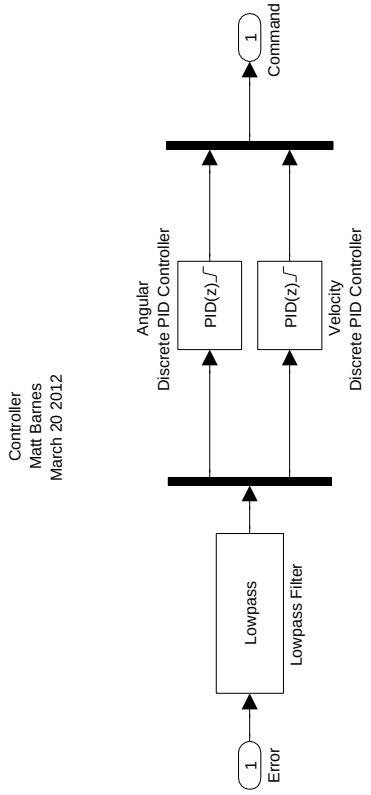

Figure A.2. Controller

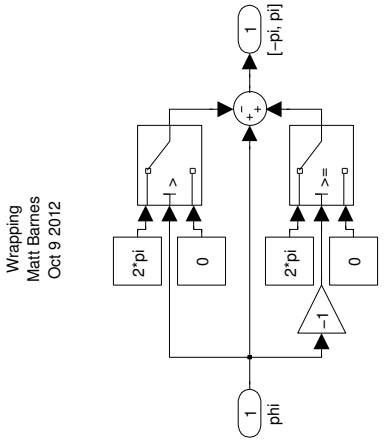

Figure A.3. Heading wrapping bounded by $(-\pi, \pi]$

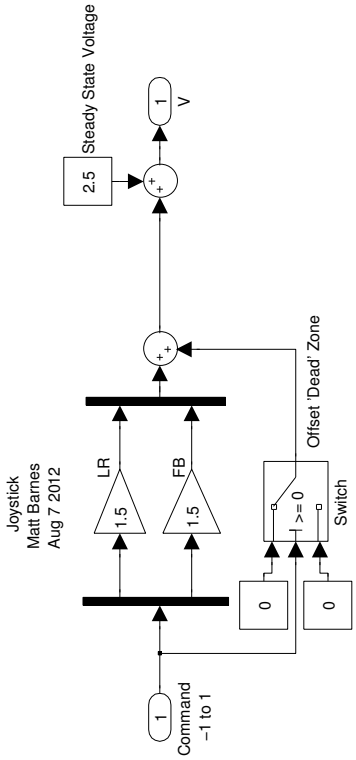

Figure A.4. Joystick emulator

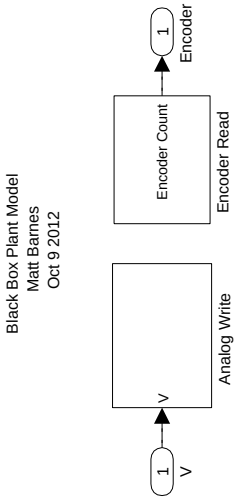

Figure A.5. Black box plant model

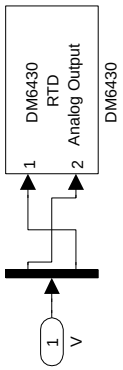

Figure A.6. Analog write for joystick commands

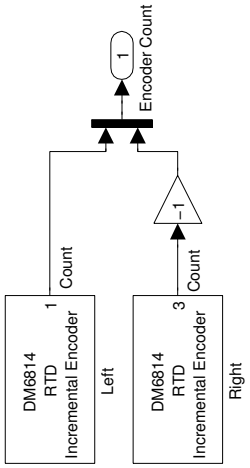

Figure A.7. Encoder read

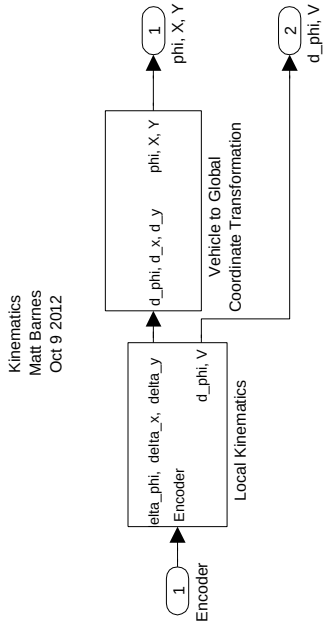

Figure A.8. Kinematics overview

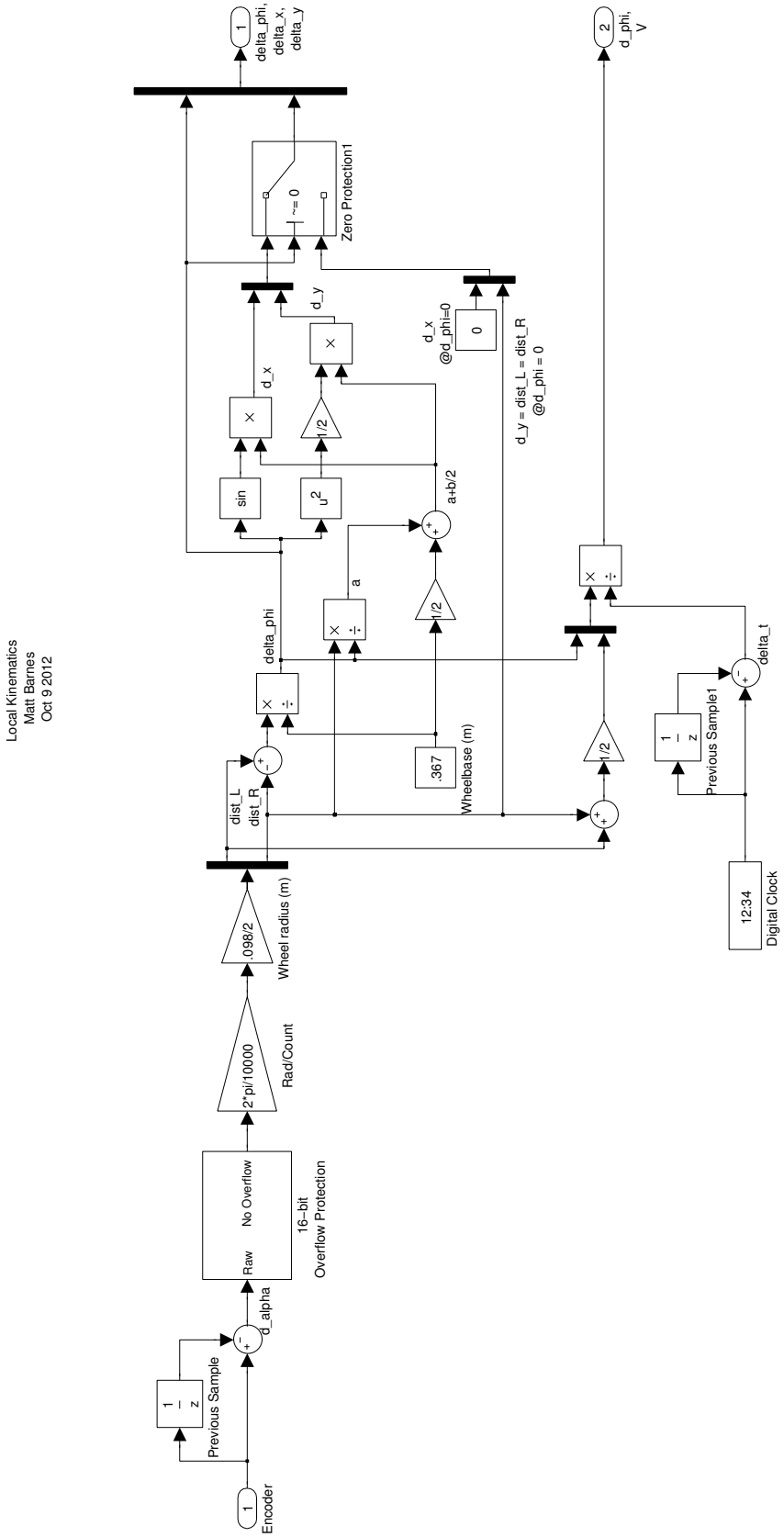

Figure A.9. Local kinematics

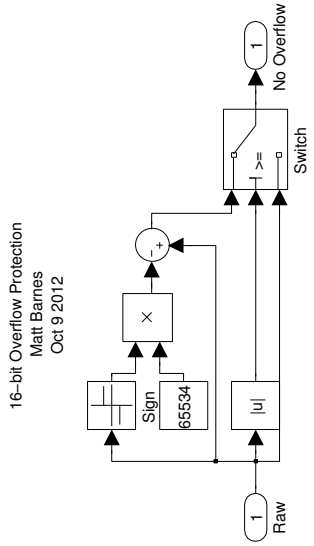

Figure A.10. 16-bit overflow protection

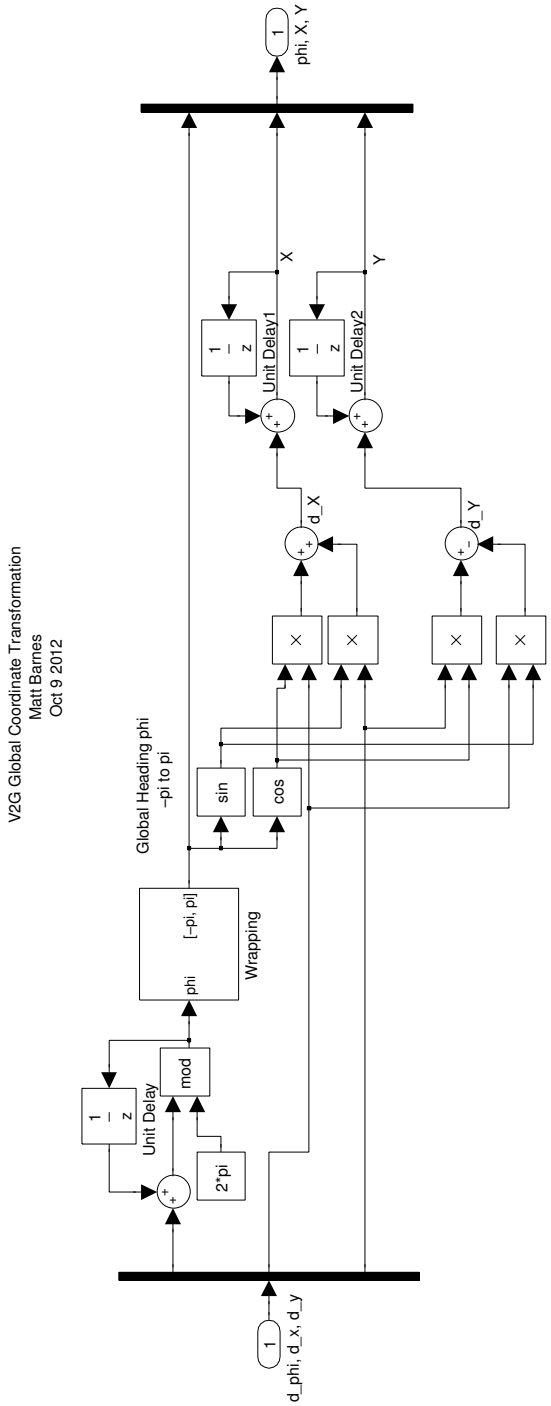

Figure A.11. Vehicle to global coordinate transformation

Appendix B

ROS Source Code

B.1 Introduction

High level controls are executed using Python code in the ROS environment, which is gaining widespread popularity in the robotics community for its open-source development and powerful libraries. Scripts (i.e. 'nodes') publish or subscribe to information (i.e. 'topics') within the framework, thus enabling convenient organization and task distribution. The large number of community supported libraries allow for rapid code development involving sensor data, SLAM, visualization, and information playback.

In following with the organization of ROS, each subsection is the code for a particular node. The Vel_Broadcaster node in Appendix B.3 implements the smooth kinematic control law from Chapter 6 and communicates with the xPC over the RS-232 serial port. The TF_Broadcaster and Joy_Broadcaster nodes in Appendices B.4 and B.5 transform the laser scan data to the wheelchair frame-of-reference and allow for manual control with a joystick, respectively. Lastly, the Arduino ROS node replaces the functionality of the xPC Target, but is able to directly publish and subscribe to ROS topics, as documented in Appendix B.6.

The code can be summarized by graphing the flow of topics, subscribers, and publishers using #rosrun rqt_graph rqt_graph, as seen in Figure B.1.

The code segments for xPC and Arduino can easily be changed to work with the other respective platform. The only difference is Arduino directly publishes and subscribes to topics, whereas the xPC requires explicit RS-232 serial communication.

Figure B.1. Topic graph

B.2 Launch File

The launch file starts all nodes and is run from the command line with #roslaunch wheelchair wheelchair.launch (where the arguments correspond to the package name and launch file name, respectively). For controlling the wheelchair from a remote computer, the joy_node would be started from the computer connected to the Xbox controller.

```
<launch>
  <node name="Joystick" pkg="joy" type="joy_node" args="_autorepeat_rate:=10" />
  <node name="Joy_Broadcaster" pkg="wheelchair" type="Joy_Broadcaster.py" />
  <node name="TF_Broadcaster" pkg="wheelchair" type="TF_Broadcaster.py" />
  <node name="Arduino" pkg="rosserial_python" type="serial_node.py" args="/dev/ttyUSB0" />
  <node name="hokuyo_node" pkg="hokuyo_node" type="hokuyo_node" />
</launch>
```

B.3 Vel_Broadcaster

The Vel_Broadcaster node is designed to communicate with the xPC Target and implements the smooth kinematic control law from Chapter 6.

```
#!/usr/bin/env python
# Vel_Broadcaster
# This node communicates with the xPC and implements the smooth kinematic control law described
#
# Matt Barnes
# April 2013

import roslib
import sys
import rospy
from numpy import *
import math
import serial
from geometry_msgs.msg import TwistStamped, PoseStamped
from std_msgs.msg import Header
import tf #ROS transformation package

class xPC_Driver:

  def __init__(self):
    #ROS publishers and messages
    self.pub_vm = rospy.Publisher('vel_meas', TwistStamped)
    self.pub_vd = rospy.Publisher('vel_des', TwistStamped)
    self.pub_pose = rospy.Publisher('pose', PoseStamped)
    self.vm_msg = TwistStamped()
    self.vd_msg = TwistStamped()
    self.pose_msg = PoseStamped()

    # Serial port communication (to/from xPC)
    self.ser = serial.Serial()
    self.ser.baudrate = 57600
    self.ser.port = '/dev/ttyUSB0'
    self.ser.timeout = 1
    self.ser.open()

```

```
36      # Constants (for control law)
37      self.k1 = 1.5
38      self.k2 = 3
39      self.B = 0.4
40      self.lam = 2
41      self.vmax = 1.2
42      self.rthres = 1.2
43
44    ## Reset the xPC ##
45    def reset(self):
46      phi = 1
47      while abs(phi)>0.05:
48        rospy.sleep(0.05)
49        data_str = self.ser.readline()
50        data = data_str.strip().split(',')
51        try:
52          phi = float(data[0])
53        except:
54          print "Warning: Serial reset packet lost"
55        self.ser.write("0"+"\t"+"0"+"\t"+"1"+"\r")
56
57    ## Stop the wheelchair ##
58    def stop(self):
59      V = phi = 1
60      while abs(phi)>0.05 or V>0.05:
61        rospy.sleep(0.05)
62        data_str = self.ser.readline()
63        data = data_str.strip().split(',')
64        try:
65          phi = float(data[0])
66          V = float(data[4])
67        except:
68          print "Warning: Serial stop packet lost"
69      self.ser.write("0"+"\t"+"0"+"\t"+"0"+"\r")
70
71    ## Move towards next target pose ##
72    def target(self, Tx, Ty, Ttheta):
73      data_str = self.ser.readline()
74      data = data_str.strip().split(',')
75      try:
```

```
76        phi = float(data[0])
77        X = float(data[1])
78        Y = -1*float(data[2])
79        w_meas = float(data[3])
80        V_meas = float(data[4])
81        t = rospy.get_time()
82        r = math.sqrt(pow(Tx - X,2) + pow(Ty - Y,2))
83        delta = math.fmod(phi - math.atan((Ty-Y)/(Tx-X))+math.pi,2*math.pi)-math.pi
84        theta = math.fmod(Ttheta - math.atan((Ty-Y)/(Tx-X))+math.pi,2*math.pi)-math.pi
85        kappa = -1/r*(self.k2*(delta - math.atan(-self.k1*theta)) + (1 + self.k1/(1 + pow((self.k
86        V = min(self.vmax/self.rthres*r, self.vmax/(1 + self.B*pow(abs(kappa),self.lam)))
87        w = V*kappa*4
88
89        # Publish results
90        self.pose_msg.header.stamp = t
91        self.pose_msg.pose.position.x = X
92        self.pose_msg.pose.position.y = -Y #SAE->ISO
93        self.pose_msg.pose.orientation = tf.transformations.quaternion_from_euler(0,0,-phi) #SAE-
94        self.pub_pose.publish(self.pose_msg)
95
96        self.vm_msg.header.stamp = t
97        self.vm_msg.twist.linear.x = V_meas
98        self.vm_msg.twist.angular.z = w_meas
99        self.pub_vm.publish(self.vm_msg)
100
101       self.vd_msg.header.stamp = t
102       self.vd_msg.twist.linear.x = V
103       self.vd_msg.twist.angular.z = w
104       self.pub_vd.publish(self.vd_msg)
105
106       # Print to serial port (to xPC)
107       self.ser.write("%.2f"%w+"\t"+"%.2f"%V+"\t"+"0"+"\r")
108       print "%.2f"%w+"\t"+"%.2f"%V+"\t"+"0"+"\r"
109     except:
110       print "Warning: Serial target packet lost"
111
112 def main(args):
113   ic = xPC_Driver()
114   rospy.init_node('Vel_Broadcaster', anonymous=True)
115
```

```
116   ic.reset()
117   ic.stop()
118   print 'xPC Reset and System Ready'
119   t0 = rospy.get_time()
120   print 'Starting in '
121   for countdown in range(5,0,-1):
122     print '\b'+str(countdown)+'...'
123     rospy.sleep(1)
124   print 'Go!'
125   t0 = rospy.get_time()
126   while not rospy.is_shutdown():
127     ic.target(10,0,0) #Desired pose (X,Y,phi)
128     rospy.sleep(0.01)
129 if __name__ == '__main__':
130   main(sys.argv)
```

B.4 TF_Broadcaster

The TF_Broadcaster node transforms the LIDAR frame of reference to the wheelchair frame of reference.

```
#!/usr/bin/env python
# TF_Broadcaster
# This node sends transformations for:
#   -The wheelchair pose relative to world
#   -The laser pose relative to the chair
#
# Matt Barnes
# April 2013

import roslib
roslib.load_manifest('wheelchair')
import rospy
import tf
from geometry_msgs.msg import PoseStamped

def handle_wheelchair_pose(msg):
    br = tf.TransformBroadcaster()
    br.sendTransform((msg.pose.position.x, -msg.pose.position.y, -msg.pose.position.z),
                     tf.transformations.quaternion_from_euler(0, 0, -msg.pose.orientation.z),
                     rospy.Time.now(),
                     "chair",
                     "world")  #changed from SAE to ISO
    br.sendTransform((.8636, .2032, .3937), #LIDAR position relative to center of wheelchair dr
                     tf.transformations.quaternion_from_euler(0, 0, 0),
                     rospy.Time.now(),
                     "laser",
                     "chair")  #ISO

if __name__ == '__main__':
    rospy.init_node('TF_Broadcaster')
    rospy.Subscriber('/pos',
                     PoseStamped,
                     handle_wheelchair_pose)
    rospy.spin()
```

B.5 Joy_Broadcaster

The Joy_Broadcaster node converts values from an Xbox 360 controller to raw analog voltages for the wheelchair joystick.

```
#!/usr/bin/env python
# Joy_Broadcaster
# This node translates Xbox 360 joystick commands to raw analog voltage values for the wheelcha
#
# Matt Barnes
# April 2013

import roslib
import sys
import rospy
from geometry_msgs.msg import Twist, PoseStamped
from sensor_msgs.msg import Joy

class Joystick_Driver:

  def callback(self,data):
    #Translate Xbox 360 joystick value to wheelchair joystick voltage
    self.vd_msg.linear.x = 2.5+1.5*data.axes[1]
    self.vd_msg.angular.z = 2.5-1.5*data.axes[0]
    self.pub_vd.publish(self.vd_msg)

  def listener(self):
    rospy.Subscriber("joy", Joy, self.callback)
    rospy.spin()

  def __init__(self):
    #ROS publishers and messages
    self.pub_vd = rospy.Publisher('vel_des', Twist)
    self.vd_msg = Twist()

def main(args):
  ic = Joystick_Driver()
  rospy.init_node('Joy_Broadcaster', anonymous=True)

  while not rospy.is_shutdown():
```

```
36      ic.listener()
37
38  if __name__ == '__main__':
39    main(sys.argv)
```

B.6 Arduino Node

The Arduino node reads encoder data, implements the kinematics from Chapter 5, publishes position and velocity information, and subscribes to the ROS control output (e.g. velocity, raw joystick voltage).

B.6.1 Arduino_ROS

The main node for the Arduino.

```
/*************************************************************
 *
 * Pennsylvania State University
 * Protected by the GNU General Public License
 *
 * This source file is developed and maintained by:
 * + Matt Barnes mjb5497@psu.edu
 *
 * File: PID_ROS.ino
 * Desc: Reads encoders, implements kinematic calculations from
 * Chapter 5, publishes pose estimate, subscribes to desired vel or
 * joystick command, writes analog voltages to joystick
 *
 *************************************************************/

#include <ros.h>  // declare the ros library
#include <geometry_msgs/PoseStamped.h>  // declare the messages library
#include <geometry_msgs/Twist.h>  // declare the messages library
#include <geometry_msgs/TwistStamped.h>  // declare the messages library
#include <std_msgs/Empty.h>

ros::NodeHandle  nh;  // initiate the ROS handle

// Create ROS objects (messages, time, publishers, and subscribers)
geometry_msgs::TwistStamped vel_meas;
geometry_msgs::PoseStamped pos;

ros::Time current_time;

ros::Publisher pub_vel("vel_meas", &vel_meas);  // initiate my publisher
ros::Publisher pub_pos("pos", &pos);  // initiate my publisher
```

```
32
33 //Function called every time new desired twist message received
34 void messageCb( const geometry_msgs::Twist& msg){
35   Joystick(msg.linear.x, msg.angular.z); //send voltage commands to joystick
36   digitalWrite(13, HIGH-digitalRead(13));   // blink the led
37 }
38
39 ros::Subscriber<geometry_msgs::Twist> sub("vel_des", messageCb );
40
41 // Constants
42 #define LeftMotor 129
43 #define RightMotor 128
44
45 #define BASE .367           // The distance between wheels in meters
46 #define R_ENC .049           // The encoder wheel radius in meters
47
48 int LeftEncoderPos;
49 int RightEncoderPos;
50
51 int OldLeftEncoderPos;
52 int OldRightEncoderPos;
53
54 // Old and new times (for derivatives)
55 unsigned long OldTime;
56 unsigned long NewTime;
57
58 float roll=0, pitch=0, yaw=0;
59
60 void setup() {
61   nh.initNode();  // initiate the ROS node
62   nh.advertise(pub_vel);  // advertise my message
63   nh.advertise(pub_pos);  // advertise my message
64   nh.subscribe(sub); //subscribe to joystick commands
65
66   SetupHardware();
67   InitializeHardware();
68 }
69
70 void loop() {
71
```

```
72   UpdateTime();
73   UpdateKinematics(); //Update position and velocities
74
75   //Publish messages
76   pub_vel.publish(&vel_meas);
77   pub_pos.publish(&pos);
78   nh.spinOnce();  // handles ROS comm callbacks
79 }
80
81
82 void UpdateTime()
83 {
84   current_time = nh.now(); //same as ros::Time::now(), but for the Arduino
85   vel_meas.header.stamp = current_time;
86   pos.header.stamp = current_time;
87 }
88 void UpdateKinematics()
89 {
90   // Old Encoder Positions
91   OldLeftEncoderPos = LeftEncoderPos;
92   OldRightEncoderPos = RightEncoderPos;
93
94   // Old and new times
95   OldTime = NewTime;
96   NewTime = micros();
97
98   // Change in encoder positions
99   UpdateEncoders(); // New encoder positions
100  long DeltaLeftEncoder = -LeftEncoderPos + OldLeftEncoderPos;
101  long DeltaRightEncoder = RightEncoderPos - OldRightEncoderPos;
102
103  // Account for 16-bit overflow
104  if(abs(DeltaLeftEncoder)>32768)
105  {
106    DeltaLeftEncoder = DeltaLeftEncoder-65536*sgn(DeltaLeftEncoder);
107  }
108  if(abs(DeltaRightEncoder)>32768)
109  {
110    DeltaRightEncoder = DeltaRightEncoder-65536*sgn(DeltaRightEncoder);
111  }
```

```
112
113   // Calculate distances
114   float d_left = DeltaLeftEncoder*R_ENC*2*3.14159/10000; // radius * rad per count
115   float d_right = DeltaRightEncoder*R_ENC*2*3.14159/10000; // radius * rad per count
116
117   // Turn radius and travel angle
118   float R = BASE/2*(d_left+d_right)/(d_left - d_right);
119   float d_phi = (d_left - d_right)/BASE;
120
121   // Angular and linear velocities
122   vel_meas.twist.linear.x = 0.5*(d_left + d_right)/((float)(NewTime - OldTime)/1000000);
123   vel_meas.twist.angular.z = d_phi/((float)(NewTime - OldTime)/1000000);
124
125   // Kinematics
126   float Rstar1, Rstar2;
127   if(d_phi == 0)
128   {
129     Rstar1 = d_left;
130     Rstar2 = 0;
131   }
132   else
133   {
134     Rstar1 = R*d_phi;
135     Rstar2 = 0.5*R*pow(d_phi,2);
136   }
137   pos.pose.position.x = cos(yaw)*Rstar1 + sin(yaw)*Rstar2 + pos.pose.position.x;
138   pos.pose.position.y = sin(yaw)*Rstar1 - cos(yaw)*Rstar2 + pos.pose.position.y; //why are the
139   pos.pose.position.z = 0;
140
141   yaw = yaw + d_phi;
142   // Yaw bounded by (-pi, pi]
143   if(abs(yaw)>3.14159)
144   {
145     yaw = yaw - 2*3.14159*sgn(yaw);
146   }
147   FromEulerAngles(roll, pitch, yaw);
148 }
149
150 static long sgn(long val) {
151   if (val < 0) return -1;
```

```
152   if (val==0) return 0;
153   return 1;
154 }
155
156 void FromEulerAngles(float roll, float pitch, float yaw)
157 {
158   // Assuming the angles are in radians.
159   double c1 = cos(yaw * 0.5);
160   double s1 = sin(yaw * 0.5);
161   double c2 = cos(pitch * 0.5);
162   double s2 = sin(pitch * 0.5);
163   double c3 = cos(roll * 0.5);
164   double s3 = sin(roll * 0.5);
165   double c1c2 = c1 * c2;
166   double s1s2 = s1 * s2;
167   pos.pose.orientation.w = 0; //(float)(c1c2 * c3 - s1s2 * s3);
168   pos.pose.orientation.x = roll; //(float)(c1c2 * s3 + s1s2 * c3);
169   pos.pose.orientation.y = pitch; //(float)(s1 * c2 * c3 + c1 * s2 * s3);
170   pos.pose.orientation.z = yaw; //(float)(c1 * s2 * c3 - s1 * c2 * s3);
171 }
172
173 /*
174 To read this in ROS, use the following commands:
175 roscore
176 --Open new tab
177 rosrun rosserial_python serial_node.py /[your serial port, most likely dev/ttyUSB0]
178 --Open new tab
179 rostopic echo pos
180 */
```

B.6.2 Hardware

Hardware related functions developed by Rich Mattes.

```
/*************************************************************
 *
 * Pennsylvania State University - Robotics Club
 * Learn more at www.psurobotics.org
 * Protected by the GNU General Public License
 *
 * This source file is developed and maintained by:
 * + Rich Mattes rjm5066@psu.edu
 *
 * Modified by:
 * + Matt Barnes mjb5497@psu.edu
 *
 * File: Hardware.pde
 * Desc: Provides the hardware-related functions for the Mechbot
 * platform.
 *
 *************************************************************/

// Pin Definitions

// Analog I/O for joystick
#define FBPin 2
#define LRPin 3

// Encoder specific defines.
#define ShiftClk 23
#define ShiftEnable 22
#define ShiftLatch 25

#define LeftEncoderData 24
#define RightEncoderData 29

#define EncoderSelect1 26
#define EncoderSelect2 27
#define EncoderOutputEnable 28

```

```
37 // Loop time for calculating velocity
38 #define DELAY_TIME_VELOCITY 10 // In ms
39 #define DELAY_TIME_SHIFTREG 4  // In us
40
41 // Define physical constants of the wheel for wheel encoding
42 #define WHEEL_CIRCUMFERENCE 0.314 // In meters
43 #define WHEEL_TICKS 8192          // The number of 'ticks' for a full wheel cycle
44 #define WHEEL_DIST .235           // The distance between wheels in meters
45
46 // "Memory" for integral and derivative terms
47 double LeftIntegral = 0;
48 double RightIntegral = 0;
49 double LastLeftError = 0;
50 double LastRightError = 0;
51 double LdVal;
52 double RdVal;
53
54 // Initialize all hardware components
55 void SetupHardware()
56 {
57
58   pinMode(FBPin, OUTPUT);   // sets the pin as output
59   pinMode(LRPin, OUTPUT);   // sets the pin as output
60
61   // Set the correct input/output modes for the encoder pins.
62   pinMode(LeftEncoderData, INPUT);
63   pinMode(RightEncoderData, INPUT);
64
65   pinMode(ShiftEnable, OUTPUT);
66   pinMode(ShiftLatch, OUTPUT);
67   pinMode(EncoderOutputEnable, OUTPUT);
68   pinMode(ShiftClk, OUTPUT);
69   pinMode(EncoderSelect1, OUTPUT);
70   pinMode(EncoderSelect2, OUTPUT);
71
72   digitalWrite(ShiftClk, LOW);
73
74 }
75
76 void InitializeHardware()
```

```
77 {
78   UpdateEncoders();
79 }
80
81 void Joystick(float fb, float lr) //Input fb and lr as voltages
82 {
83   analogWrite(FBPin, fb*255/5);
84   analogWrite(LRPin, lr*255/5);
85 }
86
87 // Gets Encoder data.  Pass two unsigned longs into this function
88 // and they'll be updated with the current values.
89 void UpdateEncoders()
90 {
91   // Clear the current readings
92   LeftEncoderPos = 0;
93   RightEncoderPos = 0;
94
95   int S1, S2;
96   // Cycle through all 4 registers (for all 32 bits of information)
97   for (int i = 0; i < 4; i++)
98   {
99     // Set the register to read on the encoder chips.
100    digitalWrite(EncoderOutputEnable, HIGH);
101    delayMicroseconds(DELAY_TIME_SHIFTREG);
102    if (i == 0){
103      S1 = 1;
104      S2 = 0;
105    }
106    else if (i == 1){
107      S1 = 0;
108      S2 = 0;
109    }
110    else if (i == 2){
111      S1 = 1;
112      S2 = 1;
113    }
114    else{
115      S1 = 0;
116      S2 = 1;
```

```
117     }
118     digitalWrite(EncoderSelect1, S1);
119     digitalWrite(EncoderSelect2, S2);
120     delayMicroseconds(DELAY_TIME_SHIFTREG);
121     digitalWrite(EncoderOutputEnable, LOW);
122     delayMicroseconds(DELAY_TIME_SHIFTREG);
123
124     // Cycle the Shift Registers to grab a value from the Encoder chips
125     digitalWrite(ShiftLatch, LOW);
126     delayMicroseconds(DELAY_TIME_SHIFTREG);
127     digitalWrite(ShiftLatch, HIGH);
128     delayMicroseconds(DELAY_TIME_SHIFTREG);
129
130     // Read the bytes from the shift registers.
131     byte LeftTmp = 0;
132     byte RightTmp = 0;
133
134     // Write the clock enable pin low so we can clock the data out.
135     digitalWrite(ShiftEnable, LOW);
136     delayMicroseconds(DELAY_TIME_SHIFTREG);
137     // Grab all 8 bits from shift register
138     for (int j = 7; j>=0; j--)
139     {
140       LeftTmp |= digitalRead(LeftEncoderData) << j;
141       RightTmp |= digitalRead(RightEncoderData) << j;
142       digitalWrite(ShiftClk, HIGH);
143       delayMicroseconds(DELAY_TIME_SHIFTREG);
144       digitalWrite(ShiftClk, LOW);
145     }
146     // Combine the four bytes into a unsigned long
147     LeftEncoderPos |= LeftTmp << (i*8);
148     RightEncoderPos |= RightTmp << (i*8);
149   }
150 }
```

Bibliography

[1] M. L. Jones and J. A. Sanford, "People with mobility impairments in the United States today and in 2010.," *Assistive technology : the official journal of RESNA*, vol. 8, pp. 43–53, Jan. 1996.

[2] D. Ding and R. Cooper, "Electric powered wheelchairs," *IEEE Control Systems Magazine*, vol. 25, pp. 22–34, Apr. 2005.

[3] C. Mandel, K. Huebner, and T. Vierhuff, "Towards an autonomous wheelchair: Cognitive aspects in service robotics," *... of Towards Autonomous Robotic ...*, 2005.

[4] D. Fox, *The dynamic window approach to collision avoidance*. Bonn: Sekretariat fur Forschungsberichte Inst. fur Informatik III, 1995.

[5] J. J. Park and B. Kuipers, "A smooth control law for graceful motion of differential wheeled mobile robots in 2D environment," in *2011 IEEE International Conference on Robotics and Automation*, pp. 4896–4902, IEEE, May 2011.

[6] J. Park, "Robot navigation with model predictive equilibrium point control," *IEEE International Conference on Intelligent Robots and Systems*, pp. 4945 – 4952, 2012.

[7] S. P. Levine, D. a. Bell, L. a. Jaros, R. C. Simpson, Y. Koren, and J. Borenstein, "The NavChair Assistive Wheelchair Navigation System.," *IEEE transactions on rehabilitation engineering : a publication of the IEEE Engineering in Medicine and Biology Society*, vol. 7, pp. 443–51, Dec. 1999.

[8] R. C. Simpson and S. P. Levine, "Voice control of a powered wheelchair.," *IEEE transactions on neural systems and rehabilitation engineering : a publication of the IEEE Engineering in Medicine and Biology Society*, vol. 10, pp. 122–5, June 2002.

[9] A. Fattouh, M. Sahnoun, and G. Bourhis, "Force feedback joystick control of a powered wheelchair: preliminary study," in *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, vol. 3, pp. 2640–2645, IEEE.

[10] T. Taha, J. V. Miro, and G. Dissanayake, "POMDP-based long-term user intention prediction for wheelchair navigation," in *2008 IEEE International Conference on Robotics and Automation*, pp. 3920–3925, IEEE, May 2008.

[11] R. Cooper, T. Corfman, S. Fitzgerald, M. Boninger, D. Spaeth, W. Ammer, and J. Arva, "Performance assessment of a pushrim-activated power-assisted wheelchair control system," *IEEE Transactions on Control Systems Technology*, vol. 10, no. 1, pp. 121–126, 2002.

[12] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, pp. 278–288, June 1991.

[13] S. M. Lavalle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," 1998.

[14] S. Bennett, *A history of control engineering, 1930-1955*. Stevenage Herts. U.K.: P. Peregrinus on behalf of the Institution of Electrical Engineers London, 1993.