Development of an Autonomous Omnidirectional Hazardous Material Handling Robot

by

Nicholas Charabaruk, B.Eng., EIT

Bachelor of Engineering (Hons), University of Ontario Institute of Technology, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

The Faculty of Engineering and Applied Science

Mechanical Engineering

Supervisor(s): Scott Nokleby, Faculty of Engineering and Applied Science
Examining Board: Remon Pop-Iliev, Faculty of Engineering and Applied Science
External Examiner: Andrew Hogue, Faculty of Business and Information Technology

THE UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

December 2015

© Nicholas Charabaruk, B.Eng., EIT 2015
Abstract

This thesis describes the prototyping and testing of an autonomous omnidirectional robot, called the OmniMaxbot. The OmniMaxbot is designed to carry cans of uranium ash at Cameco Corporation’s Port Hope Conversion Facility while minimizing worker exposure. The robot makes use of the Robot Operating System (ROS) to allow the individual components to communicate as well as control the movement. In the course of this work, the OmniMaxbot’s power distribution system was redesigned. The software for much of the hardware was developed, in part or in whole, in order to ensure the safe autonomous functioning of the system. Due to the excessive wheel slip caused by the OmniMaxbot’s Mecanum wheels, laser scan matching was implemented to generate odometry data, as opposed to using encoder data. Different Simultaneous Localization and Mapping (SLAM) and autonomous navigation packages were tested, with the ROS hector_mapping, global_planner, and base_local_planner packages being selected for use. Configuration settings were determined that produced the best performance results. A variety of tests were performed to ensure that individual hardware, subsystems, and the full system worked as required. The results showed that the OmniMaxbot is capable of autonomously navigating to a pick-up location, picking up a mock ash can, navigating to a drop-off location, putting the can down, and returning to a standby location, all in an area with dynamic and static obstacles, without collisions.
To my family and friends.
Acknowledgements

I would like to thank my supervisor, Dr. Scott Nokleby. Without his support and advice this work would never have been completed.

I also need to thank my colleagues in the Mechatronic and Robotic Systems Laboratory. Their help and companionship has been invaluable. Special mention must go to Tony Baltovski for helping with my innumerable programming problems and Jordan Gilbert for being there to bounce ideas off of.

Finally, I must also thank Cameco Corporation for their financial support of this research.
Table of Contents

Abstract ii

Acknowledgements iv

Table of Contents x

List of Tables xi

List of Figures xiii

Acronyms xiv

1 Introduction 1

1.1 Background ........................................................................... 2
  1.1.1 Uranium Production ......................................................... 2
  1.1.2 Literature Review ........................................................... 4
1.2 Problem Statement and Goal .................................................. 10
1.3 Scope .................................................................................. 15
1.4 Summary of Contributions ................................................... 15
1.5 Thesis Outline ................................................................. 16

2 OmniMaxbot System Requirements, Design, and Overview 17

2.1 System Requirements ....................................................... 17
2.1.1 Physical Requirements ........................................... 17
2.1.2 Functional Requirements ....................................... 18
2.1.3 Assumptions .......................................................... 19
2.2 System Design .......................................................... 19
  2.2.1 Mechanical Design ................................................ 19
  2.2.2 Electrical Design .................................................. 19
  2.2.3 Control and Software Design .................................... 23
2.3 System Overview ..................................................... 24
2.4 Summary ................................................................. 27

3 Hardware ................................................................. 30
  3.1 Integrated Computer ................................................ 30
  3.2 Sensors ................................................................. 31
    3.2.1 SICK LMS100 Laser Range Finders ........................ 32
    3.2.2 Xbox Kinect ...................................................... 33
    3.2.3 Bumblebee®2 Stereo Vision Camera ....................... 33
    3.2.4 Sparkfun SEN-10724 Inertial Measurement Unit ........ 34
  3.3 Motors and Encoders ............................................... 34
    3.3.1 NPC T74 ......................................................... 34
    3.3.2 Phidgets 3270 .................................................. 34
  3.4 Motor Drivers ........................................................ 35
    3.4.1 AX2850 .......................................................... 35
    3.4.2 Phidgets 1065 .................................................. 36
  3.5 Additional Elements ............................................... 36
    3.5.1 Mecanum Wheels ............................................... 36
    3.5.2 Frame ............................................................ 37
    3.5.3 Forks ............................................................. 37
  3.6 Summary ............................................................... 37
4 Software and Control

4.1 Terminology .................................................. 38

4.1.1 Node ...................................................... 39

4.1.2 Package .................................................. 39

4.1.3 Topic ...................................................... 39

4.1.4 Message .................................................. 39

4.1.4.1 geometry_msgs/Pose .................................. 40

4.1.4.2 geometry_msgs/Pose2D ................................ 40

4.1.4.3 geometry_msgs/PoseArray ............................. 40

4.1.4.4 geometry_msgs/PoseStamped .......................... 40

4.1.4.5 geometry_msgs/PoseWithCovariance .................. 40

4.1.4.6 geometry_msgs/PoseWithCovarianceStamped ............ 41

4.1.4.7 geometry_msgs/Quaternion ............................ 41

4.1.4.8 geometry_msgs/TransformStamped ..................... 41

4.1.4.9 geometry_msgs/Twist .................................. 41

4.1.4.10 geometry_msgs/TwistWithCovariance ................ 41

4.1.4.11 geometry_msgs/Vector3 ................................ 42

4.1.4.12 geometry_msgs/Vector3Stamped ....................... 42

4.1.4.13 nav_msgs/MapMetaData ............................... 42

4.1.4.14 nav_msgs/OccupancyGrid ............................. 42

4.1.4.15 nav_msgs/Odometry .................................. 42

4.1.4.16 sensor_msgs/CameraInfo .............................. 43

4.1.4.17 sensor_msgs/Image .................................... 43

4.1.4.18 sensor_msgs/Imu ...................................... 44

4.1.4.19 sensor_msgs/LaserScan ............................... 44

4.1.4.20 std_msgs/Float32 .................................... 44

4.1.4.21 std_msgs/Bool ........................................ 44
5 Testing and Results

5.1 Parts Testing ............................................. 81

5.2 Subsystems .............................................. 82
    5.2.1 Emergency Stop .................................. 82
    5.2.2 Deadman Switch ................................. 83
    5.2.3 Forklift Test ................................... 83

5.3 Navigation Tests ....................................... 84
    5.3.1 Kinematics ...................................... 84
    5.3.2 Wheel Odometry Test ........................... 84
    5.3.3 Laser Odometry Test ............................ 88
    5.3.4 Simple Navigation Test ......................... 93
    5.3.5 Static Obstacle Test ........................... 93
    5.3.6 Dynamic Obstacle Test .......................... 94

5.4 Integrated System Tests ............................... 94
    5.4.1 Simplified Ash Can Retrieval Test ............. 95
    5.4.2 Complex Ash Can Retrieval Test ............... 96
    5.4.3 Full System Test ................................ 97
    5.4.4 Summary ........................................ 122
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Power Requirements by Component</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>NPC T74 Dynamometer Results [31]</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>ROS Package/Node Summary</td>
<td>80</td>
</tr>
<tr>
<td>5.1</td>
<td>Emergency Stop Subsystem Test</td>
<td>107</td>
</tr>
<tr>
<td>5.2</td>
<td>Deadman Switch Test</td>
<td>108</td>
</tr>
<tr>
<td>5.3</td>
<td>Forklift Subsystem Test</td>
<td>109</td>
</tr>
<tr>
<td>5.4</td>
<td>Kinematics Test</td>
<td>110</td>
</tr>
<tr>
<td>5.5</td>
<td>Wheel Odometry Test</td>
<td>111</td>
</tr>
<tr>
<td>5.6</td>
<td>Laser Odometry Test</td>
<td>112</td>
</tr>
<tr>
<td>5.7</td>
<td>Simple Navigation Test</td>
<td>113</td>
</tr>
<tr>
<td>5.8</td>
<td>Static Obstacle Test</td>
<td>115</td>
</tr>
<tr>
<td>5.9</td>
<td>Dynamic Obstacle Test</td>
<td>117</td>
</tr>
<tr>
<td>5.10</td>
<td>Simple Integrated Test</td>
<td>119</td>
</tr>
<tr>
<td>5.11</td>
<td>Complex Integrated Test</td>
<td>120</td>
</tr>
<tr>
<td>5.12</td>
<td>Full Integrated Test</td>
<td>121</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Clearpath Robotics’ OTTO Heavy-Load Material Transporter [16] .... 11
1.2 An Amazon Robotics Kiva robot [17] . . . . . . . . . . . . . . . . . . . 12
1.3 The original omnidirectional base developed by the first capstone group. 13
1.4 The robot base after the additions made by the second capstone group. 14

2.1 The wiring diagram of the OmniMaxbot . . . . . . . . . . . . . . . . . . 21
2.2 The OmniMaxbot as seen from opposite corners. 1: front LRF, 2: rear LRF, 3: Bumblebee2 stereovision camera, 4: Microsoft Kinect, 5: IMU, 6: onboard computer and router, 7: Mecanum wheel . . . . . . . . . . 25
2.3 The UML state diagram of the OmniMaxbot during testing . . . . . 28
2.4 The OmniMaxbot’s subsystems . . . . . . . . . . . . . . . . . . . . . . . 29

3.1 Right and left Mecanum wheels [37] . . . . . . . . . . . . . . . . . . . 37

4.1 A visual representation of Big and Little Endian storage methods [39] . 43
4.2 The transform tree of the OmniMaxbot while autonomously navigating. 46
4.3 The AR code used to detect mock ash cans and the drop-off location. 53
4.4 An occupancy grid map of the MARS Lab made using hector_mapping. White cells are unoccupied, black cells are occupied, and grey cells are unknown space. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.5 The wheel numbering used for the OmniMaxbot . . . . . . . . . . . 61
4.6 The nodes and interactions used in the move_base package [63] . . . 66
4.7 The costmaps used by move_base: a) the global costmap, b) the local costmap, c) both costmaps. ....................................................... 68
4.8 The default recovery behaviours used by the move_base package [63]. 70
4.9 A flowchart showing how omnimaxbot_control_node lines up with a mock ash can. ................................................................. 75
4.10 The node and topic interactions for the navigation sensors. ........ 77
4.11 The node and topic interactions for the control system. ............... 78
4.12 The node and topic interactions for the navigation system. ......... 79

5.1 Result of the linear odometry test in the x direction using robot_pose_ekf. The left image is the original scan. The image on the right shows the scans aggregated over 30 seconds. ............................... 86
5.2 Result of the linear odometry test in the y direction using robot_pose_ekf. The left image is the original scan. The image on the right shows the scans aggregated over 30 seconds. ............................... 87
5.3 Result of the rotational odometry test using robot_pose_ekf. The left image is the original scan, the middle image is the scan after the robot had rotated 360° degrees, and the image on the right is the scan after it had settled. ......................................................... 89
5.4 Result of the linear odometry test in the x direction. The left image is the original scan. The image on the right shows the scans aggregated over 30 seconds. ................................. 90
5.5 Result of the linear odometry test in the y direction. The left image is the original scan. The image on the right shows the scans aggregated over 30 seconds. ................................. 91
5.6 Result of the rotational odometry test. The left image is the original scan. The image on the right is the scan taken after a 360° degree rotation. ......................................................... 92
5.7 The OmniMaxbot in its starting location. . . . . . . . . . . . . . . . . 98
5.8 The OmniMaxbot approaching the mock ash can. . . . . . . . . . . . 99
5.9 The mock ash can being lifted by the OmniMaxbot. . . . . . . . . . . 100
5.10 The position of the OmniMaxbot after reversing to simulate clearing
    the flame reactor. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
5.11 The approach position for the drop-off location. . . . . . . . . . . . 102
5.12 The OmniMaxbot after lowering the mock ash can to the ground. . . 103
5.13 The position of the OmniMaxbot after reversing to ensure that the
    forks are clear of the mock ash can. . . . . . . . . . . . . . . . . . . . 104
5.14 The test stop position. . . . . . . . . . . . . . . . . . . . . . . . . . . 105
5.15 The layout of the full system test. . . . . . . . . . . . . . . . . . . . . 106
Acronyms

AGV  Autonomous Guided Vehicle.

AR  Augmented Reality.

c  Current Capacity.

CPR  Counts Per Revolution.

DOF  Degrees-of-Freedom.

EKF  Extended Kalman Filter.

FPS  Frames Per Second.

ICP  Iterative Closest Point.

IMU  Inertial Measurement Unit.

INS  Inertial Navigation System.

IR  Infra-Red.

KLD  Kullback-Leibler Distance.

LIDAR  Light Detection And Ranging.

LiPo  Lithium Polymer.
LRF  Laser Range Finder.

mAH  MilliAmp Hours.

MARS  Mechatronic and Robotic Systems.

MCL  Monte Carlo Localization.

MDV  Multi-Directional Vehicle.

MP  MegaPixels.

ms  Milliseconds.

PID  Proportional-Integral-Derivative.

PPR  Pulses Per Revolution.

PSU  Power Source Unit.

PWM  Pulse Width Modulation.

RBPF  Rao-Blackwellized Particle Filter.

RCPRG  Robot Control and Pattern Recognition Group.

RGB  Red-Green-Blue.

ROS  Robot Operating System.

RPM  Revolutions Per Minute.

RPPRG  Robot Programming and Pattern Recognition Group.

SLAM  Simultaneous Localization and Mapping.

UOIT  University of Ontario Institute of Technology.
Chapter 1

Introduction

There are a wide variety of reasons to use robots. These range from automating simple pick and place tasks, to removing workers from hazardous environments. The purpose of this project is to reduce the risk to workers of being exposed to hazardous materials. Cameco Corporation produces uranium fuel for the nuclear industry. One of the steps in processing the uranium is fluorination. In this step, uranium tetrafluoride, UF$_4$, is combined with fluorine gas, F$_2$, in a flame reactor to form uranium hexafluoride, UF$_6$. Approximately 10% of the input material is output as waste, in the form of uranium ash and fluorine residue. This ash falls to the bottom of the reactors, where it is gathered into cans. When the cans are full, workers in radiation suits manually seal the cans with a lid and transport them by forklift to a disposal location. This part of the process poses the highest risk to workers of being exposed to radioactive and toxic materials.

The OmniMaxbot has been developed to mitigate this risk. The OmniMaxbot is an autonomous, omnidirectional robot for the transportation of hazardous materials. Using sensors, software, motors, and special wheels, the OmniMaxbot can locate and move the ash cans without requiring any user input. Mecanum wheels are used to
allow the robot to move in any direction on the x-y plane, allowing the OmniMaxbot to manoeuvre in the tight spaces between the reactors.

1.1 Background

1.1.1 Uranium Production

There are many steps in the manufacturing of uranium fuel pellets. An overview of the nuclear fuel cycle can be found at [1]. The first step is to mine the uranium. Cameco runs three mines in Canada: Cigar Lake, MacArthur River, and Rabbit Lake. At the mines, the extracted uranium is mixed with water to convert it to a thick slurry for transportation to milling facilities.

At the mill, the uranium slurry is converted from uranium ore to uranium concentrate, better known as yellowcake uranium or $\text{U}_3\text{O}_8$. The first stage in the mill is to separate the uranium from the waste rock. To do this, the slurry is mixed with sulphuric acid to dissolve the uranium. The uranium, now in solution, is separated from the waste solids through a series of thickeners. The uranium solution is then sent to the solvent extraction circuit, where a series of chemical process are used to purify and concentrate the solution. The solution is then pumped into precipitation tanks and mixed with ammonia gas to separate the uranium. The now solid uranium is pumped into an oven, to be baked at 840°C, to remove any remaining impurities. The dried yellowcake uranium is then screened and poured into a hopper, before being distributed into steel drums for transport to the refinery at Blind River.

Cameco’s Blind River Refinery receives yellowcake uranium from mines all over the world. The refinery converts the yellowcake uranium, $\text{U}_3\text{O}_8$, into uranium trioxide, $\text{UO}_3$. $\text{UO}_3$ is the feedstock of the conversion process, which is the stage this project
is related to. The U₃O₈ is mixed with water and nitric acid in a digestion tank to form a slurry. The slurry is pumped into an extraction column where it is mixed with a solvent to remove the uranium. The extract is then scrubbed to remove any impurities. The solvent is then stripped from the extract to be regenerated, and the extract, now called OK liquor, is sent to the boildown area. This area consists of heat exchangers, evaporators, and entrainment separators. This process is used to concentrate the OK liquor by boiling off the water and nitric acid, leaving uranyl nitrate hexahydrate, UNH. The UNH is then heated in denitration pots to break it down into uranium trioxide, UO₃, and nitrogen oxides, NOₓ. The UO₃ is transferred into bins for shipping to Cameco’s Port Hope Conversion Facility.

The Port Hope Conversion Facility, as the name implies, converts the uranium trioxide, UO₃, into either uranium dioxide, UO₂, or uranium hexafluoride, UF₆. Cameco’s Port Hope facility is one of only two uranium conversion facilities in North America. Natural UO₂ is used as fuel for heavy water reactors, such as CANDU reactors. For light water reactors, enriched UO₂ is required. To enrich the UO₂, it must first be converted to UF₆. The first step in the conversion process is to split hydrogen fluoride, HF, into hydrogen gas, H₂, and fluorine gas, F₂. These are used later in the process. The UO₃ is mixed with the H₂ in a fluid bed reactor to form uranium dioxide, UO₂. The UO₂ is then mixed with hydrofluoric acid, 4HF, in a wet reactor to produce a UF₄ slurry. The slurry is dried to remove the water, leaving UF₄ powder. Next, the UF₄ enters a flame reactor where it is bonded with fluorine gas, in a process known as fluorination. This forms gaseous uranium hexafluoride, UF₆. The UF₆ is then cooled until it becomes liquid. It is then pumped into transport cylinders, where it is cooled and transforms to a solid. The UF₆ is then transported to enrichment facilities around
the world. The following shows a summary of the conversion process [2]

\[ UO_3 + H_2 \rightarrow UO_2 + H_2O \]  
\[ UO_2 + 4HF \rightarrow UF_4 + 2H_2O \]
\[ UF_4 + F_2 \rightarrow UF_6 \]

1.1.2 Literature Review

The idea of using autonomous vehicles is not new. One of the most common types of autonomous vehicles found in manufacturing environments are Autonomous Guided Vehicles (AGVs). Berman and Eden [3] created a decentralized control system for multiple AGVs used in material handling. Berman and Eden used fuzzy logic and behaviour based navigation to control the robots. Rather than having a map of the area, each AGV contained a list of interconnected waypoints and a set of right-of-way rules. When a workstation needed servicing, it sent a signal to a central system. The AGVs decided which workstations to service based on the distance between the individual AGV and the closest workstation requiring servicing. The fuzzy controller determined the path to take based on the shortest distance between the AGV and the workstation to be serviced based on the distance between the waypoints. If an obstacle was detected while travelling, the AGV used right-of-way rules to determine if the obstacle was another AGV, in which case it would wait for the other AGV to bypass it, or if the obstacle was a static object, in which case the AGV would bypass it.

Stentz et al. [4] developed a method of automating a robot excavator for loading trucks in mines. They mounted two Laser Range Finders (LRFs) to the excavator, one on either side of the cab. During digging, the left LRF was used to detect obstacles, as well as to detect, localize, and measure the truck. Using this information, the excavator’s control system could determine the best location in the truck bed to
drop the load. Once finished digging, the boom swung towards the truck. During this motion, the right LRF scanned the dig face. The new surface was measure so that the control system could determine the next best location to dig. As the boom moved toward the face to start digging again, the left LRF would scan the truck bed again to determine the material distribution and determine where to put the next load. This process was repeated until the truck was full.

Sugar and Kumar [5] presented a method of moving materials using a number of co-operating robots. They had robots, the number of which could be increased if needed, which could work autonomously. If a robot determined that it could not move an item on its own, it could work with other robots in order to move the item to where it needed to go. In this case one of the robots was designated as the lead, and the others were the followers. The lead planned a path from its location to the goal location. This path was sent over wireless ethernet to the follower robots. The followers then planned their own paths to the goal location while keeping in formation with the lead robot. The lead and follower robots could switch roles if needed, as in the case of an obstacle being detected in the lead robot’s path.

A great deal of research has been performed on omnidirectional robots. Dickerson and Lapin [6] compared the maneuverability of Multi-Directional Vehicles (MDVs) to conventionally steered vehicles, such as cars. Two types of MDVs were examined. The first used Mecanum wheels to achieve omnidirectional movement. The second MDV, referred to as an all-wheel steered vehicle, used four independently powered and steered wheels. It should be noted that physical vehicles were not used, the tests were performed using mathematical models. Three parameters were examined to determine maneuverability. The first was the area required for each vehicle model to make the desired motion. The second was the path length. The final parameter
was the number of singular movements. In this case, a singular movement is one in which a small movement in some direction of the overall model requires a large motion of the propulsion system. Three situations were examined: parallel docking in a recessed space, perpendicular docking in a recessed space, and a \(90^\circ\) turn. In all three cases, the MDV models required less area, shorter paths, and fewer singular motions to complete the maneuvers than the conventionally steered model. The MDV with Mecanum wheels required fewer singular motions than the all-wheel steered vehicle. This is because Mecanum wheels are non-singular. Omnidirectional motion is achieved without requiring the wheels to move in any direction other than forward or reverse.

Salih et al. [7] used a simple robot to determine the kinematics and motion characteristics of Mecanum wheels. Their robot consisted of an aluminium chassis with four independent motors, the Mecanum wheels, a four channel motor driver, a microcontroller, and a power supply. Pulse Width Modulation (PWM) was used to control the motor speeds. While the wheels were mounted in the opposite direction as those on the OmniMaxbot, making their kinematics non-applicable, their motion characteristics will be comparable. It was found that forward/backward motion and clockwise/counter-clockwise rotation worked as expected. This is because the passive rollers on the Mecanum wheels were not used when making these motions. Translation sideways however was found to be unacceptable due to the robot wandering forward and rearward. It was determined that this was because of the rollers themselves. Due to the shape of the wheels themselves, some rocking occurs. The robot’s real position and orientation was found to be different than that planned for the path. This was due to slippage. Surface contact, floor condition, and traction were determined to all influence the amount of slip. Due to this, using visual dead reckoning for odometry and motion control was recommended to help reduce the error.
The OmniMaxbot is similar to the “Andrea’s” robot platform developed by Rovetta [8]. “Andrea’s” stands for Autonomous Navigation with Dexterity and Robotic Environmental Actions System. The robot was built as part of the European Locobot project. The goal of Locobot, which stands for Low Cost Robotic Co-Workers, is to create an industrial mobile robot which is able to work safely on an assembly line alongside human workers. The “Andrea’s” platform was designed to be able to autonomously navigate, perform pick and place operations, and work around human workers without harming them. “Andrea’s”, like the OmniMaxbot, makes use of Mecanum wheels in order to manoeuvre in tight spaces. A series of Infra-Red (IR) and ultrasonic sensors are used to detect objects in the immediate vicinity of the robot. “Andrea’s” also uses the Robot Operating System (ROS) for mapping and autonomous navigation. The mapping is performed by a rotating LRF as well as a stereovision camera. These sensors are also used, along with odometry data, for the localization. In order to improve the localization, a webcam is used in addition to wheel encoders. The webcam faces the ceiling, and is used for both odometry as well as fine navigation. An Xbox Kinect is used to detect the gestures made by workers. These gestures are interpreted into commands. As the “Andrea’s” platform was designed for pick and place operations, it is equipped with a robotic arm.

Localization, the ability to determine a robot’s location on a map, is an important part of mobile robotics. Odometry found using wheel encoders is a common method of localizing robots. Due to the amount of slip generated by Mecanum wheels, odometry found using wheel encoder dead-reckoning is generally unreliable. Velagic et al. [9] localized an omnidirectional robot using dead-reckoning from wheel encoders and an Xbox Kinect. The Kinect was used to detect landmarks and calculate the distance between the landmarks and the robot. This data, as well as the odometry data calcu-
lated using the dead-reckoning, were then passed through an Extended Kalman Filter (EKF). The EKF output the robot’s estimated position on the map. The method was tested using two different paths. The localization using this method over time was plotted against the localization data found using just odometry and the desired path. It was found that this method provided far superior localization estimates than using odometry data.

Cooney, Xu, and Bright [10] used visual dead-reckoning to calculate the odometry of an omnidirectional robot using Mecanum wheels. Their system used optical mice to provide a constant stream of x and y movements. As optical mice cannot detect rotation, two were used, one mounted to the front and one mounted to the rear. By knowing the change in x and y movement and the spacing of the mice to each other, the translation and rotation were calculated. The system was tested by running the robot and comparing the displacement measured by the optical mice to that measured with a measuring tape. The difference was found to be less than 1%.

Kim et al. [11] devised an Inertial Navigation System (INS) to be used on AGVs with Mecanum wheels. Their system used encoders, an accelerometer, and a gyro sensor. The INS is composed of two steps, a slip decision step and a position correction step. In the slip decision step, the degree of slip was calculated by determining the difference between angular velocities found using the gyro sensor and the encoders. In the position correction step, the position found using the encoders and accelerometers is corrected using the degree of slip calculated in the first section. The system was tested comparing the results of the INS to the results found using laser navigation with different motions and speeds. It was found to return accurate results.

Wulf et al. [12] developed a method for localizing a robot in a factory environment
by making a 3D map of the factory’s ceiling, rather than using a planar map. Their robot used two LRFs. The first was mounted sideways to a stepper motor. The sensor was rotated to achieve a 360° scan around the robot. Due to the sensor’s scan range, this worked well for detecting directly above the robot as well as around it. A second LRF was mounted to the robot base near the floor, facing forward. This sensor was used for obstacle detection and avoidance. After building the 3D map of the ceiling, a Monte Carlo Localization (MCL) technique was used to determine the robot’s pose on the map.

Simultaneous Localization and Mapping (SLAM) was used to build the map of the OmniMaxbot’s area. SLAM is when a robot makes a map of its environment, while at the same time localizing itself within that map. This is non-trivial because for localization a consistent map is required, while for map building the robot requires a good estimate of its position. Durrant-Whyte and Baily presented two tutorials on SLAM. The first tutorial [13] explains the background of SLAM and details two of the most common methods, the EKF and Rao-Blackwellized Particle Filter (RBPF) methods. The second tutorial [14] reviews some literature on SLAM, focusing primarily on computational complexity, data association, and environmental representation.

Bailey et al. [15] examined the consistency of the EKF-SLAM algorithm. They found that this SLAM method is theoretically inconsistent at all times due to variations caused while linearising the model to fit the Kalman filter algorithm. However, this inconsistency is not always serious. The severity of the inconsistency was found to be linked to the amount of heading uncertainty. In cases where there is a small amount of heading variance, then the inconsistency grows slowly and is manageable. When there is a large amount of heading variance however, the algorithm will suddenly and catastrophically fail. Heading variance is the difference between the robot’s true
Autonomous robots are already making their way into service, primarily in the warehousing industry. Clearpath Robotics has recently announced OTTO [16], an autonomous heavy-materials transport robot. OTTO has a load capacity of 1,500 kg, max speed of 2.0 m/s, and can rotate in place. The robot has front and rear LIght Detection And Ranging (LIDAR) sensors for obstacle detection. As well, it has light and audio prompts to inform workers of its intentions. Figure 1.1 shows OTTO.

Amazon currently uses robots to work in their warehouses [17]. Amazon purchased Kiva Robotics in 2012, and now there are 50,000 Kiva robots in 10 Amazon warehouses in the United States. These robots carry shelving racks to the pickers, who pull off products to place in bins for packaging and shipping. A Kiva robot is shown in Figure 1.2.
Figure 1.2: An Amazon Robotics Kiva robot [17].
1.2 Problem Statement and Goal

During the fluorination step outlined in Section 1.1, approximately ten percent of the input material comes out as waste, in the form of uranium ash and fluorine residue. This waste falls to the bottom of the flame reactors, where it is gathered into cans. Currently, these cans are moved by workers in radiation suits driving forklifts. This step poses the highest risk to workers of being exposed to radioactive and toxic materials.

The overall goal of this project is to develop an autonomous robotic system to supplement workers in the flame reactor room of Cameco’s Port Hope Conversion Facility in an effort to keep the workers’ exposure to hazardous materials as low as reasonably achievable. This thesis is a key step in the overall development of such a system.

This thesis builds upon the work of two fourth year capstone projects at the University of Ontario Institute of Technology (UOIT). The first, titled Design and Development of a Drive-Train for an Omni-Directional Platform [18], was worked on in the 2009-2010 school year. The mobile base was designed and constructed during this project. This included purchasing the AX2850 motor drivers, NPC T74 motors, shock absorbers, encoders, Mecanum wheels, and frame material. This group assembled the frame and mounted the components. They were able to achieve manual control of the base using a joystick and open loop control. Figure 1.3 shows the base at the end of the project. In this thesis, the base was kept substantially the same, with some modifications such as the replacement of the wheel encoders.

The second capstone project, worked on in 2012-2013, was titled Design and Development of an Automated Ash Can Replacement System for UF₆ Production [19]. This project focused on adding the forklift assembly, redesigning the power distribution
system, and attempting to automate the system. Several components were purchased for this capstone project such as the Microsoft Kinect, Phidgets 1065 motor drivers, Phidgets 3270 motors, Phidgets 3531 encoders, screw shafts, and linear rails. The forklift assembly, as it appears in this thesis project, was built during this capstone project. The only change made to the forklift assembly in this thesis was the addition of a brace bar across the top of the frame. This group replaced the lead acid batteries used by the first group with Lithium Polymer (LiPo) batteries and rewired the system. The power system devised by this group was entirely redesigned as explained in Section 2.2.2, though the same batteries were used. The group’s attempt to automate the system were rudimentary and therefore disregarded. An image of the robot at this stage can be seen in Figure 1.4.

The goal of this thesis was to build upon the mechanical system to develop a fully autonomous proof-of-concept prototype. This prototype has been named the OmniMaxbot. In order to achieve this goal sensors were added to the current robot base. Hardware and sensor drivers were integrated with an overall control system in order to build a map of the OmniMaxbot’s work environment, localize the robot on said map, and autonomously navigate. The OmniMaxbot must able to navigate from its
starting location to a mock ash can, approach the can, lift it up, autonomously navigate to a drop-off location with the can, put down the can, move back from the can until the forks are clear, then autonomously navigate to a waiting location. This must all be performed without user interference and without hitting any obstacles.

1.3 Scope

The scope of this thesis involved the development and testing of a prototype autonomous material handling robot. The robot must be able to build a map and navigate around its work area in a safe manner. It must be able to travel from a start position to a simulated reactor position, pick-up a mock ash can, carry the can to a drop-off location, and make its way back to the start position. These motions must be made autonomously and without hitting any obstacles.


1.4 Summary of Contributions

The main contributions of this work are:

- Developed a proof-of-concept prototype of an autonomous omnidirectional robot
- Designed overall control strategy
- Redesigned the power distribution system
- Developed, in part or in whole, the software to interact with the hardware and control the OmniMaxbot autonomously
- Tested, selected, and integrated odometry data generation ROS packages
- Tested, selected, integrated, and configured ROS SLAM and autonomous navigation packages
- Tested hardware, subsystems, and the complete system to ensure full functionality

1.5 Thesis Outline

The remainder of this thesis is laid out as follows: Chapter 2 outlines the requirements of the overall project, an overview of the design process, and an outline of the developed system. Chapter 3 details the hardware used to construct the OmniMaxbot. The sensors, motors, motor drivers, and other physical parts of the system are explained in this chapter. In Chapter 4, the robot’s programming is explained. This chapter starts with an overview of ROS and then delves into the software used to run the robot. Chapter 5 discusses the testing the OmniMaxbot underwent. This includes describing what each test was used for, the test plans used, and the results of the tests. Finally, in Chapter 6 conclusions are drawn, possible future work is discussed, and the lessons learned are presented.
Chapter 2

OmniMaxbot System

Requirements, Design, and Overview

This chapter outlines the OmniMaxbot’s requirements, design methodologies, and a brief overview of the system developed as part of this thesis.

2.1 System Requirements

There are a variety of requirements that the OmniMaxbot must be able to fulfil in order to perform its tasks. These requirements have been divided into physical and functional requirements.

2.1.1 Physical Requirements

The following list details the physical requirements that the OmniMaxbot must fulfil:

1. The base may be no more than $1.5 \text{ m} \times 1.5 \text{ m}$
2. The OmniMaxbot must be able to safely carry loads of up to 500 kg without deformation or failure

3. The robot must have a low centre of gravity to ensure load stability

4. All four wheels must remain on the ground at all times

5. The robot must be able to keep sufficient traction on uneven surfaces

6. Components must be able to operate in adverse environmental conditions, such as areas with dust and radiation

7. The OmniMaxbot must be capable of travelling up to 1 m/s

2.1.2 Functional Requirements

The following list details the functional requirements that the OmniMaxbot must fulfil:

1. The OmniMaxbot must be able to autonomously navigate throughout its work area safely

2. The robot must be omnidirectional to fit in the tight spaces in its work area

3. The OmniMaxbot must be able to traverse an incline of 10% while carrying the maximum load

4. The robot must be able to detect ash cans

5. The OmniMaxbot must be able to pick-up mock ash cans and deliver them to a drop-off location
2.1.3 Assumptions

Some assumptions were made for the purposes of this thesis. They were:

1. Limited alterations to the ash cans and work area are permitted

2. The effects of radiation in the work area will not impact the system

3. Deviations in elevation of the work area are small enough that a 2D map is sufficient for mapping

4. The floor of the operating area is relatively even

2.2 System Design

This section details the design process used to develop this prototype. It has been divided into the major disciplines of mechatronics: mechanical, electrical, controls, and software. As controls and software are intertwined, they have been merged into a single subsection.

2.2.1 Mechanical Design

As stated in Section 1.2, the mechanical design had been completed by two prior capstone groups. Both groups followed a formal engineering design process. As this work was performed by others, it will not be reiterated here. Those interested in the mechanical design are advised to look at the capstone project reports [18, 19]. The hardware developed for the OmniMaxbot is explained in Chapter 3.

2.2.2 Electrical Design

Similar to the mechanical design, an electrical system had already been implemented for the robot base. Unlike the mechanical parts, however, the electrical system required a complete redesign. The only components kept from the capstone projects
Table 2.1: Power Requirements by Component

<table>
<thead>
<tr>
<th>Component</th>
<th>Amperage (A)</th>
<th>Voltage (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Kinect</td>
<td>0.1875</td>
<td>12</td>
</tr>
<tr>
<td>PointGrey Research BumbleBee® 2</td>
<td>0.2083</td>
<td>12</td>
</tr>
<tr>
<td>Gigabyte GA-J1900N-D3V</td>
<td>1.833</td>
<td>12</td>
</tr>
<tr>
<td>Linksys WRT54GS</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Sparkfun SEN-10724</td>
<td>6.5×10⁻³</td>
<td>3.3</td>
</tr>
<tr>
<td>Arduino Mega2560</td>
<td>6.62×10⁻⁴</td>
<td>5</td>
</tr>
<tr>
<td>NPC T74</td>
<td>1.1</td>
<td>24</td>
</tr>
<tr>
<td>AX2850</td>
<td>0.1</td>
<td>24</td>
</tr>
<tr>
<td>Phidgets 3270</td>
<td>2.2</td>
<td>24</td>
</tr>
<tr>
<td>Phidgets 1065</td>
<td>0.1</td>
<td>24</td>
</tr>
</tbody>
</table>

1: The current here was found while the OmniMaxbot was on jacks and therefore under no load, including its own weight, and running at the user specified max speed, not the actual max speed.

were the LiPo batteries.

The first step of the electrical design was to examine the hardware components and determine how they would connect to the power system. A wiring diagram was used to visually represent the power system and its connections. The wiring diagram for the OmniMaxbot is shown in Figure 2.1.

Next, the specification sheets for the components were examined to determine their theoretical peak current draw. This was used to determine how much power the system would draw when operating at the same time. Table 2.1 shows the power requirements of each component.

The power requirement for each component was determined using:

\[ P = i \times V \]  \hspace{1cm} (2.1)

where \( P \) is the power in watts, \( i \) is the current in amps, and \( V \) is the voltage in volts.
Figure 2.1: The wiring diagram of the OmniMaxbot.
By knowing the total power required by each component, the total current required from the batteries was found by setting the voltage to 24 V and rearranging Equation (2.1) to solve for current. The sum of the currents for each component was then found. After applying a 20% buffer to account for inefficiencies in the components and converting between voltages, it was found that the current draw on the batteries was 15 A. This current value was then used to determine the wire gauge, emergency stop capacity, and fuse rating required for the system.

The batteries used on the OmniMaxbot are Turnigy nano-tech LiPo [20] batteries. The batteries supply 22.2 V, have a continuous discharge rate of 20 Current Capacity (c), a burst discharge rate of 30 c, and have a charge capacity of 5,000 MilliAmp Hours (mAH). The c value is the discharge rate of the battery as a scaled value of the battery’s capacity. The maximum continuous current provided by a single battery is found using:

\[ i = c \times \text{capacity} \quad (2.2) \]

where c is the continuous discharge rate. These batteries can provide a maximum continuous current of 100 A. The running time using a single battery is found using:

\[ T = \frac{\text{capacity}}{i} \quad (2.3) \]

where T is the running time in hours. Using a single battery, the OmniMaxbot could run for 20 minutes. In order to increase the running time, multiple batteries are connected in series. This increases the overall capacity of the system while maintaining the voltage. The new running time was found using:

\[ T = \frac{\sum_{n=1}^{\text{#of batteries}} \text{capacity}}{i} \quad (2.4) \]
Using 8 batteries increases the running time of the OmniMaxbot to 2 hours and 40 minutes.

Once the calculations were performed, the power distribution system was then assembled based on the wiring diagram. During testing, the OmniMaxbot ran for approximately 3 hours between requiring charges. The motors were not running the entire time which could account for the robot running longer than the calculated running time. The discrepancy could also be attribute to the 20% buffer estimate being too large.

2.2.3 Control and Software Design

A substantial amount of software is required to operate the OmniMaxbot. This software needed to be selected, tested, configured, and in some cases modified or created. A decision was made at the start of the project to use the Robot Operating System (ROS) as much as possible. ROS is a flexible framework for programming robots [21]. ROS provides libraries and tools for a wide variety of applications. As it is open source, many research groups contribute to the overall ROS community. This is useful as robot developers do not need to reinvent the wheel. Unfortunately, the quality of the documentation can vary widely between developers.

One of the project constraints was to use as many standard ROS packages as possible. This was to eliminate redeveloping software that had already been extensively developed for different applications. In cases where there were existing ROS packages that perform similar tasks to those required but not exactly the same, the packages were modified to work for this system. In some cases, there were no existing packages to work from. In these situations new packages were created for this thesis. Chapter 4 details the packages used for the OmniMaxbot, including the rationale for the choice
2.3 System Overview

The OmniMaxbot is an autonomous omnidirectional robot for use in handling and transporting radioactive materials. In order to achieve this goal, a variety of hardware and software is required. The OmniMaxbot makes use of various sensors and motors in order to move around its environment safely. The OmniMaxbot must be able to navigate from its start location to a pick-up location, pick-up a mock ash can, take the can to a drop-off location, put the can down, and then return to its start position. Figure 2.2 shows the completed OmniMaxbot, with the major components indicated.

Setting up and using the OmniMaxbot must be performed in two stages. The first stage involves building a map of the work environment. To do this, the robot uses distance measurements from the front LRF. By knowing the distance and angle of objects relative to the position of the LRF, the software determines where on the map objects are located. The OmniMaxbot determines its own location on the map at
any point by comparing current scans to previous scans and determining the rotation and translation between them. The OmniMaxbot is carefully directed about the area using teleoperation to build the map.

Once the map is built, the pick-up, drop-off, and stop locations are determined. These are found by manually driving the OmniMaxbot into these positions, using teleoperation, and checking the pose estimates. The pose is found for each location and are hardcoded into the control node.

When testing the OmniMaxbot, the robot autonomously navigates from its current location to the pick-up location. This position is actually set approximately 1 metre from the mock ash can. This is to provide clearance, assuring that the forks will not hit the can. The can has an Augmented Reality (AR) code attached to it. A camera detects the code and determines the can’s position and orientation relative to the camera. It was determined that modifying the mock ash cans with AR codes was a simpler way of detecting the mock ash cans and determining their position relative to the robot than using object recognition and measurement using a stereovision camera. Using the position and orientation data, the OmniMaxbot corrects its approach to the mock ash can, allowing it to move into position to lift the mock ash can without hitting it. Once the robot is in position, the forks are raised. The forks are attached to screw drives. As the motors turn, the screws rotate, causing the forks to raise or lower. The forks are mounted to linear rails in order to keep them level.

Once the mock ash can is supported by the forks, the OmniMaxbot autonomously navigates to the drop-off location. Again, this location is offset from the actual drop-off position. A second AR code is mounted to the wall behind the drop-off location. This code is detected by another camera, mounted to the top of the robot. Using
the position and orientation data, the robot is directed to the drop-off position. Once there, the forks are lowered until the can is on the ground. The OmniMaxbot then reverses until the forks are clear of the can. This is determined using the AR code on the can. The OmniMaxbot reverses until the system determines that it is 90 cm away, far enough to rotate in place without hitting the can with the forks. At this point the OmniMaxbot drives to the stop location and returns the forks to their original starting height.

Figure 2.3 shows the UML state diagram of the OmniMaxbot during testing. The state diagram shows the possible states that the robot could be in, the possible changes between states, and what causes these changes.

An image of the subsystems and associated hardware for the OmniMaxbot is shown
in Figure 2.4. This figure focuses on the hardware subsystems. The software systems are shown in Figures 4.10, 4.11, and 4.12 of Chapter 4.

2.4 Summary

This chapter outlined the requirements of the OmniMaxbot system. The system’s design process was also explained and an overview of the system was presented. The next chapter will provide details about the OmniMaxbot’s hardware.
Figure 2.4: The OmniMaxbot's subsystems.
Chapter 3

Hardware

This chapter explains the physical components of the OmniMaxbot and how they work. The computer, sensors, and power system, other than the batteries, were selected and added to the system as part of this thesis. The rest of the hardware was selected and assembled by the capstone groups that started the project. The software required to integrate the hardware with the rest of the system is described in Chapter 4.

3.1 Integrated Computer

An integrated computer is was purchased as part of this thesis to control the OmniMaxbot. This computer has a Gigabyte J1900N-D3V motherboard with an onboard Intel Celeron J1900 2.0 GHz processor [22] running Ubuntu 12.04 Precise Pangolin. The primary reason that this board was selected is due to the number and types of inputs. The motherboard has two LAN, six USB, and two serial ports. This is useful as the AX2550 motor controllers work best when connected to serial ports, the Phidgets drivers connect by USB, as does the Microsoft Kinect. The motherboard is also very durable, with better humidity, electrostatic, temperature, and power failure protections than most other motherboards. It is a mini-ITX board, so it can fit into a
smaller case, useful as it is mounted to the OmniMaxbot. The computer has 4 GB of RAM. An Intel Dual Band Wireless-N 7260 [23] wireless and Bluetooth card is used to connect the controller to the lab network. This is required so that software can be updated and to download ROS packages. This card is wireless 802.11a/g/n certified and can connect to networks on both the 2.4 and 5 GHz bands. A firewire card has also been installed on the motherboard. This card is used to connect to and power the Bumblebee® 2 camera that is described in Section 3.2.3.

A Linksys WRT-54GS wireless router is used to connect the two SICK LMS100 LRFs to the integrated computer, as well as to connect the integrated and workstation computers over a wireless network. The router is required for two reasons. First, the LRFs connect by LAN. Secondly, Ubuntu does not allow for simultaneous use of wireless and LAN.

A picoPSU-90 [24] is used to power the computer, Kinect, and the router. The picoPSU-90 takes in the 12 V DC power output from the DC-DC converter and distributes it to the motherboard and peripherals. This is an extremely small Power Source Unit (PSU), making it ideal for this application. The picoPSU-90 is connected to a DC-DC converter [25]. This device takes in 6-34 V DC power and steps it up or down to between 5-24 V based on the user’s settings. In this case, the default value of 12 V is all that is required. Using this converter allows the computer and peripherals to be connected to the OmniMaxbot’s onboard power system.

### 3.2 Sensors

Sensors are an important part of this system. They are what allow the robot to perceive its environment. A variety of sensors are used by the OmniMaxbot. This
section explains the principles behind the sensors, what they are used for, and why they were selected.

### 3.2.1 SICK LMS100 Laser Range Finders

The OmniMaxbot uses two SICK LMS100 LRFs [26]. LRFs work by emitting a laser beam at a rotating mirror [27]. The laser beam reflects off of the mirror and out of the sensor. If the laser beam hits an object, it is reflected back to the LRF, where it is detected by a receiver. The time between the emission of the laser beam and its reflection being received is recorded. As the speed of light is a constant, the distance between the LRF and a detected object is found using:

\[ d = \frac{ct}{2} \]  

(3.1)

where \( d \) is the distance to the object, \( c \) is the speed of light, and \( t \) is the time. The distance is divided by two as the time recorded is the period of time the light needed to travel to the object and back.

The advantages to LRFs is that they are highly accurate, have good resolution, and can work at a high frequency. Their downsides are that they are expensive, can miss narrow objects, and may fail to detect transparent objects. As well, LRFs scan along a plane, and will therefore miss any objects that are above or below that plane.

The OmniMaxbot uses two LRFs. One faces forward while the other faces rearward. This is because the LMS100s have a scanning angle of 270°. By using two LRFs facing opposite directions, a 360° scanning angle can be achieved. Unfortunately, due to the locations of the sensors in respect to the fork assembly on the robot, a portion of the scan is still occluded and ignored. The LMS100s have an angular resolution of
0.5°/0.25°, a frequency of 25 Hz/50 Hz, and an effective range of 20 m. The LMS100 has a variety of filters, including a particle filter. This is useful as it can filter out interference caused by particles such as dust.

### 3.2.2 Xbox Kinect

An Xbox Kinect is used to detect objects to the right of the OmniMaxbot as well as to detect the drop-off location. The Kinect is comprised of a laser emitter, an IR camera, and a Red-Green-Blue (RGB) camera [28]. The laser emits a beam, which is passed through a diffraction grating. This splits the beam and projects a constant pattern of dots of light on the scene. This pattern is detected by the IR camera and is compared to a reference pattern. The reference pattern is of a plane at a known distance from the Kinect. If an object in the scene is closer or farther away than the reference plane, the dot moves along the baseline between the laser emitter and the centre of the IR camera. These shifts are determined for each dot and a disparity image is produced. The distance between the Kinect and each pixel in the captured image can then be determined from the corresponding disparity. The RGB camera is used to capture colour images. Stereovision is used determine the orientation of the RGB camera in reference to the depth image frame. Once the orientation is determined, the colour of each point in the cloud can be found by determining its position relative to the RGB image and interpolating the colour.

### 3.2.3 Bumblebee®2 Stereo Vision Camera

A Point Grey Research Bumblebee®2 stereovision camera [29] is used to detect the cans and determine the distance between the can and the camera. While the Bumblebee®2 is a stereovision camera, only one of the cameras is actually required. The Bumblebee®2 connects to the computer using an IEEE 1394a/Firewire connection. This connection is used to provide power to the camera as well as transfer data.
The camera has a resolution of 0.8 MegaPixels (MP) and takes video at a rate of 20 Frames Per Second (FPS). The images it takes have a resolution of 1037×776.

3.2.4 Sparkfun SEN-10724 Inertial Measurement Unit

The OmniMaxbot uses a Sparkfun SEN-10724 9 Degrees-of-Freedom (DOF) Inertial Measurement Unit (IMU) [30]. This sensor contains accelerometers, gyros, and a magnetometer. The data from the IMU is used to help with estimating the odometry data. An Arduino Mega2560 board is used to control the IMU and interface with ROS.

3.3 Motors and Encoders

3.3.1 NPC T74

The OmniMaxbot uses four NPC T74 DC motors [31] as the drive motors. The T74 are brushed DC motors that run at 24 V, though they can go as high as 36 V. The T74 has a 20:1 gear ratio. Its performance parameters are shown in Table 3.1. The T74s have been fitted with US Digital E5 optical encoders [32] with 250 Counts Per Revolution (CPR). With quadrature, that equates to 1,000 Pulses Per Revolution (PPR). Quadrature is when the encoder has two channels which are 90° out of phase. By knowing which channel leads and which follows, the direction of rotation is known. The original encoders on the motors had 1,250 CPR. While these encoders had a higher resolution, the data overran the motor driver buffers at anything above 20% power, making them not practical.
Table 3.1: NPC T74 Dynamometer Results [31]

<table>
<thead>
<tr>
<th>Torque (in-lbs)</th>
<th>Current (A)</th>
<th>RPM</th>
<th>HP</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>9.2</td>
<td>248</td>
<td>0.11</td>
</tr>
<tr>
<td>101</td>
<td>18.5</td>
<td>235</td>
<td>0.38</td>
</tr>
<tr>
<td>228</td>
<td>28.6</td>
<td>219</td>
<td>0.80</td>
</tr>
<tr>
<td>301</td>
<td>43.3</td>
<td>205</td>
<td>0.98</td>
</tr>
<tr>
<td>396</td>
<td>55.6</td>
<td>189</td>
<td>1.19</td>
</tr>
<tr>
<td>507</td>
<td>68.4</td>
<td>173</td>
<td>1.39</td>
</tr>
<tr>
<td>591</td>
<td>84.2</td>
<td>157</td>
<td>1.47</td>
</tr>
<tr>
<td>702</td>
<td>95.5</td>
<td>141</td>
<td>1.57</td>
</tr>
<tr>
<td>833</td>
<td>107.5</td>
<td>125</td>
<td>1.66</td>
</tr>
<tr>
<td>914</td>
<td>123.5</td>
<td>109</td>
<td>1.58</td>
</tr>
<tr>
<td>1030</td>
<td>135.4</td>
<td>93</td>
<td>1.53</td>
</tr>
<tr>
<td>1112</td>
<td>148.9</td>
<td>77</td>
<td>1.36</td>
</tr>
<tr>
<td>1214</td>
<td>161.6</td>
<td>59</td>
<td>1.13</td>
</tr>
<tr>
<td>1480</td>
<td>210</td>
<td>Stall</td>
<td></td>
</tr>
</tbody>
</table>

3.3.2 Phidgets 3270

Two Phidgets 3270 [33] motors are used to lift and lower the forks. The 3270 is a 24 V motor with a 13:1 gear ratio. They have a max speed of 192 rpm and a rated torque of 1.4 N·m. The motors have Phidgets 3531 optical encoders [34] attached to them. These encoders have a resolution of 300 CPR, resulting in 1,200 PPR after quadrature.

3.4 Motor Drivers

3.4.1 AX2850

Two Roboteq AX2850 motor drivers [35] are used to control the NPC T74 motors. One AX2850 controls the front pair of motors, while the second controls the rear motors. The AX2850 has two channels, each able to output ±120 A. The polarity of the current dictates whether the motor rotates forward or reverse. The AX2850 is a versatile motor controller. It has multiple control modes depending on how the system
is set-up. Each channel can be run independently or mixed control can be used for skid-steering. The AX2850 has a built in Proportional-Integral-Derivative (PID) controller for closed-loop speed control. Open-loop control can also be used. To facilitate closed-loop control, the motor controller can be connected to two encoders, one for each channel. The encoder counts can also be outputted by the AX2850 to be used by a computer or microcontroller. The AX2850s are connected using RS-232 serial ports.

For this project, the motor drivers are set to use independent wheel control. This is due to how the kinematic equations for Mecanum wheels work. They have been set to use PID speed control to ensure that the motors rotate at the correct speeds.

### 3.4.2 Phidgets 1065

There are two Phidgets 1065 motor drivers [36], one to control each of the Phidgets 3270 motors. The 1065 uses PWM to control the motor speed. PWM is when the current is switched rapidly between the operating current and 0 A. The duty cycle, which is the proportion of time the current is flowing rather than stopped, dictates the speed of the motor. For example, a duty cycle of 50% means that the current is flowing half the time, therefore the motor is moving at 50% of full speed. The frequency of the switching has to be high enough so that the resulting waveform is as smooth as possible. The 1065 can output a max current ±5 A. As with the AX2850, the polarity of the current controls the motor direction.

### 3.5 Additional Elements

#### 3.5.1 Mecanum Wheels

In order to be omnidirectional, the OmniMaxbot uses Mecanum wheels. Mecanum wheels consist of a hub wheel and a series of rollers around the hub’s circumference
at a $45^\circ$ angle from the hub axis. As the wheel turns, friction causes the rollers to rotate. This results in a force perpendicular to the direction of rotation of the hub. Using four of these wheels and varying the wheel velocities allows the robot to move in any direction. Figure 3.1 shows an example of a mecanum wheel.

### 3.5.2 Frame

The frame is built from 80/20, a modular extruded aluminium structural material, giving it excellent strength with a light weight. It is available in a wide variety of shapes and standard sizes, though custom cutting is available as well. 80/20 also comes with all the fasteners and supports required, making it an excellent material for prototyping.

### 3.5.3 Forks

The forks are custom built from steel. They are driven using screw shafts attached to the Phidgets 3270 motors described in Section 3.3.2. The forks are attached to linear rails for stability.
3.6 Summary

This chapter outlined the physical components used to build the OmniMaxbot. These included the onboard computer, sensors, power system, motors, and drivers. The next chapter explains the software used to control the physical components, as well as how the robot achieves its tasks.
Chapter 4

Software and Control

As stated in Section 2.2.3, the OmniMaxbot make use of ROS. There are many versions of ROS available. ROS Hydro was used for this project as it was the most recent version when the programming portion of the project was started. It was decided to stay with Hydro when Indigo was released so that the code would not need to be updated. The first section of this chapter will present some of the common terminology used in ROS. The following sections will then be broken down by function for increased readability and flow. ROS is native to Linux, though it has been ported to other operating systems.

The software created specifically for, or modified for, the OmniMaxbot as part of this thesis can be found at the following online public repository: https://github.com/mars-uoit/omnimaxbot. Information on the standard packages can be found at: https://wiki.ros.org/.

4.1 Terminology

Some of the terminology used by the ROS community is unique and therefore a glossary is required. All of the information in this section can be found on the ROS
4.1.1 Node

A node is a single ROS executable. The node used to control the AX2580 motor drivers is called `ax2550.omnimaxbot` and is compiled from the `ax2550.omnimaxbot.cpp` file. This font will represent nodes. Nodes can be written in either C++ or Python.

4.1.2 Package

A package is a collection of related nodes. For example, the AX2550 package used to control the drive motors consists of the `ax2550.omnimaxbot`, `omni.cmd_vel`, and `omni.odom` nodes. This font will represent packages. In some cases both the package and the node will have the same name. An example of this is RVIZ, the ROS visualizer tool. To use this tool you must run the `rviz` node from the `rviz` package. In this case, only the package name will be used.

4.1.3 Topic

Topics are the named buses over which the nodes communicate. A topic is composed of publishers and subscribers. When a node publishes to a topic, it sends messages to that topic. A node that reads messages in a topic is said to be subscribed to it. It is possible to subscribe to a topic which is not being published to. Topics names generally start with a `/`. For example, `omni.cmd_vel` subscribes to the `/cmd_vel` topic, and publishes to the `/omnimaxbot/front/cmd_vel` and `/omnimaxbot/rear/cmd_vel` topics.
4.1.4 Message

Messages are the tool with which nodes communicate with each other. A message is a simple data structure which is comprised of fields. Messages are similar to C structures in that they can contain nested structures and arrays. Messages support standard primitive types and their associated arrays. They also support embedding other message types inside messages. There are a wide variety of message types. The type of message to use is determined by what information the nodes are sending. The different message types are standardized in order to facilitate communications between a wide range of hardware and manufacturers.

4.1.4.1 geometry_msgs/Pose

The Pose message contains a pose in free space. This pose consists of a point, holding the pose’s x, y, and z coordinates, and an orientation quaternion.

4.1.4.2 geometry_msgs/Pose2D

This message represents a point on a 2D plane. It consists of an x value, a y value, and a theta value.

4.1.4.3 geometry_msgs/PoseArray

The PoseArray message contains an array of geometry_msgs/Poses along with a header, which contains the sequence number of the message, a time stamp of when the message was made, and the ID of the frame that the poses are relative to.

4.1.4.4 geometry_msgs/PoseStamped

This message is similar to the geometry_msgs/Pose message, but also contains a header which indicates the sequence number of the message, a time stamp of when the message was created, and the frame that the pose is relative to.
4.1.4.5 geometry_msgs/PoseWithCovariance

The PoseWithCovariance message is similar to the geometry_msgs/Pose message, but it also contains a $6 \times 6$ covariance matrix representing the uncertainty in the pose estimate.

4.1.4.6 geometry_msgs/PoseWithCovarianceStamped

This message is similar to the geometry_msgs/PoseWithCovariance message, but it also contains a header with the sequence number of the message, a time stamp of when the message was created, and a frame ID.

4.1.4.7 geometry_msgs/Quaternion

This message represents an orientation in free space in quaternion form. It consists of an x, y, z, and w value.

4.1.4.8 geometry_msgs/TransformStamped

The TransformStamped message is used, primarily by the tf package, to denote the transform between two frames. It consists of a message sequence number, a time stamp, a frame ID of the parent frame, the child frame ID, and the tranform between the two. The transform itself is comprised of an x, y, z vector for translation and an orientation quaternion for rotation.

4.1.4.9 geometry_msgs/Twist

This message type is used to send velocity commands. It consists of linear velocities in metres per second along the x, y, and z directions, and angular velocities in radians per second about the x, y, and z axes.
4.1.4.10  geometry_msgs/TwistWithCovariance

The TwistWithCovariance message contains the same data as the base geometry_msgs/Twist message, as well as the $6 \times 6$ covariance matrix representing the uncertainty in the velocity estimate.

4.1.4.11  geometry_msgs/Vector3

The Vector3 message is used to indicate a direction in free space. It contains x, y, and z values used to construct a vector starting from the origin.

4.1.4.12  geometry_msgs/Vector3Stamped

This message is similar to the geometry_msgs/Vector3 message but includes a header, which contains the message sequence number, the frame ID that the vector is relative to, and a time stamp of when the message was made.

4.1.4.13  nav_msgs/MapMetaData

The basic information about an occupancy grid map is held in this message. This information includes the time at which the map was loaded, the map resolution, the width and height of the map, and the origin of the map. The resolution is the length of the size of each cell in metres, while the width and height of the map is in number of cells.

4.1.4.14  nav_msgs/OccupancyGrid

This message represents a 2D grid map, where each cell contains the probability that it is occupied. It contains a header with the message sequence number, time stamp, and frame ID, the map metadata as well as an array in row-major order containing the probabilities.
4.1.4.15 nav_msgs/Odometry

The Odometry message holds the estimated position and velocities of an object in a child frame with reference to a parent frame, with covariance. It consists of a message sequence number, time stamp, the parent frame ID, the child frame ID, the pose in the form of a geometry_msgs/PoseWithCovariance message, and the velocities in the form of a geometry_msgs/TwistWithCovariance message.

4.1.4.16 sensor_msgs/CameraInfo

The CameraInfo message is used to hold the configuration information for cameras. This data is used in conjunction with sensor_msgs/Image messages. The message contains a sequence number, the frame ID of the optical frame of the camera, a time stamp of when the image was taken, calibration data, and operational parameters. In the case of stereovision cameras, there is a separate message for each lens.

4.1.4.17 sensor_msgs/Image

This message holds the information used to record an uncompressed image. The message contains a message sequence number, time stamp of when the image was taken, the frame ID of the camera that took the image, the height of the image based on the number of pixel rows, the width of the image based on the number of pixel columns, how the image is encoded, whether it is Big Endian or Little Endian, the length of each row in bytes, and the data matrix. Endianess is important as it dictates the order in which the data is stored. In Big Endian, the most significant byte is stored in the first first memory address, while the rest follows in the subsequent memory addresses. Little Endian is the opposite, with the least significant byte stored in the first memory address. A visual explanation of this is in Figure 4.1
4.1.4.18 sensor_msgs/Imu

The Imu message is used to hold data from an IMU. The message contains a message sequence number, the frame ID of the IMU, a time stamp of when the message was created, a geometry_msgs/Vector3 to hold the linear accelerations in $m/s^2$, a second geometry_msgs/Vector3 to hold the rotational velocities in rad/s, and a geometry_msgs/Quaternion to hold the orientation. The Quaternion and Vector3s are all in row major order.

4.1.4.19 sensor_msgs/LaserScan

The LaserScan message is used to hold information from LRFs that can be used by ROS. The message consists of a message sequence number, the sensor’s frame ID, time stamp, the start and end angle of the scan in radians, the angular distance between measurements in radians, the time between measurements in seconds, the time between scans in seconds, the min and max range of the sensor in metres, an array holding the ranges, and another array holding the scan intensities.

4.1.4.20 std_msgs/Float32

This message holds a signed 32 bit floating point number.
4.1.4.21 std_msgs/Bool

This message holds a boolean value of either true or false.

4.1.4.22 tf/tfMessage

The tfMessage message contains an array of geometry_msgs/TransformStamped messages.

4.1.5 Launch File

Most packages contain one or more launch files. A launch file is an XML file that is used to set up and run other packages and nodes. Using a launch file removes the need for opening multiple terminal windows. Launch files have the .launch extension.

4.2 tf

Robots are comprised of a variety of 3D coordinate frames. These frames are used to relate the pose of individual parts of the robot to each other. Some of these frames, such as the /odom frame, change over time. The tf package [40] is used to keep track of these frames over time. It maintains the frames in a transform tree, as seen in Figure 4.2. Many of the other packages and nodes cannot work if they do not know where the sensors are in relation to other parts of the robot. The tf package subscribes and publishes to the /tf topic.

4.3 omnimaxbot_description

The omnimaxbot_description package was built as part of this thesis. It consists of a single node, tf_broadcaster. tf_broadcaster publishes the transform data relating
Figure 4.2: The transform tree of the OmniMaxbot while autonomously navigating.
the position and orientation of the individual parts to the centre of the robot. This data is published to the /tf topic.

### 4.4 lms1xx

The lms1xx package [41] is used to read the data from the SICK LMS100 LRFs. The package was originally developed by the former Robot Control and Pattern Recognition Group (RCPRG), which is now the Robot Programming and Pattern Recognition Group (RPPRG), under the name RCPRG_laser_drivers. lms1xx was renamed and is currently being maintained by Clearpath Robotics. Two new nodes, `LMS1xx_node_front` and `LMS1xx_node_rear`, were made for the OmniMaxbot by adapting the base `LMS1xx_node`. These nodes were created in order to limit the scans. Due to the position of the fork assembly relative to the LRFs, the fork assembly is always seen as an obstacle right beside the robot on the edges of the scans. This prevents the robot from being able to navigate as the system always believes that it is directly beside an obstacle and any motion will cause a collision. By limiting the ends of the scans, the fork assembly is no longer detected.

#### 4.4.1 LMS1xx_node

This is the base node of the package. As inputs it takes in the IP address and frame ID of the sensor. The frame ID is required so that ROS knows where the scans are originating from in relation to the rest of the robot. The node outputs a sensor_msgs/LaserScan message. The min and max scan angles, angle increment, time between measurements and scans, and the min and max ranges are all taken from the sensor’s configuration file. The number of elements required to hold the range and intensity data is calculated using the min and max scan angles and the angle increments. These arrays are then filled using the sensor data. Finally, the
message is published to ROS.

4.4.2 LMS1xx_front_node and LMS1xx_rear_node

It was found while working with the sensors that certain versions of the LMS100 sensors cannot dynamically change the min and max scan angles. This means that the sensors will output the entire 270° scan range, even if some of it is not desired. That is the case here. As previously stated, due to the placement of the sensors and the fork assembly, it always appears as though there is an obstacle to the right side of the robot. In order to remedy this, LMS1xx_front_node and LMS1xx_rear_node were created for this thesis by modifying the LMS1xx_node. The nodes work essentially the same way as the original node, but have the min and max angles hard coded in the programming, and have had the loops used to transfer data into the range and intensity arrays modified so that the data in the undesired areas is ignored.

4.5 rosserial_python

Three packages are used for the IMU. The first is the rosserial_python serial_node.py node [42]. This node allows ROS to communicate with the Arduino microcontroller, that the IMU is connected to, through a USB port. This node takes care of such things as setting the port that the Arduino is connected to and the baud rate used for communication. As well, running this node activates the ros_arduino_imu node, which is saved on and run from the Arduino. This node takes all of the raw data from the SEN-10724 IMU and publishes it as a custom message to the /raw_imu topic. The ros_raw_imu node was created by Tony Baltovski in the Mechatronic and Robotic Systems (MARS) lab for use in his MASc thesis project.
4.6  ros_arduino_imu

The ros_arduino_imu raw_imu_bridge_node [43] is used to sort and republish the raw data from the IMU. This package was also created by Tony Baltovski. The node takes the raw IMU data, published on the /raw_imu topic, and converts it to messages usable by ROS. The gyro and accelerometer data is converted to a sensor_msgs/Imu message and published to the /imu/data_raw topic, while the magnetometer data is converted to a geometry_msgs/Vector3Stamped message and is published to the /imu/mag topic.

4.7  imu_filter_madgwick

The imu_filter node of the imu_filter_madgwick package [44] is used to create an orientation quaternion from the raw IMU data. It does this by fusing the raw angular velocities and linear accelerations from the sensor_msgs/Imu messages on the /imu/data_raw topic, and magnetometer data in the geometry_msgs/Vector3Stamped messages on the /imu/mag topic in the method described by Madgwick [45]. The node republishes the raw IMU data, along with the calculated quaternion, as a sensor_msgs/Imu message on the /imu/data topic.

4.8  laser_scan_matcher

Due to the large amount of slip the OmniMaxbot experiences from its Mecanum wheels, odometry estimated using wheel encoder dead-reckoning is unreliable. Unfortunately, many packages require odometry data to be available. A transform between the /odom frame and the /base_link frame of the robot is required for amcl, described in Section 4.17, and move_base, described in Section 4.20, requires a geometry_msgs/Odometry message in order to know the OmniMaxbot’s speed. In order
to provide this data, the `laser_scan_matcher` package is used in lieu of the wheel encoder data. This package, found at [46], compares successive laser scans in order to determine the change in translation and rotation between the scans. The documentation for the package is for ROS Fuerte, but the code itself has been updated to run with Hydro. The base package contains only a single node, the `laser_scan_matcher` node, which only provides the transform required by `amcl`. As part of this thesis, a second node, `laser_scan_odometry_node`, was created in order to make the `geometry_msgs/Odometry` messages required for `move_base`. `laser_scan_matcher` uses PLAICP, an Iterative Closest Point (ICP) variant using a point-to-line metric presented by Censi [47]. In order to reduce errors caused by sensor noise, the package uses keyframe matching rather than direct scan-to-scan matching. What this means is that the first scan is designated as a keyframe and all subsequent scans are compared to that scan. The keyframe is only updated if the scan it is being compared to has rotated or translated by a user specified amount that is greater than any displacement that sensor noise would account for.

### 4.8.1 `laser_scan_matcher`

This node is used to determine the transform between `/odom` and `/base_link`. The node locks onto the transform of the first LaserScan message it receives, therefore only the front sensor is used. The node subscribes to the `/front_scan` topic in order to receive the scans and the `/imu/data` topic in order to get the IMU data. The node outputs the transform between the `/odom` frame and the `/base_link` frame. It also publishes the robot’s current position relative to its starting position as a `geometry_msgs/Pose2D` message on the `/pose2D` topic and as a `geometry_msgs/PoseStamped` message on the `/pose_stamped` topic.
4.8.2 laser_scan_odometry_node

This node was developed as part of this thesis for the laser_scan_matcher package in order to build an odometry message, which is required for move_base to know the OmniMaxbot’s linear and angular velocities. In order to achieve this, the node subscribes to the /pose_stamped topic in order to get the Pose2D message. Consecutive messages are compared to determine changes in the x and y translation, and the yaw rotation. The velocities are then determined using finite differencing. The node publishes the results as a geometry_msgs/Odometry message on the /odom topic.

4.9 camera1394stereo

The Point Grey Bumblebee®2 stereovision camera is run using the camera1394stereo package [48]. This is a slightly modified version of the camera1394 package, but with support for stereovision cameras. This node is used as it meets the IEEE 1394 IIDC standard, which is what the Bumblebee®2 uses for communication. The package publishes the raw images from each camera as sensor_msgs/Image messages on the /stereo_camera/left/image_raw and /stereo_camera/right/image_raw topics. The calibration data for each camera is published as sensor_msgs/CameraInfo messages on the /stereo_camera/left/camera_info and /stereo_camera/right/camera_info topics.

4.10 camera_calibration

To calibrate the Bumblebee®2, the camera_calibration package [49] was used. This package can be used to calibrate both standard and stereo cameras. The calibration data is used to remove distortion from the edges of the images obtained from each camera in a process known as rectification. The package uses OpenCV’s camera calibration method. The calibration data is used in the sensor_msgs/CameraInfo
messages published by camera1394stereo.

4.11 stereo_image_proc

In order to perform the actual rectification, the stereo_vision_proc package [50] is used. This package is used between the stereo camera drivers and image processing nodes in order to rectify and colorize the images. It can also create the disparity image and a point cloud, though in this case that is unnecessary. The package subscribes to the /stereo_camera/left/image_raw, /stereo_camera/left/camera_info, /stereo_camera-right/image_raw, and /stereo_camera/right/camera_info topics. The sensor_msgs/Image message published on the /image_raw topics are rectified and colorized using the data from the sensor_msgs/CameraInfo messages published on the /camera_info topics. While there are many outputs, for this project only the rectified and colorized image from the right camera is required. This is sent as a sensor_msgs/Image message on the /stereo_camera/right/image_rect_color topic.

4.12 openni_launch

The openni_launch package [51] is used to control the Microsoft Kinect sensor and provide depth images. This package contains several other nodes and managers, making it more of a black box than most ROS packages. Due to this, a nodelet manager is used to control the required nodes. The only input required by this package is the serial number of the Kinect. It outputs a wide variety of data such as raw and rectified images, the depth image, and the point cloud. For this project, only four of the outputs are actually required. The first is the depth image, which is published as a sensor_msgs/Image message on the /camera/depth/image_rect topic and its associated sensor_msgs/CameraInfo message, published to /camera/depth/camera_info. The other outputs are the rectified image, which is also a sensor_msgs/Image mes-
sage but published to the /camera/rgb/image_rect_color topic, and its associated sensor_msgs/CameraInfo message, published on the /camera/rgb/camera_info topic.

### 4.13 depthimage_to_laserscan

The depthimage_to_laserscan package [52] is used to convert the depth image from the Kinect into a laser scan. This simulated scan is then used by move_base for obstacle detection. The package takes in the depth image from the Kinect as a sensor_msgs/Image message on the /camera/depth/image_rect topic and outputs a sensor_msgs/LaserScan message on the /scan topic. The user specifies a distance, in pixels, from the centre line of the image that is to be considered. The nearest point in each pixel column, and in the distance from the centre specified, is taken as the range value for that point in the scan. This is repeated for the length of the scan line. The user specifies the max and min ranges to be considered.

### 4.14 ar_sys

The single_board node of the ar_sys package [53] is used twice, once to determine the position of the mock ash can relative to the Bumblebee® 2 camera, and again to determine the OmniMaxbot’s relative position to a fixed board so that the robot can get into position to put down the mock can. In order to use the same node twice, they need to be given unique names. In this case, the nodes are referred to as can_ar_sys and drop_ar_sys. This package is used as it provides a reasonably accurate estimate of the position between the AR code and the OmniMaxbot, allowing the control node to move into position to pick-up or drop-off the mock ash can easily. The AR board consists of a series of markers on a board. By knowing the actual size of the markers, the node can determine the position and orientation of the board relative to the camera being used to detect the board. Figure 4.3 shows the AR code used in this project.
For determining the position of the mock ash cans relative to the Bumblebee® 2 camera, can_ar_sys subscribes to the /stereo_camera/right/image_rect_color and /stereo_camera/right/camera_info topics. For approaching the drop-off location, drop_ar_sys subscribes to the /camera/rgb/image_rect_color and /camera/rgb/camera_info topics.

4.15 omnimaxbot_teleop

The omnimaxbot_teleop package is used to control the OmniMaxbot using a joystick. This package was created as part of this thesis and consists of a single node, omnimaxbot_joystick_teleop. This node is used to teleoperate the OmniMaxbot using a joystick, as well as to use a joystick button as a deadman switch during autonomous navigation. The node subscribes to the /joy, /omni_cmd_vel, and /plan_fork_position topics, and publishes to the /cmd_vel topic. The trigger button of the joystick works as the deadman switch for teleoperation. While this button is being depressed, the node...
converts the raw joystick data to usable commands and republishes it. The thumb button is used as the deadman switch for autonomous navigation. While it is being depressed the node takes in the data on the /omni_cmd_vel and /plan_fork_position topics and republishes it. If neither button is being depressed, the node constantly publishes all zeros.

The omnimaxbot_joystick_teleop node makes use of joy joy_node. This node takes input from a joystick and publishes it to the /joy topic. This data includes which buttons are currently being pressed, how far the joystick is being pushed along the x and y axis respectively, and how much it is being twisted around the z axis. The distance along each axis is sent as a value between 0 and ±1, where the value denotes the joystick’s position between the origin (0) and max distance along the axis (1), and the sign shows direction. Similarly, the message also includes the amount of rotation about the z axis, which is sent in the same format as the x-y data.

4.16 hector_mapping

The hector_mapping node of the hector_mapping package [54] was used to build the map of the OmniMaxbot’s environment. This package was selected as it does not require odometry to build the map, which is useful for this application as the odometry data from the Mecanum wheels is generally so inaccurate as to be unusable. Rather than rely on uncertain odometry, the package uses the laser scan data to estimate the robot’s position. The package updates the odometry data as fast as the update rate of the sensor allows. The method used in this package is described by Kohlbrecher et al. [55]. The node builds a 2D occupancy grid map of the area detected by the sensor. As the name implies, an occupancy grid map is a map built out of a grid. Each cell is of a size specified by a user and contains a value representing the probability that
the cell is occupied. In this case, the length of the edge of one cell was set to 5 cm. Each cell is considered to either be free, occupied, or unknown. Figure 4.4 shows the occupancy grid map of the MARS Lab at UOIT.

The node receives sensor_msgs/LaserScan messages by subscribing to the /front_scan topic for the scans from the front LRF. It also subscribes to the /tf topic in order to get the required transforms. It outputs the map as a nav_msgs/OccupancyGrid message on the /map topic. It publishes just the map metadata to the /map_metadata topic as a nav_msgs/MapMetaData message. The node publishes the estimated position of the robot, both without and with covariance, as a geometry_msgs/PoseStamped message on the /slam_out_pose topic and as a geometry_msgs/PoseWithCovarianceStamped message on the /poseupdate topic, respectively. It also provides the transform from /map to /odom to the /tf topic.

Figure 4.4: An occupancy grid map of the MARS Lab made using hector_mapping. White cells are unoccupied, black cells are occupied, and grey cells are unknown space.
The amcl package [56] was used to localize the OmniMaxbot in the map of its environment. While there are several localization packages available with ROS, most of them do not provide global localization. The only other package that performs global localization is the humanoid_localization package, which is used for localizing humanoid robots and therefore requires a 3D occupancy grid map. There is only one node in amcl, the amcl node. This package uses adaptive Monte-Carlo localization with Kullback-Leibler Distance (KLD)-sampling, as described by Fox [57]. The base Monte-Carlo filter is described in detail by Thrun, Burgard, and Fox [58]. The package subscribes to the /scan topic in order to get sensor_msgs/LaserScan messages from both the front and rear sensors. It also subscribes to the /tf topic in order to receive the transform information required to know where the scans are originating from, the /map topic to get the information about the map it is localizing the robot in, and the /initialpose topic. This last topic is used to initialize or reinitialize the particle filter if needed, otherwise it defaults to an initial position estimate of (0,0) and localizes from there. It was found during testing that using the default initial position estimate worked well, therefore no initial pose is sent to amcl.

As outputs, the package publishes the pose estimate of the robot along with its covariance matrix as a geometry_msgs/PoseWithCovarianceStamped message on the /amcl_pose topic, the set of pose estimates being sampled from by the filter as a geometry_msgs/PoseArray message on the /particlecloud topic, and the transform between the /map frame and the /odom frame on the /tf topic.

This package was selected as it provides a reasonable pose estimate using only the laser scan data. As only laser scanners are being used, this is all that is necessary. As well, it only produces a 2D pose estimate, rather than 3D, which is all that is required.
for this application. Both of the above features result in faster processing times due to reduced overhead load.

4.18 phidgets

The Phidgets 1065 motor drivers are controlled using the phidgets motor_control-hc_1065 node. This is a modified version of the motor_control_hc node from the phidgets package [59]. The original motor_control_hc was designed for use with two motors and the kinematics were designed for skid steering. For this thesis, the node was heavily modified to use PID position control. Two instances of the motor_control_hc_1065 node are required, one for the front fork motor and another for the rear fork motor. The nodes were name front_fork_driver and rear_fork_driver, respectively.

Both nodes subscribe to the /fork_position topic, as they are started with the same zero position and need to move the same distance in order to keep the mock cans level. When a value is published to /fork_position, it is converted into encoder counts using:

\[
\text{unroundedGoal} = \text{rawPosition} \times \text{conv} \times \text{gear\_ratio} \times \text{TPI} \times \text{cpr} \quad (4.1)
\]

where unroundedGoal is the number of counts, rawPosition is the desired position published to /fork_position in metres, conv is a conversion factor from metres to inches, gear_ratio is the gear ratio of the Phidgets 3760 motor, TPI is the number of threads per inch of the screw shaft, and cpr is the number of encoder counts per revolution of the rear motor shaft. The number of encoder counts must be an integer value, but the unroundedGoal is calculated as a double. Due to this, it must then be rounded up or down to the nearest integer. Once that is done, it is saved as a global variable, targetPosition. As well, the time at which the command was received is saved. This time
is used to determine if the timeout period is ever exceeded. The node must receive a position command every two seconds or it will time out and the motors will stop. This is to ensure that the forks will not keep moving if the driver loses communication.

One of the base functions in the Phidgets API [60] is the EncoderUpdateHandler. This functions checks the change in encoder counts every 8 Milliseconds (ms). While this function was not used in the base node, it was found to be useful for the PID controller. Using this function, the number of counts are added together in order to determine the overall change in position of the forks since the node was started. It then publishes the total summed encoder counts, as well as the change in encoder counts over the 8 ms, and publishes it to the /phidgets/<serial>_fork_encoder topics, where <serial> is the serial number of the motor driver. Once this is completed, the actual position, in encoder counts, is sent to the PID function.

The PID function is what actually controls the motor movement. When this function is entered, it first checks to see if the timeout period has been exceeded. If so, the function skips to the end and exits. If the timeout has not been exceeded, then the function calculates the duty cycle and determines if the forks are in the correct position. It does this by calculating the error using:

$$\text{error} = \text{targetPosition} - \text{actualPosition} \quad (4.2)$$

where targetPosition is the desired position and actualPosition is the current position of the fork as calculated by the EncoderUpdateHandler function. Once the error is found, the duty cycle is calculated using:

$$\text{dutyCycle} = (K_p \times \text{error}) + (K_i \times \text{integral}) + (K_d \times \text{derivative}) \quad (4.3)$$
where the duty cycle is the percent time that the motor is on, integral is the sum of
the errors from each cycle, derivative is the current error minus the previous error all
divided by the time period, and $K_p$, $K_i$, and $K_d$ are the proportional, integral, and
derivative gains, respectively. If the duty cycle is outside of the range of $\pm 100\%$, it is
set to the appropriate bound. The error is then checked to see if it is within a specified
tolerance. If it is, the motors are stopped, the integral and derivative terms are reset
to zero, and the Boolean variable goalReached is set to true and published to the
/phidgets/<serial>/fork_goal_reached topic. If the error is outside of the tolerance,
then the motor is set to the calculated duty cycle.

4.19  ax2550

The AX2850 motor drivers are controlled with the ax2550 package [61]. While this
package was already built, it was designed for running two motors using skid-steer
control. Due to this, three new nodes were created for this thesis to allow the use
of two AX2850 motor drivers simultaneously running four motors. The package is
called ax2550 despite being used to control AX2850 motor drivers. This was an
oversight on the part of the original package’s designer. The only difference between
the AX2550 and AX2850 motor controllers is that the AX2850 has inputs for optical
encoders preinstalled, while the AX2550 does not. This detail was not discovered
until after the code had been updated and integrated with the rest of the system, and
it was decided to be minor enough not to change the package name.

4.19.1  omni_cmd_vel

The omni_cmd_vel node was built as a part of this thesis to determine the velocities
at which each motor should run. The node works by taking as an input the geometry_msgs/Twist messages published to the /cmd_vel topic. The velocities in these
messages are for the entire robot as a whole, therefore, the node must break them down into individual motor velocities. This is achieved by using the inverse kinematic equations, which were taken from Doroftei et al. [62], which are outlined below.

The linear velocities of each wheel, in m/s, are found using:

\[
W_1 = V_x - V_y + W_z \times -k
\]
\[
W_2 = V_x + V_y + W_z \times +k
\]
\[
W_3 = V_x + V_y + W_z \times -k
\]
\[
W_4 = V_x - V_y + W_z \times +k
\]

where \( W_i \) is the linear velocity of the indicated wheel. The wheel numbers can be seen in Figure 4.5. \( V_x \) and \( V_y \) are the linear velocity of the robot in the x and y directions, \( W_z \) is the angular velocity about the z axis, and \( k \) is the sum of the distances between the coordinate of the wheel and the x and y axes. The equations differ slightly from those given by Doroftei et al. [62] as ROS uses left as the positive direction along the Y axis, while they used right as the positive Y direction.

Next, the node converts the linear motor velocities into rotational velocities, in Revolutions Per Minute (RPM) using:

\[
RPM_i = W_i(m/s) \times \frac{60 \text{sec}}{1\text{min}} \times \frac{1\text{rev}}{2 \times \pi \times r}
\]
\[
RPM_i = W_i \frac{60}{2 \times \pi \times r}
\]

where \( r \) is the radius of the wheel. The conversion is a rearrangement of the circumference of a circle calculation and converting from seconds to minutes.
Figure 4.5: The wheel numbering used for the OmniMaxbot.

Once this is complete, each motor velocity needs to be converted from RPM to a relative value between -127 and 127, as this is how the AX2850 drivers require the input. This is performed using:

\[ rel_i = \frac{RPM_i \times 250 \times 11}{58593.75} \]  

where \( rel_i \) is the relative speed value of the given motor, \( RPM_i \) is the motor’s rotational velocity in RPM, 250 is the number of encoder counts per revolution, 11 is the time base, and 58,593.75 is a conversion factor provided by the AX2850 operators manual [35].

After calculating the relative speed, the value is checked against a max and min bound. A value of ±35 was set as the bound. This corresponds to a maximum speed of approximately 0.5 m/s. If one motor is found to be set above the max bound, or below the min bound, then a relative value between the out of bounds value and the exceeded bound is found. The motor that exceeds is set to either the max or min
bound and the other three motor speeds are then scaled up or down based on the calculated relative value. This check is run for each motor velocity.

Given the complexity of the kinematic equations and associated conversions, shown in Equations 4.4, 4.5, and 4.6, MATLAB was used to verify that the values generated matched expected values. These tests were run with various values and it was found that in all cases the calculated and expected values were within acceptable error. Once the relative speed values are found and the bounds checked, the speeds are published. The speeds for the front motors must be sent separately than those to the rear motors due to the fact that they are on separate drivers. The values for the A motors, which are on the right side of the OmniMaxbot, are multiplied by -1 as these motors are facing backwards. The speeds are then published as geometry_msgs/Twist messages on the /omnimaxbot/front/cmd_vel and /omnimaxbot/rear/cmd_vel topics.

4.19.2 omni_odom

The omni_odom node was going to be used to calculate and publish the odometry data using dead-reckoning. Unfortunately, it was found during testing that there was too much slip in the wheels for this node to be usable. The work is included here for completions sake as it was developed as part of this thesis.

In order to calculate the odometry data the node subscribes to the /omnimaxbot/front/encoders and omnimaxbot/rear/encoders topics. These topics hold ax2550/StampedEncoders messages, made expressly for this package. These messages contain the number of encoder counts that have been counted for each motor as well as the amount
of time that has passed. The individual wheel velocities are found using:

\[
\begin{align*}
    v_w1 &= (\text{wheel1\_new} \times \text{dist\_per\_tick})/\text{dt\_front} \\
    v_w2 &= (\text{wheel2\_new} \times \text{dist\_per\_tick})/\text{dt\_rear} \\
    v_w3 &= (\text{wheel3\_new} \times \text{dist\_per\_tick})/\text{dt\_rear} \\
    v_w4 &= (\text{wheel4\_new} \times \text{dist\_per\_tick})/\text{dt\_front}
\end{align*}
\]

where \( v_{wi} \) is the velocity of the \( i \)th wheel, \( \text{wheeli\_new} \) is the number of encoder counts for the \( i \)th wheel, \( \text{dist\_per\_tick} \) is the distance the wheel turns per encoder count, \( \text{dt\_front} \) is the time between checking the front encoder counts, and \( \text{dt\_rear} \) is the time between checking the rear encoder counts.

Once the individual wheel velocities are found, the overall velocity of the OmniMaxbot is calculated using the kinematic equations from Doroftei et al. [62]:

\[
\begin{align*}
    vx &= (\text{wheel\_radius}/4) \times (v_w1 + v_w2 + v_w3 + v_w4) \\
    vy &= (\text{wheel\_radius}/4) \times (v_w1 - v_w2 - v_w3 + v_w4) \\
    vth &= (\text{wheel\_radius}/(4 \times k)) \times (-v_w1 + v_w2 - v_w3 + v_w4)
\end{align*}
\]

where the linear velocities of the OmniMaxbot in the \( x \) and \( y \) directions are denoted by \( vx \) and \( vy \) respectively, the angular velocity about the \( z \)-axis is denoted by \( vth \), and \( k \) is the sum of the distances between the wheel location and the \( x \) and \( y \) axes.

The OmniMaxbot’s displacement must be found using the velocity of the robot. This is achieved by multiplying the OmniMaxbot’s calculated velocities by the time be-
tween checking encoder counts:

\[
\begin{align*}
\delta_x &= v_x \times \text{avg}_\text{dt} \\
\delta_y &= v_y \times \text{avg}_\text{dt} \\
\delta_{\text{th}} &= v_{\text{th}} \times \text{avg}_\text{dt}
\end{align*}
\]  

where \( \delta_x \) and \( \delta_y \) are the displacements in the x and y directions, \( \delta_{\text{th}} \) is the angular displacement, and \( \text{avg}_\text{dt} \) is the average time between checking the front encoders and checking the rear encoders.

Finally, the overall displacement is found by adding the change in displacement to all of the other changes in displacement. The overall displacement is published as a geometrymsgs/Odometry message on the /odom topic. As well, the transform between the /odom frame and /base_link is published on /tf. However, as noted, there is too much slip, so the odometry data cannot be determined properly using this node. Due to this, the laser_scan_matcher package, described in Section 4.8 is used.

### 4.19.3 ax2550_front and ax2550_rear

These nodes, again built for this thesis, are slightly modified versions of the original ax2550_node. They are essentially the same as the original node, but with the kinematic and odometry sections removed. Two nodes are required, one for each motor driver. The nodes subscribe to the /omnimaxbot/front/cmd_vel and /omnimaxbot/rear/cmd_vel topics, respectively. The node sends the motor velocities to the AX2850 motor drivers, publishes the encoder counts as ax2550/StampedEncoders messages to the /omnimaxbot/front/encoders and omnimaxbot/rear/encoders topics, and stops the motors if the time between velocity commands being received exceeds
the timeout period. If the motors do not receive a command once every second, the motors will stop. This is to ensure the OmniMaxbot will not drive out of control if there is a communications issue.

4.20 move_base

The move_base package [63] is used to control the autonomous navigation. Figure 4.6 shows the interactions between the nodes in order for move_base to work. The move_base package requires several inputs. These are: a map of the robot’s environment, sensor data, transforms, and odometry. The package outputs velocity commands to the robot controller. There are five plugins to move_base. These are the local and global costmaps, local and global planners, and the recovery behaviours.

4.20.1 Global Costmap

Costmaps are used to detect and inflate obstacles so that the robot will avoid them. Both costmaps use the costmap_2d package [64]. The global costmap plugin takes as input the nav_msgs/OccupancyGrid from the /map topic and the sensor_msgs/LaserScan messages from the /scan topic. Using this data, the node finds the edges indicated
on the map and inflates them by a user specified radius. The path cannot enter the inflated area, therefore the robot will avoid walls and static obstacles. As the path is based on the centre of the robot, the inflation radius must be set large enough that the robot will not hit any obstacles even if the path skirts the edge of the inflated area. This node also uses the sensor data to detect obstacles around the robot. These objects are added to the global costmap so that even after passing out of sensor range, the global planner will know to avoid the obstacles. If, when the robot returns to that area of the map, the obstacle is no longer there it will be removed from the global costmap. The global costmap is output to the global planner and recovery behavior plugins. Figure 4.7a shows an example of a global costmap. The light grey area, surrounding the occupied cells and objects found using the laser scans, is the inflated area that the path may not pass through.

### 4.20.2 Local Costmap

The local costmap works similar to the global costmap, but only in the area immediately around the robot. The plugin only requires the sensor_msgs/LaserScan messages from the /scan topic as input. Objects detected by the sensors are inflated by a user specified radius. The output from this node is sent to the local planner and recovery behavior plugins. Figure 4.7b shows an example of the local costmap. The light square centered on the robot is the costmap area, in this case 4 m by 4 m. The dark grey areas surrounding the laser scans denote the inflated areas that the local path cannot pass through.

### 4.20.3 Recovery Behaviors

The recovery behaviors plugin is used to determine what actions the robot should take if it were to get too close to an obstacle. The default recovery behaviours are shown in Figure 4.8. If the robot enters the inflated radius of an object on one of the
costmaps, the first behaviour used to attempt a recovery is a conservative reset of the costmaps. This uses the `clear_costmap_recovery` package [65]. This package attempts to clear space by removing obstacles from the costmaps outside of a certain radius, in this case 3 metres, and reverting the costmaps to the static map provided. Objects within the radius are kept to prevent the robot from hitting them. The local and global planners can then attempt to plan a path through the newly cleared space.

If a valid path still cannot be found after the conservative reset is attempted, the plugin attempts a clearing rotation. This is only performed if there are no obstacles close enough for the robot to impact. The clearing rotation action is controlled by the `rotate_recovery` package [66]. This package directs the robot to rotate 360°, allowing the sensors to clear obstacles that are on the costmaps but are not physically there. Once this is complete, the local and global planners again try to plan a new path.

If there is still no valid path, the plugin attempts an aggressive reset. This is the same as the conservative reset, but with a smaller radius around the robot. Once this
Figure 4.8: The default recovery behaviours used by the move_base package [63].

is completed, the planners attempt to find a new path.

In the case that the robot is still stuck, the plugin will attempt a final clearing rotation. If this fails to clear enough space for the robot to plan a valid path, the motion is aborted. In this case, another goal can be sent to move_base and the package can attempt to move autonomously, or an operator has to move the robot manually via teleoperation.

4.20.4 Global Planner

The global planner is used to plan a path from the robot’s current location to a specified goal location. For this project, the goal location is always on a known map, though the goal could also be in unknown space when performing SLAM. This plugin takes the global costmap and the goal as inputs, and outputs a path. The global costmap is used to let the plugin know what areas the global plan cannot pass through. A global costmap can be seen in Figure 4.7a. The global plan found using this plugin cannot enter the light grey areas in the figure, otherwise there is the risk of crashing the robot.

move_base uses an action server for receiving goals. While it can subscribe to topics
for simple goals, the package will not provide status updates that way. By using
the action server, it is possible to query the action server for move_base’s status.
There are several global planners in ROS. Rather than using the default navfn
package, the aptly named global_planner package [67] was used. This package
was built to be a more flexible replacement for navfn. It was found during testing
that global_planner resulted in better, more consistent, paths. The package
uses Dijkstra’s algorithm. This algorithm is used to find the optimum shortest path
between points. An in depth explanation of the algorithm is provided by Sniedovich
[68].

4.20.5 Local Planner

The local planner is used to determine the actual velocity commands to send to the
robot base. Again, there are various local planners to choose from. For this application
the default base_local_planner was used. Both base_local_planner and
dwa_local_planner were tested, and it was found that base_local_planner
performed better. As inputs, the package takes in the path provided by global_pla-
nner, the nav_msgs/Odometry message from the /odom topic, and the local costmap.
The global path is required as base_local_planner needs to know where it is so
that the local plan can stay close to it, the odometry information is needed so that
the package knows how fast and in what directions the OmniMaxbot is currently
moving, and the local costmap is used by the package to ensure that the local plan
avoids obstacles in the robot’s immediate vicinity. As with the global planner, the
path found using the local planner is based on the centre of the robot. Therefore, the
local path cannot pass through the inflated object areas of the local costmap. The
plugin outputs velocities in the form of geometry_msgs/Twist messages. These are
usually published on the /cmd_vel topic and sent straight to the robot controller, in
this case ax2550. However, in order to use omnimaxbot_teleop as a deadman
switch, the velocities are published to the /omni_cmd_vel topic.

The base_local_planner package works in several steps:

1. The package takes discrete samples from a set of achievable velocities.

2. Using the robot’s current state as a starting point, forward simulation is used to determine the robot’s end state if the sample velocities are applied over a short sample time.

3. Each sample trajectory is evaluated based on its proximity to the global plan, proximity to the goal, proximity to any obstacles, and its speed.

4. The highest scoring trajectory is selected and its associated velocities are sent to the robot controller.

5. The above steps are repeated until the goal is reached.

4.21 omnimaxbot_control

The omnimaxbot_control package was built as part of this thesis. As the name implies, is used to control the OmniMaxbot. This package contains the launch files used to start the nodes required to run the OmniMaxbot. It also has a single node, omnimaxbot_control_node. This node controls the robot’s movement and the lifting and lowering of the forks. The node subscribes to the /can_ar_sys/pose, /drop_ar_sys/pose, /phidgets/299103/encoder, /phidgets/299104/encoder, /phidgets/299103/goal, and /phidgets/299104/goal topics. It publishes to the /fork_position and /omni_cmd_vel topics.

The omnimaxbot_control_node is responsible for sending goals to move_base, lining up and approaching the mock can and drop-off location, and telling the phidgets
nodes when and how far to raise and lower the forks. A C-structure is used to record the goals. The structures hold an x and y coordinate on the map frame, as well as the z and w parts of an orientation quaternion. As yaw is the only rotation required, the x and y part of the quaternion can be ignored. The goals correspond to the pick-up, drop-off, and stop positions that were selected for testing.

Once the goals are set, the position of the forks is checked. This is done by looking at data being published to the /phidgets/299103/encoders and /phidgets/299104/encoders topics. If the current position of each fork is at zero, nothing happens. Otherwise, the forks are instructed to return to their zero positions by publishing a std_msgs/Float32 message with a value of 0 to the /fork_position topic. The node waits until it has received confirmation that the forks have reached the specified position. This confirmation is found by subscribing to the /phidgets/299103/goal and /phidgets/299104/goal topics. Once the std_msgs/Bool message being published to both topics are true, the forks are in position.

Once the forks have been zeroed, the first goal, the can pick-up location, is sent to the move_base action server. After sending the goal, the node waits until the action server returns that the OmniMaxbot has either successfully moved to the goal location or that it has failed and aborted the motion. In the latter case, the OmniMaxbot can be teleoperated to a clear location and move_base will try again.

If move_base returns that the OmniMaxbot has moved into position, the node instructs the OmniMaxbot to approach the can. It does this by using the data from the /can_ar_sys/pose topic, described in Section 4.14. The node first lines up the OmniMaxbot with the can in the x direction. Once the OmniMaxbot is lined up with the can, the node then determines if the robot is lined up with the can rotationally.
If not, the OmniMaxbot is directed to rotate in place. The flow chart in Figure 4.9 shows how this is achieved. As new updates come in from \texttt{ar\_sys}, the robot continuously checks whether it is still lined up with the can and makes corrections as needed.

Next, the node directs the OmniMaxbot to approach the mock ash can. The data from \texttt{/can\_ar\_sys/pose} is used to determine the distance between the OmniMaxbot and the ash can. While approaching the can, the node checks to ensure that the robot is still properly lined up with the can in the x direction and makes adjustments as necessary.

Once the OmniMaxbot reached the pick-up position, the distance that the forks need to raise to lift the mock ash can is found using:

\begin{equation}
goal = z\text{Dist} + \text{offset} + \text{dist} - \text{preset} \tag{4.10}\end{equation}

where \text{goal} is the distance the forks need to raise in metres, \text{zDist} is the distance between the Bumblebee\textsuperscript{®}2 camera and centre of the AR code mounted to the mock can, \text{offset} is the distance between the centre of the AR code and the bottom of the can flange, \text{dist} is the extra distance to lift to make sure that there is ground clearance, and \text{preset} is the distance between the fork’s zero height and the Bumblebee\textsuperscript{®}2 camera. This goal value is published as a \texttt{std\_msgs/Float32} message to the \texttt{/fork\_position} topic. The node repeatedly publishes the message until it receives confirmation that the forks have reached the goal height. Once the confirmation is received, the OmniMaxbot reverses to clear the area.

At this point, the second goal position is sent to the \texttt{move\_base} action server. This position corresponds to the the drop-off location. Again, the node waits until \texttt{move\_base} confirms that it is in position. The OmniMaxbot then moves to the
Figure 4.9: A flowchart showing how omnimaxbot_control_node lines up with a mock ash can.
drop-off point using the same algorithm as approaching the can, but using data from 
/drop_ar_sys/pose and with different goals. Once in position, the forks are instructed 
to drop to their zero position. As the cans are a standard size, this will always result 
in the can resting on the ground and no longer being supported by the forks. Once 
the phidgets nodes have again confirmed that the forks have reached their goal, 
the robot backs away until the forks are clear of the mock can. To do this, the node 
uses the data from /can_ar_sys/pose to determine the OmniMaxbot’s distance from 
the can. Once it shows that the robot is 90 cm from the can, the third goal position 
is sent to move_base. This position is the end position. Once the OmniMaxbot has 
moved to this position, the node closes.

Figures 4.10-4.12 show the node and topic interactions used to control the Omni-
Maxbot as a whole. Figure 4.10 displays how the sensors used for the navigation 
system interact. The control system interactions are exhibited in Figure 4.11. Fi-
nally, the navigation system is presented in Figure 4.12.

4.22 Summary

This chapter explained the software drivers used to interface the sensors and mo-
tors with ROS and the programming used to control the system as a whole. Table 
4.1 outlines the nodes and packages that were developed specifically for the Omni-
Maxbot, those that were adapted for its use, and those that were used in an unmodi-
fied state. Though unmodified, significant amounts of configuration were required in 
some instances. In the next chapter, the test plan and test results for verifying the 
functionality of the OmniMaxbot are presented.
Figure 4.10: The node and topic interactions for the navigation sensors.
Figure 4.11: The node and topic interactions for the control system.
Figure 4.12: The node and topic interactions for the navigation system.
Table 4.1: ROS Package/Node Summary

<table>
<thead>
<tr>
<th>Newly Developed</th>
<th>Modified</th>
<th>Unmodified</th>
</tr>
</thead>
<tbody>
<tr>
<td>omni_cmd_vel</td>
<td>ax2550_node_front</td>
<td>rosserial_python</td>
</tr>
<tr>
<td>omni_odom</td>
<td>ax2550_node_rear</td>
<td>ros_arduino_imu</td>
</tr>
<tr>
<td>omnimaxbot_description</td>
<td>lms1xx_node_front</td>
<td>imu_filter_madgewick</td>
</tr>
<tr>
<td>omnimaxbot_teleop</td>
<td>lms1xx_node_rear</td>
<td>topic_tools</td>
</tr>
<tr>
<td>omnimaxbot_control</td>
<td>motor_control_hc_1065</td>
<td>laser_scan_matcher_node</td>
</tr>
<tr>
<td>laser_scan_odometry_node</td>
<td></td>
<td>openni_launch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>depthimage_to_laserscan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ar_sys</td>
</tr>
<tr>
<td></td>
<td></td>
<td>camera1394_stereo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stereo_image_proc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>map_server</td>
</tr>
<tr>
<td></td>
<td></td>
<td>amcl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>move_base</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hector_mapping</td>
</tr>
</tbody>
</table>
Chapter 5

Testing and Results

Extensive testing was required to ensure that the OmniMaxbot worked properly. This testing was performed in three stages. In the first stage, the individual components were tested. This was to ensure that the parts and their programming worked on their own. After the parts were individually tested, different subsystems were tested to verify that the parts worked together properly. Finally, the entire system was tested to make sure that the whole system worked as one unit and that it was able to perform its job.

5.1 Parts Testing

In order to verify that the parts and their programming work as expected, it was important to test all of the parts on their own. In this way, if a part worked on its own but stopped working after being integrated into the system, there would be a starting point for troubleshooting. This part involved testing the parts to ensure that they work, becoming familiar with their programming, and calibrating them if needed.
5.2 Subsystems

Each major subsystem required testing to ensure that the parts all worked together when integrated, and that the OmniMaxbot behaved as expected. This section gives details on what tests were performed, the expected results, and the actual results.

5.2.1 Emergency Stop

The first subsystem tested was the emergency stop as a safety measure. It was import to ensure that the motors could be powered down immediately if the robot started acting in an unpredictable or unsafe manner. Table 5.1 details the test plan and results used when testing the e-stop button. For this test, the OmniMaxbot was placed on jack stands. This was so that if the robot acted unpredictably and the e-stop did not work, the robot would not actually move. The OmniMaxbot was wired in such a way that the motors would lose power when the e-stop is pressed, but the AX2550 motor drivers and the integrated computer would not. This can be seen in Figure 2.1. Wired this way, if the e-stop is pressed the motors will stop but the controllers would still be running, allowing for trouble shooting or proper shutdown of the rest of the robot. Unfortunately, the Phidgets 1065 motor drivers, used to control the forklift motors, had to be wired in after the e-stop due to how they work. Once everything was connected, a launch file was run which allowed for the OmniMaxbot to be controlled by joystick or autonomously. Using the joystick, the robot was instructed to drive forward. As the wheels were turning, the e-stop button was pressed to ensure that it would cut power to the motors. It was found that the emergency stop worked as planned, with the motors stopping but the control systems still operating properly.
5.2.2 Deadman Switch

As explained in Section 4.15, a joystick was used as a deadman switch for both teleoperation and autonomous navigation. Tests were performed to ensure that both deadman switches worked properly. In order for this test to be passed, the motors must stop immediately after the deadman switch is released. Table 5.2 shows the test plan and results. Three tests were performed. The first test used a basic version of the omnimaxbot_teleop teleop node to test the teleoperation deadman switch. It was found using the original node that the motors would stop moving, but only after the timeout period of two seconds was exceeded. This was deemed to be unacceptable. The node was edited so that when the deadman switch was released the node would publish a value of 0 to the motors. In this way, rather than waiting for the motor drivers to time out they would be sent an immediate stop value. The second test used the updated node to test the teleoperation deadman switch. This time, the motors stopped immediately after the switch was released. The final test was to ensure that the deadman switch worked for autonomous navigation. Again, it was found that the robot stopped immediately upon release of the deadman switch.

5.2.3 Forklift Test

The forklift system was tested to ensure that the OmniMaxbot would be able to lift the mock ash cans. Table 5.3 shows the test plan and results. The first test was to determine how accurately ar_sys can estimate the position of an AR code relative to the camera. This was determined by comparing the results from ar_sys to distances found using a measuring tape. The AR code was moved to various locations for more data. It was found that for distances in the y direction, ar_sys is accurate to $\pm 5$ cm. For distances in the x direction, ar_sys is accurate to $\pm 0.1$ cm.

The second test was to make sure that the phidgets motor_control hc_1065 nodes
raised and lowered the forks when instructed and moved the correct distance. This was tested using the Ubuntu command line and the rostopic tool to manually input distances for the forks to raise and lower. It was found that the forks raised and lowered the correct distance when directed.

The final test was to integrate the ar_sys and phidgets packages together into a working subsystem. For this test, a mock ash can with an AR code mounted on the side was manually moved towards the camera. For the test to be passed, the forks needed to raise when the can was within 20 cm. This test worked as expected, with the forks lifting the can once it was 20 cm from the camera.

5.3 Navigation Tests

The following tests were performed in order to ensure that the navigation systems worked properly.

5.3.1 Kinematics

Testing was performed to ensure that the kinematic equations produced the expected motion. The test plan and results can be seen in Table 5.4. A joystick was used to instruct the OmniMaxbot to move forward, rearward, left, right, and to rotate clockwise and counter-clockwise. It was found that the robot moved in all of the expected directions.

5.3.2 Wheel Odometry Test

Much of the literature states that odometry data found using dead-reckoning from encoders on Mecanum wheels is highly inaccurate. As explained in Section 4.20, the odometry data is important for the autonomous navigation. Tests were run in order
to determine how inaccurate the odometry data from the encoders was. For this test, the `robot_pose_ekf` package was used to determine the robot’s odometry data. Table 5.5 details the test procedure and results. The results clearly show that using odometry data generated using wheel encoders is completely unacceptable.

Three tests were performed to verify the odometry data. The first was to test the odometry in the x direction. For this test, the OmniMaxbot faced a flat section of wall. An image was taken of the initial scan taken by the front LRF using RVIZ. Next, the decay time of the laser scan messages, which is how long a scan stays on the screen before being deleted, was set to thirty seconds. This was so that scans gathered over that period would be aggregated together. The robot was then driven toward the wall. If the odometry were perfect, the aggregated scans should overlap each other perfectly at the edge of the wall, appearing as a single scan. Errors in the odometry, such as from slip, would result in the scan lines moving. The result of this test is shown in Figure 5.1. As can be seen in the figure, the scan lines spread out a great deal, indicating an excessive amount of drift. There is also some rotational drift, despite the robot being driven straight. As well, the figure appears to show the robot as being stationary, with the wall moving toward the robot, rather than the robot moving toward the wall.

In the second test, the odometry in the y direction was examined. The same process as before was used, except that the robot was at a 90° angle from the wall and strafed sideways towards it. Again, the scans spread out over a large area and some rotational drift is evident as can be seen in Figure 5.2. Again, `robot_pose_ekf` made it look as though the wall was moving toward the OmniMaxbot rather than the other way around.
Figure 5.1: Result of the linear odometry test in the x direction using robot_pose_ekf. The left image is the original scan. The image on the right shows the scans aggregated over 30 seconds.

Figure 5.2: Result of the linear odometry test in the y direction using robot_pose_ekf. The left image is the original scan. The image on the right shows the scans aggregated over 30 seconds.
Figure 5.3: Result of the rotational odometry test using robot_pose_ekf. The left image is the original scan, the middle image is the scan after the robot had rotated 360° degrees, and the image on the right is the scan after it had settled.

The final test using the wheel encoders was to check the odometry during rotation. For this test, an image of the initial scan was taken using rviz. Next, the OmniMaxbot was rotated in place by 360° degrees. An image of the laser scan was taken after the rotation as well. If the odometry were perfect, the initial and final laser scans would coincide. As can be seen in Figure 5.3, this was not the case. In addition to rotational drift, there appeared to be an issue with the encoder counts being buffered. As can be seen in the centre portion of Figure 5.3, which was taken once the 360° degree rotation was completed, the scans were off by roughly 60° degrees. At this time, the scans were still rotating despite the robot being still. After the scans settled, they were only off by 15° degrees from the original scan.

5.3.3 Laser Odometry Test

Once it was determined that the wheel odometry was as poor as expected, the decision was made to use the laser_scan_matcher package, described in Section 4.8, to simulate the odometry data. The same three tests used to determine the accuracy for the wheel odometry were used to test the laser odometry. Table 5.6 outlines the test plan and results. The results clearly show that using the laser_scan_matcher...
Figure 5.4: Result of the linear odometry test in the x direction. The left image is the original scan. The image on the right shows the scans aggregated over 30 seconds.

The results of the test in the x direction can be seen in Figure 5.4. The image shows that the aggregated scans spread a small amount.

The results of the test in the y direction can be seen in Figure 5.5. This image also shows an insignificant amount of spreading of the scan lines.

The final rotation test is displayed in Figure 5.6. As can be seen in the image, the initial and after-rotation scans almost perfectly line up.

5.3.4 Simple Navigation Test

To ensure that the autonomous navigation worked as expected, simple navigation tests were performed. The omnimaxbot_control nav_test.launch file was used to run all of the required nodes. Using rviz, the OmniMaxbot was given navigation goals on
Figure 5.5: Result of the linear odometry test in the $y$ direction. The left image is the original scan. The image on the right shows the scans aggregated over 30 seconds.

Figure 5.6: Result of the rotational odometry test. The left image is the original scan. The image on the right is the scan taken after a 360° degree rotation.
the map. The robot needed to be able to make its way from its starting position to the indicated goal poses without user interference. A tolerance of 0.152 m in translation and 0.087° Radians in rotation was permitted. The position of the OmniMaxbot at the end of each test was measured relative to the origin of the map using measuring tapes. These results were compared to the x and y position values sent from rviz to move_base. While a ground truth system could have been used to achieve a more accurate result than manual measurement, that equipment was unavailable and it was determined that manual measurement returned acceptable results. Table 5.7 shows the test plan and results. The test was run ten times, using different start and goal locations. The OmniMaxbot passed every test.

5.3.5 Static Obstacle Test

The simple navigation tests were performed in a cleared area. In practical use however, the OmniMaxbot may have to avoid obstacles in its path. This set of tests were used to determine if the OmniMaxbot could avoid static obstacles in its path. A similar procedure to the one used in Section 5.3.4 was used here, but with a section of Sonotube placed between the OmniMaxbot and the goal pose. In order for the test to be passed, the OmniMaxbot needed to be able to navigate around the obstacle if possible, or stop if not, without colliding with it. A binary pass/fail was used due to safety concerns. If the OmniMaxbot were to hit a worker while in use, that would be a severe failure regardless of how hard the actual impact was. As the robot cannot tell what is an inanimate object and what is a worker, it was determined that any impacts count as a failure from a safety point of view, regardless of what the actual severity of the situation would be. Different start, stop, and obstacle locations were used in order to simulate a wider range of situations. The test was run, and passed, ten times.
5.3.6 Dynamic Obstacle Test

Once it was determined that the OmniMaxbot could handle static obstacles, a dynamic obstacle was introduced to the testing area. This was to simulate a worker walking through the area while the OmniMaxbot is operating. The process was the same as that used in Section 5.3.5, but with a moving obstacle rather than a static obstacle. To make the obstacle, a section of Sonotube was pushed across the room on a cart while the OmniMaxbot moved from its start location to a navigation goal set using rviz. Again, the OmniMaxbot needed to avoid hitting the obstacle to be considered successful. As with the static obstacle test, a binary pass/fail metric was used for safety reasons. This test was performed ten times. All of the tests passed, though in some cases the robot approached very close to the Sonotube. The rolling window for the local costmap was increased in size, at which point this ceased. In some cases the robot aborted its motion entirely due to not being able to fit in the space between the obstacle and the desks.

5.4 Integrated System Tests

The final set of tests were used to determine how well the OmniMaxbot performed as a fully integrated system. Three tests were performed in this set. These were the simplified mock ash can retrieval test, the complex retrieval test, and the full system test. In all of the tests, the OmniMaxbot’s performance was given a rank of pass or fail. In the case of a failed test, the reason and severity of the failure is explained.

5.4.1 Simplified Ash Can Retrieval Test

The first test was a simple retrieval of the mock ash can. This test was used to ensure that the OmniMaxbot would follow simple navigation goals and make sure that all of the parts of the system worked together properly. The test followed these steps:
1. Autonomously navigate to a preprogrammed pose near the mock ash can

2. Line up with and approach the mock ash can

3. Raise the forks to lift the mock ash can

4. Reverse to simulate the OmniMaxbot moving clear of the flame reactor

5. Autonomously navigate to a stop position

6. Lower the mock ash can to the ground

In order for an instance of the test to be considered successful, the OmniMaxbot needed to be able to achieve each step without requiring interference by an operator and without making contact with the can or any obstacles.

This test was run ten times. Between each test the ROS cache was cleared to ensure that data from previous tests would not interfere with subsequent tests. Nine of the ten tests passed. In the one failed test, the OmniMaxbot hit the mock ash can with one of the forks. This was determined to have occurred due to an error in the localization. The OmniMaxbot believed itself to be further back than it actually was. Once this occurred, the deadman switch was released and the OmniMaxbot halted its movement. The robot was moved away from the can manually, after which the OmniMaxbot successfully manoeuvred around the can and finished the test without further errors. In an actual production environment, this would not have been a severe failure as the robot was stopped as soon as the fork touched the can. It is unknown, however, how much the OmniMaxbot would have pushed the can had it not been under observation, which is why the test was considered a failure. As well, had it been a person the robot hit rather than an ash can, the consequences would be worse.
5.4.2 Complex Ash Can Retrieval Test

Once it was determined that the OmniMaxbot could retrieve the mock ash can and bring it to a general location, a new test was devised to ensure that the OmniMaxbot could drop off the can in a specified spot. This test simulated retrieving a can from a flame reactor and placing it onto a conveyor. The test used the following steps:

1. Autonomously navigate to a preprogrammed pose near the mock ash can
2. Line up with and approach the mock ash can
3. Raise the forks to lift the mock ash can
4. Reverse to simulate the OmniMaxbot moving clear of the flame reactor
5. Autonomously navigate to a second preprogrammed pose near the drop-off location
6. Line up with and approach the drop-off location
7. Lower the mock ash can to the ground
8. Reverse until the forks are clear of the mock ash can
9. Autonomously navigate to a stop position

As with the simple test, in order for an instance of the test to be considered successful, the OmniMaxbot needed to be able to achieve each step without requiring interference by an operator and without making contact with the can or any obstacles.

This test was also performed ten times to ensure repeatability, with the ROS cache being cleared between each run. Of these ten tests, nine were completed successfully. In the one failed test, the OmniMaxbot moved too close to the can while approaching it for pick-up. The robot approached until the mock ash can was hit by the stop
bar, a section of 80/20 placed across the fork assembly frame near the bottom to prevent mock ash cans from getting too close to the Bumblebee®2 camera. After hitting the can, the OmniMaxbot immediately stopped before raising the forks and completing the rest of the test successfully. While being counted as a failure because the OmniMaxbot hit the can, this would not have been a critical failure under working conditions. This is because the OmniMaxbot stopped moving almost immediately after hitting the can without the deadman switch being released.

Figures 5.7-5.14 show the OmniMaxbot’s progression through this test. Figure 5.7 shows the OmniMaxbot in its starting position. The OmniMaxbot is displayed approaching the mock ash can in Figure 5.8. Figure 5.9 presents the OmniMaxbot after lifting the mock ash can. The OmniMaxbot is shown in Figure 5.10 after reversing to simulate clearing the flame reactor. Figure 5.11 shows the OmniMaxbot in position to approach the drop-off location. The OmniMaxbot is in the drop-off position in Figure 5.12. In Figure 5.13, the OmniMaxbot has reversed to clear the mock ash can. Finally, Figure 5.14 displays the OmniMaxbot in the stop position after finishing the test.

5.4.3 Full System Test

The final test was the full system test. This test was used to show all of the parts of the OmniMaxbot working in unison. The test consisted of running the complex test outlined in Section 5.4.2, with both static and dynamic obstacles. Figure 5.15 shows the layout of the test. Two sections of Sonotube were placed between the robot’s start location and the pick-up location to force the OmniMaxbot through a small bottleneck. After the OmniMaxbot started moving, a lab member walked past that space twice, once when the robot was nearing the bottleneck on its way to pick-up the can and again when the robot was moving to the drop-off location. In order to pass,
Figure 5.7: The OmniMaxbot in its starting location.

Figure 5.8: The OmniMaxbot approaching the mock ash can.
Figure 5.9: The mock ash can being lifted by the OmniMaxbot.

Figure 5.10: The position of the OmniMaxbot after reversing to simulate clearing the flame reactor.
Figure 5.11: The approach position for the drop-off location.

Figure 5.12: The OmniMaxbot after lowering the mock ash can to the ground.
Figure 5.13: The position of the OmniMaxbot after reversing to ensure that the forks are clear of the mock ash can.

Figure 5.14: The test stop position.
Figure 5.15: The layout of the full system test.

the robot needed to stop and wait for the lab member to pass before continuing the steps outlined in Section 5.4.2. The OmniMaxbot passed the test perfectly, in both cases stopping and waiting for the inflated area around the lab member to clear the planned path before proceeding.
Table 5.1: Emergency Stop Subsystem Test

| Hardware: | AX2550 (2), NPC-T74 (4), emergency stop button, Turnigy LiPo batteries (12), Phidgets 1065 (2), Phidgets 3270 (2) |
| Setup: | Put the OmniMaxbot on jack stands  
Connect NPC T74 motors to the AX2850 motor drivers  
Connect Phidgets 3270 motors to the Phidgets 1065 motor drivers  
Connect motor drivers to the integrated computer  
Connect components and batteries as in Figure 2.1  
Engage the emergency stop button  
Run omnimaxbot_control nav_test.launch |
| Expected Output: | The motors should turn off, but the controllers should remain powered |
| Test: | Action: | Pass | Fail | Comments |
| 1 | Run the motors for 10 seconds, then push the emergency stop button | X |  |  |
Table 5.2: Deadman Switch Test

<table>
<thead>
<tr>
<th>Hardware: AX2550 (2), NPC-T74 (4), emergency stop button, Turnigy LiPo batteries (12), joystick, integrated computer</th>
</tr>
</thead>
</table>
| Setup:  Put the OmniMaxbot on jack stands  
Connect NPC T74 motors to the AX2850 motor drivers  
Connect motor drivers to the integrated computer  
Connect components and batteries as in Figure 2.1  
Engage the emergency stop button  
Run omnimaxbot_control nav_test.launch |
<p>| Expected Output: The OmniMaxbot should move forward when the deadman switch is depressed, and stop immediately once the deadman switch is released |</p>
<table>
<thead>
<tr>
<th>Test:</th>
<th>Action:</th>
<th>Pass</th>
<th>Fail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hold down the deadman switch, push the joystick forward, then release the deadman switch</td>
<td>X</td>
<td></td>
<td>Motors continued running until they timed out</td>
</tr>
<tr>
<td>2</td>
<td>Hold down the deadman switch, push the joystick forward, then release the deadman switch</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Hold the deadman switch, give the OmniMaxbot a navigation goal using RVIZ, then release the deadman switch</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table 5.3: Forklift Subsystem Test

| Hardware: | Phidgets 1065 (2), Phidgets 3270 (2), emergency stop button, Turnigy LiPo batteries (12), Bumblebee2 camera, integrated computer, mock ash can, measuring tape |
| Setup: | Attach Phidgets 3270 motors to Phidgets 1065 motor controllers  
Connect Phidgets 1065s and Bumblebee2 to the integrated computer  
Run omnimaxbot_control forklift_test.launch  
Engage the emergency stop button |
| Test: | Action: | Expected Output: | Pass | Fail | Comments |
| 1 | Place the mock can in front of the Bumblebee2, check the distance values found using ar.sys, and confirm them with the measuring tape | The measured distances should be similar to those reported by ar.sys | X |  | Distance accurate to ±5 cm, horizontally accurate to ±0.1 cm |
| 2 | Use the command line to enter distances to raise and lower the forks | The forks should raise/lower the specified distance | X |  |  |
| 3 | Manually move the ash can into pick-up range | The forks should raise to lift the mock ash can when it is within 20 cm of the camera | X |  |  |
Table 5.4: Kinematics Test

| Hardware: AX2550 (2), NPC-T74 (4), emergency stop button, Turnigy LiPo batteries (12), joystick, integrated computer |
| Setup: Connect AX2550s to the integrated computer  
  Turn on the AX2550s  
  Run nav_test.launch  
  Engage the emergency stop button |

<table>
<thead>
<tr>
<th>Test:</th>
<th>Action:</th>
<th>Expected Output:</th>
<th>Pass</th>
<th>Fail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hold the deadman switch and push the joystick forward</td>
<td>The OmniMaxbot should move forward</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Hold the deadman switch and pull back on the joystick</td>
<td>The OmniMaxbot should move rearward</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Hold the deadman switch and push the joystick to the right</td>
<td>The OmniMaxbot should strafe to the right</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Hold the deadman switch and push the joystick to the left</td>
<td>The OmniMaxbot should strafe to the left</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Hold the deadman switch and twist the joystick counter clockwise</td>
<td>The OmniMaxbot should rotate counter clockwise</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Hold the deadman switch and twist the joystick clockwise</td>
<td>The OmniMaxbot should rotate clockwise</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td>Action</td>
<td>Expected Output</td>
<td>Pass</td>
<td>Fail</td>
<td>Comments</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>----------------</td>
<td>------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>1</td>
<td>In RVIZ, set the laser scan decay time to 30 seconds. Drive the OmniMaxbot directly towards a flat wall</td>
<td>The thickness of the aggregated scans should stay relatively consistent. Ideally, they should appear the same width as a single scan</td>
<td>X</td>
<td>The scans spread out over a large area. See Figure 5.1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>In RVIZ, set the laser scan decay time to 30 seconds. Drive the OmniMaxbot sideways towards a flat wall</td>
<td>The thickness of the aggregated scans should stay relatively consistent. Ideally, they should appear the same width as a single scan</td>
<td>X</td>
<td>The scans spread out over a large area. See Figure 5.2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>In RVIZ, watch the laser scans and rotate the robot in place 360°</td>
<td>The scans from before and after the rotation should be within ±2° of each other</td>
<td>X</td>
<td>The start and end scans were ~60° degrees apart. See Figure 5.3</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: Wheel Odometry Test

Hardware: AX2550 (2), NPC-T74 (4), emergency stop button, Turnigy LiPo batteries (12), joystick, LMS100 (1), IMU, integrated computer

Setup: Attach LMS100, IMU, and AX2550s to the integrated computer
Turn on the LMS100 and AX2550s
Run omnimaxbot_control nav_test.launch
Engage the emergency stop button
Table 5.6: Laser Odometry Test

| Hardware: | AX2550 (2), NPC-T74 (4), emergency stop button, Turnigy LiPo batteries (12), joystick, LMS100 (1), IMU, integrated computer |
| Setup: | Attach LMS100, IMU, and AX2550s to the integrated computer  
Turn on the LMS100 and AX2550s  
Run omnimaxbot_control laser_odom_test.launch  
Engage the emergency stop button |
| Test: | Action: | Expected Output: | Pass | Fail | Comments |
| 1 | In RVIZ, set the laser scan decay time to 30 seconds. Drive the OmniMaxbot directly towards a flat wall | The thickness of the aggregated scans should stay relatively consistent. Ideally, they should appear the same width as a single scan | X | | The scans spread out slightly, but an acceptable amount. See Figure 5.4 |
| 2 | In RVIZ, set the laser scan decay time to 30 seconds. Drive the OmniMaxbot sideways towards a flat wall | The thickness of the aggregated scans should stay relatively consistent. Ideally, they should appear the same width as a single scan | X | | The scans spread out slightly, signifying a small amount of drift. See Figure 5.5 |
| 3 | In RVIZ, watch the laser scans and rotate the robot in place 360° | The scans from before and after the rotation should be within ±2° of each other | X | | The start and end scans were off an insignificant amount. See Figure 5.6 |
Table 5.7: Simple Navigation Test

<table>
<thead>
<tr>
<th>Hardware: AX2550 (2), NPC-T74 (4), emergency stop button, Turnigy LiPo batteries (12), joystick, LMS100 (2), IMU, Kinect, integrated computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup: Connect components  Engage the emergency stop button  Run omnimaxbot_control nav_test.launch</td>
</tr>
<tr>
<td>Expected Output: The OmniMaxbot should autonomously drive from its current position to the goal position</td>
</tr>
<tr>
<td>Test:</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
Table 5.8: Static Obstacle Test

| Hardware: | AX2550 (2), NPC-T74 (4), emergency stop button, Turnigy LiPo batteries (12), joystick, LMS100 (2), IMU, Kinect, integrated computer, Sonotube |
| Setup: | Connect components  
Engage the emergency stop button  
Run omnimaxbot_control nav_test.launch |
| Expected Output: | The OmniMaxbot should autonomously drive from its current position to the goal position while avoiding the obstacle |

<table>
<thead>
<tr>
<th>Test:</th>
<th>Action:</th>
<th>Pass</th>
<th>Fail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>In rviz, set a 2D nav goal. Place the Sonotube section between the OmniMaxbot and the goal position. Hold the deadman switch</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>In rviz, set a 2D nav goal. Place the Sonotube section between the OmniMaxbot and the goal position. Hold the deadman switch</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>In rviz, set a 2D nav goal. Place the Sonotube section between the OmniMaxbot and the goal position. Hold the deadman switch</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>In rviz, set a 2D nav goal. Place the Sonotube section between the OmniMaxbot and the goal position. Hold the deadman switch</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>In rviz, set a 2D nav goal. Place the Sonotube section between the OmniMaxbot and the goal position. Hold the deadman switch</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 5.9: Dynamic Obstacle Test

<table>
<thead>
<tr>
<th>Hardware:</th>
<th>AX2550 (2), NPC-T74 (4), emergency stop button, Turnigy LiPo batteries (12), joystick, LMS100 (2), IMU, Kinect, integrated computer, Sonotube</th>
</tr>
</thead>
</table>
| Setup:    | Connect components  
Engage the emergency stop button  
Run omnimaxbot_control nav_test.launch |
| Expected Output: | The OmniMaxbot should autonomously drive from its current position to the goal position, avoiding the obstacle |

<table>
<thead>
<tr>
<th>Test:</th>
<th>Action:</th>
<th>Pass</th>
<th>Fail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot’s path</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot’s path</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot’s path</td>
<td>X</td>
<td></td>
<td>Stopped close. Had to reverse to rotate.</td>
</tr>
<tr>
<td>4</td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot’s path</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot's path</td>
<td>X</td>
<td>Stopped close. Aborted move, not enough space to pass obstacle.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------------------------------------------------------</td>
<td>---</td>
<td>------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot's path</td>
<td>X</td>
<td>Stopped close. Had to reverse to rotate.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot's path</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot's path</td>
<td>X</td>
<td>Stopped. Move aborted, not enough space to bypass obstacle.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>In rviz, set a 2D nav goal. Hold the deadman switch. Push the section of Sonotube across the OmniMaxbot’s path</td>
<td>X</td>
<td>Stopped. Move aborted, not enough space to bypass obstacle.</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.10: Simple Integrated Test

| Hardware: | AX2550 (2), NPC-T74 (4), Phidgets 1065 (2), Phidgets 3270 (2), emergency stop button, Turnigy LiPo batteries (12), joystick, LMS100 (2), IMU, Kinect, integrated computer, mock ash can |
| Setup: | Connect components  
Run omnimaxbot_control omnimaxbot_move.launch  
Engage the emergency stop button |
| Expected Output: | The OmniMaxbot should autonomously navigate to a location near the mock ash can. It should then line up and approach the can. Next, the forks should raise 7.62 cm, lifting the can. The robot should then reverse 90 cm, then autonomously navigate back to its start location. Finally, the forks to lower the can to the floor |
| Test: | Action: | Pass | Fail | Comments |
| 1 | Hold deadman switch | X |
| 2 | Hold deadman switch | X |
| 3 | Hold deadman switch | X |
| 4 | Hold deadman switch | X |
| 5 | Hold deadman switch | X |
| 6 | Hold deadman switch | X |
| 7 | Hold deadman switch | X | Had trouble with localization. Approached very close to the mock ash can, but stopped and backed away before hitting it |
| 8 | Hold deadman switch | X | Hit the mock ash can with the front fork. Attributed to localization error |
| 9 | Hold deadman switch | X |
| 10 | Hold deadman switch | X |
Table 5.11: Complex Integrated Test

<table>
<thead>
<tr>
<th>Test</th>
<th>Action</th>
<th>Pass</th>
<th>Fail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td>Moved too close to the mock ash can during approach step. The mock ash can hit stop bar. The OmniMaxbot stopped almost immediately, then proceeded to lift the can and complete the test</td>
</tr>
<tr>
<td>10</td>
<td>Hold deadman switch</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.12: Full Integrated Test

<table>
<thead>
<tr>
<th>Test</th>
<th>Action</th>
<th>Pass</th>
<th>Fail</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Place Sonotube between the OmniMaxbot’s start location and the pick-up location. Hold deadman switch. Once the OmniMaxbot starts moving toward the pick-up location, have lab member walk across robot’s path.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.4.4 Summary

This chapter explained the testing used to ensure that the individual parts and sub-systems worked, that the OmniMaxbot is capable of autonomously navigation, that the OmniMaxbot can avoid static and dynamic obstacles, and to verify that the system is able to perform its goals. The next chapter details the conclusions found from this testing and recommends future work.
Chapter 6

Conclusions and Recommendations for Future Work

This thesis described the development of the OmniMaxbot, a fully functioning proof-of-concept prototype autonomous omnidirectional robot for handling hazardous materials. The hardware, software, and testing have been examined in depth. Omnidirectional motion was achieved using Mecanum wheels. This allows the OmniMaxbot to navigate in tight spaces. The power distribution system was redesigned for improved safety. An overall control strategy was developed. Several ROS packages were developed or adapted to interact with the OmniMaxbot’s hardware and to control the robot autonomously.

Different methods of generating odometry data were tested, with the ROS laser_scan_matcher package being selected as the most accurate. Due to the excessive amount of slip caused by the Mecanum wheels, generating odometry data using encoder counts proved to be infeasible.

Different SLAM packages were examined, with the ROS hector_mapping package
being selected as it does not require odometry data to build the map. Similarly, different path planning packages were tested to determine which worked best. It was found that base_local_planner provided the best paths. Testing was performed in order to determine the configuration settings that produced the best performance results.

Extensive testing was performed in order to ensure that the individual components, subsystems, and the full system worked as required. It was found that the developed control strategy enabled the OmniMaxbot to autonomously navigate to a mock ash can, pick the mock ash can up, navigate to a drop-off location, put the mock ash can down, back away until the forks are clear of the mock ash can, and navigate to a standby position. These tasks can be achieved in open areas, as well as areas containing static and dynamic obstacles.

In addition, much of the hardware and software used in the development of the OmniMaxbot could also be used to automate an appropriately modified forklift. Some retrofitting would be required in order to add the sensors, communications, and processing equipment. Adapting the high level navigation and control software would be relatively straightforward.

6.1 Lessons Learned

Several observations were made during the course of this thesis. The main ones were:

- Modifying the ash cans with AR codes was a far simpler way of detecting the cans and determining their position relative to the OmniMaxbot than using object recognition with a stereovision camera.

- Odometry data estimation from wheel encoders is generally poor, especially
with Mecanum wheels. Using a system to calculate the odometry independent of slip resulted in far better estimation.

- Configuring software nodes is an iterative, time consuming process, but doing it properly once takes less time than having to reconfigure the node every time something changes.

### 6.2 Recommendations for Future Work

This work shows a key step in the overall development of the OmniMaxbot system. The next step would be to build a high fidelity prototype to be tested in the actual work environment under normal working conditions with fully loaded ash cans. This would be a two stage process, building and testing the initial prototype in the lab to ensure that it works, before moving it to Cameco for in-service tests.

There are some upgrades that the next prototype could implement in order to make the system more efficient or user friendly. As alluded to earlier, the power system could be improved. Currently, each battery needs to be removed and charged separately. This takes a great amount of time and would be impractical in a manufacturing environment. Designing a power system for the OmniMaxbot that could charge the batteries simultaneously without requiring their removal would be ideal. As well, due to the way the batteries are wired, it is possible for the batteries to charge each other if the emergency stop button is engaged and the batteries are not all charged to the same level. This could reduce the lives of the batteries, therefore preventing this from occurring would be helpful.

The OmniMaxbot currently only uses the two LMS100 LRFs and the Kinect for sensing its environment. Objects above or below the sensors are not detected, which
could pose a risk. There are two possible solutions for this. One would be to add secondary sensor systems to detect objects above or below the plane of the LRFs. The other would be to implement nodding or rotating LRFs in order to generate 3D scans.

Currently, the lids on the ash cans are clamped in place by workers. To further increase worker safety, a system should be developed and added to the OmniMaxbot to automate this step as well.

The base_local_planner package used for planning the local path prefers to act as a skid-steer drive, only reversing or strafing only when there is not enough room to rotate and move forward. This limits the usefulness of the Mecanum wheels. Further examination of path planning algorithms is recommended for better effectiveness of the omnidirectional properties of the Mecanum wheels.
References


Appendix A

Installing OmniMaxbot Software

These instructions explain how to install the OmniMaxbot’s software. It is assumed that ROS Hydro has already been installed and the catkin workspace initiated. If not, please follow the instructions shown at http://wiki.ros.org/hydro/Installation and http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment.

- Run the following in a terminal window: 
  
  ```
  sudo apt-get install ros-hydro-serial ros-hydro-serial-utils ros-hydro-sound-play libusb-1.0-0-dev
  ```

- Install the libphidget drivers using the instructions shown at http://www.phidgets.com/docs/OS_-_Linux

- Run the following commands in a terminal window:

  ```
  - sudo apt-get install git
  - git clone https://github.com/mars-uoit/omnimaxbot.git ~/catkin_ws/src/omnimaxbot
  - cd ~/catkin_ws
  - catkin_make --pkg phidgets
  - catkin_make
  ```
Appendix B

Start Up Procedure

The instructions listed here are to start the OmniMaxbot and run the full test launch file:

1. Ensure that the hardware on the OmniMaxbot is connected to the integrated computer
2. Connect the DC/DC converter to the power connector for the batteries
3. Connect a battery to each of the LRFs
4. Turn on the integrated computer
5. Turn on the AX2850s
6. From the command line on the integrated computer, run `roslaunch omnimaxbot_control camera.launch`
7. Close the terminal window, `ctrl-c`
8. On the workstation computer, attach to the OmniMaxbot’s wireless network
9. In a terminal window on the workstation computer, SSH into the OmniMaxbot and run `rosrun`
10. In another terminal window on the workstation computer, SSH into the Omni-
Maxbot and run `roslaunch omnimaxbot_control kinect.launch`

11. In a third terminal window on the workstation computer, run `roslaunch
omnimaxbot_control omnimaxbot_move.launch`