Matching Points to Lines: Sonar-based Localization for the PSUBOT

Kevin Blythe Stanton
Portland State University

Recommended Citation

10.15760/etd.6514
AN ABSTRACT OF THE THESIS OF Kevin Blythe Stanton for the degree of Master of Science in Electrical and Computer Engineering presented February 12, 1993.

Title: Matching Points to Lines: Sonar-based Localization for the PSUBOT.

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

Marek A. Perkowski, Chair

Michael A. Driscoll

Jean Scholtz

Eugene A. Enneking

The PSUBOT (pronounced pea-es-you-bought) is an autonomous wheelchair robot for persons with certain disabilities. Its use of voice recognition and autonomous navigation enable it to carry out high level commands with little or no user assistance. We first describe the goals, constraints, and capabilities of the overall system including path planning and obstacle avoidance. We then focus on localization-the ability of the robot to locate itself in space. Odometry, a compass, and an algorithm which matches points to lines are each employed to accomplish this task. The matching algorithm (which matches "points" to "lines") is the main contribution to this
work. The "points" are acquired from a rotating sonar device, and the "lines" are extracted from a user-entered line-segment model of the building. The algorithm assumes that only small corrections are necessary to correct for odometry errors which inherently accumulate, and makes a correction by shifting and rotating the sonar image so that the data points are as close as possible to the lines. A modification of the basic algorithm to accommodate parallel lines was developed as well as an improvement to the basic noise removal algorithm. We found that the matching algorithm was able to determine the location of the robot to within one foot even when required to correct for as many as five feet of simulated odometry error. Finally, the algorithm's complexity was found to be well within the processing power of currently available hardware.
MATCHING POINTS TO LINES: SONAR-BASED LOCALIZATION
FOR THE PSUBOT

by

KEVIN BLYTHE STANTON

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
In
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1993
TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Kevin Blythe Stanton presented February 12, 1993.

Mark A. Perkowski, Chair

Michael A. Driscoll

Jean Scholtz

Eugene A. Enneking

APPROVED:

Rolf Schaumann, Chair, Department of Electrical Engineering

Roy W. Koch, Vice Provost for Graduate Studies and Research
ACKNOWLEDGEMENTS

In a pursuit such as this, there are so many people who contributed in so many ways that it is difficult to know where to begin my acknowledgement of them. But I first want to give my heart-felt thanks to my wife Coreen. She has been my biggest fan, my greatest encourager, and is not a bad proofreader either. I thank her for all her hard work these last 4½ years and for her selfless dedication to our daughter. Next, a big thanks to Dr. Perkowski for his creativity, his conception of the PSUBOT, and for his flexibility in allowing me to pursue the topics of the PSUBOT which interested me. He has also been a great encouragement and has believed in me from the beginning. I would like to thank the rest of my committee: Dr. Michael Driscoll, Dr. Eugene Enneking, and Dr. Jeanne Schötz. Your input during the development and critique at the end of my research greatly improved the quality of my presentation and methods and are greatly appreciated. Specifically, Dr. Driscoll for his critical approach to scientific research and measurements and for helping me find missing pieces in my arguments; Dr. Enneking for his numerous explanations and helping me understand the mathematics beneath and around sum-of-squares procedures; Dr. Schötz for her "take a step back, and does this really make sense" approach and suggestions. Then to Doug Hall, for generously sharing with me his office, computer, counsel, wisdom, and an occasional chocolate chip cookie over the past two years. I have learned much from our conversations, and benefited even more from his friendship. To Brian Heerwagen for meticulously proof-reading the text of this document and suggesting numerous improvements, but more importantly, for his friendship which I highly value. Mom and dad, I could never say "thank you" enough. You're the greatest. Also to my sisters, extended family and many friends. You've been supportive and encouraging; I am very fortunate to have you all. Finally, I thank my God and Savior for giving me the ability, the opportunity, and the strength to finish what I had started.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>I  INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II GENERAL SYSTEM OVERVIEW OF THE PSUBOT</td>
<td>5</td>
</tr>
<tr>
<td>User Interface</td>
<td>5</td>
</tr>
<tr>
<td>Motor Control</td>
<td>8</td>
</tr>
<tr>
<td>Other External Sensors</td>
<td>10</td>
</tr>
<tr>
<td>Software Operating System</td>
<td>11</td>
</tr>
<tr>
<td>III PATH PLANNING/OBSTACLE AVOIDANCE</td>
<td>15</td>
</tr>
<tr>
<td>Overview of Path Planning Methods</td>
<td>15</td>
</tr>
<tr>
<td>Path Representation for the PSUBOT</td>
<td>23</td>
</tr>
<tr>
<td>Following the Path</td>
<td>25</td>
</tr>
<tr>
<td>Coordinating the Behaviors (The Pilot)</td>
<td>28</td>
</tr>
<tr>
<td>IV LOCALIZATION</td>
<td>31</td>
</tr>
<tr>
<td>Introduction</td>
<td>31</td>
</tr>
<tr>
<td>The need for Localization</td>
<td>32</td>
</tr>
<tr>
<td>Localization Techniques</td>
<td>32</td>
</tr>
<tr>
<td>Relevant Literature</td>
<td>36</td>
</tr>
<tr>
<td>Comparison of Localization Techniques</td>
<td>38</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Localization for the PSUBOT</td>
<td>40</td>
</tr>
<tr>
<td>Development of the Sonar-Based Matching Algorithm</td>
<td>48</td>
</tr>
<tr>
<td>Comments on the Matcher</td>
<td>59</td>
</tr>
<tr>
<td>Modifications for Parallel Target Lines</td>
<td>63</td>
</tr>
<tr>
<td>V TESTING THE MATCHER</td>
<td>67</td>
</tr>
<tr>
<td>Methodology</td>
<td>67</td>
</tr>
<tr>
<td>Expected Results</td>
<td>70</td>
</tr>
<tr>
<td>Types of Rooms</td>
<td>71</td>
</tr>
<tr>
<td>Sample Matching Sessions</td>
<td>72</td>
</tr>
<tr>
<td>Regions of Convergence</td>
<td>78</td>
</tr>
<tr>
<td>VI EVALUATION OF MATCHER RESULTS</td>
<td>105</td>
</tr>
<tr>
<td>Accuracy</td>
<td>105</td>
</tr>
<tr>
<td>Matching Frequency</td>
<td>106</td>
</tr>
<tr>
<td>Use of the Compass</td>
<td>107</td>
</tr>
<tr>
<td>Combining Localization Estimates</td>
<td>108</td>
</tr>
<tr>
<td>Asymptotic Complexity</td>
<td>108</td>
</tr>
<tr>
<td>Overall Execution Time</td>
<td>110</td>
</tr>
<tr>
<td>VII CONCLUSIONS/FUTURE WORK</td>
<td>114</td>
</tr>
<tr>
<td>Conclusions</td>
<td>114</td>
</tr>
<tr>
<td>Future Work</td>
<td>115</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>119</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>122</td>
</tr>
<tr>
<td>TABLE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>I</td>
<td>43</td>
</tr>
<tr>
<td>II</td>
<td>44</td>
</tr>
<tr>
<td>III</td>
<td>45</td>
</tr>
<tr>
<td>IV</td>
<td>73</td>
</tr>
<tr>
<td>V</td>
<td>74</td>
</tr>
<tr>
<td>VI</td>
<td>76</td>
</tr>
<tr>
<td>VII</td>
<td>77</td>
</tr>
<tr>
<td>VIII</td>
<td>78</td>
</tr>
<tr>
<td>IX</td>
<td>109</td>
</tr>
<tr>
<td>X</td>
<td>110</td>
</tr>
<tr>
<td>XI</td>
<td>111</td>
</tr>
<tr>
<td>XII</td>
<td>112</td>
</tr>
</tbody>
</table>

LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I  Odometry Results of Forward Motion (Distances in Feet)</td>
<td>43</td>
</tr>
<tr>
<td>II Odometry Results of Counter-Clockwise Motion (Distances in Feet)</td>
<td>44</td>
</tr>
<tr>
<td>III Odometry Results of Clockwise Motion (Distances in Feet)</td>
<td>45</td>
</tr>
<tr>
<td>IV Matching Trace of Good Match for Room 0: ( \Delta = (-5.0, -5.0, 0.5236), F = 1.5 )</td>
<td>73</td>
</tr>
<tr>
<td>V  Matching Trace of Bad Match for Room 0: ( \Delta = (5.0, -5.0, 0.5236), F = 1.5 )</td>
<td>74</td>
</tr>
<tr>
<td>VI Matching Trace for Room 1: ( \Delta = (6.0, 4.0, 0), F = 1.5 )</td>
<td>76</td>
</tr>
<tr>
<td>VII Matching Trace for Room 2: ( \Delta = (-3.0, 0.0, 0.1745), F = 1 )</td>
<td>77</td>
</tr>
<tr>
<td>VIII Matching Trace for Room 3: ( \Delta = (-3.0, -2.0, 0.122), F = 1 )</td>
<td>78</td>
</tr>
<tr>
<td>IX Asymptotic Complexity with ( n = 100 )</td>
<td>109</td>
</tr>
<tr>
<td>X  Asymptotic Complexity with ( m = 10 )</td>
<td>110</td>
</tr>
<tr>
<td>XI Asymptotic Complexity with ( N = 100 ) (Parallel)</td>
<td>111</td>
</tr>
<tr>
<td>XII Asymptotic Complexity with ( M = 2 ) (Parallel)</td>
<td>112</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>8</td>
</tr>
<tr>
<td>2.</td>
<td>11</td>
</tr>
<tr>
<td>3.</td>
<td>15</td>
</tr>
<tr>
<td>4.</td>
<td>16</td>
</tr>
<tr>
<td>5.</td>
<td>17</td>
</tr>
<tr>
<td>6.</td>
<td>20</td>
</tr>
<tr>
<td>7.</td>
<td>23</td>
</tr>
<tr>
<td>8.</td>
<td>25</td>
</tr>
<tr>
<td>9.</td>
<td>26</td>
</tr>
<tr>
<td>10.</td>
<td>27</td>
</tr>
<tr>
<td>11.</td>
<td>30</td>
</tr>
<tr>
<td>12.</td>
<td>37</td>
</tr>
<tr>
<td>13.</td>
<td>39</td>
</tr>
<tr>
<td>14.</td>
<td>40</td>
</tr>
<tr>
<td>15.</td>
<td>42</td>
</tr>
<tr>
<td>16.</td>
<td>43</td>
</tr>
<tr>
<td>17.</td>
<td>43</td>
</tr>
<tr>
<td>18.</td>
<td>46</td>
</tr>
<tr>
<td>19.</td>
<td>46</td>
</tr>
<tr>
<td>20.</td>
<td>48</td>
</tr>
<tr>
<td>21.</td>
<td>49</td>
</tr>
</tbody>
</table>

1. Wheel encoder outputs
2. Object hierarchy and interaction diagram
3. Sample building layout
4. Visibility graph
5. Cell decomposition
6. Voronoi diagram
7. (a) A quad (b) A channel-path and obstacles
8. Typical bug2 path trace
9. Goal seeking parameters
10. Robot reaching edge of obstacle
11. Posture diagram
12. Flexibility vs. complexity
13. Accuracy vs. complexity
14. Calculating new posture based on wheel distances
15. Lateral odometry errors of three trials
16. Angle errors for counter-clockwise rotation
17. Angle errors for clockwise rotation
18. Reliable odometry distance
19. Internal compass signals
20. Sample room layout
21. Simple hypothetical sonar scan of a room
22. Infinite line representation .................................................................................................. 52
23. Distance from a point to a line segment ............................................................................ 53
24. Theoretical shape of sum of mean squares ......................................................................... 56
25. Parallel target lines .............................................................................................................. 61
26. Example of parallel target lines ............................................................................................ 63
27. Generation and evaluation of matcher errors ...................................................................... 66
28. Simulating odometry errors/corrupting an image ............................................................... 67
29. Sum of the squared errors versus ΔP example .................................................................... 69
30. Room 0: Good matching session .......................................................................................... 71
31. Room 0: Bad matching session ............................................................................................ 73
32. Room 1: Matching in a large rectangular room ................................................................. 74
33. Room 2: corrupted corridor image .................................................................................... 76
34. Room 3: intersecting corridors, raw data .......................................................................... 76
35. Room 0: region of convergence ............................................................................................ 78
36. Room 0: region of convergence ............................................................................................ 78
37. Room 0: Sum of squares versus ΔP ..................................................................................... 80
38. Room 0: region of convergence (simplified room layout) ................................................... 80
39. Room 0: region of convergence ............................................................................................ 81
40. Room 1: region of convergence ............................................................................................ 82
41. Room 1: region of convergence ............................................................................................ 82
42. Room 1: region of convergence ............................................................................................ 82
43. Room 1: region of convergence ............................................................................................ 83
44. Room 1: region of convergence ............................................................................................ 83
45. Room 1: region of convergence ............................................................................................ 83
46. Room 1: region of convergence ............................................................................................ 83
47. Room 1: region of convergence ............................................................................................ 84
ix

48. Room 1: region of convergence................................................................. 85
49. Room 1: region of convergence................................................................. 85
50. Room 1: region of convergence................................................................. 86
51. Room 1: region of convergence................................................................. 86
52. Room 1: region of convergence................................................................. 86
53. Room 1: region of convergence................................................................. 86
54. Room 1: region of convergence................................................................. 87
55. Room 1: region of convergence................................................................. 87
56. Room 1: region of convergence................................................................. 87
57. Room 1: region of convergence................................................................. 87
58. Room 1: region of convergence................................................................. 88
59. Room 1: region of convergence................................................................. 88
60. Room 1: region of convergence................................................................. 88
61. Room 1: region of convergence................................................................. 88
62. Room 1: region of convergence................................................................. 89
63. Room 1: region of convergence................................................................. 89
64. Room 1: region of convergence................................................................. 89
65. Room 1: region of convergence................................................................. 89
66. Room 1: region of convergence................................................................. 90
67. Room 1: region of convergence................................................................. 90
68. Room 2: region of convergence................................................................. 91
69. Room 2: region of convergence................................................................. 91
70. Room 2: region of convergence................................................................. 92
71. Room 2: region of convergence................................................................. 92
72. Room 2: region of convergence................................................................. 92
73. Room 2: region of convergence................................................................. 92
74. Room 2: region of convergence ........................................................................ 93
75. Room 2: region of convergence ........................................................................ 94
76. Room 2: region of convergence ........................................................................ 94
77. Room 2: region of convergence ........................................................................ 95
78. Room 2: region of convergence ........................................................................ 95
79. Room 2: region of convergence ........................................................................ 95
80. Room 2: region of convergence ........................................................................ 95
81. Room 2: region of convergence ........................................................................ 96
82. Room 2: region of convergence ........................................................................ 96
83. Room 2: region of convergence ........................................................................ 96
84. Room 2: region of convergence ........................................................................ 97
85. Room 2: region of convergence ........................................................................ 97
86. Room 2: region of convergence ........................................................................ 97
87. Room 2: region of convergence ........................................................................ 97
88. Room 2: region of convergence ........................................................................ 98
89. Room 3: region of convergence ........................................................................ 99
90. Room 3: region of convergence ........................................................................ 99
91. Room 3: region of convergence ....................................................................... 100
92. Room 3: region of convergence ....................................................................... 100
93. Room 3: region of convergence ....................................................................... 100
94. Room 3: region of convergence ....................................................................... 100
95. Room 3: region of convergence ....................................................................... 101
96. Room 3: region of convergence ....................................................................... 102
97. Room 3: region of convergence ....................................................................... 102
98. Room 3: region of convergence ....................................................................... 103
99. Room 3: region of convergence ....................................................................... 103
100. Room 3: region of convergence ................................................................. 103
101. Room 3: region of convergence ................................................................. 103
102. Room 3: region of convergence ................................................................. 104
103. Time complexity vs. model size ............................................................... 110
104. Time complexity vs. model size ............................................................... 111
105. Time complexity vs. model size ............................................................... 112
106. Time complexity vs. model size ............................................................... 113
CHAPTER I

INTRODUCTION

The ability to move freely is taken for granted by most people. If we want a snack, we walk to the concession area of our building, deposit the appropriate fee, and go back to our office. It's simply an every-day occurrence. However, there are people whose freedom to travel freely is restricted for physical reasons.

Electric wheelchairs are used by many who are unable to move from one room to the next without mechanical assistance. To operate such a wheelchair the user generally must be able to control both speed and direction by moving a joystick with their hand. Changes in inclines or friction on the wheels require the user to adjust the position of the joystick to keep the wheelchair heading in the desired direction.

Certain people who need such transportation are unable to effectively provide such control due to their physical condition. Some people with Cerebral Palsy, for example, have difficulty with hand-eye coordination and therefore cannot safely operate such a vehicle.

People without the use of their arms and legs cannot steer such a wheelchair and must resort to linkages connected to their head to control vehicle movement. The ultimate example of the shortfalls of conventional electric wheelchairs is a person who is both blind and unable to walk. Such a person would require outside assistance to guide them and their wheelchair as they would be unable to determine where to steer the wheelchair.

The Portland State University roBOT (PSUBOT) is an attempt to provide a system with which such persons could gain greater independence, freedom, and safety in navigation. [41], [40].
The goal of the PSUBOT project is to develop a computer-assisted wheelchair which is voice activated and is able to navigate autonomously throughout a known building with minimal interaction with the user. It is not intended for use outdoors as many serious dangers as well as major additional technical (and financial) challenges would be introduced.

There have been other attempts at robotic assistance for the disabled. Engelhardt has published a wealth of papers on human-machine integration (interaction) specifically targeted to the elderly and disabled (see [16] for an overview). The robot “INCH” [39] was an intelligent wheelchair with the ability to interrupt user control in the event of an impending collision, and Madarasz et al developed an image processing system for a wheelchair which was able to navigate in hallways by looking for the converging lines of baseboards [30].

To accomplish our goal, several additions were made to an Everest & Jennings electric wheelchair. Incremental optical encoders were connected to the wheels with a sprocket and chain to acquire the rotation of each wheel independent of the other. An electronic compass was added to provide heading information, and a sonar device was attached to a motor above the wheelchair to acquire distance information. Finally, all computation occurs on an 80386-based IBM clone on-board.

The first step in reaching a goal is determining the robot’s location. Unless a robot can “see” its goal, it does not know where to begin in pursuit of it. If the robot is given the command: “go to the elevator” it must first know its location before planning a path. Once the path is planned it must continuously know its location to determine whether the plan is being successfully followed. Odometry (monitoring the rotation of the wheels) is used in the short term (around 10 feet) and is accurate in that range. However, location information must come from outside the robot to keep odometry errors (which inherently accumulate) from getting out of hand. This is accomplished by scanning the area with the sonar and matching the resulting two-dimensional image to the known dimensions of the room. In this way the robot determines its position relative to the walls of the room (which don’t often change) and the result is a more
reliable location estimation.

Requiring the robot to know the layout of a building is realistic for two reasons. First, for the PSUBOT to be useful in a building, it would need to know which hallways lead where so it can plan the shortest route. Aliases for locations will also be necessary to help the user specify the desired goal instead of being required to indicate the location in terms of coordinates. The second justification for requiring a map of the building is that computer-based drafting tools are quickly becoming the standard for generating and maintaining building floorplans, though it will probably be necessary to first verify the accuracy of such drawings. A translation program could be developed to generate a map in the form used by the PSUBOT from the drafting files.

It is also important to realize that the problem of localization is not specific to the PSUBOT but is a requirement of robots in general. Therefore this research may be applied to any mobile robot operating in a mapped building.

The PSUBOT embodies many possible research topics for future students: From vision to voice recognition to user interfaces to path planning. We do not attempt to solve them all, but only outline the needs and present possible solutions. Voice recognition, for example, is performed by a very simple, commercially available product. The user interface is currently very crude, and only the simplest paths are planned. Localization, however, is the major thrust of this research and therefore much of this work is dedicated to describing, developing, testing, and evaluating its methods.

The following chapters are organized as follows: Chapter II gives a more detailed overview of the PSUBOT subsystems and the software used to control it. Chapter III presents the problem of planning a path from one location in a building to another and then demonstrates current methods for generating and following them. Chapter IV presents localization, the major contribution of this work. Here several methods are reviewed, one is chosen and developed, and improvements are made. In chapter V the localization algorithm is tested on real data taken from the Portland Center for Advanced Technology (PCAT) building where the robot was developed.
Tests are performed which demonstrate the algorithm's ability to find the robot's location in spite of various errors. In chapter VI we evaluate the performance of the algorithm and relate the results to the other types of localization used by the PSUBOT. Finally, conclusions are drawn in chapter VII and improvements to both the localization algorithm and the PSUBOT in general are suggested.

As the primary developer of the PSUBOT the author includes in the Appendix: technical details, program examples, and "words of wisdom" for all those who follow.
CHAPTER II

GENERAL SYSTEM OVERVIEW OF THE PSUBOT

The PSUBOT system is broken into four subsystems: User Interface, Motor Control, Other External Sensors, and the Software Operating System. The following sections discuss these fundamental components of the PSUBOT.

USER INTERFACE

Due to the limited mobility of those who would use the PSUBOT, the conventional hand-operated mechanical interface (joystick) was deemed inappropriate. Other existing methods include control by: breath, eye movement, and linkages connected to the chin for those able to move their head freely.

However, if the robot were able to listen to the user's commands, then such mechanical and sometimes awkward mechanisms could be replaced by a microphone. The term listen need not be restricted to what would be considered plain English, but could be some other specialized method of interface that would better fit the particular user. For the sake of discussion we will use speech to refer to this broader mode of communication.

Input Structure

The input language is well structured and hierarchical, further simplifying the task of recognition since only a subset of all possible words need to be considered at a particular position in the input statement. The highest level of the command tree has three categories: 'Utility', 'Information', and 'Move'.
The Utility and Information subtrees will be discussed only briefly, while the motion commands will be addressed in more detail since they are more relevant to this work.

Utility. The computer will have a large amount of state information, some of which may be of interest to the user. For example, the PSUBOT will know the time and date, battery life remaining, and perhaps even the weather conditions outside. All this and more will be accessible by the user from the Utility subsystem. The computer could also store phone numbers and appointments, play a game, and even play the radio. All this is for the purpose of making the PSUBOT as friendly as possible and to make it act more like an able companion than a machine.

Information. It is also our goal to make the user feel like an intelligent partner in the operation of the PSUBOT rather than simply an extension of a machine. To this end (among others) the user will be given the opportunity to provide information to the PSUBOT to help it make decisions. This may be done voluntarily (from the "information" subtree) or in response to a request for information by the PSUBOT. While additional information may add to the performance of the overall system, it must not be necessary; i.e. the PSUBOT must have the ability to navigate without human involvement, yet critical decisions may well be aided by the cooperation of something significantly more perceptive than itself (and many many times more complex!).

As an example, consider the PSUBOT and user traveling down a corridor and meeting a pile of boxes nearly obstructing the hallway. The PSUBOT could ask the user whether he/she suggests they try going around to the left or the right. Not only could the user likely make a decision in a glance (faster than the PSUBOT), but the user could probably take more factors into account than could a machine algorithm. Clearly a blind person would have a difficult time making such a suggestion in general, but the capability could still prove useful in certain circumstances.

In summary, we want to make the user feel helpful and not simply a person riding helplessly on a machine. Conversely, it would seem a terrible waste to limit the PSUBOT to internal decision making strategies only, when a wonderfully complex and intelligent human is available
Move. Voice commanded motion is available through both high and low level constructs. The low level commands simply replace and extend the basic joystick capabilities. These include, but are not limited to: forward, stop, turn, right, left, slower, and faster. These capabilities allow the user fine grained control over wheelchair motions where the necessary higher level commands are absent or found to perform unsatisfactorily in a particular situation.

High level commands extend the capabilities of the wheelchair to an autonomous mobile robot. Commands include: "go to room 312", "find the drinking fountain", "go to my office". Clearly the execution of these commands requires an intelligent navigation system (see Chapter IV).

Natural language systems are currently in place for user/computer interaction within restricted application domains, for example, a lathe in a machine shop where the user gives a verbal description of the part to be machined. The requirement of a narrow domain is currently necessary to simplify the number of words which must be searched and limit the context in which those words may be used in order to eliminate many of the ambiguities which would otherwise result. Although the PSUBOT would likely have a small enough vocabulary for a natural language interface, we chose a more rigid grammar for the current version of the PSUBOT as such refinement is not the major thrust of this work.

A simplified grammar is shown below for some commands. Literals are in quotes " " and optional parameters are in brackets {}.

"stop"
"forward {fast, slow}"
"reverse {fast, slow}"
"turn {left, right}"
"veer {left, right}"
The only high-level command the PSUBOT is currently able to execute is "goto" as it allows testing of localization and motion planning algorithms. Goto is clearly a more difficult command to execute than those above and requires "intelligence".

Speaker Dependence

The voice input system is pseudo-speaker independent, i.e. the computer is trained to recognize the user voice, though it will also recognize anyone if they pronounce the commands similarly and use similar inflections. True speaker independent recognition (designed to understand everyone without training) is an unnecessary luxury since there will usually be only one user of the robot, and thus only one speaker. Speaker dependence also adds a small amount of security since a non-user shouting commands to the PSUBOT would need to sound similar to the rightful owner for the PSUBOT to respond. Voice fingerprinting is a possible extension to the current voice system. In this way the characteristics of the user's voice would serve as a password to protect against foul play.

For an example of a voice controlled robot (albeit an arm) see a work by Mital [35].

MOTOR CONTROL

Motion of the PSUBOT is accomplished by two large rear wheels each of which is powered by a permanent magnet motor and belt. The motors are Pulse-Width Modulated (PWM) to provide variable speed by a solid-state control module mounted on the back of the wheelchair.
Normally, the input to the module is a joystick positioned for easy access by the user's right hand. The two potentiometers within the joystick are connected at right angles and oriented in an "X" so that a forward thrust on the joystick displaces each potentiometer by the same angle. The output for each axis (each wheel) is a push/pull pair with min/max values of 2.8 Volts ($V$) and 3.6 V respectively. In other words, the voltages $V_{L1}$ and $V_{L2}$ are related by the equations:

\[
\frac{V_{L1} + V_{L2}}{2} = 3.2V
\]  

and

\[2.8V < V_{Li} < 3.6V\]  

where $i = 1, 2$ and $V_{L1}$ and $V_{L2}$ are the push/pull control voltages for the left wheel. Of course the same relationship is true for the right wheel controller. It becomes the task of the computer to generate these signals in place of the joystick. This is accomplished with the PSUBOT motor interface described in the Appendix which employs a pair of Digital-to-Analog Converters (DAC's).

**Feedback**

As with any feedback control system, the PSUBOT has sensors to measure the controlled quantity (speed). This is accomplished by a pair of optical encoders connected to each wheel by a chain and sprocket. The encoders require +5V and ground and output two TTL-level square waves which are proportional to the angular rotation, and are 90° out of phase. From the gear ratios, encoder resolution, and tire size, we calculate the distance traveled for each cycle of the square waves. From their phases we can determine direction. "A" before "B" implies forward and "B" before "A" implies reverse as shown in Figure 1. For a good description of this and other types of encoders the reader is referred to [21].

A custom feedback board counts up or down depending on the phase such that the aggregate distance is available to the computer each time the control loop is executed.
Control Algorithm

A PID (Proportional + Integral + Derivative) feedback control algorithm runs on the PC and is used to maintain accurate wheel velocities. The control algorithm executes one loop (samples distance, calculates new control values, and outputs them), then exits allowing other functions to be performed on the (currently) uniprocessor PSUBOT (See section on the Software System).

OTHER EXTERNAL SENSORS

In addition to motor feedback sensors, other maneuver-based sensor systems assist with localization and obstacle avoidance. A sonar device is located at a height of approximately six feet above the floor on a stepper motor to provide rotation. The output of the sonar device is a distance and an angle which represent the number of feet to the nearest obstruction at the direction in which the sonar transducer points. This data is used to determine the location and heading of the robot as discussed in chapter IV.

There are also sonar transducers planned which will be located one foot above the ground around the perimeter of the PSUBOT for use with close proximity (less than ten feet) obstacle avoidance as described in chapter III.

A flux-gate compass is another of the contributing sensors to the localization system. Global heading is thus determined at a resolution of eight bits (0-255). The compass was a commercial product for automobiles sold by Radio Shack and modified by us to allow the computer to digitize and process the necessary analog signals to extract heading information. Again, the
SOFTWARE OPERATING SYSTEM

Object Oriented

All the major software subsystems were designed using an object oriented approach as described in [23]. This is easily translated into classes and member functions when implemented using C++. It is stressed that a good object oriented software solution comes from a good Object Oriented Design (OOD).

A good OOD comes by asking four questions:

- What are this object's responsibilities
- What hidden information does it hold
- How does it let others view its hidden data
- How does it allow others to modify its hidden data (if at all)

rather than

- What does this function receive as parameters
- What global constructs does it access
- What operation does it perform on the data
- What does it return

As a result of this type of thinking about the problem, a larger picture of the overall system emerges, related functions are grouped together, and a well-defined interface between classes (objects) is defined and enforced. It is the author's view that a bad OOD will probably yield worse program code than its corresponding functional design, but a good OOD lends itself to cleaner code in addition to fewer opportunities for programmer error. Cox in [5] also draws out some of the specific advantages of using an object oriented language such as C++ to write programs which deal with physical systems, especially robotics, since objects can be made to represent real systems (like a wheel).
Another advantage to C++ is guaranteed initialization. The Sonar object, for example, performs a *home* operation on startup. There is no way to invoke the Sonar object without this initialization occurring automatically (unless the Sonar object itself is changed).

Finally, the ability to test lower-level objects quickly and independently of others was a great advantage during the development stage.

**Framework**

The software for the PSUBOT is described in terms of an object diagram as seen in Figure 2.

![Object hierarchy and interaction diagram.](image)

**Figure 2.** Object hierarchy and interaction diagram.

The boxes represent the objects which perform the particular task and the arrows indicate the
direction in which messages are initiated. For example, the Pilot may send a message to the
Wheels telling them to increase their speed, or may send a message asking how fast the left wheel
is turning. So the flow of information is actually bidirectional but the Pilot is in control.

For simplicity, we will refer to objects by their names as though they were people carrying
out tasks and communicating with other objects.

A more detailed picture would include all the message names, parameters, and the return
values. The reader is referred to the Appendix for additional information.

Execution starts with the User object which interacts with the user through voice commu­ni­cation. Depending on the type of command or question spoken, one of the next three objects
interpret and perform the desired operation. If the command is a "motion" type, the Planner
takes over and begins the process of path planning and execution. The Planner first asks for a
Global Path (sequence of rooms to reach the goal) from the World Database. Then the Planner
asks the World Database for a model of the current Room. The Room is a small part of the world
model making up the entire building. The planner than generates a local path through the current
room by retrieving geometric information from the Room. To say the robot operates in a known
environment means it must know the dimensions of the current room and the obstacles present. It
is at this point that such information comes into play. Path planning is discussed in Chapter III.

Next, the Pilot begins by asking the Navigator where it is and commanding the Wheels as it
follows the local path. It is the task of the Pilot to reach the local goal in the room while avoiding
collisions.

The Wheels implement the PID controller which controls the two differentially controlled
wheels of the wheelchair.

The Navigator is the main emphasis of this work, as it is responsible for localizing (finding
the current location) based on sensor data. It also gives the Pilot information on impending colli­
sions while drawing on information from the Sonar and Compass.
Other types of commands - information and utility - are not further investigated, though information will certainly be an integral part of the Pilot in the future. The Secretary will have the responsibility to handle commands of utility type.

It can be noted that if the User is able to receive commands while the Pilot is working, there must be some level of multitasking present in the structure. The current solution is a custom non-preemptive scheduler which, when asked, tells the object whether it may execute or not. So the execution sequence in the object tree begins with the user, passes control to the Secretary (which may or may not try to do anything useful), then to the Planner, to the Pilot, and so on until all the objects are given the chance to execute. This in turn requires the ability for the objects to save state information so that each time a module is told to work it can begin where it left off. More information on the specifics of this scheme may be found in the Appendix.
CHAPTER III

PATH PLANNING/OBSTACLE AVOIDANCE

For a robot to navigate intelligently within the real world it must be able to plan a path to its goal and deal with unexpected events. Often these events are obstacles in its path due to displaced furniture, people walking down the hall, or a disoriented robot. This chapter describes what a path is, methods of generating it, following it, and what to do if something blocks the path.

OVERVIEW OF PATH PLANNING METHODS

A path is something which shows the robot where to go to reach its goal. Ideally, a path for the PSUBOT would be free from obstacles and would minimize the distance traveled (or some other quantity). There are two fundamental prerequisites of a path: (1) an initial position, and (2) a final position (the goal). The initial position of the PSUBOT is derived from its surroundings (see Chapter IV) and the goal is supplied by the user ("go to the elevator"). A path, then, should serve to connect these two points in such a way that the application of the Path to a Pilot will result in the robot achieving the goal state. Given the layout of the building and the location and sizes of known obstacles, the problem may be reduced to a graph search.

However, for an entire campus the number of nodes could number in the thousands, requiring significant computation to find an optimum solution. This excess in complexity is unnecessary with a little hierarchical decomposition of the world model. First, a graph is generated containing all the rooms of the current building as nodes, and openings between them represented as arcs. This graph describes all possible routes in the building. A tree searching algorithm may
then be used to produce the set of nodes (rooms) through which the distance (or some other pertinent variable) to the goal is minimized. This procedure may then be repeated for buildings and their interconnections as shown in Figure 3.

Note that some rooms (corridors) may be subdivided ("E" and "F" in the example) to allow for a more regular room description. A search would therefore involve about as many nodes as there were rooms and corridors in the building - in the order of tens or hundreds. An example Global Path from room "B" to room "G" is \{B,E,F,G\}. Actually the global decomposition starts at a higher level - buildings, floors, rooms - but is simplified for this discussion. See [33] and [10] for a complete description of global planning methods tailored especially to the PSUBOT.

![Figure 3. Sample building layout.](image)

Once a sequence of rooms is obtained, a local path is generated for the first room which terminates at the entrance to the next room in the Global Path. The pilot then attempts to follow the path. If successful, a path through the next room is planned, and so on until the ultimate goal is reached. In this way the complexity of the local Path Planner is greatly reduced since much tree-pruning is done at the global level.

There are two major ways to represent the world in robotics research today. The first (both historically and in this presentation) is the spacial representation where the environment is laid out on a two (or three) dimensional grid. The resulting algorithms for operating within such a framework are inherently numerical in nature. The second method of representing the
environment is more symbolic in nature and is based more on artificial intelligence techniques than equations and highly processed sensor data.

Spacial representations are well suited for the PSUBOT. In brief, this is because we would like the PSUBOT to follow the best path to the goal, and Spacial representations of the world and the path allow such paths to be generated and followed. In contrast, while symbolic representations of the world are more adaptive, they do not easily find the shortest distance between two lines. We will first describe several dominant spacial methods and then briefly discuss some which are more symbolic in nature.

The Visibility Graph Method

Lozano Pérez first formalized the visibility graph approach in his landmark paper: [28]. The procedure begins with all obstacles modeled as rectangles, each of which has four vertices.

Then, for any given set of obstacles, a visibility graph is generated in the following way:

**Algorithm 1: Visibility Graph**

FOR each vertex $u$

Connect $u$ with an arc to every other vertex $v$ which is "visible" from $u$, where two nodes are "visible" if the line connecting them does not intersect any obstacles.

ENDFOR

By definition, each vertex is visible from the two adjacent vertices of the same obstacle. Figure 4 shows a possible visibility graph.

Once the graph is generated - which is necessary only when the room changes - the initial and goal nodes are added into the graph with arcs to each "visible" node. If there is a simple path between the initial and goal nodes (a straight line), that path will be part of the whole visibility graph (and thus a solution). The graph may then be searched for a path meeting the minimization criterion.

In all but the trivial case, the path will take the robot zero distance from the obstacles
This is not a problem if the robot is a point, but for real systems there must be another step included to keep the robot a safe distance from the obstacles. The process proposed in the above paper is called "obstacle growing" and involves increasing the dimensions of each obstacle artificially so that the real robot (when touching the grown, virtual obstacle) will still be a safe distance from the real obstacle. In other words, the robot may then be treated as a point.

If the robot is not circular, additional problems arise (how much should the obstacles grow) since different orientations of the robot yield different distances from its center to its exterior. This is also addressed in the paper by Lozano-Pérez [28].

Cell Decomposition

This method, also known as "quad-tree", differs from the visibility graph method in several respects. First, obstacles may be of arbitrary shape. Second, a path need not follow the outline of obstacles which are blocking the path, but may take a more natural curving path as seen in Figure 5.
To perform Cell Decomposition the space is split into several rectangular pieces and labeled "empty" or "full" depending on whether no obstacle is within the borders or at least one obstacle is within the borders of the "cell", respectively. The cells labeled "full" are then decomposed further by dividing each of them into smaller areas, each of which are labeled "full" or "empty" in the same way. The process continues until some minimum cell size is reached.

Next, every cell becomes a node in a graph and adjoining sides become arcs between the nodes. The arcs commonly are weighted with the distance between cells. The graph (as in the visibility graph method) may then be searched for a sequence of cells from the cell containing the initial position to the cell containing the goal. The actual path may then be constructed by connecting the midpoints of the sides of adjacent cells in the sequence [27].

A disadvantage to this approach is the complexity involved when there are many obstacles, since the number of nodes (and arcs) in the graph may grow very large. An advantage is that gross planning may be performed with little computation if the minimum sized cell is sufficiently large, increasing the resolution until either a path is found or the minimum cell size is reached.

Figure 5. Cell decomposition.
(path not found). A side effect of this strategy is that a path which exists in practice may remain
unfound due to a lack of resolution.

Although the resulting path is often smoother than with the Visibility Graph method, it is
still piece-wise linear and produces jerky movements on execution. The method discussed in the
next section, Voronoi diagrams, generates smooth paths: which are equidistant from the nearest
obstacles.

A variation of the cell decomposition method is called occupancy grids [14], [15]. As with
Cell Decomposition, space is divided into rectangular (or square) cells. However, instead of a
cell being either "occupied" or "not occupied", the cell contains the probability that it is occu­
pied. This information is generally deduced from a range sensor. As the robot moves through
space it scans the area and makes modifications to the relative "occupancy" based on the sensor
readings. If the ranging device indicates an obstacle at grid location (12,27), the probability of
occupation for that cell is increased. But for the cells between the robot and (12,27), the proba­
bility of occupation is reduced. Planning is then accomplished by finding the sequence of cells to
the goal which corresponds to the lowest probability of collision with an obstacle. An advantage
of this method over simple Cell Decomposition is that raw sonar data may be used directly in
determining where obstacles are, and there is a history effect since the results of multiple readings
are combined from different locations from which more reliable conclusions may be drawn.

This method does not require a priori knowledge of the environment and it is therefore
more difficult to specify a goal in absolute coordinates. However, its ability to adapt quickly to
changes in the structure of the environment (i.e. people, vehicles, clutter) makes it useful in
unstructured environments and outdoors as in the CMU Mobile robotics lab [42].

Voronoi Diagrams

Another approach has been used to generate paths through a known set of arbitrary shaped
obstacles and is known as the Voronoi diagram [37]. With its origins in mathematics the Voronoi
diagram has found its way into numerous applications such as the simulation of crystal growth and locating a business as far as possible from any of its competitors.

A Voronoi diagram is the locus of points equidistant from the nearest two obstacle surfaces (where walls are obstacles too) as illustrated in Figure 6. After initial and goal positions are added to the graph, the whole diagram is searched for the best path.

Two additional segments are added to connect the initial and final points to the Voronoi curve and then the whole curve is searched for the optimum path. If the obstacles are all piecewise linear then the diagram will consist of line segments and parabolic arcs [26].

An advantage to this method is that the path stays as far from obstacles as possible. A modification known as “Simplified Voronoi Diagrams” was customized for robotic path planning by lowering the computational complexity and dealing with deformities to the diagram because of rotations of the robot (due to growing the obstacles) [11].

Figure 6. Voronoi diagram.
Potential Fields

To understand this method, one can think of the robot and all obstacles as negatively charged particles and the goal as a positively charged particle. Additionally, the robot is the particle allowed to move. [24]. For this method a mathematical model is generated which describes the strengths of the repelling negative fields generated by the obstacles and the attracting positive field generated by the goal so that if the robot particle were allowed to travel along the gradient of the curve it would eventually find itself at the global minimum (the goal). If, that is, there are no local minimums in which to become trapped.

Local minimums are the major problem with the Potential Fields approaches to path planning, and only special algorithms designed to detect and eliminate such null areas are able to spur the robot on to the ultimate goal. Another problem with this approach is the numerical computation required to generate and search the Potential Fields function for the solution path. For an overview of the Potential Fields method of path planning, see [25].

Symbolic Representations

A major and controversial research direction coming from the MIT Artificial Intelligence Laboratory is the subsumption approach to robot planning and motion. Using this approach, the robot knows nothing of the geometry and layout of a building in space, but rather moves from one place to another by finding, labeling, and moving between landmarks as in the robot Toto [31] which consists of several "behaviors" acting in parallel which result in the emergent behavior. The basic philosophy of the subsumption architecture is that by assimilating the operation of simple biological systems such as ants, bats and cockroaches, surprisingly versatile and robust systems result. [3]. Though the resulting robots have received much media attention for their innovation and very small hardware requirements (one robot "Squirt", though only slightly larger than one cubic inch, is able to hide in dark corners and run from loud noises [19]) some question the usefulness of such simple and reflex-oriented robots. It is, however, difficult to
discount their ability to operate and adapt to unknown and changing environments.

PATH REPRESENTATION FOR THE PSUBOT

The geometric method of representing an environment is particularly attractive to a system such as the PSUBOT where the structure of the building is known beforehand. For a thorough theoretical description of the above methods refer to [26].

A problem with many geometry-based methods of path planning is that they are very rigid (made up of line segments and arcs). This means that if an obstacle of non-zero width is found in the path, replanning of the entire remaining path is often required - which often is very computationally expensive. It would seem that a simpler and more flexible method would be advantageous.

Not Lines, Channels

The path description chosen for the PSUBOT is a combination of the Visibility Graph method and a construct of our own, called "Quad Channels" (not to be confused with "quad trees"). The basic notion of a path is hence expanded from one dimension to two. Figure 7. shows the structure of both a Quad and a Quad Channel.

A Channel is an obstacle-free region in which the robot may proceed to the local goal (the goal of the current room). Our use of the term "goal" may refer to the local goal or the global goal depending on the context in which it is used.

The Channel is made up of a sequence of quadrilaterals ("Quads") each of which has one or two of their four sides in common depending on whether they are at the beginning, middle, or end of the Channel. A Quad Channel may be simplified to a piece-wise linear path by connecting the midpoints of the common sides of the Quads to each other and to the initial and goal points. The advantage of the Channel over the resulting linear path is that the robot may maneuver
around an unexpected (unmapped) obstacle without replanning. Such obstacle avoidance is allowed as long as the robot does not move outside the Channel. This provides flexibility for the robot while ensuring that it doesn’t wander off indefinitely in an attempt to maneuver around a wide, unforeseen obstacle.

Figure 7. (a) A quad (b) A channel-path and obstacles.

Path Generation

A modified visibility graph algorithm would seem the best prospect for generating Channels since the borders of obstacles make natural boundaries for the Channel, and all obstacles are to be represented as rectangles in the World Model. If needed, a three sided Quad could be generated by setting the length of one side to zero. In this case all other sides would be required to be non-zero.

The input to the Channel generator (path planner) is the dimensions of the room, all obstacles - approximated by rectangles - and the initial and goal positions within the room. The output is an ordered collection of Quads which make up the Channel.

The actual algorithm for generating the Channel is left for future researchers as it is not within the scope of this work and is included only as a part of the overall framework for the PSU-
FOLLOWING THE PATH

To solve the real world path-following problem, two fundamental questions must be answered: (1) "Where are we?", and (2) "Where should we go now?". If we cannot answer question (1), we cannot know when the goal is reached. Clearly "where to go" is a fundamental question simply because motion is the main function of the robot. We now discuss localization (where are we) followed by navigation (where to go now).

Localization

The first question is known as the "Localization Problem" and has been studied by many researchers, including: [5], [10], and [9]. It is formulated as follows: Given the world as perceived by available sensors (often with conflicting conclusions), what is the best estimate of the robot's position and orientation. In our case we have only sonar (distance, $\phi$), a compass (heading), and odometry (wheel rotations). If the intention is to navigate within a Channel, the computer must have the ability to deduce the robot's current position to ensure compliance. Localization is the major emphasis of this work, and is discussed extensively in chapter IV.

Commanding Motion

Once the robot has the ability to determine its current position in a room, the next function it will require is the ability to follow the path (channel). A simple algorithm for following a path can be imagined where the robot identifies an immediate goal at the midpoint of the opposite side of the Quad and reaches it through one rotation and one forward movement - assuming there are no unexpected obstacles. In reality, however, rooms often contain people and other unmapped obstacles. Therefore, to perform acceptably the robot must have the ability to deal with obstacles and/or an incorrectly specified path (one which even contains known obstacles). Lumelsky describes a method for maneuvering around unexpected obstacles. He names the algorithms
"bug1" and "bug2" [29]. Bug1 is shown to have a better worst case time complexity while bug2 has a lower average time complexity. Both guarantee a solution when one exists.

Simplified algorithm definitions follow:

**ALGORITHM 2: bug1:**

1. Move toward the goal until:
   (a) the goal is reached. The procedure stops.
   (b) an obstacle is encountered, define current location as H. Goto 2.
2. Follow the obstacle boundary. If the goal is reached, stop. After having traversed the entire boundary of the obstacle (H reached again), calculate the point along the boundary which was closest to the goal. Move back to that point. Go to 3.
3. If the goal is deemed "unreachable", stop. Otherwise go to step 1. The goal is unreachable if the line from H to the goal intersects the current obstacle.

**ALGORITHM 3: bug2:**

1. Move along the straight line which connects the starting point to the goal point until:
   (a) The goal is reached. The procedure stops.
   (b) An obstacle is encountered at point H. Go to step 2.
2. Follow the obstacle boundary until one of the following occurs:
   (a) the target is reached. The procedure stops.
   (b) the original line defined in step 1 is reached. Go to step 1.
   (c) the point H is reached. The goal is unreachable. The procedure stops.

The choice to implement the bug2 algorithm for the PSUBOT is due to its lower average execute time. It takes an artificially contrived example to demonstrate better performance with bug1.

A robot controlled by the bug2 algorithm moves directly toward the goal until it is reached, or until an obstacle is encountered. If the goal is reached, the algorithm terminates. However, if an obstacle is encountered the robot turns and follows the perimeter of the obstacle until the line from the starting point to the goal is reached. Then it resumes motion along the original straight-line path until the goal is reached or until another obstacle is encountered, and so on as is illustrated in Figure 8. This algorithm requires - in addition to localization - two capabilities: Goal Seeking, and Wall Tracking. These can be thought of as fundamental behaviors which the Pilot can switch between as necessary such that one of the two is controlling the robot at all times during navigation.
Goal Seeking

Goal seeking consists of following the line between the initial position and the goal, and can be split into three steps: Head the robot toward the goal, move forward until the goal is reached (or an obstacle), and a final rotation once the goal is reached. The maneuver may be characterized by two rotations: $\alpha_1$ and $\alpha_2$, and the distance between the points: $D$ (see Figure 9 for a graphical illustration of the distance and angles).
These are found in terms of the initial and final points (A and B) and the initial and desired final headings: $\theta_1$ and $\theta_2$.

\[ \alpha_1 = \tan^{-1} \left( \frac{y_b-y_a}{x_b-x_a} \right) - \theta_1 \]  
\[ D = \sqrt{(x_b-x_a)^2 + (y_b-y_a)^2} \]  
\[ \alpha_2 = \theta_2 - \alpha_1 \]

Currently, the Pilot is restricted to goal seeking as its only method of path execution.

**Wall Tracking**

If an obstacle is passable, then tracking its exterior will generally result in reaching the other side. Yet, the question remains: when an obstacle is encountered, should the robot turn left or right? As mentioned in Chapter II, the user may be queried at this point to provide a suggestion. If the user does not, or is unable to do so, range finders may be used to determine the direction which would most likely yield the shortest path. A simple test would be to find the nearest corner and head toward it. If Wall Tracking were to take the robot outside the Channel, the other direction would be tried.

To track a wall (i.e. the boundary of an obstacle) the robot must periodically receive readings from at least 2 sensors: one on the side (sensing the distance to the wall, $D_r$), and one to the front (preventing collisions with a concave or additional obstacle, $D_f$) as illustrated in Figure 10.

If, for example, the distance to the right side of the robot was found to jump suddenly to a large number, the robot would conclude that the edge of the obstacle had just been passed.

**COORDINATING THE BEHAVIORS (THE PILOT)**

The objective of the Pilot is to guide the robot to the goal. While carrying out this task it may become necessary to maneuver around obstacles found in the path. It could either attempt to
characterize the obstacle, include it in the obstacle set and replan the path, or simply try to go around it. The latter appears more appealing since obstacle characterization is a difficult problem, and the dimensions of the obstacle (needed to replan the path) would not be available from a single orientation anyway. This is not to say that an unmapped obstacle found in a room should not be entered into the World model after maneuvering around it.

The Pilot may also be expandable so that new behaviors may be added as found necessary. Such things as moving through doors (a special kind of maneuver), moving into and operating an elevator, and approaching a charging dock are natural candidates for specialized behaviors which could be incorporated into the Pilot.

The following Pilot algorithm shows how Goal Seeking and Wall Tracking behaviors may be used to move the robot to the goal. Again, the direction the robot turns may be dependent on the input given by the rider. Here we assume the robot turns to the right first.
ALGORITHM 4: Pilot:

FOR each Quad (starting from the current one)
   A. Turn toward local-goal (endpoint on far side of Quad)
      Move forward UNTIL:
         a. The goal is reached;
         CONTINUE procedure for next Quad.
         b. Obstacle is found obstructing path;
         Turn Right
      B. Invoke Track_Wall behavior UNTIL:
         a. Local Goal is reached;
         REPEAT for NEXT Quad
         b. Obstacle no longer blocking Goal_Seeking;
            Go To A
         c. At Quad boundary;
            IF Quad boundary is reached for a second time
            give up.
            ELSE
               Rotate 180°
               Go To B
            ENDIF
      ENDUNTIL
ENDUNTIL
ENDFOR

Comments

If an obstacle is found to obstruct the entire Channel, the pilot would give up, allowing the
Channel generator (path planner) to generate a less restrictive (wider) Channel. This is an impor-
tant point. It is conceivable that a room with entirely unknown obstacles could be successfully
navigated by taking control from the path planner and giving it to the pilot's two obstacle
avoidance behaviors (Goal Seek and Wall Tracking). To accomplish this, the Path Planner could
create a Channel consisting of a single Quad which would contain the entire room. The robot
would then be allowed to move freely about the room following the boundaries of obstacles in its
pursuit of the goal.
CHAPTER IV

LOCALIZATION

INTRODUCTION

DEFINITION 1:
Localization- To make local; limit or confine to a particular place, area or locality [2].

In robotics localization is the process of using sensors to determine the posture of a robot as illustrated in Figure 11.

DEFINITION 2:
Posture- The triple (x, y, θ) representing the location of the robot and its heading with θ = 0 along the x axis and θ increasing counter-clockwise as illustrated in 11.

![Posture diagram](image)

Figure 11. Posture diagram.

Navigation, which includes both localization and course planning, has been studied extensively in the fields of aircraft, missiles, spacecraft, and marine vehicles [4].

Methods commonly employed on mobile robots include the use of: odometry, beacons, vision, range finders, compass, and the triangulation of satellite signals. Early robots followed magnetic or radioactive strips embedded in floors, creating a sort of virtual railroad track.
Localization for such a robot was a one-dimensional problem.

The intended use of the robot is a major influence on the methods employed in its localization. If the robot is to explore new territory on Mars then matching sensor data with stored images makes little sense (unless it had been there before), while a robot designed to operate within a structured environment can gain much information when comparing sensor reading to a stored model.

THE NEED FOR LOCALIZATION

Suppose a robot is given a goal. If the robot can "see" the goal it need not know its own location to reach it. However, if the robot cannot "see" the goal, it must know its own location before moving toward the goal. A goal such as "x = 25ft, y = 37ft, θ = 0°" cannot be "seen" by the robot and thus the PSUBOT must first determine its own position so that it can compute a path to the goal.

LOCALIZATION TECHNIQUES

Methods of localization for a mobile robot may be divided into two major categories: Zero drift estimation and non-zero drift estimation. Put simply, the expected error from a non-zero drift method is proportional to the distance traveled. A zero-drift method exhibits bounded error regardless of the distance traveled. We first give an intuitive example and later present a more rigorous definition.

An analogy can be drawn between a robot maneuvering and driving a car on a trip. If the odometer on the car loses 0.1 mile in 100 miles, it will likely loose 0.2 miles after traveling 200 miles - odometry is a non-zero drift location estimator. In contrast, if the driver looks at the city name on mailboxes (assuming mailboxes have city names on them) as they are passed, one likely could circle the continent several times and still know the name of the current city just as well after years of traveling as when the trip was begun. The latter method of localization has the
property of zero-drift (though its resolution is not nearly as good).

Mathematically, each component of the posture may be modeled independently by:

\[ \Psi_n = \Psi_{n-1} + \Phi + \chi \]

where \( \Psi_n \) is the estimate of the parameter \( \Psi \) at time \( n \), \( \Phi \) is the true change of the quantity which \( \Psi \) estimates since time \( n-1 \), and \( \chi \) is a random variable. If \( \text{E}(\chi) \) (the expected value or average of \( \chi \)) is zero, the estimator is said to have zero drift. If \( |\text{E}(\chi)| > 0 \), the estimator is said to have non-zero drift. The \( \text{E}(\chi) \) and \( \text{E}^2(\chi) \) parameters are intrinsic characteristics of each type of localization.

We make one additional observation before describing several localization methods: Just as odometry in your car can be trusted almost whole-heartedly in the short term, so other non-zero drift localization methods often have much smaller variances then their zero-drift counterparts. It will be to our best interests to use this fact when integrating several sources of location estimation.

**Odometry**

This source of location information is available in any wheeled robot where the wheels are driven using feedback control. Odometry is often implemented with optical encoders mechanically connected to a turning shaft (motor, gears, or the drive wheel itself) where a direct relationship between the rotation of the encoder and the distance traveled by the wheel may be derived. One variant on this method is to place a passive knife-edge measurement wheel directly on the floor behind the robot. Most robots utilize odometry based localization since the data is readily available and it provides an accurate and quickly acquired short term estimation of the robot's posture. The major problem with odometry, as with any dead-reckoning system, is the inherent drift associated with it; i.e. it is a non-zero drift estimator and the error grows without bound due to slippage and measurement errors. It is therefore necessary to provide additional means by which to determine position if trips longer than a few meters are required.
Beacons

These devices belong to one of two classes: active or passive. Active beacons send out unique code streams which are received by the robot and used to find its position relative to the devices through triangulation. The robot must know the location of the beacons. Passive beacons are often reflectors or bar-coded strips which the robot can scan (with a light source) to determine its location. There are two requirements for the use of beacons for localization: (1) The location of all beacons must be known by the robot beforehand and (2) each building in which the robot is to navigate must be prepared by placing beacons before localization using this method is possible. The second of these requirements rules out beacons as a navigation aid for the PSUBOT since modification to the building is not allowed.

Vision

Work has been done at Portland State University toward the incorporation of vision into the PSUBOT system [17]. Our vision system uses the notion of “features”: The attributes of an image which are deemed important and unique to a particular “scene”. Features from a picture taken beforehand are stored in a database along with the location of the robot when the picture was taken. Then periodically during normal navigation the vision system takes a picture, matches it to the database of stored “scenes”, and determines the corresponding posture of the robot. However, vision is not currently used by the PSUBOT due to its slow speed and currently unrealized benefits. Faster algorithms and hardware and a better understanding of how to incorporate different estimations of position into a single representation will be necessary before vision is of more than minimal use to the PSUBOT.

Range Finders

These devices return the pair \((r, \theta)\) where \(r\) is the distance to the closest object of some minimum normal size and reflection coefficient, and \(\theta\) is the angle relative to the device’s platform on the robot. Range finders are usually optical or acoustic, but the general principles of
operation are the same. A wave (acoustic or electromagnetic) is sent from a source in the direction $\theta$. The wave is reflected off an object at distance $r$ and returns to a receiver (often the same device as the source). Auxiliary circuitry then calculates the distance to the reflecting object. Time-of-flight is used for acoustic ranging devices (speed of sound $\times$ time of flight), while laser/inferred rangers measure the phase difference between the outgoing and incoming beam. The range of the acoustic devices is often limited by reflection off walls, ceilings, and floors because of its wide cone (20 degrees), whereas laser/inferred rangers are limited by the effect of aliasing when there is a phase shift of more than $2\pi$ between the source and returning signals over long distances. For a good review of most ranging systems available today, see a work by CDR Everett [18].

Global Positioning Systems

A Global Positioning System (GPS) can calculate the module’s global, worldwide position through the triangulation of signals from orbiting satellites. This is an example of active beacon-based localization where the beacons are satellites orbiting the earth. Accuracy can be relatively good (within ten feet), and they are not prohibitively expensive. Even within this remarkable range, however, better methods will still be required for providing a more accurate estimate of position. Future versions of the PSUBOT may be equipped with GPS to provide rough position estimation.

Compass

A compass is related to GPS in its global nature (it operates almost anywhere in the world), but is much simpler and has a smaller average error if such a comparison is possible. Using the orientation of the earth’s magnetic field a compass can give the robot’s heading. Magnetic compasses are plagued with errors when placed near ferrous metals or other induced magnetic fields. The electronic flux-gate compass, however, is more immune to these perturbations and thus provides a more accurate estimate of heading. A compass is used in the PSUBOT since it is both inexpensive (under $40$) and provides a low-noise, zero-drift estimate of the $\theta$ parameter.
RELEVANT LITERATURE

There have been a number of methods developed for localizing a mobile robot using range data. Though all perform some level of matching, the approaches vary widely.

Mataric

This was a masters thesis completed at MIT [31]. I mention this first because it is both fundamentally different from the following works, and is representative of an entire class of navigation algorithms based on the subsumption architecture investigated primarily at MIT which is not further mentioned in this work. The subsumption architecture is based on the philosophy that small is better than big and distributed intelligence is better than central intelligence for certain application domains. The robot Toto is equipped with a ring of sonar and a compass. Where it begins to differ from more traditional approaches is that it has no concept of space. It first learns a building by wandering around, following one wall or another. As it moves it generates a list of nodes or landmarks based on sensor readings. Then a goal (in terms of a landmark) may be reached by visiting each landmark between the current and final ones. This means that the notion of a "best path" is limited to those locations where the robot had been. There are, however, many advantages to this system. For one, it is implemented with a fraction of the computational power of the others. Secondly, if the robot becomes lost, the next landmark it reaches will automatically direct it in the correct direction - without replanning. Finally, landmarks are very simple, seldom changing characteristics of the building - leading to robust landmark recognition.

Elfes

This is perhaps the most famous experiment in sonar-based navigation. The Neptune robot [13] used a pair of cameras and a ring of sonar devices to generate and maintain a world model. Several levels of abstraction were used to represent the accuracy of the sonar, geometric features, and the resolution of the model. The system was very versatile since it needed no a priori infor-
motion and was also appropriate for (and demonstrated) outdoors. One problem was that odometry was assumed accurate so that errors in the estimated location were transferred to the generated model. The hope was therefore that over time odometry errors would eventually cancel out and the model would reflect reality. Since occupancy grids were the medium for representing both sonar data (and the model) simplifications and pre-processing were required to keep the time complexity at a reasonable level.

Drumheller

In this research, line segments were extracted from raw sonar data to produce contours [12]. Sonar-segments (points making up short lines) were then matched to a priori line-segment walls. This method was rather immune to spurious noise since data points not falling in a line were ignored. Also of interest was the sonar barrier test where the algorithm used the fact that sonar cannot penetrate walls to eliminate certain otherwise possible orientations. There was not the notion of "small-step" which was used by subsequent researchers to decrease search space for a solution.

Crowley

This method is much the same as that of Drumheller’s above, but with the added ability to update the model. In addition, it can generate a model from only observances [8] and [9]. Additionally, Kalman filtering [32] is used to update the robot’s believed position based on the orientation and reliability of each line extracted. The process of building a map based on odometry and then using it later to estimate position and correct odometry errors has been called “pulling ones self up by ones bootstraps” [9].

Cox

Conceptually this is perhaps the simplest of the methods discussed thus far. Instead of abstracting lines from data points, data points are matched directly to line-segments which are
known *a priori* [7]. While less flexible than those which are able to update the model, it is based on the assumption that many buildings in which a robot operates seldom change. The algorithm is based on a least-squares matching of the data points to their nearest line. It is this method on which we base our localization algorithms due to similar assumptions (static building, indoor operation, on-board processing) and its relative computational simplicity. It also makes good sense in terms of the "small step" assumption - that corrections are small when performed often enough. Cox used an infrared laser ranger which exhibits lower noise and higher resolution (at a higher cost) than ultrasonics [34]. The algorithm was tested on a robot named Blanche and was shown to perform acceptably.

**COMPARISON OF LOCALIZATION TECHNIQUES**

We now demonstrate, compare, and contrast several parameters indicative of various localization techniques. First we compare flexibility and complexity in Figure 12.

![Figure 12. Flexibility vs. complexity.](image)

Flexibility is a measure of how well the robot deals with inaccuracies in the model, sensor noise, and extra or missing obstacles.

Behavior based robots [3] tend to be very good at operating in unstructured environments since they generally have little knowledge about the geometric or spacial layout of the building.
anyway. They simply obey a set of rules and "learn" about the environment as they move along. Some simply respond to hard-coded reflexes. From the abstract of [19], Flynn describes the robots as those which: "hide in dark corners and venturing out in the direction of last heard noises, only moving after the noises are long gone." Others have more sophisticated capabilities such as finding goals. This simplicity is also reflected by the amount of memory necessary to operate the robot: Toto from MIT (a fairly capable robot with sonar and a compass) uses only 51K bytes for a "moderate sized run" [31].

Although this approach is flexible, it lacks the ability required by the PSUBOT to find a "best path" from one room to another since it knows of locations only if it has been there before and deemed it a "landmark". To move to location \((x,y)\) would not be easily accomplished since it would have no notion of Cartesian coordinates.

Global methods (GPS, compass, etc.) have slightly higher complexity - at least in terms of hardware - but are much more flexible than any other type of localization scheme presented since they need know nothing about their surroundings or where they have been. The first reading at a particular point is as reliable as the 50th.

Vision-based localization and navigation systems are very complex. The sheer quantity of data requires one or more dedicated high-powered processors in order to deduce useful position information from an image. However, the potential for using vision as a navigation aid (especially stereo vision) is easily seen by realizing that much of the natural kingdom uses binocular vision to navigate.

The two localization techniques near the center of the chart (compare data to model) use range data to see neighboring walls much like a bat uses sonar to determine the shape, location, and speed of its prey.

A big advantage of using this type of sensor is that the amount of data is small and can be reduced further by lowering the resolution of the rotating device (if any) without severe degra-
tion of interpreted results.

Figure 13 shows how the accuracy and complexity of the same localization techniques relate. We can see that one of the simplest (odometry) is highly accurate in the short term but the most inaccurate in the long term. Matching algorithms which operate on raw and abstract data are again near the center of the graph and offer fairly good accuracy and complexity. Vision systems are placed lower on the accuracy scale simply because algorithms have not achieved consistent results due to significantly higher complexity.

![Accuracy vs. complexity diagram](image)

**Figure 13.** Accuracy vs. complexity.

**LOCALIZATION FOR THE PSUBOT**

The current version of the PSUBOT uses odometry, a compass, and ultrasonic ranging to determine its posture within the known building. Both odometry and ultrasonic matching algorithm determine a complete posture while the compass gives only heading information. Combining these estimates optimally is a difficult problem and a solution is not presented here. Instead we use the compass to find the initial heading, we use odometry during maneuvers, and the sonar matching algorithm is applied periodically.

It is interesting to note that since none of the sensors require light, the robot is exactly as
capable in complete darkness as in a well lit room.

Odometry

Necessary equations. Position estimation through odometry for a robot with two drive wheels has been studied by Iijima [22] as cited by Wang [43]. Their results are used without modification and are restated here for completeness.

We must know the distances traveled by the left and right wheels; \( \Delta D_L \) and \( \Delta D_R \), respectively. Fortunately, these values are available directly from the feedback hardware and from them we can derive the distance traveled \( \Delta D \) and the change of heading \( \Delta \theta \). Figure 14 presents the model of motion where \( A \) and \( C \) are the initial wheel locations, and \( A' \) and \( C' \) are the final wheel positions. \( B \) denotes the center of the rear axle whose width is \( L \).

\[
\Delta D_n = \frac{\left( \Delta D_L + \Delta D_R \right)}{2}
\]

(7)

\[
\Delta \theta_n = \frac{\left( \Delta D_L - \Delta D_R \right)}{L}
\]

(8)
The new position and heading can then be calculated by adding to the parameter estimates at time 

\[ n-1: \]

\[
x_n = x_{n-1} + \frac{\Delta \theta_n}{2} \cdot \Delta D_n \cdot \cos \left( \frac{\theta_{n-1} - \Delta \theta_n}{2} \right)
\]

\[
y_n = y_{n-1} + \frac{\Delta \theta_n}{2} \cdot \Delta D_n \cdot \sin \left( \frac{\theta_{n-1} + \Delta \theta_n}{2} \right)
\]

\[
\theta_n = \theta_{n-1} + \Delta \theta_n
\]

Thus we find our posture at time \( n \) to be:

\[ P_n = (x_n, y_n, \theta_n) \]  

(12)

Notice that if \( \Delta D_l = \Delta D_r \) the equations for \( x_n \) and \( y_n \) cannot be simply computed. In this instance we use the property:

\[ \lim_{x \to 0} \left[ \frac{\sin(x)}{x} \right] = 1 \]

(13)

to obtain meaningful results. This situation occurs when the robot is moving forward without turning.

Since the location of the robot is determined through these recurrence relations, the result is drift; i.e. a non-zero mean error will be added to each iteration. Therefore odometry-based error grows without bound.

**Odometry measurements.** We set out to test the accuracy of the odometry-based localization. The test included a forward motion in addition to clockwise and counterclockwise rotations. For each motion the wheelchair was moved by hand the desired amount and then asked for its estimated location based on the equations above. This process was repeated three times for each of the three motions and the results averaged and displayed in the next three tables.

Table I shows a forward motion of 33 feet. The "r" column represents the absolute
(radius) distance between the actual and odometry-based locations.

**TABLE I**

**ODOMETRY RESULTS OF FORWARD MOTION (DISTANCES IN FEET)**

<table>
<thead>
<tr>
<th>Actual(feet)</th>
<th>x</th>
<th>y</th>
<th>θ(rad)</th>
<th>error r</th>
<th>ft</th>
<th>r</th>
<th>ft</th>
<th>θ</th>
<th>ft</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>3.0</td>
<td>3.0</td>
<td>0.2</td>
<td>0.00</td>
<td>0.23</td>
<td>0.076</td>
<td>0.001</td>
<td>0.076</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>6.0</td>
<td>6.0</td>
<td>0.3</td>
<td>-0.07</td>
<td>0.34</td>
<td>0.056</td>
<td>-0.012</td>
<td>0.056</td>
<td>-0.012</td>
<td>0.012</td>
</tr>
<tr>
<td>9.0</td>
<td>9.0</td>
<td>0.4</td>
<td>-0.07</td>
<td>0.44</td>
<td>0.048</td>
<td>-0.007</td>
<td>0.048</td>
<td>-0.007</td>
<td>0.007</td>
</tr>
<tr>
<td>12.0</td>
<td>12.0</td>
<td>0.5</td>
<td>-0.06</td>
<td>0.52</td>
<td>0.043</td>
<td>-0.005</td>
<td>0.043</td>
<td>-0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>15.0</td>
<td>15.0</td>
<td>0.6</td>
<td>-0.08</td>
<td>0.62</td>
<td>0.041</td>
<td>-0.005</td>
<td>0.041</td>
<td>-0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>18.0</td>
<td>18.0</td>
<td>0.6</td>
<td>-0.09</td>
<td>0.65</td>
<td>0.036</td>
<td>-0.005</td>
<td>0.036</td>
<td>-0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>21.0</td>
<td>21.0</td>
<td>0.6</td>
<td>-0.11</td>
<td>0.62</td>
<td>0.029</td>
<td>-0.005</td>
<td>0.029</td>
<td>-0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>24.0</td>
<td>24.0</td>
<td>0.6</td>
<td>-0.08</td>
<td>0.63</td>
<td>0.026</td>
<td>-0.003</td>
<td>0.026</td>
<td>-0.003</td>
<td>0.003</td>
</tr>
<tr>
<td>27.0</td>
<td>27.0</td>
<td>0.6</td>
<td>-0.11</td>
<td>0.60</td>
<td>0.022</td>
<td>-0.004</td>
<td>0.022</td>
<td>-0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>30.0</td>
<td>30.0</td>
<td>0.6</td>
<td>-0.12</td>
<td>0.62</td>
<td>0.021</td>
<td>-0.004</td>
<td>0.021</td>
<td>-0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>33.0</td>
<td>33.0</td>
<td>0.6</td>
<td>-0.08</td>
<td>0.64</td>
<td>0.019</td>
<td>-0.003</td>
<td>0.019</td>
<td>-0.003</td>
<td>0.003</td>
</tr>
</tbody>
</table>

From Table I we see that the robot did very well at maintaining its forward location. Assuming an operator error of one inch, the estimate of x shows no accumulating errors while traveling 33 feet (which is not to say that there were none, we were simply unable to detect them in the distance traveled).

The lateral (y) location, however, grew initially and then seemed to stabilize at 0.6 feet for the remainder of the 33 feet traveled. It would seem that there was some startup error, followed the remainder of the motion in which no errors occurred. However, from Figure 15 which shows the y-error for the three separate forward experiments, we see that the maximum distance errors were much more erratic and accumulative. The worst-case error after 33 feet was 1.1 feet and the overall maximum y error was 1.3 feet (at x=24 ft). The x errors were all zero- within the accuracy of measurement.

In Table II we tabulate the average of the three experiments for a 360° (2π) counter-clockwise rotation. Table III shows a clockwise rotation. The values of "Actual Rotation" are
assumed to be accurate to within 0.04 radians or 2.4° (which corresponds to placement of each wheel within 1/2 inch).

TABLE II

ODOMETRY RESULTS OF COUNTER-CLOCKWISE MOTION (DISTANCES IN FEET)

<p>| Odometry Trace (Counter-Clockwise) |
|-------------------------------------|---|---|---|---|---|</p>
<table>
<thead>
<tr>
<th>Actual(radians)</th>
<th>x</th>
<th>y</th>
<th>θ(rad)</th>
<th>error r</th>
<th>error θ</th>
<th>error θ/radian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.0</td>
<td>0.0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
</tr>
<tr>
<td>1.57</td>
<td>0.1</td>
<td>0.0</td>
<td>1.54</td>
<td>0.11</td>
<td>-0.036</td>
<td>-0.023</td>
</tr>
<tr>
<td>3.14</td>
<td>0.1</td>
<td>0.1</td>
<td>3.11</td>
<td>0.13</td>
<td>-0.032</td>
<td>-0.010</td>
</tr>
<tr>
<td>-1.57</td>
<td>0.0</td>
<td>0.0</td>
<td>-1.57</td>
<td>0.05</td>
<td>0.005</td>
<td>0.001</td>
</tr>
<tr>
<td>0.00</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.01</td>
<td>0.04</td>
<td>-0.006</td>
<td>-0.001</td>
</tr>
</tbody>
</table>

We find from the two rotation tables that significant rotation errors were accumulated in a single rotation. In Figures 16 and 17 we show the angle errors of each of the three trials for both counter-clockwise and then for clockwise rotations. The maximum rotation error was 0.113 radians which occurred at -1.57 radians during one of the counter-clockwise experiments. This maximum is also significant relative to the measurement accuracy.
TABLE III

ODOMETRY RESULTS OF CLOCKWISE MOTION (DISTANCES IN FEET)

<table>
<thead>
<tr>
<th>Actual (radians)</th>
<th>x</th>
<th>y</th>
<th>θ (rad)</th>
<th>error r</th>
<th>error θ</th>
<th>error θ (rad)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.0</td>
<td>0.0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>-1.57</td>
<td>-0.0</td>
<td>0.0</td>
<td>-1.56</td>
<td>0.03</td>
<td>0.006</td>
<td>-0.004</td>
</tr>
<tr>
<td>-3.14</td>
<td>-0.0</td>
<td>0.0</td>
<td>-3.13</td>
<td>0.02</td>
<td>0.013</td>
<td>-0.004</td>
</tr>
<tr>
<td>1.57</td>
<td>-0.1</td>
<td>0.0</td>
<td>1.55</td>
<td>0.08</td>
<td>-0.025</td>
<td>0.005</td>
</tr>
<tr>
<td>0.00</td>
<td>0.0</td>
<td>-0.0</td>
<td>-0.05</td>
<td>0.02</td>
<td>-0.045</td>
<td>0.007</td>
</tr>
</tbody>
</table>

Figure 16. Angle errors for counter-clockwise rotation.

Rotation is much more critical than forward motions in the long term, since an error of only 0.066 radians (3.8°) will cause the robot to be wrong by 2 feet after moving forward perfectly for 30 feet. A compass (the subject of the next section) with appropriate accuracy and resolution may therefore be of great help in determining the actual heading of the robot.

Rotations are also most prone to error since both wheels slide during the entire rotation (rather than roll). It is also difficult to accurately determine the effective wheelbase of the robot since each wheel is nearly two inches wide and it is not clear about which point they pivot.
Another cause of odometry error is changes of the diameter of the wheels. When a heavy
person is on the wheelchair, it has a smaller effective wheel radius than when a child is on board.
Also, changes in the tire pressure resulting from leaks or even temperature changes affect the
radius of the wheel and thereby change the number of feet per rotation. It was found that calibra-
tion was required periodically (over several days) to maintain the accuracy found in the Tables
and Figures above.

Long-term-error estimation. We now use a simple geometric model which may be used to
anticipate long-term odometry errors from the measurements made above.

Let $e_{\text{max}}$ denote the maximum tolerable translation error after a forward motion. Let $e_{\theta}$
denote the maximum rotation error after a particular rotation. We wish to find $r_s$ which
represents the distance we may travel after a rotation and still be within a "safe" distance from
our desired location. Figure 18 gives the geometric interpretation of these quantities. Let $e_{\text{r/ft}}$
denote the maximum translation error accumulated for each foot traveled. Then $r_s$ may stated by

$$r_s = \frac{e_{\text{max}}}{\tan(e_{\theta})} - r_s e_{\text{r/ft}}$$

so that
Figure 18. Reliable odometry distance.

This model for odometry errors will be used together with the measurements in Chapter VI to determine realistic system operating parameters.

Compass

The flux-gate compass in use on the PSUBOT was manufactured for public use in automobiles and modified to allow acquisition of signals through a digital computer. Two signals from the compass are sampled by the computer and will be called A and B, and their theoretical shape may be seen from Figure 19.

The signals are modeled as trigonometric functions: 

\[ A = A_F \sin(\theta_A) + A_o \]

\[ B = B_F \cos(\theta_B) + B_o \]

and are both needed to determine the heading of the robot since neither function is reflexive. Solving for \( \sin(\theta) \) and \( \cos(\theta) \) and dividing we have the tangent from which we can then find the heading.
The parameters $A_0$ and $A_{FS}$ are determined experimentally during configuration of the robot to a particular environment. There is currently no method of updating these values during the course of navigation, though a method of doing so should be rather simple.

The accuracy of the compass is currently rather poor (around 20°), though we attribute this primarily to the simplicity of the model. The repeatability is much better than 20° but the poles (N,S,E,W) give the appearance of being warped. We expect that a lookup table or a model which also adjusts for shifts in the above sinusoids could improve the accuracy of the compass to within 5°.

So, the output of the compass is the angular heading of the robot with 0° mapped to the direction of the x axis and positive headings measured counter-clockwise.

**Sonar Matcher**

Of primary interest in this work is the reduction of odometry error by matching a set of range points to a model of the world. Although the application is applied to matching sonar range points to line segment, any two-dimensional point-based image may be matched to a line segment model using this method.

The remainder of this chapter is dedicated to the development and discussion of the algorithm which determines the best match from points to lines, hereafter called the “matcher”.

**DEVELOPMENT OF THE SONAR-BASED MATCHING ALGORITHM**

The matching algorithm makes an assumption about the magnitude of the errors it corrects - it assumes them to be small. Points are moved to the closest line segment, so if they are too distant from their correct location, they may be moved to the wrong place. This assumption is a
valid one for this application, however, since the errors result from odometry and odometry is accurate in the short term. Therefore if the matching algorithm is applied often enough, the so called small step assumption will be valid.

World Model

The World is modeled by a collection of line segments $A_iB_i$ where $A_i$ and $B_i$ are points represented as column vectors $A=[A_x,A_y]^T$ and $A_i\neq B_i$. Figure 20 shows an example map.

![Figure 20. Sample room layout.](image)

A more complex representation of the world is allowed, provided it can be decomposed into a collection of line segments as described above.

Sonar System

Distance measurements are achieved through the use of sonar (SOUND NAVIGATION AND RANGING). Since the frequency of the sound is higher than is generally audible (> 20,000 Hz) we will also call it an “ultrasonic ranging system”. A Polaroid ultrasonic transducer [1] is used which both sends and receives signals. The system also incorporates a stepper motor for rotation, external counting/measuring circuitry, and a PC interface. The value returned from the sonar control software is a (distance, angle) pair $(d, \alpha)$. Collecting two or more sonar-derived points at different angles is called a sonar scan. Scans are performed periodically. Range samples can be taken while the robot is in motion by using the robot’s current estimation of its posture to
transform each point into global coordinates at the time it is received. Once the desired scan is completed, the matcher has the responsibility of finding a correction vector which matches the points to the line segment model. The correction represents the matcher's estimation of the accumulated position error since the previous match and is used to hold the ever-growing odometry position error to an acceptable level.

Matcher Algorithm

A common matching procedure involves minimizing the sum of the squared errors. The quantity to be minimized is demonstrated by a simple example.

Consider the robot in the following room with its estimated position shown \((P = (x, y, \theta))\). The robot performs a scan consisting of 5 points like the one pictured in Figure 21.

![Figure 21. Simple hypothetical sonar scan of a room.](image)

Based on the sonar data, we can see that the robot is closer to the bottom of the image and is heading differently than it thinks. Now keeping the model constant and treating the robot and sonar scan points as a rigid body (a translation of the points moves the robot by the same amount), apply a shift in the negative \(x\) and \(y\) directions and a clockwise rotation such that the points are placed as close to their closest line as possible. Changing the orientation of the points with respect to each other is prohibited; i.e. the points must be shifted and rotated by the same
amount and around the same point. Thus $p_1$ and $p_2$ should be placed on line $b$ and $p_3$, $p_4$, and $p_5$ should be moved to line $a$.

**Definition 3:**

**Target.** The line segment from the world model which is geometrically the closest to a particular sonar-based data point.

Each point has a unique target. If two lines are equidistant from the point, the target is chosen at random from them. The algorithm for finding targets for the points is given below.

**Algorithm 5: Find-Targets**

```
FOR each point $p_i$
    FOR each line segment $l_j$
        LET $\text{min} = \infty$
        LET $d$ be the shortest distance from $p_i$ to $l_j$
        IF $d < \text{min}$ THEN
            LET $\text{target}# = j$
            LET $\text{min} = d$
        ENDIF
    ENDFOR
    The target of $p_i$ is $l_{\text{target}#}$
ENDFOR
```

Next we show the pseudo-code for the matcher algorithm. The mathematics required to implement specific steps are developed in the following section. Noise removal is the topic of a future section in this chapter.
Algorithm 6: Matcher

LET $p_0$ be the believed position of the robot
LET $\theta_0$ be the believed orientation of the robot
LET Incremental_Correction = 0

Rotate the points by $\theta_0$
Add $p_0$ to the points (transform them to world coordinates)

Repeat the following:

- Find the target line of each point
- Calculate the average and mean of the distance between each point and its target
- Remove outliers based on average and standard deviation of distances between pts and targets

With the resulting filtered points DO:

- IF all resulting targets are parallel THEN
  LET Incremental_Correction = the result of the Parallel Matching algorithm
- ELSE
  LET Incremental_Correction = the result of the Non-Parallel Matching algorithm
- ENDIF

Apply Incremental_Correction to all points (even those filtered out previously)
LET Total_Correction = Total_Correction + Incremental_Correction

Until Incremental_Correction is sufficiently small
Calculate the Robot-Relative Total Correction

Mathematical Fundamentals of the Sonar Matcher

We now develop some of the mathematics necessary to accomplish the above task. The following method is based on a work by Cox [36], and [5].

It will be our convention to denote vectors by bold symbols ($\mathbf{p}$). A lower case italic character will be used for a line segment ($l$), and an upper-case italic character for the infinite line ($L$) which coincides with the segment $l$. Scalars are also in italics and their meaning will be clear from the context in which they are used. Also, since the vectors discussed are usually of dimension 2x1 and carry physical $x/y$ coordinate information, an $x$ or $y$ subscript will refer to the first or second element of the vector respectively.
DEFINITION 4:
Point- The point \( p \) is represented by a vector of the form: \([x, y]' = [x, y] \).

DEFINITION 5:
Line Segment- A line segment \( l \) is defined by two endpoints \( a \) and \( b \) where \( a \neq b \).

DEFINITION 6:
Infinite Line- The infinite-length line \( L \) which passes through the endpoints \( a \) and \( b \) of line segment \( l \). Also called "a line".

A line segment \( l \) has exactly one infinite line \( L \) which also passes through its endpoints. Such a line can be represented by the pair: \((u, r)\) where \( u \) is the unit vector orthogonal to the infinite line \( L \), and the point \( r \cdot u \) is on \( L \) (\( r \) is the orthogonal distance from the origin to the line) as demonstrated in Figure 22.

\[ \text{Figure 22. Infinite line representation.} \]

To aid in future derivations we derive the relationship between a line segment \( l \) with endpoints \( a \) and \( b \), and its infinite line \( L \). We begin with the general equation for a line:

\[ Ax + By + C = 0 \]

The coefficients \( A, B, C \) of \( L \) can be calculated in terms of \( a \) and \( b \) as follows:

\[ A = -(b_y - a_y) \quad \text{(18)} \]
\[ B = b_x - a_x \quad \text{(19)} \]
\[ C = a_x(b_y - a_y) - a_y(b_x - a_x) \quad \text{(20)} \]

from which the following relationships can be seen:
If \( L \) is Horizontal \( A = 0 \)

If \( L \) is Vertical \( B = 0 \)

If \( L \) passes through \((0,0)\) \( C = 0 \)

Then from geometry we find that the components \((u, r)\) are:

\[
\begin{align*}
    u_x &= \pm \frac{A}{\sqrt{A^2 + B^2}} \\
    u_y &= \pm \frac{B}{\sqrt{A^2 + B^2}} \\
    r &= \pm \left( \frac{-C}{\sqrt{A^2 + B^2}} \right)
\end{align*}
\]

where the \( \pm \) sign is chosen opposite to \( C \). If \( C = 0 \), use the sign of \( B \).

**DEFINITION 7:**
*Distance to Infinite Line* - The distance from a point \( p \) to an infinite line \( L \) represented by \((u, r)\) is

\[
\text{distance} = (p \cdot u - r)
\]

**DEFINITION 8:**
*Distance to Line Segment* - The distance from a point \( p \) to a line segment \( l \) (defined by points \( a \) and \( b \)) is more complicated since there are three possible cases. Consider the line which is perpendicular to the infinite line \( L \), passes through the point \( p \) and intersects \( L \) at point \( q \) as in Figure 23.

![Figure 23](image)

Which of the three cases is valid can be determined by testing the order of the points \( a \), \( b \), and \( q \) along the line \( L \).

Case 1: \( qab \) \( \text{dist} = \text{abs}(\|a\| - \|p\|) = \sqrt{(p_x - a_x)^2 + (p_y - a_y)^2} \)

Case 2: \( aqb \) \( \text{dist} = \text{abs}(p \cdot u - r) \)
Case 3: \( a \leq q \leq b \) dist = \( \text{abs} \left( \frac{b-\text{abs}(b-p)}{a-p} \right) \)

where \( \text{abs} \) denotes absolute value. If \( q \) is equal to either \( a \) or \( b \), Case 2 is used for efficiency.

Thus we may calculate the distance between each point and each line segment thereby determining the target of each. It is the sum of the squares of this distance which we are trying to minimize with the matcher.

**DEFINITION 9:**

*Centroid:* The centroid \( c \) of a set of points \( p_i, i=1,2,3,...,n \) (also called their "center of gravity" or "mean") is defined by:

\[
\begin{align*}
\mathbf{c} &= \frac{1}{n} \sum_{i=1}^{n} \mathbf{p}_i \\
&= \frac{\sum_{i=1}^{n} \mathbf{p}_i}{n}
\end{align*}
\]  


**DEFINITION 10:**

*Point Rotation:* It is well known from geometry that to rotate a point \( p \) about the origin by \( \theta \), one simply multiplies its coordinates by the following orthogonal transformation matrix \( R_0(\theta) \):

\[
R_0(\theta) = \begin{bmatrix} 
\cos(\theta) & -\sin(\theta) \\
\sin(\theta) & \cos(\theta) 
\end{bmatrix}
\]

To rotate \( p \) about an arbitrary point \( c \) we must first shift the origin to \( c \), multiply by the rotation matrix, and then shift the origin back to \( c \):

\[
\mathbf{p}' = R_0(\theta) \cdot (\mathbf{p}-\mathbf{c}) + \mathbf{c}
\]

**DEFINITION 11:**

*Pseudo-Rotation:* For linearity reasons to be explained later, it proves useful to approximate the true rotation \( R_0(\theta) \) by the first term of its Taylor series expansion:

\[
R_p(\theta) = \begin{bmatrix} 1 & -\theta \\
\theta & 1 \end{bmatrix}
\]

Pseudo-rotation is equivalent to moving a point along the line tangent to an arc rather than along the arc itself. Note also, that for very small \( \theta \), \( R_p(\theta) \approx R_0(\theta) \). This is true if the small step assumption is true.
Minimize Mean Squared Error

We are now ready to derive expressions for the correction \((t, \theta)\) which, when applied to the points of the sonar scan image, matches them to the line segment model in a mean squared sense. Define \(S\) as the sum of the squared distances between the points, each rotated about the centroid by \(\Delta \theta\) and translated by \(\Delta t\), and the target lines (not line segments):

\[
S = \sum_{i=1}^{n} \left( [ R_o(\Delta \theta)(p_i - c) + (c + \Delta t) ] \cdot u_i - r_i \right)^2
\]

where there are \(n\) scan points \(p_i\) in the image whose centroid is \(c\). \(u_i\) and \(r_i\) represent the target line of the point \(p_i\), and \(R_o()\) is the rotation matrix.

Removing the \(\Sigma\) and the square operator, the above equation is seen to be in the form:

\[
\text{Distance} = (P' \cdot u - r)
\]

where \(P\) is the point \(p\) rotated by \(\Delta \theta\) about the centroid \(c\) and then displaced by the translation vector \(\Delta t\). The distance is then squared, and summed over the points.

Since the partial derivatives of \(S\) would be linear were it not for the trigonometric functions from \(R_o()\), the pseudo-rotation matrix \(R_p()\) is used to simplify the solution. However, since \(R_p()\) is only an approximation of the true congruence, it will be necessary to iterate to obtain a solution. So \(t\) and \(\theta\) are calculated by summing all intermediate corrections obtained over the necessary number of iterations:

\[
t = \sum \Delta t \quad \theta = \sum \Delta \theta
\]

Now, \(S\) represents the square of the total of the distances between target lines and data points shifted by \(\Delta t\) and pseudo-rotated by \(\Delta \theta\). So, to find the minimum error we must differentiate \(S\) with respect to the independent variables \(\Delta t\) and \(\Delta \theta\) and set the resulting equations to zero.

The extremum found will be the global minimum for the following reason: There will be only one finite extremum, and there not a finite maximum. We know there will be only one finite
extremum because the derivatives are first order equations \( S \) is a quadratic), so there is a single zero-crossing. Additionally, as the points are moved further and further away from the model, the sum of the squared distances increase without bound, i.e. \( \lim_{|M| \to \infty} S = \infty \). Figure 24 illustrates the theoretical shape of the \( S \) curve while holding one variable constant (only two independent variables). The third behaves similarly.

Figure 24. Theoretical shape of sum of mean squares.

Setting the partial derivatives to zero:

\[
\frac{\partial S}{\partial \Delta t_x} = 0 \quad \frac{\partial S}{\partial \Delta t_y} = 0 \quad \frac{\partial S}{\partial \Delta \theta} = 0
\] (32)

yields a third order system of linear equations of the form \( A \mathbf{x} = \mathbf{B} \):

\[
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{bmatrix}
\begin{bmatrix}
\Delta t_x \\
\Delta t_y \\
\Delta \theta
\end{bmatrix} =
\begin{bmatrix}
B_{11} \\
B_{21} \\
B_{31}
\end{bmatrix}
\] (33)
where
\[ A = \begin{bmatrix}
\sum_{i=1}^{n} u_i^2 & \sum_{i=1}^{n} u_i v_i \\
\sum_{i=1}^{n} u_i u_j & \sum_{i=1}^{n} c_i u_i^2 - c_i u_i u_j + u_i u_j v_i - u_j^2 v_i \\
\sum_{i=1}^{n} c_i u_i^2 - c_i u_i u_j + u_i u_j v_i - u_j^2 v_i & \sum_{i=1}^{n} (c_i u_i - c_i u_j + u_i v_i - u_j v_j)^2
\end{bmatrix} \]

\[ B = \begin{bmatrix}
\sum_{i=1}^{n} r u_i - u_i v_i - u_i u_j v_i \\
\sum_{i=1}^{n} r u_i - u_i u_j v_i - u_j v_i \\
\sum_{i=1}^{n} (u_i v_i - u_j v_i + c_i u_i - c_i u_j)(u_i v_i + u_j v_j - \tau)
\end{bmatrix} \]

Gaussian elimination produces the following results:

\[ \Delta t_x = -A_{23} A_{32} B_{11} + A_{22} A_{33} B_{11} - A_{13} A_{22} B_{11} + A_{12} A_{32} B_{11} + A_{12} A_{33} B_{31} + A_{13} A_{23} B_{31} \]

\[ \Delta t_y = -A_{13} A_{22} A_{31} + A_{12} A_{33} + A_{13} A_{23} B_{21} + A_{13} A_{32} B_{21} + A_{12} A_{23} A_{31} + A_{13} A_{23} A_{32} + A_{12} A_{23} A_{33} + A_{13} A_{32} A_{33} + A_{12} A_{33} A_{33} \]

\[ \Delta \theta = -A_{22} A_{33} B_{11} + A_{22} A_{33} B_{11} + A_{13} A_{22} A_{31} + A_{12} A_{32} A_{31} + A_{13} A_{22} A_{32} + A_{12} A_{32} A_{32} + A_{13} A_{32} A_{32} + A_{12} A_{33} A_{32} + A_{13} A_{33} A_{32} + A_{12} A_{33} A_{33} + A_{13} A_{33} A_{33} \]

from which we can find the minimum point of the curve.

**Accounting for Rotation About c**

Recall that the final solution vector after all necessary iterations \((t, \theta)\) was acquired by rotating the points about \(c\) by \(\theta\) and shifting them by \(t\) such that the sum of squared errors was minimum. However, what we really want is the \((t', \theta')\) which yields an equivalent correction when rotated by \(\theta\) about the robot’s location \(R\) (as opposed to the centroid). Were \(\theta\) zero, such a distinction would not exist.

It is clear that a rotation cannot be replaced by translation, so \(\theta' = \theta\). We now derive the
expression for $t'$. Let $q_i$ be the result of point $p_i$ translated by $t$ and rotated about $c$ by $\theta$:

$$q_i = R_\theta(p_i-c)+(c+t)$$

(39)

Next we wish to find the new translation vector $t'$ which yields the same results when rotated about the robot's posture $P$:

$$q_i = R_\theta(p_i-P)+(P+t')$$

(40)

By subtracting and rearranging the above equations we arrive at our result:

$$t' = t - R_\theta(c-P)+(c-P)$$

(41)

The updated posture of the robot may then be found as:

$$P(n) = (x_{n-1}+t'_x, y_{n-1}+t'_y, \theta_{n-1}+\theta)$$

(42)

We now review the matching process:

ALGORITHM 7: Matcher (Summary)

1. A sonar scan produces a set of data points in polar form $(d_i, \alpha_i)$ which are transformed into the Cartesian world coordinates - $p_i = (x_i, y_i)$ - based on the believed posture of the robot (the posture which is to be corrected).

2. Repeat the following until $(\Delta t, \Delta \theta)$ is very small.
   2.1 Outliers removed (see next section: "Removal of Outliers").
   2.2 A target line $L_j$ is assigned to each point $p_i$ by finding the line segment $l_j$ whose distance from the point is minimum.
   2.3 Find $(\Delta t, \Delta \theta)$ which minimizes the mean squared distance between points and their targets using pseudo-rotation.
   2.4 Apply $(\Delta t, \Delta \theta)$ to the sonar data points using true rotation.

3. Add up all the corrections applied to the points under step 3. This is the solution vector $(t, \theta)$.

4. Find translation and rotation (based at the current position of the robot) which yields the same translation and rotation based at the centroid of sonar data points.

5. Apply the solution vector to the old posture to calculate a new matcher-corrected posture of the robot.

Very small is currently 0.2 ft for distances and 0.1 radians for angles, though smaller values could be used if run on faster hardware.
COMMENTS ON THEMatcher

The algorithm as described is subject to several downfalls due to the straight ahead mathematical approach, so let us take several steps back for a moment to expose these problems and their solutions.

Finding the correction vector

Notice that all correction vectors of step 3 are summed in step 4 to yield the final correction vector. We are able to do this because of a property of points rotated about their centroid: to translate and rotate a set of points by \( \Delta t_i, \Delta \theta_i \) about their centroid a number of times is equivalent to translating by \( \sum \Delta t_i \) and rotating by \( \sum \Delta \theta_i \). This property is not true for rotations about any other point.

Removal of Outliers

Due to the noisy nature of the sonar range data it was necessary to remove certain data points to allow closer matches. The rationale used here is that some points are not simply "inaccurate", but rather are wrong (the data doesn't correspond to any physical distance we are interested in measuring). There are several causes for such extraneous data. One is the effect of echos, especially near corners where the sound waves may reflect off of several walls before returning to the transducer. Echoing leads to misleading readings of long distances and should be removed.

Another cause of erroneous range data is perhaps illustrated best with an example: Suppose the robot is in a corridor having a width of 8 feet and a height of 12 feet (not unlike the PCAT building in which testing occurred). Additionally assume that the sonar device is pointing in the direction of a wall at the end of the corridor which is out of range. Clearly a finite distance reading will be incorrect, yet this often occurs due to reflections of the ultrasonic sound wave off the walls, ceiling, and floor. Again, it is emphasized that we are not trying to remove data points
which are slightly inaccurate, but those which have no meaningful physical relationship to the world.

The method employed by Cox [5] is straight-forward: remove any points whose distance from their target line segment (the error) is greater than some pre-defined constant. This method is valid whenever the small step assumption is valid. However, to increase the range for which the small step holds, we chose a statistical method. If the error is greater than the cutoff point, all points would be lost. We assume most of the data points are valid and that we should discard only those whose error is far from the mean. A shifted image results in a non-zero mean, so those points near the mean are assumed to be valid. So the points farther than some distance from the mean could be removed. This procedure would accommodate larger steps (corrections).

However, another improvement would be to adjust the threshold automatically for particularly noisy data. This is accomplished by calculating the standard deviation of the errors. So, those points which are a fixed multiple of the standard deviation of the error from the mean error are removed. We denote the average by $\mu$ and the standard deviation by $\sigma$.

\begin{align}
\mu &= \frac{1}{n} \sum_{i=1}^{n} \text{error}_i, \\
\sigma^2 &= \frac{1}{n} \sum_{i=1}^{n} (\text{error}_i - \mu)^2
\end{align}

Then we remove those data points whose distance from their target line segment is outside the range $(\mu - F\sigma, \mu + F\sigma)$, where $F$ is determined experimentally to reflect the quantity of erroneous data points. It would perhaps be possible to find some correlation between the shape of the room and this quantity, though we have made little attempt to do so and have chosen $F$ to have a value of 1.5. This value of $F$ retains about 87% of the points, assuming the distribution of the errors is Gaussian [20].
Subtle implementation/intention discrepancy

Recall the meaning of the error term which is minimized by the matcher: The distance between a point and its infinite length target line. In other words, while the target of a point is determined by its distance from the line segments (as expected), the actual minimization equation effectively extends the target line segment to infinity in both directions and then finds the distance from the point to this line. The result is that points are drawn toward phantom lines (drawn to an infinite length line rather than toward the true line segment).

The desired and calculated distances are different only if the distance from the point \( p \) to its target line segment \( I \) does not lie on the perpendicular through \( p \) to \( L \). However, since target lines are determined based on distances to line segments, the influence of this undesirable side effect is reduced when outliers are removed. In a typical continuous line segment world model, such troublesome points occur mainly near outside corners. Also, an algorithm could be imagined which would identify and eliminate these data points.

The problem of phantom lines becomes more pronounced when many short lines are included in the model. Therefore it is better to use simple models (more long lines, fewer short ones). In Chapter V is an example which demonstrates better performance with a simpler model. In Chapter VII we also find that execution time is proportional to the number of lines in the model.

Indeterminate Results

The system matrix in section 3.3.2 is singular \( \begin{pmatrix} 0 \\ 0 \end{pmatrix} \) when all target lines in the calculation of the correction vector are parallel, or nearly parallel as is the case in Figure 25. Such a situation would not be uncommon with a robot traveling down a corridor. \( \begin{pmatrix} 0 \\ 0 \end{pmatrix} \) implies an infinite number of solutions. This makes intuitive sense when the robot is too far from the ends of a corridor to determine its lateral position. Clearly the matcher should be capable of determining the correc-
tion perpendicular to the walls, but the position along the hallway cannot be determined from the available sonar data points. Cox's solution to this problem is to make modifications to the system matrix to increase its stability which included calculating the singular value decomposition. Following is a quote from Cox in [7]. The meaning of the following symbols is not of significance here.

Third, the SVD requires a relatively large computation time. There is probably a faster simpler method to achieve the same purpose, omitting the use of $f$ and the SVD as described, and based on direct comparison of $M_2$, $\tilde{m}_1$, and $m_0$. We hope to develop this in the future.

We believe we have found such a method. Our approach is to adapt the basic algorithm used above to this special case. The modified algorithm is the topic of the following section.

**MODIFICATIONS FOR PARALLEL TARGET LINES**

As stated earlier, we assume that most points are near the line segment of the model they represent in the real world. This is generally a valid assumption since the purpose of the sonar matcher is to provide an update to the dead reckoning so that the posture errors do not grow without bound. However, even when the small step assumption is satisfied, the matching
algorithm described in the previous section yields an infinite number of solutions if all the target lines are parallel. Consider a simple example shown in Figure 26.

Clearly the data points need to move in the negative $y$ direction and rotate. Any value for $x$, however, yields a minimum error term and the system matrix $A$ is found to be singular.

There are three ways such a situation could occur in practice:

1. Data points very distant from the model (the robot is lost). In this case one line segment is generally the closest one and becomes the target of all the points.

2. The robot traveling down a corridor: Long, narrow, parallel line segments become the targets of the points.

3. Only a partial scan taken. If the robot was following a wall in a very large room, it may decide to acquire data only where it expected to find it valid (or the other walls could be too far away). As in 1, all data points would likely have the same line segment as their target.

The translation vector $t$ becomes under-determined so we reduce the dimension of the problem from three to two; i.e. we allow the data points to move only perpendicular to their target lines. Rotation, of course, is also performed.

The vector $v$ is assigned perpendicular to the (parallel) target lines. $K$ becomes the multiplier by which the points are moved along $v$. The "sum of the least squares" equation then becomes
\[ S = \sum_{i=1}^{n} \left[ R(\Delta \theta)(\mathbf{p}_i - \mathbf{c}) + (\mathbf{c} + \Delta \mathbf{K} \mathbf{v}) \right]' \mathbf{u}_i - r \]  \overset{\text{(45)}}{=} \]

Following the same procedure as before, we find the values for \( \Delta \mathbf{K} \) and \( \Delta \theta \) which minimize \( S \) by setting the partial derivatives to zero:

\[ \frac{\partial S}{\partial \Delta \mathbf{K}} = 0 \quad \frac{\partial S}{\partial \Delta \theta} = 0 \]  \overset{\text{(46)}}{=} \]

and solving the second order system of linear equations:

\[ \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{K} \\ \Delta \theta \end{bmatrix} = \begin{bmatrix} B_{11} \\ B_{21} \end{bmatrix} \]  \overset{\text{(47)}}{=} \]

where

\[ A = \begin{bmatrix} \sum_{i=1}^{n} (\mathbf{u}_i \cdot \mathbf{p}_x + \mathbf{u}_i \cdot \mathbf{p}_y)^2 & \sum_{i=1}^{n} (\mathbf{p}_x \mathbf{u}_i \cdot \mathbf{p}_y \mathbf{u}_i) (\mathbf{c} \mathbf{u}_i - \mathbf{c}_y \mathbf{u}_i + \mathbf{u}_i \mathbf{v}_i - \mathbf{u}_i \mathbf{v}_i) \\ \sum_{i=1}^{n} (\mathbf{u}_i \cdot \mathbf{p}_x + \mathbf{u}_i \cdot \mathbf{p}_y) (-\mathbf{u}_i \mathbf{v}_i + \mathbf{u}_i \mathbf{c}_y + \mathbf{u}_i \mathbf{v}_i - \mathbf{u}_i \mathbf{c}_y) & \sum_{i=1}^{n} (-\mathbf{u}_x \mathbf{v}_i + \mathbf{u}_i \mathbf{c}_y + \mathbf{u}_i \mathbf{v}_i - \mathbf{u}_i \mathbf{c}_y)^2 \end{bmatrix} \]  \overset{\text{(48)}}{=} \]

\[ B = \begin{bmatrix} - (\mathbf{p}_x \mathbf{u}_i \cdot \mathbf{p}_y \mathbf{u}_i) (\mathbf{r} + \mathbf{u}_i \mathbf{v}_i - \mathbf{u}_i \mathbf{v}_i) \\ - (\mathbf{u}_i \cdot \mathbf{v}_i + \mathbf{u}_i \mathbf{v}_i - \mathbf{r}) (-\mathbf{u}_x \mathbf{v}_i + \mathbf{u}_i \mathbf{c}_y + \mathbf{u}_i \mathbf{v}_i - \mathbf{u}_i \mathbf{c}_y) \end{bmatrix} \]  \overset{\text{(49)}}{=} \]

The solution vector may then be determined:

\[ \Delta \mathbf{K} = \frac{B_{21} - B_{11} A_{22}}{A_{21} - A_{11} A_{22}} \]  \overset{\text{(50)}}{=} \]

\[ \Delta \theta = \frac{A_{21} (B_{21} - B_{11} A_{22})}{A_{21} - A_{11} A_{22}} \]  \overset{\text{(51)}}{=} \]

and the correction is:

\[ P_n(n) = \left[ x_{n-1} + K \mathbf{v}_x , y_{n-1} + K \mathbf{v}_y , \theta_{n-1} + \theta \right] \]  \overset{\text{(52)}}{=} \]

where the subscript of \( n \) denotes the matching session and
\[ K = \sum \Delta K \quad \theta = \sum \Delta \theta \]  

over the number of iterations. As in the 3D case, the effect of rotating the points about \( e \) must be negated before applying the correction to the robot’s location (equation 41).

If all the target lines are parallel, the appropriate matching algorithm is used (two dimensional one). However, if it is due to reason (1) above, moving the points closer to the model could result in some non-parallel targets. For this reason the targets are tested during each iteration of the algorithm to determine which minimization equations to use.

Recall that the match is calculated with rotations relative to the centroid of points. This brought up an interesting problem, since when the total correction is applied to the robot’s position, it yields a movement not only in the direction orthogonal to the parallel target lines but also parallel to them. This is due to the rotation effect when transforming coordinates from the centroid-based correction and the robot-based correction.

To prevent this side-effect, the matcher algorithm was modified to use the robot’s position as the center of rotation instead of the centroid of points when calculating the correction vector in the parallel-targets case. The additional computation necessary is negligible and the intent of the original algorithm is preserved.
CHAPTER V

TESTING THE MATCHER

METHODOLOGY

Testing of the sonar matching algorithm was performed by: Mapping a room, gathering sonar data, corrupting it, applying the matcher, and analyzing the results. From these results we hoped to characterize the performance of the matching algorithm under different conditions, in different rooms, and under bounded odometry errors.

Figure 27 illustrates the steps used to test the matcher algorithm.

First, a blueprint was secured from the architecture department within the physical plant at PSU. The geometric origin was arbitrarily chosen as the NW corner of the building with zero degrees pointing south. Next the walls of a room on the blueprint were turned into line segments by finding the coordinates of their endpoints. It was found that some walls on the blueprint were inaccurate and others appeared to be simply in the wrong place. This showed that with a building with many temporary partitions and multiple reconstructions, drawings cannot be relied upon wholeheartedly and required verification by a tape measure and a friend. We suspect that a
computer-drawn floorplan would have been more accurate and would certainly have been more
easily updated.

Since the scope of this work did not include generating, searching, or retrieving line segment
models from the World Database, relevant lines were manually chosen and entered into a
file for each room investigated.

The next step was to move the robot to a known location and perform a sonar scan. We call
this known position and heading the actual posture. These scans were then transferred from the
PC to a Unix system for analysis.

To analyze the algorithm it was necessary to induce positional errors into the data to see
how the matcher performed in spite of them. Since all the points of a particular scan were taken
from the same location (the robot was stationary), a simple shift and rotation of the image did just
that. Additional noise was not added since the data already contained "real" noise. The amount
of shift in $x$ and $y$ and the amount of rotation ($\theta$) about the robot's location is called the induced $\Delta$ ("Delta"), defined as

$$\Delta=(\Delta^x, \Delta^y)$$

where

$$\Delta^p = (\Delta^x, \Delta^y)$$

The matching algorithm could then be employed on the corrupted data in an attempt to
negate the effects of the induced $\Delta$. The difference between the result of the matcher and the
actual posture is called the "error" of that matching session. It should be noted that a $\Delta$ of
$(0,0,0)$ does not, in general, yield an error of $(0,0,0)$ due primarily noisy sonar measurements.

Figure 28 shows a $\Delta^p$ grid with an assigned $\Delta^\theta$ of $20^\circ$ and illustrates how an image is
corrupted (shifted and rotated) by adding a particular $\Delta^p$ and rotating by $\Delta^\theta$.

Matching each corrupted image from each setting of each room would produce a seeming
myriad of data between which relationships could be shown. For example: How does the $\Delta^x$ (or
Figure 28. Simulating odometry errors/corrupting an image.

$\Delta y$ or $\Delta \theta$ affect the ability of the matcher to converge; how does the number (or closeness or length or one-sidedness) of the lines affect the convergence (or average $x,y$ or $\theta$ errors) of the matcher; how does the change in the parameter $F$ in the noise-removal equation affect the convergence (or $x,y,$ or $\theta$ errors) of the matcher, and so on. Worse still, it is not clear of what use the data would be. Therefore we simply chose to test the algorithm’s ability to correct for position and rotation errors and demonstrate locations from which the matcher converged correctly. The investigation of other relationships mentioned above would perhaps lead to improved performance by tuning parameters based on what is known about the world or data.

So, first we demonstrate the matcher on real data for various rooms in an attempt to give the reader a feel for the process of matching. Then we develop and present a broader approach where multiple $\Delta P$’s are shown at the same time.

The final, and ultimate test of the matcher is to combine it with a pilot algorithm on the robot and observe its ability to maneuver within a room using odometry and occasionally matching a sonar scan to the model.
EXPECTED RESULTS

Since the matching algorithm operates by converging on the nearest minimum sum of mean squared errors, we expected to see a small constant error when inducing small $\Delta P$'s, and large errors for large $\Delta P$'s. The relationship was not expected to be linear but binary: either the matcher converges on the correct local minimum or it converges on the wrong one (yielding a larger error).

To illustrate this point, we show a 3D plot of the sum-of-squared-distances over a large region. As can be seen from the plot in Figure 29, the global minimum is in the region we would expect (where $\Delta P = (0,0)$ at the center of the surface). However, as $\Delta P$ increases, the small-step assumption becomes less and less valid which yields many locations around the perimeter of the plot where the algorithm could get stuck in local-minima. Also note that this graph was generated for a particular $\Delta \theta$. Changing the induced rotational error would yield a family of surfaces from which the matching algorithm attempts to find the minimum.

Figure 29. Sum of the squared errors versus $\Delta P$ example.
TYPES OF ROOMS

Four types of rooms were used in our analysis of the matcher: small rectangular, large rectangular, corridor (two parallel walls), and a combination of the above. Each type has associated good and bad points which are discussed below.

A small rectangular room is one in which the matcher performs the best. Its small size allows the sonar device to locate all four walls, yielding a well-founded match. It was found that the matcher often converged correctly despite very large translations (greater than 10 feet). A problem with rectangular rooms is that a rotation of the image by $45^\circ$ or more often caused the matcher to converge on a correction whose heading was off by $90^\circ$.

Large rectangular rooms present the additional problem of range. If walls are too far to be reached by the sonar, somewhat shorter distances are recorded which must be removed as noise. Also, if only one wall is found with the sonar, the distance along the wall remains unknown until a non-parallel wall is reached. This is the problem of parallel (single) target lines as seen in the next room type. Another more subtle characteristic of scans taken in larger rooms is that a small heading error moves distant readings much farther from their true location than in smaller rooms. The result is lower rotational noise-immunity.

Parallel walls present a situation where the position along the hall is non-deterministic and is assumed to be the position given by the odometry. However, $\Delta \theta$ may be as large as $90^\circ$ without an incorrect convergence.

Intersecting corridors and other complex rooms are perhaps the most difficult to match due to the fact that a small $\Delta P$ can place points close to the wrong side of either or both hallways leading to an incorrect convergence. In other words, there are many false local minimums near the correct one when many mutually close parallel lines make up the local line segment model.

One could certainly produce other room types from which additional conclusions could be drawn, but we chose to limit our observations to the simple-yet-complete types shown for
simplicity. We also believe that these rooms are characteristic of those most often occurring in "real" buildings. This is especially true of corridors through which the robot would spend much of its time navigating.

SAMPLE MATCHING SESSIONS

We begin our presentation of the results by showing simulations of several concrete examples from which we can begin to evaluate the matching algorithm in practice. All four types of rooms are represented and various $\Delta$'s are used.

Room 0

Correct Convergence. The matching algorithm is first demonstrated on Dr. Perkowski's basement lab: A small rectangular room in which much of the PSUBOT was developed. The room has shelves covering nearly 3 of the walls and other assorted clutter which was not entered into the model of the room. The wall on the left is smooth, and in the scan shown, the author was standing near the doorway located near the right side of the bottom wall (as may be seen by eight data points which come out from the bottom wall). First we show the results of the matcher with $\Delta = (-5, -5, 0.5236)$; i.e. the estimated position is wrong by 5 feet in both $x$ and $y$ directions, and the believed heading is wrong by $20^\circ$ as seen in Figure 30. The second image shows the progress of the matcher just before the last correction. From it we can see that the noise removal algorithm was successful in eliminating those points resulting from the person standing near the far wall. Table IV tabulates the statistics of the matching session shown.

We can see from Table IV that initially there were many points dropped, as there were many points far from their targets. As they came closer to their target lines the number was reduced to those which were truly noisy or simply wrong (for a distinction see Chapter IV/Removal Of Outliers). Notice also, that the process terminated when the maximum change in $x$ and $y$ was less than 0.2 feet and the maximum change in $\theta$ was less than 0.1 radians ($5.7^\circ$).
Figure 30. Room 0: Good matching session.

TABLE IV

MATCHING TRACE OF GOOD MATCH FOR ROOM 0: $A=((-5.0, -5.0, 0.5236), r=1.5$

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Average Dist</th>
<th>Std Dev of Dist</th>
<th># Pts. Dropped</th>
<th>Correction (distances in feet)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x$</td>
<td>$y$</td>
<td>$\theta$ (rad)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1.537</td>
<td>3.148</td>
<td>13</td>
<td>2.230 2.478 -0.032</td>
</tr>
<tr>
<td>2</td>
<td>-0.247</td>
<td>2.744</td>
<td>12</td>
<td>0.741 1.372 -0.111</td>
</tr>
<tr>
<td>3</td>
<td>0.076</td>
<td>3.054</td>
<td>10</td>
<td>0.541 0.963 -0.119</td>
</tr>
<tr>
<td>4</td>
<td>0.451</td>
<td>3.545</td>
<td>11</td>
<td>0.558 0.341 -0.113</td>
</tr>
<tr>
<td>5</td>
<td>0.658</td>
<td>3.394</td>
<td>10</td>
<td>0.435 0.104 -0.127</td>
</tr>
<tr>
<td>6</td>
<td>0.909</td>
<td>3.362</td>
<td>9</td>
<td>0.150 -0.054 -0.063</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>4.658 5.206 -0.568</td>
</tr>
<tr>
<td>Error</td>
<td></td>
<td></td>
<td></td>
<td>-0.3 0.2 -0.0</td>
</tr>
</tbody>
</table>

Faster matching sessions may be achieved by changing the termination condition at an expense of accuracy.

As can be seen both pictorially and numerically the matching algorithm found the correct orientation of the robot despite the initial error.

Incorrect Convergence. We now demonstrate a situation where the matcher converges on the wrong local minimum. If $A=(5.0, -5.0, 0.5236)$ the matcher continues to rotate the image further rather than rotate it back. The result is a relatively good (yet incorrect) match for this rather
square-like room as it is similar after rotating 90° as seen in Figure 31. Also notice in Table V that it takes many more iterations for the algorithm to settle. This is because of the pseudo-rotation used in the derivation of the sum of least squares equations.

![Figure 31. Room 0: Bad matching session.](image)

**TABLE V**

MATCHING TRACE OF BAD MATCH FOR ROOM 0: $A=(-5.0,\ -5.0,0.5236)$, $F=1.5$

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Average Dist</th>
<th>Std Dev of Dist</th>
<th># Pts. Dropped</th>
<th>Correction (distances in feet)</th>
<th>$x$</th>
<th>$y$</th>
<th>$\theta$ (rad)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.380</td>
<td>4.945</td>
<td>12</td>
<td>-0.147</td>
<td>0.801</td>
<td>0.106</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.782</td>
<td>4.905</td>
<td>12</td>
<td>-1.353</td>
<td>1.459</td>
<td>0.039</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.258</td>
<td>4.235</td>
<td>12</td>
<td>-2.859</td>
<td>1.221</td>
<td>0.094</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.230</td>
<td>3.660</td>
<td>15</td>
<td>-1.614</td>
<td>0.040</td>
<td>0.075</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.474</td>
<td>2.570</td>
<td>11</td>
<td>-1.055</td>
<td>-0.222</td>
<td>0.097</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.097</td>
<td>2.350</td>
<td>11</td>
<td>-1.078</td>
<td>-0.597</td>
<td>0.139</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.065</td>
<td>2.645</td>
<td>11</td>
<td>-0.660</td>
<td>-0.449</td>
<td>0.138</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>-0.569</td>
<td>2.857</td>
<td>10</td>
<td>-0.259</td>
<td>-0.169</td>
<td>0.126</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>-0.500</td>
<td>3.076</td>
<td>11</td>
<td>-0.055</td>
<td>-0.167</td>
<td>0.103</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-0.419</td>
<td>3.002</td>
<td>11</td>
<td>0.023</td>
<td>0.048</td>
<td>0.018</td>
<td></td>
</tr>
</tbody>
</table>

Total: -9.060 1.96 0.939

Error: -4.1 -3.0 1.5

Looking at the two matching sessions we observe that, for this image the matcher worked for $A=(-5,-5.0,0.5236)$ but did not work for $A=(5,-5.0,0.5236)$. From this we see that convergence
depends on the direction of displacement and not on its magnitude only (the magnitudes were exactly the same).

We forgo showing incorrect matchings for the remainder of the rooms. Our purpose in this section is to give the reader a feel for what types of images can be expected from various room, not to show every possible matching session.

Room 1

The second room demonstrated is a large rectangular room. It is the main lobby of the PCA T building. It is uncluttered except for some chairs which are in a line about 4 feet from the walls on the right and left. There are also some short tables which do not enter into the image because of their height. The walls behind the chairs are nearly ideal for sonar. They have vertical slats spaced about one inch apart which allows the ultrasonic pulse to bounce back to the transmitter. As an added bonus corkboard lines the back side of the slats for a reduction in echo. From this example it is clear that walls used for matching will not always be the same as those used for obstacle avoidance (they cannot be seen by the spinning sonar device, but still must be avoided).

Figure 32 shows the robot near the double doors. Several people were watching as the scan was made, as can be seen by what looks like a wall above and to the left of the robot. Notice that most of this noise was removed in the second image.

Room 2

Next we demonstrate the matcher in a corridor. To the matcher, there are only two parallel target lines, so it uses its parallel (two dimensional) solution. This means that the matcher will correct the position of the robot only along the corridor; in the y direction. (The total correction in the table is non-zero due to rounding errors.) Table VI tabulates the statistics of the matching session shown.
Figure 32. Room 1: Matching in a large rectangular room.

TABLE VI

MATCHING TRACE FOR ROOM 1: $A=(6.0,4.0,0), F=1.5$

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Average Dist</th>
<th>Std Dev of Dist</th>
<th># Pts. Dropped</th>
<th>Correction (distances in feet)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x$</td>
<td>$y$</td>
<td>$\theta$ (rad)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.342</td>
<td>12.098</td>
<td>29</td>
<td>-5.626 -4.114 0.053</td>
</tr>
<tr>
<td>2</td>
<td>-2.304</td>
<td>12.166</td>
<td>25</td>
<td>-0.007 -0.195 -0.101</td>
</tr>
<tr>
<td>3</td>
<td>-1.925</td>
<td>12.864</td>
<td>25</td>
<td>-0.028 -0.009 -0.031</td>
</tr>
<tr>
<td>Total</td>
<td>-5.661</td>
<td>-4.318</td>
<td>-0.079</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>0.3</td>
<td>-0.3</td>
<td>-0.1</td>
<td></td>
</tr>
</tbody>
</table>

The corridor is lined on both sides by glass. For this reason the sonar data is much noisier than in previous rooms, and therefore the matcher does not perform as well. Since it is long and narrow and due to the spreading effect of the sonar beam, measurements along the hallway are nearly always much shorter than they are in reality.

Also note that the algorithm is more tolerant of noise in this setting than in the previous ones. This is because the standard deviation of the distance from each point to its target line is much larger. However, that noise which remains after filtering tends to be cancelled out by points near the opposite wall.
Following is a matching session from a corridor. Table VII tabulates the statistics of the matching session shown.

![Corridor Image](image)

**Figure 33.** Room 2: corrupted corridor image.

**TABLE VII**

**MATCHING TRACE FOR ROOM 2:** $\Delta=(-3.0,0.0,0.1745)$, $F=1$

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Average Dist</th>
<th>Std Dev of Dist</th>
<th>#Pts. Dropped</th>
<th>Correction (distances in feet)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x$</td>
<td>$y$</td>
<td>$\theta$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-2.036</td>
<td>1.455</td>
<td>27</td>
<td>2.048 0.0 -0.025</td>
</tr>
<tr>
<td>2</td>
<td>-0.312</td>
<td>1.418</td>
<td>22</td>
<td>0.326 0.0 -0.033</td>
</tr>
<tr>
<td>3</td>
<td>-0.223</td>
<td>1.349</td>
<td>21</td>
<td>0.243 0.0 -0.054</td>
</tr>
<tr>
<td>4</td>
<td>-0.008</td>
<td>1.189</td>
<td>23</td>
<td>0.015 0.0 -0.021</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>2.632 0.0 -0.135628</td>
</tr>
<tr>
<td>Error</td>
<td></td>
<td></td>
<td></td>
<td>-0.4 0.0 0.0</td>
</tr>
</tbody>
</table>

**Room 3**

The final room type investigated is located at the "T" of the above corridor and another short corridor. Again we show the matching process pictorially and numerically. Table VIII tabulates the statistics of the matching session shown.
As an example of the effect of the termination condition on the resulting error, the above matching session converges with an error of \((x=-0.122484, \ y=0.539079, \ \theta=-0.073256)\) in six iterations when the maximum allowable change in \(x, y,\) and \(\theta\) is set to 0.1, 0.1 and 0.01 respectively (half of the values used above). It will require 50% more time to compute, however.

**REGIONS OF CONVERGENCE**

A large number of pictures and tables as shown above with varying \(\Delta\)'s would not be adequate in our attempt to characterize the matchers performance or to ensure with reasonable doubt
that it would correct for expected odometry errors. Therefore we chose to plot the region of convergence for various rooms.

To generate a region of convergence we first determined the error resulting from a matching session with $\Delta P = (0, 0)$. We call this the nominal error. Then, for $\Delta P = (x, y)$ where $x \neq 0, y \neq 0$, we determined whether or not the algorithm converged to within the nominal error from the actual position. If it did, we placed a dot at $(x, y)$. Otherwise a dot was not plotted at $(x, y)$. The result is a grid of points with dots missing where the algorithm did not converge correctly.

All $\Delta P$'s were not be tested, nor were all $\Delta \theta$'s. The tests were performed at intervals of 1 foot. For each diagram $\Delta \theta$ is constant (as indicated above each diagram), and the $\Delta P$ region of test is bounded by a dotted line.

Since such a diagram could be misleading if the matcher converged incorrectly even for $\Delta = (0, 0, 0)$ we show the absolute distance error and angle error (in radians) for each diagram.

Also note that the points are superimposed over the line-segment room and the robot's actual location is designated by a box. The actual heading of the robot can be seen by the solid arrow, and $\theta + \Delta \theta$ is shown as a dotted arrow, both originating at the center of the box representing the robot. All the following convergence diagrams were generated with the noise-removal multiplier $F$ set at 1.5 unless otherwise indicated. Also, all angles are in radians and distances are in feet.

**Room 0**

Now we show the region of convergence for the sonar scan from room 0 shown above.

Notice that the matcher does not converge for $\Delta = (-2, 1, 0.5236)$. For this instance the matcher does not rotate the image back to where it should be, but rather the top wall gets stuck in the upper left corner, where it stays. We found two cures for this particular situation. But first, we show the 3D plot of sum-of-squared-distance versus $\Delta P$ from which we see the "landscape"
\[ \Delta \theta = 0 \]

\[ \Delta \theta = 0.5236 \]

The first cure for the incorrect convergence was found by changing the termination condition. Since the slope along \( \Delta \theta \) in the region of \( \Delta = (-2, 1, 0.5236) \) was too small, the incremental heading correction was smaller than the termination condition. By changing the termination
condition, iteration continued until the $\theta$ correction was less than 0.08 radians (instead of 0.1) and the algorithm eventually converged correctly.

Secondly, a change to the model also resulted in a correct convergence where the algorithm otherwise did not converge correctly. Figure 37 shows the changes made to the model and the resulting convergence diagram.

![Figure 37. Room0: Sum of squares versus $\Delta P$.](image)

It is also important to note that by approximating several lines by a diagonal one, the resulting match takes less time (fewer line segments to consider). The lesson here is that simple rooms (long line segments) allow better performance than rooms with every feature modeled. This was predicted in Chapter IV.

Also, notice on the 3D plot that there is a local minimum near $\Delta = (5,-5,0.5236)$ - on the right hand side of the graph. We now see why the algorithm did not converge correctly in that region.

For the remaining rooms show more combinations of $\Delta \theta$ and more room locations than with
\( \Delta \theta = 0.5236 \)

20

15

10

5

0

-5

-5

0

5

10

15

20

\[
\begin{array}{cccccc}
0 & 5 & 10 & 15 & 20 \\
-5 & 0 & 5 & 10 & 15 & 20
\end{array}
\]

\text{error.r(), error.theta()} = 0.468808, -0.051575

\textbf{Figure 38.} Room 0: region of convergence (simplified room layout).

Room 0. Quite simply, it was found in Room 0 that the matcher converged for all reasonable \( \Delta \)'s, and therefore we chose to show only a few diagrams.

\[ \Delta \theta = -0.5236 \]

20

15

10

5

0

-5

-5

0

5

10

15

20

\[
\begin{array}{cccccc}
0 & 5 & 10 & 15 & 20 \\
-5 & 0 & 5 & 10 & 15 & 20
\end{array}
\]

\text{error.r(), error.theta()} = 0.418362, -0.087793

\textbf{Figure 39.} Room 0: region of convergence.
Room 1

Next we show various $\Delta \theta$'s from different locations within the large rectangular room. Notice that line segments representing hidden walls were removed from the model. This action was performed manually and was found to increase the performance of the algorithm both computationally and functionally.

$\Delta \theta = 0$

$\Delta \theta = 0.1745$

Figure 40. Room 1: region of convergence.

Figure 41. Room 1: region of convergence.
$\Delta \theta = -0.1745$

$\Delta \theta = -0.349$

Figure 42. Room 1: region of convergence.

error.r(), error.theta() = -.848192, -.06814

epsilon = 0.349

Figure 44. Room 1: region of convergence.

epsilon = 0.5236

Figure 43. Room 1: region of convergence.

error.r(), error.theta() = .849812, -.068644

Figure 45. Room 1: region of convergence.

error.r(), error.theta() = .850624, -.068823
\[ \Delta \theta = -0.5236 \]

\[ \Delta \theta = 0.1745 \]

\[ \Delta \theta = 0 \]

\[ \Delta \theta = -0.1745 \]

error.r(), error.theta() = 0.8491, -0.068422

error.r(), error.theta() = 0.487171, 0.051137

Figure 46. Room 1: region of convergence.

Figure 47. Room 1: region of convergence.

Figure 48. Room 1: region of convergence.

Figure 49. Room 1: region of convergence.
\[ \Delta \theta = 0.349 \]

**Figure 50.** Room 1: region of convergence.

\[ \Delta \theta = 0.5236 \]

**Figure 52.** Room 1: region of convergence.

\[ \Delta \theta = -0.349 \]

**Figure 51.** Room 1: region of convergence.

\[ \Delta \theta = -0.5236 \]

**Figure 53.** Room 1: region of convergence.

\[
\begin{align*}
\text{error.r()}, \text{error.theta()} &= 0.486863, 0.051149 \\
\text{error.r()}, \text{error.theta()} &= 0.487194, 0.051136 \\
\text{error.r()}, \text{error.theta()} &= 0.23865, -0.021963 \\
\text{error.r()}, \text{error.theta()} &= 0.238653, -0.021964
\end{align*}
\]
\[ \Delta \theta = 0 \]

\begin{align*}
50 & - \quad 50 \\
40 & - \quad 40 \\
30 & - \quad 30 \\
20 & - \quad 20 \\
80 & \quad 80 \\
90 & \quad 90 \\
100 & \quad 100 \\
110 & \quad 110 \\
120 & \quad 120
\end{align*}

error.r(), error.theta() = .924715, .012336

\textbf{Figure 54.} Room 1: region of convergence.

\[ \Delta \theta = -0.1745 \]

\begin{align*}
50 & - \quad 50 \\
40 & - \quad 40 \\
30 & - \quad 30 \\
20 & - \quad 20 \\
80 & \quad 80 \\
90 & \quad 90 \\
100 & \quad 100 \\
110 & \quad 110 \\
120 & \quad 120
\end{align*}

error.r(), error.theta() = .961147, -.004026

\textbf{Figure 56.} Room 1: region of convergence.

\[ \Delta \theta = 0.1745 \]

\begin{align*}
50 & - \quad 50 \\
40 & - \quad 40 \\
30 & - \quad 30 \\
20 & - \quad 20 \\
80 & \quad 80 \\
90 & \quad 90 \\
100 & \quad 100 \\
110 & \quad 110 \\
120 & \quad 120
\end{align*}

error.r(), error.theta() = .873726, .012117

\textbf{Figure 55.} Room 1: region of convergence.

\[ \Delta \theta = 0.349 \]

\begin{align*}
50 & - \quad 50 \\
40 & - \quad 40 \\
30 & - \quad 30 \\
20 & - \quad 20 \\
80 & \quad 80 \\
90 & \quad 90 \\
100 & \quad 100 \\
110 & \quad 110 \\
120 & \quad 120
\end{align*}

error.r(), error.theta() = .918394, .012362

\textbf{Figure 57.} Room 1: region of convergence.
\[ \Delta \theta = -0.349 \]

50 ──────

40 ─
30 ─
20 ─

\[ \text{error}_r(), \text{error}_\theta() = 0.827203, -0.030383 \]

\[ \Delta \theta = 0.5236 \]

50 ──────

40 ─
30 ─
20 ─

\[ \text{error}_r(), \text{error}_\theta() = 0.918394, 0.012302 \]

\[ \Delta \theta = -0.5236 \]

50 ──────

40 ─
30 ─
20 ─

\[ \text{error}_r(), \text{error}_\theta() = 0.960746, -0.004027 \]

**Figure 58.** Room 1: region of convergence.

**Figure 59.** Room 1: region of convergence.

**Figure 60.** Room 1: region of convergence.

\[ \text{error}_r(), \text{error}_\theta() = 0.382047, -0.023003 \]

**Figure 61.** Room 1: region of convergence.
\[ \Delta \theta = 0.1745 \]

\[ \Delta \theta = 0.349 \]

\[ \Delta \theta = -0.1745 \]

\[ \Delta \theta = -0.349 \]

\[ \text{error}_r(), \text{error}_\theta() = 0.381688, -0.022927 \]

\[ \text{error}_r(), \text{error}_\theta() = 0.382692, -0.02313 \]

\[ \text{error}_r(), \text{error}_\theta() = 0.382693, -0.02313 \]

Figure 62. Room 1: region of convergence.

Figure 63. Room 1: region of convergence.

Figure 64. Room 1: region of convergence.

Figure 65. Room 1: region of convergence.
\[ \Delta \theta = 0.5236 \]

\[ \Delta \theta = -0.5236 \]

Figure 66. Room 1: region of convergence.

\[ \text{error.} r(), \text{error.} \theta() = 0.381668, -0.022923 \]

Figure 67. Room 1: region of convergence.

\[ \text{error.} r(), \text{error.} \theta() = 0.387294, -0.023734 \]
Room 2

Room two is a corridor with parallel lines. We found that the glass windows on both sides of the hallway were poor reflectors to sonar and the upright window frames (not modeled) caused distances to congregate together.

Notice that the algorithm converges only for changes in the $x$ direction. This is due to the fact that there is no correction in the $y$ direction due to the parallel target lines (see chapter IV).

We show convergence diagrams for three locations in the hallway. The first setting exhibits rather poor convergence due to a bad combination of factors. In particular, the wall on the left appears in the scan to be slightly diagonal which affects the convergence when $\Delta \theta$ is anything but small and negative.

The second and third settings exhibit superior performance to the first setting due to a more accurate scan.

\[ \Delta \theta = 0 \]

\[ \Delta \theta = 0.1745 \]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig68}
\caption{Room 2: region of convergence.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig69}
\caption{Room 2: region of convergence.}
\end{figure}
$\Delta \theta = -0.1745$

$\Delta \theta = -0.349$

$\Delta \theta = 0.349$

$\Delta \theta = 0.5236$

Figure 70. Room 2: region of convergence.

Figure 72. Room 2: region of convergence.

Figure 71. Room 2: region of convergence.

Figure 73. Room 2: region of convergence.
\[ \Delta \theta = -0.5236 \]

\[ \text{error}_r, \text{error}_\theta = 0.043004, -0.268834 \]

**Figure 74.** Room 2: region of convergence.
The above session was unsuccessful in determining the correct heading of the robot, but notice how the following setting performed much better. We attribute the difference to the quality of images acquired from the sonar device and differences in the texture of the wall at the two locations.

\[ \Delta \theta = 0 \]

\[ \Delta \theta = 0.0873 \]

Figure 75. Room 2: region of convergence.

Figure 76. Room 2: region of convergence.
$\Delta \theta = -0.0873$

$\Delta \theta = -0.1745$

Figure 77. Room 2: region of convergence.

$\Delta \theta = 0.1745$

$\Delta \theta = 0.2618$

Figure 78. Room 2: region of convergence.

Figure 79. Room 2: region of convergence.

Figure 80. Room 2: region of convergence.

error.r(), error.theta() = .088327, -.026898

error.r(), error.theta() = .088327, -.026905

error.r(), error.theta() = .283823, .071112

error.r(), error.theta() = .283824, .071117
\[ \Delta \theta = -0.2618 \]

\[ \Delta \theta = 0 \]

\[ \text{error}(r), \text{error}(\theta) = 0.0329, -0.063183 \]

**Figure 81.** Room 2: region of convergence.

\[ \text{error}(r), \text{error}(\theta) = 0.015202, -0.078792 \]

**Figure 82.** Room 2: region of convergence.

\[ \Delta \theta = 0.1745 \]

\[ \text{error}(r), \text{error}(\theta) = 0.102635, -0.048868 \]

**Figure 83.** Room 2: region of convergence.
\( \Delta \theta = -0.1745 \)

\( \Delta \theta = -0.349 \)

\( \Delta \theta = 0.349 \)

\( \Delta \theta = 0.5236 \)

Figure 84. Room 2: region of convergence.

Figure 85. Room 2: region of convergence.

Figure 86. Room 2: region of convergence.

Figure 87. Room 2: region of convergence.
\[ \Delta \theta = -0.5236 \]

Figure 88. Room 2: region of convergence.

\[ \text{error}(r), \text{error}(\theta) = .138838, -.146269 \]
Room 3

Room three is the most difficult since small rotations often moved points close to an incorrect line - and the matcher tried to move them closer. For this reason the matcher often converged on the incorrect heading, though the position was often correct. We expect that improvements mentioned in the conclusion would have a positive effect on the performance of the matcher on this type of room.

\[ \Delta \theta = 0 \]

\[ \Delta \theta = 0.1745 \]

error.r(), error.theta() = .075339,.023946

error.r(), error.theta() = .147611,.134927

Figure 89. Room 3: region of convergence.

Figure 90. Room 3: region of convergence.
\[ \Delta \theta = -0.1745 \]

\[ \Delta \theta = 0.349 \]

\[ \Delta \theta = 0.5236 \]

Figure 91. Room 3: region of convergence.

Figure 93. Room 3: region of convergence.

Figure 92. Room 3: region of convergence.

Figure 94. Room 3: region of convergence.
$\Delta \theta = -0.5236$

error.$r()$, error.$\theta()$ = .076232, -.012514

Figure 95. Room 3: region of convergence.
The algorithm sometimes performs poorly in the following setting since there is not a strong \( y \) component in the data, i.e. the sonar device found almost no \( x \) walls during its scan. However, the algorithm still does well when \( \Delta \theta \) is small.

\[
\Delta \theta = 0
\]

\[
\Delta \theta = 0.1745
\]

Figure 96. Room 3: region of convergence.

Figure 97. Room 3: region of convergence.
\[ \Delta \theta = -0.1745 \]

Figure 98. Room 3: region of convergence.

\[ \Delta \theta = -0.349 \]

Figure 99. Room 3: region of convergence.

\[ \Delta \theta = 0.349 \]

Figure 100. Room 3: region of convergence.

\[ \Delta \theta = 0.5236 \]

Figure 101. Room 3: region of convergence.
For the following plot, we do not achieve a correct convergence even when $\Delta P = (0,0)$. Incidentally, all the points in the dotted region below the robot converged to the same place yielding an error of $(r = 0.97807, \theta = -0.127)$

\[ \Delta \theta = -0.5236 \]

\[ \text{error.} r(), \text{error.} \theta() = 0.313273, -0.076026 \]

**Figure 102.** Room 3: region of convergence.
CHAPTER VI

EVALUATION OF MATCHER RESULTS

ACCURACY

In chapter IV an algorithm was developed for matching sonar-based range data to a line-segment model of the world. In chapter V the algorithm was demonstrated on real data from several rooms, and regions of convergence for particular locations (settings) were generated.

There are two figures of merit applied to the matching algorithm. First, how close to the actual location is the algorithm able to converge, and secondly, in what region (within how many feet and degrees) does the algorithm converge to that value.

For rooms 0 and 1 the algorithm converged to within one foot and 4' for nearly all values of $A$ tested (five feet and $\pm 30'$). We had predicted that the algorithm would converge well in these rooms.

For rooms 2 and 3, the algorithm converged quite accurately, but sometimes over a smaller region. Except for one setting, which converged to within 3 inches and 30' (not very good), the algorithm converged to within 3 inches and 8'. However, sometimes a $\Delta P$ of only one foot caused the algorithm to converge incorrectly, and in one case it didn't converge correctly even for a $\Delta P$ of 0 (though $\Delta \theta$ was $-30'$ at the time).

Additionally, in rooms 2 and 3 and for $\Delta \theta = 0$, the region of convergence was at least one foot (though usually much more) and for small $\Delta \theta = \pm 10'$ the region was usually at least two feet. For $\Delta \theta = \pm 20'$ the region was usually at least one foot.

In general, the matcher always converged to within 1 foot and with an angle error of less than 5' when $\Delta \theta$ was zero. Also, even when the final heading errors were large, they were always
less than the induced $\Delta \theta$. In other words, the matcher reduced the heading error.

So for the first two rooms the algorithm had a very large region of convergence and acceptable accuracy, and in the second two rooms the algorithm had good accuracy but sometimes a poor convergence region.

Now for typical maneuvers, a location estimation within one foot would be satisfactory. Moving through doorways and other more precise maneuvers would require additional sensor/feedback algorithms tailored specifically to the motion. Finally, if because of position error the robot were about to collide with an obstacle, the obstruction would be seen by collision avoidance sensors and obstacle avoidance algorithms would be invoked; the robot would not simply collide with the obstacle because of localization errors.

MATCHING FREQUENCY

Now the question arises: “How often should the robot use the matcher algorithm?” There is an upper limit which is bounded by both computational power of the CPU and physically obtaining the sonar soundings. The lower limit was developed in chapter IV, equation 15 where cumulative odometry errors were related to the total distance from the desired path. These are primarily due to errors occurring from a rotation, followed by a forward motion which then accumulates distance errors. Since odometry is more accurate than the sonar matcher in the short term, the frequency of matching sessions should be low enough that odometry errors are large enough to bother correcting. Recall that the standard deviation of the error of a matching session is much larger than the standard deviation of the odometry error in the short term.

We therefore set out to put an upper bound on the distance ($r_s$) the robot may travel forward (after a rotation) between sonar matching sessions. (Symbols and data used are taken from the section on Odometry in chapter IV.)

First we assume that a rotation is never more than 180°. In this way we can put a realistic
bound on the rotation error $e_9$. From the discussion following Table III we found that the rotation error after 360° when rotating counter-clockwise was 0.113 radians or 6.5° which occurred after 270°, so we interpolate to get a $e_9$ of 0.075. In doing this we assume odometry-error is a linear function of the distance traveled by each wheel - which is not unreasonable since slippage and measurement errors are equally likely everywhere (assuming smooth motions).

From the same section in Chapter IV we found that the largest translation error was 1.3 feet at a distance traveled of 24 feet. So the maximum error per foot is $(e_{tft})$ of 0.054. We will assign the maximum allowable displacement error to $e_{max} = 1$ ft. We can now calculate $r_s$

$$r_s = \frac{e_{max}}{(e_{tft} + 1) \tan(e_b)}$$  \hfill (56)

$$r_s = \frac{1}{(0.054+1)\tan(0.075)}$$ \hfill (57)

$$r_s \approx 12.6 	ext{ feet}$$ \hfill (58)

So from a worst-case analysis we find that to be reasonably sure the robot is within 1 foot of where it thinks it is after a maximum rotation of 180° we should acquire a sonar scan and match it within every 12.6 feet.

Now, if the robot is off by one foot and travels another 13 feet, it could conceivably be off by 2 feet. But, if the matcher is able to converge correctly with $\Delta P = 2$ feet, the robot would again know its location to within a maximum of one foot based on our measurements.

**USE OF THE COMPASS**

Since there is a compass on board the PSUBOT we can reduce the heading error before employing the matcher. As was seen from the convergence diagrams, reducing the initial $\Delta \theta$ greatly increases the performance of the matcher.

In addition to reducing the affect of odometry-based heading errors, the compass can be used as a litmus test when determining whether the matcher converged correctly or not. On the other hand, metal near the compass sensor can skew the results, leading to further errors.
COMBINING LOCALIZATION ESTIMATES

Kalman filtering [32] is to be used to fuse the estimates of the robot's posture from various sources (odometry, sonar matcher, compass, etc.). The filter requires an estimate of the average and standard deviation of the errors produced with each method, and combines their estimates in a statistically optimal manner. So, in the short term, odometry will be weighted strongly against the matcher (since it has a much smaller average error) but since the matcher has an average error of zero, it allows the filter to keep the estimate from drifting over the long term.

ASYMPTOTIC COMPLEXITY

One has trouble characterizing the computational complexity of the matching algorithm due to its iterative nature. The number of iterations required is dependent on the rotation error, the noisiness of the data, and the resolution of the solution desired. However, a single iteration of the algorithm is quite well characterized by a complexity analysis as can be seen from time measurements shown at the end of this section.

The computational complexity of the algorithm is dominated asymptotically by the "find-targets" algorithm. The current implementation calculates the distance to each line for each point in order to find the closest one, resulting in a complexity of $O(mn)$ where $m$ is the number of line segments in the model and $n$ is the number of data points.

Calculating the system matrix $A$ has a lower computational complexity of $O(n)$ but has a larger effect for small $m$. For each iteration of the matcher algorithm the system matrix is generated by accumulating the effect of each data point. This results in $32n$ multiplications for the 2-D algorithm and $114n$ for the 3-D solution. For small $m$ this dominates the simpler operation of finding the distance from a point to a line segment $m \cdot n$ times.

The PSUBOT acquires 96 data points $(n)$ during each sonar scan. Although the number of points usually diminishes when calculating the system matrices due to filtering, all the target of
each point must still be determined before outliers may be removed (which occurs at each iteration). Therefore the asymptotic complexity is not affected by the removal of outliers.

Table IX and Figure 103 show how execution time is affected by the number of lines in the model. We would expect to see a linear relationship between the number of lines and the execution time when the number of points is held constant. We would expect a similar result when looking at the number of points while holding the model constant.

<table>
<thead>
<tr>
<th>TABLE IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYMPTOTIC COMPLEXITY WITH $n = 100$</td>
</tr>
</tbody>
</table>

<p>| Non-Parallel Matcher Algorithm | 100 points ($n = 100$) |
| --- | --- | --- |</p>
<table>
<thead>
<tr>
<th>$n$ (# lines)</th>
<th>time(seconds)</th>
<th>seconds per line</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.961</td>
<td>0.1922</td>
</tr>
<tr>
<td>10</td>
<td>1.302</td>
<td>0.1302</td>
</tr>
<tr>
<td>15</td>
<td>1.615</td>
<td>0.10767</td>
</tr>
<tr>
<td>20</td>
<td>1.950</td>
<td>0.0975</td>
</tr>
<tr>
<td>25</td>
<td>2.212</td>
<td>0.08848</td>
</tr>
<tr>
<td>30</td>
<td>2.609</td>
<td>0.08697</td>
</tr>
</tbody>
</table>

The data was generated by generating the appropriate number of random points and lines and performing 10 iterations on them. This process was repeated three times and the execution times were averaged for each Table entry.

Next, Table X presents the data and Figure 104 shows the graphical illustration of the run times with the number of lines set at 10 and the number of points varying from 10 to 200. Again we see a rather nice linear relationship, especially as $n$ becomes large.

Finally we test the matcher on parallel lines. This algorithm is simpler (same order of complexity, but smaller multipliers) so we expect faster execution. The relationship between execution time and the number of points is shown for $m = 2$. This is a natural choice since the parallel solution will be used most often in a hallway (2 lines). Table XI and Figure 105 show the data
Non-Parallel Matcher with 100 points

![Graph showing time complexity vs. model size.]

**Figure 103.** Time complexity vs. model size.

**TABLE X**

**ASYMPTOTIC COMPLEXITY WITH \( m = 10 \)**

<table>
<thead>
<tr>
<th>Non-Parallel Matcher Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 lines ((m = 10))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n (# \text{points}) )</th>
<th>time(seconds)</th>
<th>seconds per point</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>.156</td>
<td>0.0156</td>
</tr>
<tr>
<td>50</td>
<td>.663</td>
<td>0.01326</td>
</tr>
<tr>
<td>100</td>
<td>1.298</td>
<td>0.01298</td>
</tr>
<tr>
<td>150</td>
<td>1.935</td>
<td>0.01290</td>
</tr>
<tr>
<td>200</td>
<td>2.572</td>
<td>0.01286</td>
</tr>
</tbody>
</table>

Numerically and pictorially when the number of *points* is held constant. Table XII and the graph in Figure 106 present the relationship when the number of *lines* is held constant.
Non-Parallel Matcher with 10 lines

![Graph showing time complexity vs. model size.](image)

**Figure 104.** Time complexity vs. model size.

**TABLE XI**

**ASYMPTOTIC COMPLEXITY WITH n = 100 (PARALLEL)**

<table>
<thead>
<tr>
<th>$n$ (# lines)</th>
<th>time(seconds)</th>
<th>seconds per line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5013</td>
<td>0.5013</td>
</tr>
<tr>
<td>2</td>
<td>0.569</td>
<td>0.2845</td>
</tr>
<tr>
<td>3</td>
<td>0.620</td>
<td>0.2067</td>
</tr>
<tr>
<td>4</td>
<td>0.692</td>
<td>0.173</td>
</tr>
</tbody>
</table>

**OVERALL EXECUTION TIME**

As can be seen above, for the number of points used by the PSUBOT ($n = 96$) and for a reasonable number of lines (less than 10), the matcher requires around one second for each iteration. It has been our experience that the matcher seldom requires more than five iterations to converge adequately for reasonable odometry errors. Also, a sonar scan may be acquired in about six or
Parallel Matcher with 100 points

![Graph showing time complexity vs. model size.](112)

Figure 105. Time complexity vs. model size.

**TABLE XII**

**ASYMPTOTIC COMPLEXITY WITH \( m = 2 \) (PARALLEL)**

<table>
<thead>
<tr>
<th>( n ) (# points)</th>
<th>time(seconds)</th>
<th>seconds per point</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.0810</td>
<td>0.00810</td>
</tr>
<tr>
<td>50</td>
<td>0.291</td>
<td>0.00582</td>
</tr>
<tr>
<td>100</td>
<td>0.566</td>
<td>0.00566</td>
</tr>
<tr>
<td>150</td>
<td>0.837</td>
<td>0.00558</td>
</tr>
<tr>
<td>200</td>
<td>1.093</td>
<td>0.00547</td>
</tr>
</tbody>
</table>

seven seconds. So to perform a scan and match it to a line segment model (assuming no other load on the CPU) should require less than 11 or 12 seconds. If the robot were moving at one foot per second while performing the scan (assuming similar processing speeds), an sonar-based matcher estimate of the robot's location could be delivered to the navigator at least every 12 feet using the current processor power. This is in the same range as the desired value as calculated
earlier in this chapter.

However, if the robot is to perform other tasks while executing the matcher algorithm, a more powerful computer would be necessary to maintain the same forward speed. Fortunately, it should not be difficult to find more processing power than what is delivered by the PSUBOT's 20MHz, 80386 computer. An unloaded Sparc 2 performed 100 complete matches of a model consisting of 15 line segments and a sonar scan of 96 points in 15 seconds. That's 0.15 seconds per match which means that we could easily perform one matching session every second with other systems operating.
CHAPTER VII

CONCLUSIONS AND FUTURE WORK

CONCLUSIONS

The purpose of the PSUBOT is to provide autonomous mobility for people who are otherwise unable to control a wheelchair. The target environment is the interior of buildings since outdoor travel would be much more hazardous and complex. As a further simplification, we assume the robot knows the layout of the building beforehand. Modifications to the building which help the robot maneuver easier are not allowed so that new buildings may be incorporated quickly and easily. In other words, we restrict the robot to operate in buildings which have not been modified to its advantage. Hands-free operation is possible through the use of voice recognition.

With the building layout stored in the computer, the PSUBOT may compute the shortest path from one room to another. Several methods of representing and generating paths within a single room were examined. A new, flexible method of representing the path was introduced which uses the notion of a "channel" in which the robot is free to move around unforeseen obstacles.

Odometry is used in most robots to determine their location while navigating. We found that this method works fine in the short term, but in the long term some zero-drift method is necessary to keep positional errors from growing without bound. Walls are assumed fixed, so determining the robot's location with respect to them seemed a logical solution. Several methods of accomplishing this objective were described briefly and the one by Cox was chosen as best-suited for the PSUBOT.

The algorithm matches sonar-based range points to a line-segment model by minimizing the
mean-squared distance between the points and lines. Iteration is required due to a linearization step which decreases the complexity of the underlying equations. If such linearization were not performed, an iteration approach would still be necessary since a gradient search would be required to find the closest minimum of the more complex (and sinusoidal) underlying equations. Noise is removed by the statistical method of filtering out those points which are a fixed multiple of the standard deviation from the mean error. This was found to perform acceptably.

A new minimization algorithm was developed by modifying the equations of the general solution. The result is a correction which is restricted to the direction perpendicular to the parallel lines plus any rotation correction as before.

The chosen localization algorithm was implemented on the PSUBOT. It was tested first by simulating odometry errors and observing the results. We found that the correct location was found with good accuracy except in the most difficult situations, and possible solutions to the latter have been suggested including the use of the compass to decrease rotation errors. Finally a simple pilot algorithm was implemented and given voice commands. In nearly every case the robot reached its destination with the exceptions due to limitations of the motor control and pilot; not the localization algorithm.

FUTURE WORK

Finally we make suggestions for future work. These potential improvements are organized by the chapter in which the topic was presented. But first, I list some of my accomplishments regarding the PSUBOT:

- Acquired schematics of control board and worked with Paul Sherman to design and build the analog interface to the existing motor control circuitry
- Purchased a digital I/O board and developed the software necessary to drive our custom D/A board to drive the motors.
- Worked with Mark Rohwedder on constructing the feedback hardware necessary to read
wheel rotations. Conceptual design of the digital feedback board, and made modifications after it was built to make it work.

- I worked with several EE 406 students over two years. The result was: A sonar device (Robert Gatlin), the algorithm for a digital PID controller (Karl Radestam), system integration and cabling (cables, boxes, tie-wraps, and mounting hardware) by David Underwood.
- A flux-gate compass was purchased and reverse-engineered to produce a pair of analog signals to a commercial A/D interface board.
- Software was written: For wheel control, user input, piloting, navigating, and for “matching points to lines”.
- One local television station featured the PSUBOT on the evening news, and the PSUBOT also appearance in the PSU newspaper; the Vanguard.
- The framework for path planning and obstacle avoidance was developed.
- Numerous C++ classes were developed, including: Point, PointArray, Line, LineArray, Room, Wheels, Sonar and Compass.

Chapter II

The user interface needs to be fully implemented and extended. The current commands are primitive and need to be made more friendly. Though it will be easy to do so, the low-level motion commands need to be implemented and incorporated into the user interface. Additional motion commands will undoubtedly be found useful.

Motion control should be modified so that the trajectory is incorporated into the feedback path. Currently the algorithm only attempts to keep the wheels rotating at their desired speed. Trajectory feedback would allow one wheel to slow down to keep the robot on target. Also, some method of ramping the speed up and down would undoubtedly result in tighter control than the current on/off method.
Additional ultrasonic transducers should be mounted around the perimeter of the robot to detect impending collisions. Currently the only obstacles seen are those around five feet tall.

Software system modifications will reflect the improvements to other areas. Specifically, the World Model should be given the ability to extract only those lines from the room's representation which are "observable" from the robot's location, that is, the lines representing walls which are not obscured by other walls from the current viewing location. This would improve the stability of the matching algorithm.

Chapter III

Path planning currently consists of no more than finding the straight line between the current and goal positions. The first improvement to the Pilot should be the World Model since path generation will be based heavily upon the representation of the room. However, further development of the "wall tracking" algorithm may advance in parallel since, by definition, knowledge of the room is not necessary. After the representation of the room is developed, path generation may be pursued. Throughout these developments should be consultation with the work by Dieudonné on the global path representation and planning [33].

Chapter IV

Vision as a method of localization has been investigated by Espinosa [17]. Of primary concern is the time required to extract lines from images. Until the computation is reduced, the methods developed will not be of much use to the PSUBOT unless a simpler approach is taken.

The Matching algorithm could be improved in the area of noise-reduction. Although our method for removing noise works well for large steps as well as small, it may not be optimal for small corrections. A solution would be to add a second set of iterations which uses Cox's method of removing the points whose distance from their target is greater than some fixed distance. We expect this would improve the accuracy (not convergence) of the overall algorithm. Another possibility would be to decrease the multiplier $F$ as matcher converged, thus eliminating more and
more noise. Which approach to take is not clear and would be decided by testing real data.

Another improvement could reduce the computational complexity of the find-targets algorithm (the one with the highest complexity) by generating a Voronoi diagram representation of the line-segments (which is different from a Voronoi diagram path described in Chapter II, though based on the same mathematics). Such a diagram may be generated in $O(m \log(m))$ time and determining the closest line segment to a point could then be calculated in $O(n \log(m))$ time [38] and [5].

Another improvement to the model of the room would be to change how line-segments meet at corners. Currently both lines making up a corner share a common point. The difficulty arises when a point lies outside the corner. Both line segments are then equidistant from the point so one is arbitrarily chosen (the order of comparison determines the target). An improvement would be to separate the lines slightly at the corners so that the closer line would be assigned to the point as its target.

Finally, if the matching algorithm (or navigator algorithm) was able to detect a "bad convergence" then the standard deviation of the error of the matching algorithm could be reduced, further adding validity to its results.
# REFERENCES


APPENDIX
SOFTWARE OVERVIEW

All the software for the PSUBOT is object oriented. This means that modules are isolated from one another and well-defined interfaces are implemented and enforced. Different objects have little or no knowledge about the inner workings of others objects. The following Figure shows a diagram of the current PSUBOT1 object organization.

![PSUBOT1 Object Diagram]

The software called "PSUBOT1" is multi-tasking in that several objects can share the processor at the same time by using a polling strategy. For example, the PSUBOT1 object repeatedly tells the User and the Pilot objects to work by calling their work() functions. For example, this is the main PSUBOT1 member function:
void PSUBOT1::go() {
    Command command;
    do {
        user.work();
        if (user.commandReady()) {
            command = user.readCommand();
            pilot.command(command);
        }
        pilot.work();
    } while ((pilot.status() != DONE) && (pilot.status() != EXIT));
}

From the listing it is clear that little is known about the specific objects, only what their interfaces consist of.

1. Parameters

Included in the PSUBOT1 system are several utility objects. First, the Parameters object. The Parameters object is initialized with a file name (called the param.dat file since that is usually its name) and is able to retrieve values for different objects from the file. The values may be of the type int, float, char, or string. Most other objects require that a Parameters pointer "Parameters *param" be defined somewhere so that all objects use the same file (though in certain circumstances several Parameters objects may be constructed). The param.dat file looks like this:

matcher multiplier 1.5
matcher maxerrorx 0.2
matcher maxerrory 0.2
matcher maxerrorth 0.1
matcher maxtries 15
navigator safeDistanceAhead 6
navigator roomNumber 1
navigator maxangleerror .75
compass northerlyangle 1.5
forward speed 2
rotate RotateSpeed 1
stop maxspeed 0.01
pilot TooFarOff 3
pilot pauseperiod 5
sonar totalSteps 96
sonar stepdelay 4
sonar maxdist 35
pid kp 0.4
pid ki 10
pid kd 0
wheel minperiod 0.1
wheels PIDForm v
nice filename nices.dat
world filename roomfile.dat

The file is a space-separated list of descriptors and parameters. The first descriptor is-by
convention- the object type. The second descriptor is the parameter type of that object, and
the value is at the end of the line.

For example, when the Navigator is constructed, it asks the Parameters object for the
value the user specified as the default safe distance ahead before pausing for the obstacle.
The declaration would be something like this:

```cpp
float safeDist = param->getFloat("navigator","safeDistanceAhead");
```

2. Nice

Another utility in the PSUBOT system is the Nice object. This is a crude implemen-
tation of a priority arbitrator. Each object, when it is asked to work(), polls the Nice object
to get permission. The Nice object collects the requests and after the correct number of
refusals, is given the go-ahead. So, some objects are allowed to run every time (the PID
object), while some are delayed many times (the navigator) to allow the more time-critical
objects a larger percentage of the processor time.

3. Data Objects

There are several specialized data-objects in use on the PSUBOT. First, the Point
stores an (x,y) pair. Operations such as +, -, *, radius, theta, rotate, <<, >>, etc. are defined
for the point. A Line is a pair of Points, and it knows how to calculate distances between
points and lines or line segments, how to input and output itself, and how to make a grap
file suitable for displaying itself.
The next stage of data abstraction involves arrays of the above two data types. The PointArray and LineArray use similar code for operations on sets of points and lines. For example, it is possible to add a point to a PointArray, to rotate all the points in the array a certain amount around a certain point, to input and output the array, dynamically change sizes, and access members just as if it were a "real" array. For example, if there were a PointArray called scan, printing the x member of each point could be accomplished by:

```cpp
PointArray scan;
/* Read in the scan somehow */
for (int i=0; i < scan.num(); i++) {
    cout << (scan[i]).x << endl;
}
```

Operations are similar for LineArrays.

4. Interface Objects

Objects which interface with the outside (Disk or I/O) include the World object and the Voice object. The World object knows how to create a LineArray from a room file and the room number. The Voice object knows how to call DOS interrupt vectors to read the output of the voice-recognition software. Many of the objects in the next section could also be called Interface Objects, but are more related to specific hardware devices.

5. Device Objects

The device objects consist mainly of the Sonar object, the Compass object, and the Wheel objects. These actually read or change the environment using a hardware device on the PSUBOT. Each of these objects is discussed in more detail in the following sections.
6. Service Objects

The service objects perform tasks necessary for the robot to operate. They may be thought of as the managers or decision makers. The Pilot, for example, is given commands from the User via the PSUBOT1, requests information from the Navigator object, and gives motion commands to the Motion object in an attempt to carry out the user’s desires. The Pilot is basically a big state machine which performs some operation when it is called depending on the state it was last in, on any new commands, and on inputs from the Navigator and Motion objects.

The Navigator object basically takes care of localization (other than odometry) and directly controls the Sonar and Compass objects.

The User object interprets user commands received by the Voice object and places them neatly into a Command object. It then gives the Command object to the PSUBOT1 object which finally gives it to the Pilot which takes action on it. Notice that the PSUBOT1 object does not know anything about specific commands, only how to receive and send a command to the appropriate place.
MOTOR CONTROL AND FEEDBACK

The PSUBOT is controlled by a computer instead of the usual joystick. To accomplish this several modifications were required. First, we had to remove the joystick and replace it with an electronic circuit which would emulate the characteristics of the joystick. This circuit is then controlled by the CPU. Secondly, we needed to be able to measure rotation of the wheels. To accomplish this task, a sprocket and chain were employed to transfer rotation to a pair (one for each wheel) of optical incremental shaft encoders. Basically they are small tin wheels with 100 slits in them, with an optical device which sends pulses as the slits go past the sensors. So, given the number of pulses per revolution of the wheel and the diameter of the wheel the computer can calculate how far the wheelchair has moved. Additionally, given the time elapsed the computer can determine the speed of each wheel, and speed is what the robot tries to "control".

1. Joystick Interface

1.1. Hardware

As described in Chapter II, a custom D/A board drives a push/pull pair for each wheel (as required by the joystick interface). This board is currently very delicate and should probably be rebuilt for robustness and to include some trimmers so that the "null" space of each of the wheel controllers can be adjusted to be equal.

An engineer from Everest and Jennings was kind enough to supply us with the schematics of the motor control circuitry from which we determined the functional requirements of the board. Attempts to "reverse engineer" the motor control unit without the schematics were usually met with disappointment and sometimes with smoke too. A technical report by Paul Sherman describes the circuitry with full schematic diagrams (the same is true for the feedback board).
1.2. Software

The motor control board simply reads in a pair of 8-bit values and converts each of them to the appropriate analog push/pull pair. Since both forward and reverse are needed, the full range is divided into 127 forward and 128 reverse. So, a byte of 128 is stop, 255 if full forward, and 0 is full reverse.

It is important to note that there is a dead or 'null' region around 128 for each wheel, where nothing happens. This is because the voltage must reach a threshold internally on the on-board (original equipment) PWM controller so that the driver relays turn on. Once the relays kick in, the PWM drivers slowly begin ramping up. There was some attempt to reduce the null region in software by causing a jump from 128 to 140 or so rather than 129 when a forward rotation is desired.

The operation of each half of the circuit is straight forward. The digital to analog converter (DAC) has two current outputs: \( I \) and \( \tilde{I} \) which are proportional to the digital value applied to the inputs. These currents are then the inputs to a pair of op-amps which level shift and attenuate them to the correct offsets and ranges. These voltages are then sent directly to where the joystick used to be.

2. Feedback

2.1. Hardware

There are two components to the feedback hardware. First, the optical encoders transform rotation into a pair of square waves whose frequency is proportional to the speed of rotation, and whose phase is directly related to the direction of rotation. Of the two square waves A and B, if A leads B, then rotation is forward. If B leads A then the rotation is reverse. The feedback board consists of a direction indicator (a JK flip-flop) and two 4-bit bidirectional counters with the direction of count automatically determined by the direction indicator. The outputs of the counters go through a 8-bit latch which is connected to
the digital I/O board of the CPU. For the computer to read the values of the counters, it first
clocks the data from the counters into the latches. At the same time (actually a couple of
gate-delays later) the counters are all preset to binary 10000000 or 128. This allows count­
ing forward or backward without rollover. The pulse which clocks the latches and loads
128 into the counters should be as small as possible, since any data coming in on A and B is
lost during that time. This was not found to be a problem due to the long period of A and B
and the ability of the CPU to generate a very short latch pulse.

There are several unused features on the feedback board as it is currently constructed.
First, an overflow/underflow bit was initially designed using a flip-flop. This was later dis­
abled during testing due to a short problem and since reasonable speeds won’t need such
large counts between updates. Also, we initially designed a multiplexer into the circuit so
that a bit could be chosen from the counter and used as an interrupt to the computer. First,
we have not implemented any interrupts, though a dedicated microcontroller would be well
suited for this task. Secondly, since the counters are preset at 128, one count-down pulse
would set the lower seven bits high so that the intent of the original design is lost. The ori­
ginal assumption was that the count would be positive, and starting at 0. Experience man­
dated otherwise.

Finally, noisy and slow transitions from one of the optical encoders caused one of the
counting circuits to count erratically. As a fix, I added a hex Schmitt trigger to each input
to “square them up”. This was an oversight in the original design. Noise in general was
ignored in much of the early work performed on the PSUBOT.

2.2. Software

The program periodically latches and reads the values from both wheel encoders
simultaneously. The values are multiplied by a conversion factor to yield feet. Remember
that the process of latching the data from the board inhibits counting, so the pulse to latch
the data should be as short as possible.
3. Control Software/Odometry

3.1. Wheel Object

The software responsible for controlling the wheels takes the form of an Object. The object is called “Wheel”, and is constructed with a parameter which denotes either the left or right side, as in:

Wheel left(LEFT), right(RIGHT);

where LEFT and RIGHT are enums. Each Wheel object has a PID object which calculates the command (8-bit byte) to send to the control board, and is based on the desired speed, the current speed, and the history of current and desired speeds of previous invocations (the integral term sums up the errors to provide a theoretical zero steady-state error). Member functions of the wheel are:

Wheel(SIDE s); //Constructor
~Wheel(); //Destructor
void work(); //Keep the wheel moving at the correct speed
void setSpeed(Speed s); //Set desired speed
Distance travel(); //How far has the wheel traveled
void resetTravel(); //Start accumulating at 0 now
Speed speed(); //What is the current speed of the wheel
void motorOff(); //Turn the motor off
void markTime(); //Don’t control, just read feedback

The work() member function should be called as often as possible. It maintains some minimum period internally, but if called too seldom (the exact number depends on the parameters of the PID controller) the system goes unstable and oscillates wildly.

So, to use this object to make the left wheel turn at 1 foot per second, the required program would be simply:

```c
#include "wheel.h"
#include "param.h"

Parameters *param;

main() {
```
The Parameters object is described in Appendix A, but basically allows objects to retrieve values from a data file (like "pulses-per-foot" or "wheelbase") with a call like:

```cpp
float pulsesPerFoot = param->getInt("wheel","pulsesperfoot");
```

where "wheel" is the object name and "pulsesperfoot" is the parameter name and the param.dat file has an entry like this:

```
wheel pulsesperfoot 76.5
```

### 3.2. PID Object

The PID object implements both the positional and velocity forms of a PID controller (the param.dat file determines which one is used at runtime). Here is a list of its member functions:

- `PID();` // Constructor
- `PID();` // Destructor
- `float work(Speed &desired, Speed &current, float &T);`
- `void reset();` // Reset any history of past session

As can be seen from the above public member functions, the PID controller has a very simple interface. Other control algorithms could easily be implemented with this interface WITHOUT ANY CHANGE TO ANY OTHER PART OF THE PROGRAM.
THE SONAR SYSTEM

The sonar system consists of an ultrasonic transducer (acts as both emitter and receiver), driver hardware, a stepper motor which spins the sonar device, a stepper motor driver circuit, and a home-detect device.

Following is the software interface of the Sonar System (including stepper motor, etc).

```plaintext
Sonar();
"Sonar();

Distance distance(); //Distance in current direction
Angle angle(); //What is current direction
void setDirection(Angle ang); //In radians
void home(); //Point forward
void step(int steps);//Positive is CCW
PointArray& scan(); //Generate 360 degree scan
int numOfScanPoints(); //Dependent on motor resolution
```

1. Reading Distances

Ultrasonic transducers may be obtained from Polaroid Ultrasonic Group (phone number (617)577-4681) for around $10 apiece. The circuitry to generate pulses and turn time-of-flight into an eight-bit word is described in a report by Robert Gatlin. Kits are also available from a variety of places including Polaroid and also from the Circuit Cellar Kits ($79.00, phone (203) 875-2751, kit# TI01).

We found that the twisted pair from the sonar circuitry to the transducer was very sensitive to noise. For example, when the module was mounted at waist height, just above the power-motor controller, the range of distance measurements was reduced to under 4 feet when the motors were operating. There also seemed to be a connection between sonar accuracy and whether the computer monitor was on or not. So to correct this problem we moved the modules to the top of the stand so the connecting wires could be made shorter.
1.1. Hardware

As mentioned above, all the hardware for the sonar device may be purchased as a kit and were not custom built. The output of the device to the CPU is an 8-bit 7-segment code which must be decoded in software. There is also a "start bit" input and a "finished bit" output.

1.2. Software

The process of generating a sounding involves asking the circuit to send out a "chirp" and then waiting for the echo to be detected as indicated by an echo signal. Following is the simplified algorithm for reading the distance from the sonar.

```
Distance Sonar::distance()
reselTransducer();
transmit_sonar();
int x=0;
do {
    echo = ~(0x20 & inportb(THREE_C));
    if (x++ > 30000) {
        return 999;
        // Tired of waiting.
    }
} while(!echo);
return sonar_port_value();
```

So, the transducer is asked to Transmit, and then the distance may be read once an echo is received.

2. Rotating the Sonar device

The transducer is attached to a vertically mounted stepper motor to allow a 360° scan to be taken. The motor and driver circuitry were purchased as a kit and modified so that the otherwise 12V inputs could be generated with the computer (5V).
2.1. Hardware

At the heart of the motor circuit is a stepper motor controller chip. Another stepper motor controller chip is manufactured by SGS-Thompson Microelectronics, part number TEA3717.

The hardware requires two signals from the CPU: Step, and Direction. Both these signals are level-sensitive. Half-stepping and full-stepping are currently achieved by a switch on the stepper motor controller enclosure. When half-stepping (the mode we use), the motor steps 96 times in one revolution. Full-stepping would yield 48 steps. Note that to change the operation of the entire robot so that 48 data points are used, one must simply throw the switch on the box and edit the param.dat file to reflect the new number of steps per revolution. The matching algorithm, sonar scans, etc. all adapt automatically.

There is a "limit switch" which was intended to let the computer know when the sonar was in the "home" position. It is simply a photo-interrupter switch which is connected to one bit of the digital I/O board. Currently there is nothing connected to the sonar pedestal which passed through the interrupter, but all the software and hardware are in place to make use of it once the arm is constructed and placed.

2.2. Software

The Sonar object also currently controls the stepper motor, since it is seen as a rotating distance measuring device. If a heading is given to the object (point in the direction of 1.57 radians (90°)), the angle is rounded to the closest number of steps, and the motor is commanded "intelligently" to that point. "Intelligently" means that the motor is never allowed to turn more than 360° without rotating back around to keep the cord from wrapping around the post (unlike many animals when tied to a stake). Following are the lines of code necessary to step the motor "step" times in a particular direction:

```c++
void Sonar::step(int steps) { //steps may be positive (CCW) or negative (CW)
```
int direction;
if (steps > 0)
  direction = 0x02;
else
  direction = 0x00;
for (int i = 0; i < abs(steps); i++) {
    outportb(THREE_C,(0x01|direction));
    outportb(THREE_C,(0x00|direction));
    delay(StepDelay);
}