DESIGN AND CONTROL OF A MANIPULATOR FOR AUTONOMOUS JOINING OF FEATURELESS PANELS

By

Andrew Theodore Bouchard

Thesis
Submitted to the Faculty of the Graduate School of Vanderbilt University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Mechanical Engineering

May 2009

Nashville, Tennessee

Approved: Date:
I would like to thank all the people who helped me to complete this thesis, but that is a long list and this is already a long thesis. In particular, I would like to thank my advisor Dr. Julie Adams, who supported my research with her time and ample effort. Without her help I never would have been able to complete it in the time that I did. I would like to thank Dr. Sarkar for his patience as it took me some time to find my place in the graduate program, Dr. Barth, who taught me a great deal as I assisted in his Instrumentation lab, and both of them for taking the time to be on my thesis committee. And a hearty thanks to all the members of the mechanical engineering graduate program at Vanderbilt, students and professors, who taught, supported, and guided me along the last two years.

Thanks as well go to my family and friends, without whose love and friendship I would never have made it this far with my sanity. My mother, who has sacrificed a great deal to help me get a good education and get where I want to be. My siblings, who remind me that there is more to life than engineering and book smarts. Thanks to all my fellow dancers, Olin community members (particularly Mel, whose programming help was invaluable), and Victory A Cappella vocalists - you have helped me more than you know.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. LITERATURE REVIEW</td>
<td>4</td>
</tr>
<tr>
<td>Suction Effectors</td>
<td>4</td>
</tr>
<tr>
<td>Hybrid Automata</td>
<td>5</td>
</tr>
<tr>
<td>Mobile Manipulation</td>
<td>6</td>
</tr>
<tr>
<td>Robotic Construction</td>
<td>9</td>
</tr>
<tr>
<td>III. SYSTEM DESIGN</td>
<td>12</td>
</tr>
<tr>
<td>Mechanical Manipulator Design</td>
<td>12</td>
</tr>
<tr>
<td>Goals and Considerations</td>
<td>12</td>
</tr>
<tr>
<td>Support Beam</td>
<td>13</td>
</tr>
<tr>
<td>Holding Arm</td>
<td>16</td>
</tr>
<tr>
<td>Joining Arm</td>
<td>19</td>
</tr>
<tr>
<td>Full Manipulator</td>
<td>22</td>
</tr>
<tr>
<td>Mobile Platform</td>
<td>23</td>
</tr>
<tr>
<td>Hybrid Controller Design</td>
<td>24</td>
</tr>
<tr>
<td>Panel Identification</td>
<td>25</td>
</tr>
<tr>
<td>Search</td>
<td>26</td>
</tr>
<tr>
<td>Approach</td>
<td>27</td>
</tr>
<tr>
<td>Align</td>
<td>28</td>
</tr>
<tr>
<td>Acquire</td>
<td>29</td>
</tr>
<tr>
<td>Join</td>
<td>30</td>
</tr>
<tr>
<td>IV. SIMULATION PARAMETERS</td>
<td>32</td>
</tr>
<tr>
<td>USARSim Environment</td>
<td>32</td>
</tr>
<tr>
<td>MOAST Controller Framework</td>
<td>33</td>
</tr>
<tr>
<td>Implementation</td>
<td>34</td>
</tr>
<tr>
<td>V. EXPERIMENTAL DESIGN AND RESULTS</td>
<td>38</td>
</tr>
<tr>
<td>Experiment 1: Panel Joining</td>
<td>38</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Experimental Results: Joining Angle Range</td>
<td>41</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A general image of a hybrid automaton [15].</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>Support beam design rendered in SolidWorks®.</td>
<td>14</td>
</tr>
<tr>
<td>3.</td>
<td>Holding arm design rendered in SolidWorks®.</td>
<td>17</td>
</tr>
<tr>
<td>4.</td>
<td>Feature for suction effector mating.</td>
<td>18</td>
</tr>
<tr>
<td>5.</td>
<td>Joining arm design rendered in SolidWorks®.</td>
<td>20</td>
</tr>
<tr>
<td>6.</td>
<td>Manipulator design rendered in SolidWorks®.</td>
<td>22</td>
</tr>
<tr>
<td>7.</td>
<td>Hybrid automaton for autonomously joining featureless panels.</td>
<td>24</td>
</tr>
<tr>
<td>8.</td>
<td>Manipulator design as represented in simulation.</td>
<td>36</td>
</tr>
<tr>
<td>9.</td>
<td>Data from the Approach behavior.</td>
<td>39</td>
</tr>
<tr>
<td>10.</td>
<td>Data from the final configuration of the panels.</td>
<td>40</td>
</tr>
<tr>
<td>11.</td>
<td>Overhead view of a joining operation for two panels at 40 degrees.</td>
<td>42</td>
</tr>
<tr>
<td>12.</td>
<td>Percentage of successful joins and failures for five trials each of multiple approach angles for a 45 degree joint angle.</td>
<td>44</td>
</tr>
<tr>
<td>13.</td>
<td>Percentage of successful joins and failures for five trials each of multiple approach angles for a 90 degree joint angle.</td>
<td>45</td>
</tr>
<tr>
<td>14.</td>
<td>Percentage of successful joins and failures for five trials each of multiple approach angles for a 135 degree joint angle.</td>
<td>46</td>
</tr>
<tr>
<td>15.</td>
<td>Percentage of successful joins and failures for five trials each of multiple approach angles for a 180 degree joint angle.</td>
<td>47</td>
</tr>
<tr>
<td>16.</td>
<td>Percentage of successful joins and failures for five trials each of multiple approach angles for a 225 degree joint angle.</td>
<td>47</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

This thesis presents a design for a mobile robotic manipulator and associated control system to accomplish the task of joining featureless panels for construction. The mechanical design is robust to a wide range of joining angles and construction materials, overcoming the need in existing autonomous construction systems for special markings or other features to support part identification, handling, and placement. The control system is able to find, navigate to, and join panels from a range of starting positions using algorithms flexible to a variety of sensing methods. Relative to existing autonomous construction systems, this design features greater flexibility with fewer requirements on the mobile robot base and associated sensing capabilities.

Recent research in the field of autonomous vehicles has provided robust solutions to many of the basic questions in robotic localization and navigation. The field of mobile robotics is carving out a path that leads to the use of mobile manipulators on robotic platforms to complete tasks flexibly and reliably without being constrained to a traditional stationary working envelope typically found in industrial robot deployments. However, the practical application of systems for mobile manipulation still faces a number of hurdles before the technology can be generally implemented, and most experimental systems remain constrained either by strict limitations on their environment or by special requirements for their targets. The goal of practicality, therefore, demands research focus on ways to generalize these mobile autonomous systems, removing the restrictions to their use.

One growing research application for these mobile manipulators is construction, especially in hazardous environments such as space or extreme terrain on earth. Such
applications are relevant to NASA, which is pursuing the establishment of a permanent presence on the moon by 2020 [30]; to the military, which often must work in hostile territory; and international aid organizations, which also work in unfavorable environments and prefer to focus on the human element rather than constructing supporting structures. While some research has been conducted in autonomous construction using mobile manipulators, the most advanced current technology relies heavily on specialized features as part of the construction materials. These special features include the use of fiducial markings to improve precision in visual-feedback manipulation, pins and cones for self-alignment, and holes or brackets to ease attachment between the construction element and the manipulator. While these allowances have produced solutions with some degree of success, the need for such special features is a limitation to the design. Existing production methods typically create simple extruded shapes such as drywall sheets, I-beams, lumber boards, and sheet metal. The addition of extra features for large production volumes requires either costly modifications to production or expensive post-production machining work. Shifting the system complexity to the robot manipulator enables the system to use standard materials for construction with greater flexibility.

This research contributes a mobile manipulator design for joining panels in robotic construction that lowers the overall requirements on the environment and construction materials, leading to a more robust and general solution to the question of robotic construction. Chapter II presents the current state of mobile manipulation, autonomous construction, and related technologies by examining the existing literature. Chapter III details the system design of a manipulator for generalized joining of featureless panels and an associated controller for the manipulator and mobile robot base. Chapter IV establishes a simulation framework for verification of the design and demonstrates the relevance of the results. Chapter V presents experimental simulations and
results to assess and characterize the functionality of the design. Finally, Chapter VI provides conclusions and future work.
In order to allow a robot to manipulate a target object, it must have an end effector that creates a connection between the part and the robot. The design of these effectors are as varied as the applications in which they are used, and conform to the object being manipulated and the robot itself. Flat, featureless objects present a particular challenge to robotic end effector design because they lack special parts for a traditional gripper to grasp. However, industrial applications and research work have shown vacuum cups to be a proven technology flexible to many applications and materials.

Industrial applications must frequently manipulate flat objects, both in the form of raw material (i.e. sheet metal) and final product (i.e. plywood board, glass panes), most notably in the auto industry. Walker solved both problems for General Motors with a suction end effector for a robot arm [40]. The design uses a plurality of suction cups supplied with vacuum pressure to both transport raw sheet metal stock into place on a die and to remove the molded part after it is formed. A similar device was designed by Pearson for Ford Motor Company to pick and place glass panels [25]. This implementation uses two sets of four compliant vacuum cups to manipulate two glass sheets for placement in irregularly shaped containers.

Suction effectors have also been used in a variety of research applications. A robot designed to aid disabled humans with office work during rehabilitation uses suction cups both to hold books and to turn individual pages [7]. Another end effector design implements suction cups to pick individual tomatoes from clusters [23]. In both
of these instances, suction cups were chosen for their flexibility in accommodating different materials and applications.

**Hybrid Automata**

A hybrid automata is a model of computation that is uniquely able to represent both continuous and discrete state information. In the most straightforward implementation, the discrete state information is encoded as the discrete states of the system, while the continuous state information is described as a differential equation [4]. It is important to note that using a differential equation to describe the evolution of the continuous state ensures that the continuous state of the system remains continuous, which is essential to accurately model any kind of physical system. Figure 1 shows a general model for a hybrid automaton.

![Figure 1: A general image of a hybrid automaton [15].](image)

The discrete states of the system are represented graphically as the states \( q \) and \( q' \), while the evolution of the continuous state \( x \) changes according to a differential equation that depends on both the continuous and discrete state of the system (i.e. \( \dot{x} = f(x, q) \)). In this way, a hybrid automaton can capture both the continuous and piecewise features of a hybrid system.
More formally, Egerstedt and Hu defined a hybrid automaton [16]:

**Hybrid Automaton** A hybrid automaton is considered to be a collection \((Q, X, I, f, E)\) where \(Q\) and \(X\) are sets of discrete and continuous variables respectively. \(I\) is a set of initial states, while \(f\) describes the continuous and \(E\) the discrete evolution of the states.

Egerstedt and Hu showed that hybrid automata can be used for behavioral control of autonomous mobile robots [16]. Their work involved the development of a regularized hybrid automaton to handle simple obstacle avoidance in a mobile robot through the use of attractive and repulsive fields. Under normal circumstances, the robot traveled straight toward a goal point, using a simple algorithm that pointed the robot toward the goal point and drove it with some velocity. Within some range of an obstacle, however, either an obstacle avoidance or a blend of obstacle avoidance and goal attraction were used to calculate a new trajectory, either simply away from the obstacle in the former case, or along the edge of the obstacle’s boundary field in the latter case. Results indicated that, when normalized to avoid rapid switching between states, the hybrid automaton provided good results for robot navigation toward a goal point while avoiding obstacles.

**Mobile Manipulation**

Control of fixed-base robotic manipulators is a well-researched topic, having a long history in industrial domains and other applications. Control of independent robotic vehicles has also been thoroughly explored, and following the Defense Advanced Research Projects Agency (DARPA) Grand Challenge competitions in autonomous vehicles, many consider the problem of autonomous navigation to be largely solved [22]. The field of mobile manipulators naturally extends from the work done in both of these areas, combining the flexibility and manipulation capabilities of a traditional
robot manipulator with the robustness of autonomous vehicles to unconstrained environments. However, the blending of these technologies presents unique and difficult challenges.

Yamamoto and Yun preliminarily addressed the problem of vehicle-manipulator coordination [44]. They considered the dynamics of a mobile manipulator as consisting of a structure with coarse, slow dynamics (the mobile base) and a structure having faster dynamics (the manipulator). In their work, they developed a controller to maintain a driven manipulator in a position of greatest manipulability by moving the mobile base to which the manipulator was mounted. Wang and Kumar used screw theory to decompose the dynamics of the manipulator and the base for velocity control of a mobile manipulator [41]. This approach was verified both in simulation and using a PUMA 260 manipulator on a LabMate platform made to follow a straight line path with some initial offset. Later, Yamamoto and Yun extended their original work to incorporate obstacle avoidance while maintaining an optimal configuration [45]. In simulation, a PUMA-style arm was shown to closely track a desired trajectory while avoiding an obstacle incident on the desired path, simultaneously using a mobile base and its own dynamics to maintain a nearly optimal configuration. Further development provided a unified analysis of mobile manipulation, using a single control algorithm to incorporate both mobility of the mobile platform and manipulability of the mounted manipulator [46].

Petersson, Egerstedt, and Christensen proposed the use of hybrid automata for the control of mobile manipulators [27]. Their approach used a behavior-based architecture to define manipulation primitives, each of which used its own control algorithm according to the structure of the automaton. Using a PUMA560 arm and Nomad XR4000 mobile platform, they demonstrated the approach as successful in holding the end effector still while rejecting disturbances to the mobile platform. Chang, Huang, Fu, et. al. also used a hybrid automaton to control a mobile manipulator,
creating a behavioral hybrid automaton of nine states to control a wheeled mobile manipulator [13]. The resulting controller, while significantly more sophisticated, is similar in structure to the automaton developed in this thesis, having an initial search behavior, an approach behavior, both course and fine movements for positioning, and a separate behavior for executing critical manipulation tasks.

Brock and Khatib further extended work in mobile manipulation, defining an “elastic strip” framework to handle path planning for a mobile manipulator [8]. Their approach generates homotopic paths to the initial robot trajectory and chooses between the paths during execution based on changes in the environment. The controller simultaneously handles task behavior, obstacle avoidance, and posture control, and has the ability to suspend any of these tasks when the environment renders them infeasible.

A natural extension of the existing work on single mobile manipulators is the coordination of multiple mobile manipulators. Adams, Bajcsy, Kosecka et. al. created a four-robot system to cooperatively manipulate a long pipe-like object under human supervisory control [1]. The system was experimentally validated using a pair of observation agents with stereo vision cameras and other sensors in coordination with a two mobile manipulators; one having a PUMA 260 6-axis manipulator and the other having a Zebra Zero 6-axis manipulator, each mounted on a separate TRC mobile platform. The human agent used data from the observation agents to “monitor, advise, and intervene when necessary.” Khatib, Yokoi, Chang, et. al. created a decentralized control system appropriate for multiple autonomous mobile manipulators through the use of force sensing and the augmented object and virtual linkage models [20]. The augmented object model uses additive properties of dynamics to describe the dynamics of the operational point for the system from the dynamics of the individual manipulators. The virtual linkage approach models the forces through an object handled by multiple manipulators as a series of links connecting the grasp
locations. Sugar and Kumar made further progress in the field, developing a system for cooperative transportation of large objects through tight coordination of multiple mobile robots, using compliant manipulators to control grasp forces and account for errors in the platform [34].

Recent work has focused on refinement of mobile manipulators for more general use, particularly generalization to platforms with fewer than four wheels. Christensen and Case have presented a mobile manipulator consisting of a KUKA robot arm on a Segway mobile platform [14]. The combination of the manipulator with a balancing mobile platform was designed to force consideration of mobility and manipulation as an integrated problem rather than a "move then manipulate" design, as has been implemented in this research.

Robotic Construction

The culmination of any robotic technology is its use in solving a practical real-world problem. The concept of applying mobile manipulation to construction tasks is not new, but NASA’s plans to build a lunar base have increased recent interest in autonomous construction [28]. On a larger scale, there are many applications for autonomous construction technologies, including locations rendered hazardous by natural or man-made disaster, extreme terrestrial environments, or other space locations.

Early efforts to use mobile robots for construction tasks simplified the problem by controlling agents to push materials into desired configurations. Wawerla, Sukhatme, and Mataric used Pioneer 2DX robots to push cardboard cubes into a barrier arrangement [42]. Their solution used a behavior-based controller with ultrasound sensors and a laser range-finder for sensing, and they tested both a single-robot solution and two multi-robot solutions. The system performed well and was able to complete the task of pushing together ten blocks into a barrier formation, and also demonstrated
that a multi-robot solution incorporating communication between agents gave the best results. Parker, Zhang, and Kube explored a blind bulldozing approach to collective construction inspired by ants [24].

Extension of autonomous construction to more difficult tasks typically has relied upon more complex sensing abilities in the robots. Simmons, Singh, Hershberger, Ramos, and Smith used a heterogeneous three-robot team to successfully emplace a beam in three dimensions [32]. Their approach requires a 6-DOF crane for gross, high-strength manipulation, a mobile manipulator (skid-steer mobile platform with 5-DOF robotic arm) for dexterity and precision, and a roving eye for relative sensing. The materials and mobile manipulator are all marked with fiducials to allow for easy identification by the stereo vision system on the roving eye. Using a “foreman” agent to coordinate tasks between the robots, the authors showed that the robot team can successfully complete the task. Another system developed by Werfel, Bar-Yam, Rus, and Nagpal employs an ER1 (Evolution Robotics) wheeled base with gripper, RFID reader/writer, and camera to pick up and place blocks with RFID tags in defined configurations [43].

An alternative to greater sensing capabilities is the use of building materials or manipulators with special features to ease construction. One early construction robot developed by Gambao, Balaguer, Barrientos, et. al. used a large hydraulic arm with a specially designed gripper to place blocks for autonomous construction tasks [17]. Terada and Murata have proposed a modular building system using specially designed interlocking blocks with a special robot effector [36]. The Robotic Construction Crew (RCC), developed at NASA’s Jet Propulsion Laboratory (JPL), is an autonomous robotic construction system [33]. The JPL solution consists of a heterogeneous robotic team with two different types of robots, the Sample Return Rover (SRR) and the SRR2K [29]. Both robots feature a four degree-of-freedom robotic arm, stereo camera pair, and a 3-axis force-torque sensor at the base of the gripper.
Construction is accomplished using rigid beams and flexible panels, both of which have a length that is four times the width of the robot. The construction elements each have two grasping points where the robots grasp them, several interlocking cones for alignment and locking, and a set of three fiducials to provide position and orientation information [33]. The robots are controlled using a behavior-based architecture to compensate for uncertainties due to sensor noise and limited processing capabilities. Cooperative manipulation and positioning of parts is accomplished in iterative steps, and the system relies heavily upon relative position information provided by the stereo camera system to correct errors [33].

The existing research demonstrates that much of the foundational work needed for autonomous construction using mobile robots has been under development for many years, and that great progress has been made. Coordination of a flexible manipulator with a mobile platform has been achieved, and computer algorithms exist to handle planning for construction tasks. However, the existing systems for construction using robots are highly specialized, and the need for complicated sensing equipment or custom-fabricated construction materials hampers the realization of this technology in real-world applications. Although significant contributions are still to be made in the foundational aspects of coordination, control, and planning, a great opportunity exists in the development of generalized systems and technology to make robotic construction more accessible.
CHAPTER III

SYSTEM DESIGN

Mechanical Manipulator Design

Goals and Considerations

The intention of the manipulator design is to create a joining manipulator for a construction task that joins two featureless panels. We define featureless as having no special markings, connection points, or other manufactured elements to aid in manipulation. The long term objective of the manipulator is for use on a mobile robotic platform as part of a larger heterogeneous team, including other robots for panel placement and creating the physical connection between the panels once joined (e.g. welding).

The design of the joining manipulator is undertaken with an eye toward practicality and ease of implementation. In order to be widely useful, the manipulator is designed to be relatively easy to incorporate into existing mobile platforms or sets of construction components. Therefore, the following design criteria are established:

- Accommodate a range of joining angles, that is, the angle between the panels at the joint.
- Allow for manipulation of a variety of construction materials.
- Employ standard and easily available power requirements.
- Employ a structure that is independent of the mobile base and does not rely on a particular mobile base to function.
- Support repetition for assembling larger structures.
As a result of these criteria, the resulting design is easily manufacturable, flexible to a variety of applications, and straightforward to implement in a physical system. The mechanical design consists of three major components, which we designate as the support beam, holding arm, and joining arm. Fully detailed mechanical drawings for all parts are included in Appendix B.

Support Beam

The support beam of the manipulator connects at one end to the mobile platform and at the other to the two arms that manipulate the panels. The support beam houses the motors that control the angle of the arms, the encoders that sense the position of the arms, and provides the primary structural support of the manipulator. In addition, the support beam extends the arms away from the body of the mobile platform, allowing the arms to pivot behind the front plane of the beam to accommodate joining angles over 180 degrees without interference.

Figure 2 shows the support beam design from four different viewing perspectives. The major mechanical support is provided by two horizontal aluminum plates, which serve as the top and bottom pieces of the beam. Additional stiffness to these plates is provided by a vertical plate that is attached at the center along the length of the beam with countersunk machine screws on the top and bottom, as shown in Figure III.2(d). Additionally, a vertical motor mount plate, attached in the same manner and shown in Figure III.2(c), provides a mounting surface for the drive motors and gives additional support to the horizontal plates. Approximating the support beam as an I-beam of dimensions given in the CAD package included in Appendix B, the moment of inertia for the beam is given by Equation 3, where $A$ is the thickness and $B$ is the width of the top and bottom horizontal members, and $C$ is the thickness
Figure 2: Support beam design rendered in SolidWorks®.
and $H$ is the height of the vertical support member, all dimensions in inches.

\[
I = \frac{B(H + 2A)^3 - (B - C)H^3}{12} \quad (1)
\]

\[
I = \frac{(6.0 \text{ in})(5.0 \text{ in}) + 2(0.5 \text{ in})^3 - ((6.0 \text{ in}) - (0.375 \text{ in}))(5.0 \text{ in})^3}{12} \quad (2)
\]

\[
I = 49.41 \text{ in}^4 \quad (3)
\]

Using the equation for deflection of a cantilever beam, shown by Equation 6, we can find the vertical deflection at the end of the support beam for a given point load at the end. For this equation, $W$ is the load in pounds, $l$ is the length of the beam in inches, $E$ is the Young’s modulus for the beam material in PSI, and $I$ is the beam moment of inertia in inches$^4$. Again using the dimensions from the Appendix B and assuming a construction of aluminum alloy with $E \approx 10^7$ PSI,

\[
d = \frac{Wl^3}{2EI} \quad (4)
\]

\[
d = \frac{(2000 \text{ lbs})(13.5 \text{ in})^3}{2(10^7 \text{ psi})(49.41 \text{ in}^4)} \quad (5)
\]

\[
d = 0.005 \text{ in} \quad (6)
\]

Therefore, for a load of one ton, the support beam would only deflect 0.005 inches. This is far more than the anticipated load on the end of the support beam, both because the manipulator drags the panels rather than lifting them and because such a load can not be balanced by the weight of the mobile platform. Therefore, the weight of the mobile platform will be a limiting factor long before the stiffness of the support beam, indicating that the beam design provides more than sufficient strength to support the manipulator.

Two motors for rotating the manipulator arms are housed in the support beam. They are face mounted to the motor plate with socket head cap screws such that the shafts pass through the clearance holes. The motors each drive their respective arm
pivot via a pair of conical gears at a right angle. Depending on motor selection, a gear hub may be necessary to mount the gear to the motor shaft. The gearmotors specified in the drawings in Appendix B have a 5/6” diameter shaft, while the conical gears have a 0.5” diameter bore, therefore an adapter is required and included in the drawing package. These gearmotors were chosen for small package size which will easily fit inside the envelope of the support beam while providing ample torque to pivot the unloaded arms (23 lb-in) at a slow and controllable speed (10 rpm). Ball bearings are press fit at the end of the support beam for mounting of the two arms and to facilitate rotation. The particular ball bearings shown in the design were chosen because they have a bore sufficient for the pivot shaft without additional machining and because they feature a flange for additional axial support. However, these bearings can easily be replaced by other similar bearings with small modifications to the design. The final feature of the support beam is a pair of encoders mounted atop the arm shafts with machine screws. The encoders provide the angular position of the arms to the system. The particular encoders included in the design were chosen because they have a large bore to accommodate the arm shaft with no need for extra machining, and feature a wide array of customization options to fit variations in connectors, mounting, supply voltage, and other parameters for interfacing with the mobile platform and controller. Resolutions of 10-10,000 counts per revolution are available, and 2000 counts per revolution was chosen because it is the maximum resolution available without extra cost and because it should be more than sufficient for this application.

**Holding Arm**

The first arm, shown in Figure 3 from four viewing perspectives, simply holds the right panel in place while the joining arm pulls the left panel toward the joint. The decision to hold the right panel in place was made in order to accommodate larger assemblies in which pulling the assembled wall in addition to the single new panel is
impossible. Note also that the decision to pull the left panel and hold the right still assumes a right-to-left assembly order. While this increases the utility of the system for later expansion to join more than two panels, it decreases the flexibility of the system for each individual joint, as the left panel is the only one that can be moved for alignment. Therefore, the left panel must be placed with excellent alignment relative to the right panel for successful joining.

![Figure 3: Holding arm design rendered in SolidWorks®.](image)

The shaft of the holding arm is the subassembly’s connection point to the support beam and is press fit into the ball bearings with the long end extending above the top beam to attach to the encoder. The main body of the holding arm consists of an aluminum plate with two circular features at the rotation axis, creating a hinge about the shaft, most clearly seen in Figure III.3(d). A driven conical gear, which
receives power from the motor’s conical gear, is attached to the shaft just above the bottom hinge feature, as shown in Figure III.3(c).

![Figure 4: Feature for suction effector mating.](image)

Four suction effectors provide the holding surface for the manipulator and are mounted to the front of the main body via the features shown in Figure 4. A shallow recess provides a mating area for the suction effector base and constrains rotation, while a circular through hole allows passage of a vacuum supply. The vacuum supply system is not shown for simplicity. The suction cups in Figure 3 are Anver VC-series suction cups, but can easily be replaced by other commercially available suction cups with appropriate modifications for mounting. The Anver VC-series were chosen because the larger cup diameter provides greater grasping surface, the nitrile cup is flexible for improved grasping, the aluminum hub gives a stable mounting surface, and a large inlet port can accommodate larger vacuum supply systems to handle heavy and even porous materials. In addition, the aluminum hubs are designed with flats on either side corresponding to the recess in the design to prevent rotation of the held material.
In order to minimize the weight of the arm and limit the forces required to actuate it, the design of the main body of the arm is trimmed around the necessary features, resulting in an H-shape to support the four suction effectors. The final design presented in Appendix B weighs 5.76 pounds when made of 6061 aluminum alloy. Adding 0.10 pounds each for four suction cups and 1.77 pounds for the arm pivot shaft connecting the assembly to the support beam, the entire right arms weighs about 8 pounds, which is easily supported and actuated by the support beam.

**Joining Arm**

The joining arm of the manipulator is responsible for pulling the left panel to the right such that the two panels create a continuous joint. This design accomplishes the task by using two major components: a hinge to control the angle of the arm similar to the design for the holding arm, and a sliding suction effector to provide prismatic motion along the line of the arm.

The design of the joining arm is shown in Figure 5 from four viewing perspectives. The hinge of the joining arm is designed to mirror that of the holding arm, and functions in the same manner. The face of this part does not contain any elements to interact with the panel. Since this part does not move along the line of the arm, such features will interfere with the desired motion. One bracket and two pillow blocks are mounted to the rear side of the arm and support the prismatic motion of the arm. Each of the pillow blocks support and constrain the motion of a single guide rail along the line of the arm. The specific pillow blocks chosen for this design can support a dynamic load of 230 pounds, which is more than sufficient to support the sliding effector. The bracket attaches to a pneumatic cylinder, which provides the prismatic motion to actuate the sliding suction effector to move the panel. This pneumatic cylinder must provide sufficient force to drag the left panel into the right; therefore, the cylinder chosen for this design can pull with 200 pounds of force. The
Figure 5: Joining arm design rendered in SolidWorks®.
force limitation of the actuator is probably the most significant mechanical limitation of the manipulator, but should be sufficient to drag medium-sized panels over firm terrain. For larger payloads, the entire manipulator could be scaled to fit a larger actuator.

The sliding effector design mirrors the effector face of the holding arm, having four vacuum suction cups in the same configuration with the same mounting features. In addition, excess material is again limited using the H-shape design around the four suction mounting features. As on the hinge portion of the joining arm, one bracket and two pillow blocks are mounted, connecting to the opposite ends of the corresponding guide rails and pneumatic actuator. The sliding effector moves along the guide rails driven by the pneumatic cylinder to extend or retract along the line of the arm. Detail in Figure III.5(c) shows a side view of the pneumatic cylinder and bracket as well as the conical gear to rotate the arm, and details in Figure III.5(d) show the two ends of the pneumatic cylinder with their attachment brackets. The overall weight of the sliding effector and the suction cups is less than five pounds, and to incorporate the weight of the guide rails and bearing blocks we estimate an end load on the guide rails of 20 pounds. The guide rails are made of precision steel rods with Young’s modulus approximately $3 \times 10^7$ PSI. Using the area moment of inertia for a cylinder in Equation 9, for a diameter $d$ of 0.5 inches,

$$I = \frac{\pi}{64} d^4$$

$$I = \frac{\pi}{64} (0.5 \text{ in})^4$$

$$I = 0.003 \text{ in}^4$$

Using an end load of 20 pounds and a maximum extension of 12 inches, we use the
beam bending equation in Equation 12.

\[
d = \frac{Wl^3}{2EI}
\]

(10)

\[
d = \frac{(20 \text{ lbs})(12 \text{ in})^3}{2(3 \times 10^7 \text{ psi})(0.003 \text{ in}^4)}
\]

(11)

\[
d = 0.192 \text{ in}
\]

(12)

This calculation indicates that, at full extension, one of the guide rails at 12 inches extension will bend less than one-quarter of an inch. With two guide rails and a linear actuator, the actual bending will be less.

**Full Manipulator**

![Manipulator Design](image)

Figure 6: Manipulator design rendered in SolidWorks®.
The full design of the manipulator is shown in Figure 6 with all subassemblies included. The most difficult physical challenge for the manipulator according to its current design is acquiring the panels without causing them to fall over. The design assumes that the panels are already in position to be joined, meaning that some other component of a larger construction system will have handled panel placement and alignment. With this in mind, the clearest solution seems to be that the robots that place and align the panels continue to hold them in place until the joining robot has acquired them. Another option is to turn the vacuum supply on to the suction cups for some time prior to when the robot reaches the panels, such that if the distance estimate is incorrect, the robot simply acquires the panel sooner than anticipated. In both of these cases, the only remaining case under which the manipulator can fail to grasp the panels is due to a misalignment in the arms being sufficiently large that the suction cups cannot seal against the panels. The best way to protect against a misalignment error is to use more sophisticated sensing techniques to identify the panel when both endpoints are not visible, as is often the case when the laser rangefinder is very close to the joint. However, this situation did not present itself as a significant problem during simulation-based testing.

**Mobile Platform**

The system was designed to be easily generalizable between different mobile platforms. In the simulation implementation, the robot base is an iRobot ATRV Jr.; however, any mobile base meeting the following requirements can be used with the designed manipulator and control system.

Obviously, the platform must be sufficiently massive to stably support the manipulator. In addition, the mobile base must be able to supply the power requirements of the manipulator, which includes electric power for the motors controlling arm rotation and pneumatic power for both the vacuum effectors and the linear actuator.
In order to identify the panels and perceive the surrounding environment, the mobile platform must have some sensing hardware. The current controller is designed to use a laser range-finder and pose sensors, both of which are common to autonomous vehicles. However, because the panel detection algorithm is implemented as a function that returns the panel endpoints, the laser range finder can be replaced by a camera or other imaging system assuming that the panel detection algorithm is replaced to suit the new sensing method.

**Hybrid Controller Design**

The controller scheme used for this research is a behavioral hybrid automaton with five states to handle five different subtasks. In addition, the states share a common memory of the panel positions that can be used by the last three behaviors if they are too close to identify the panels using the available sensors. The full system is shown in Figure 7.

![Figure 7: Hybrid automaton for autonomously joining featureless panels.](image-url)
Panel Identification

The robot identifies the panels by executing a simple Hough transform on the laser range-finder data, which returns the equation of the most prominent straight line in $p$-$\theta$ notation, where $p$ is the distance from the origin to the line along its perpendicular and $\theta$ is the angle that the line makes with the positive $x$-axis. The algorithm works by calculating the $p$-$\theta$ coordinates for all lines through each data point at some angular resolution ($1^\circ$ for $\theta$ in this case) and rounded to some resolution for $p$ (0.02 m in this case). Each of these lines are entered into a voting array indexed by the $p$-$\theta$ coordinates. The entry with the highest number of votes indicates the line shared by the most number of points. The points on the line are identified as those having an error of less than the distance resolution of the transform from the line equation.

The system then identifies the continuous portions of the line by sorting the data points and comparing the distance of adjacent gaps. A gap that is more than 50% larger than the one next to it is considered a break in the line. Finally, the average and standard deviation of the distance of points from the identified line is taken in the direction perpendicular to the line, and points falling outside two standard deviations from the mean are excluded. This final step removes points on the edges of panels, leaving only the data from the panel face. The endpoints are identified and the length of the line is compared to the known width of the panels. If the line length falls within 20% of the panel width, the line is assumed to represent a panel, and the endpoints are returned.

The above procedure is executed repeatedly, each time removing points from the data identified as not on a panel or on a panel already identified, until two panels are found. Using the yaw of the vehicle to establish the relative angles of each of the panels to the vehicle, the panels are identified as right and left, then the endpoints for each are also identified right and left. The panels are identified by left and right directions for two key algorithmic reasons. The first is that the left endpoint of the
right panel is designated as the joint location because the right panel does not move during joining. The second is because identification by right and left endpoints allows for a consistent definition of the panel angles relative to the joint. The lines describing the panels can be seen as having an angle of $\theta$ with respect to the x-axis or $\theta + \pi$. Defining the lines as going from the end toward the joint to the one away results in panel angles that give an accurate measure of the angle between the panels. The panel angles are described as

$$\theta_L = \tan^{-1} \left( \frac{y_{LL} - y_{LR}}{x_{LL} - x_{LR}} \right)$$

$$\theta_R = \tan^{-1} \left( \frac{y_{RR} - y_{RL}}{x_{RR} - x_{RL}} \right)$$

where $\theta_L$ and $\theta_R$ are the angles of the left and right panels, respectively, $(x_{LL}, y_{LL})$ is the left endpoint of the left panel, $(x_{LR}, y_{LR})$ is the right endpoint of the left panel, $(x_{RL}, y_{RL})$ is the left endpoint of the right panel, and $(x_{RR}, y_{RR})$ is the right endpoint of the right panel. The panel endpoints and panel angles are stored in memory each time they are successfully identified, and are accessible by any behavior.

This approach to panel identification assumes linear panels of a known width, and therefore will not work with curved or irregularly shaped panels. However, similar pattern-matching algorithms abound for visual data processing that can easily replace the Hough transform method to accommodate other building materials. In addition, because any two points can determine the line, this implementation of the algorithm establishes a minimum threshold of five points to provide sufficiently accurate data to identify a panel. Lines with fewer points are discarded regardless of length.

**Search**

The automaton Search behavior (Figure 7) assumes that the panels are within range of the robot’s laser range-finder sensor, and assumes that they are the first two
panels immediately identifiable by the robot. Therefore, the robot spins in place in order to locate the intended joint and exits this state when two adjacent panels are identified. In order to avoid potential timing issues with taking laser range-finder data while rotating, the current implementation spins the vehicle in 45-degree increments beginning with the initial vehicle yaw, stopping to check if the two panels are visible at each position. Since the laser range-finder has a 180-degree field of view, every point within range of the sensor is checked approximately four times, which has proven an effective solution to consistently identify panels at various joining angles.

**Approach**

The search behavior leaves the robot in an orientation from which the two panels to be joined are visible. However, in order to join the panels, the robot must be in a position with the intersection of the arms at the desired joint location and in such an orientation that both arms can attach to their respective panel. The Approach behavior (Figure 7) detects the panels again and calculates the bisector by averaging the two panel angles as described in Equation 14. The behavior then identifies an approach position on a line parallel to the bisector and passing through the intended joint location, that is, the left endpoint of the right panel. This approach position is defined as the point 1.0 meters from the joint location along the line and closest to the vehicle. The algorithm finds both points along the line and 1.0 meters from the joint location, then calculates the distance from each to the estimated position of the vehicle and sets the closest as the target location.

Before moving to the location, the robot opens both arms of the manipulator so that they do not protrude and potentially collide with the panels prematurely. Using the obstacle avoidance and point-to-point navigation included in the autonomous mobility echelon of the MOAST (Mobility Open Architecture Simulation and Tools) controller [5], the robot moves to this approach location. The path was found to be
more accurate when the robot traced a straight-line path from its current location to the target, so the vehicle initially rotates to a heading along the line from the vehicle to the approach location before using the MOAST navigation. In addition, instead of navigating directly to the target, the controller navigates to a point along the line from the vehicle to the target and 2.0 meters from its current location, then recalculates its approach. This replanning was found to greatly improve robot placement at the target for two reasons. First, approaching in stages allows the robot to periodically compensate for drift in the navigation by checking against sensed data. More significantly, this approach scheme also gathers higher-resolution data to describe the panels as the robot approaches because the panels occupy a greater portion of the sensor’s field of view.

**Align**

Before executing the Align behavior (Figure 7), the implementation of the controller extends the prismatic link of the joining arm so that it can later be retracted to pull the panel into a joined configuration. The joining arm extension is executed here because, due to artifacts in the simulation, extending the simulated prismatic link corresponding to the pneumatic piston often causes the robot to move slightly. Introducing such an error at this point means that it is corrected automatically in the Align and Acquire behaviors without the need for additional controller compensation. While the particular behavior seen in the simulation is not physically possible, it is possible to imagine that the motion of a mass the size of the joining arm can have inertial effects that disturb the position of the vehicle. Therefore, it is valuable in both simulation and reality to extend the prismatic joint at this point in the execution of the controller.

It is likely that the Approach behavior will place the robot in the correct position,
but with an orientation that is not conducive to joining the panels. The Align behavior detects the panels, this time with greater resolution from a closer perspective, and orients the robot such that its central axis is pointed toward the intended joint location. This alignment centers the robot on the joint, giving an equal angular offset from each arm to its respective panel, making optimal use of the angular range of the manipulator arms.

The laser range-finder’s field of view is limited to 180 degrees and has a finite angular resolution; therefore, it is possible that the sensor’s perspective will leave it unable to detect the panels. This is particularly true in cases in which the joining angle is greater than 180 degrees. The system attempts to recover from an inability to detect the panels by using the panel information stored in memory from the last successful detection in the Align and all subsequent behaviors.

Acquire

In order to pull the panels together, the Acquire behavior (Figure 7) moves the manipulator close enough to the intersection that the suction effectors can effectively grasp the panels and pull them together. Knowing the robot’s orientation and the manipulator’s physical parameters, the line describing the position of an arm can be described by the angle \( \theta \) passing through the intersection of the two arms \((x, y)\),

\[
\begin{align*}
    x &= l \cos \psi + x_{veh} \\
    y &= l \sin \psi + y_{veh} \\
    \theta &= \psi + \phi
\end{align*}
\]

where \((x_{veh}, y_{veh})\) is the location of the vehicle center, \(\psi\) is the vehicle yaw, \(l\) is the distance from the vehicle center to the intersection of the joining arms, and \(\phi\) is the angle of the arm with respect to the vehicle. Using this estimate, the arms are opened.
using a simple proportional controller with \( \dot{\phi} = 0.5(\theta - \theta_{panel}) \) until they are parallel to their corresponding panel.

Actual panel acquisition is achieved via a control loop that iteratively handles corrections to vehicle alignment and arm alignment while driving the vehicle forward. In each case, a proportional controller was found to be sufficient for reliable success. Therefore, in each iteration of the control loop, the following control laws are executed, in order:

\[
\text{Vehicle: } \dot{\psi} = 0.5(\psi - \text{atan2}(y_{int} - y_{veh}, x_{int} - x_{veh})) \tag{18}
\]

\[
\text{Arms: } \dot{\phi} = 0.5(\theta - \theta_{panel}) \tag{19}
\]

\[
\text{Drive: } v = 0.25\sqrt{(x_{int} - x)^2 + (y_{int} - y)^2} \tag{20}
\]

where \((x_{int}, y_{int})\) is the intersection point of the panels. The tolerances on each control are 0.005 radians, 0.01 radians, and half the initial distance from the panel intersection to the manipulator arms’ intersection for that iteration, respectively. Therefore, the control loop causes the robot to take smaller and smaller movements closer and closer to the panels, providing accuracy at a cost of time. The control loop exits when both of the manipulator arms are close enough to grasp the panels, 2 cm in this case. The adjustments to the control loop are executed serially because of limitations in the MOAST framework, which are discussed in Chapter IV.

**Join**

The Join behavior (Figure 7) initiates a suction event, causing the effector arms to “grasp” the panels. Currently, there is no means of confirming that the panels have been acquired, because it is guaranteed in simulation. However, in a real-world system implementation, a successful grasping of the panels can be confirmed before completing the join. Options to accurately sense panel connection include using a
touch sensor to establish contact or sensing the change in pressure that occurs as a
result of the vacuum effectors sealing against a flat surface.

Once the manipulator has acquired both panels, the prismatic joint retracts,
pulling the left panel with it into the right panel and creating the joint. Complet-
tion of the motion to pull one panel into the other can be sensed by a change in
the level of force required to retract the pneumatic cylinder, indicating a collision
between the panels. Another solution uses the laser rangefinder to determine the size
of the gap between the panels, because a successful join can be defined as the point
where there is no gap. After this stop condition is reached, the system returns the
appropriate success or failure code and exits the automaton.
CHAPTER IV

SIMULATION PARAMETERS

USARSim Environment

Due to cost and time considerations, the verification of the system was completed in simulation. The Unified System for Automation and Robotics Simulation (USARSim) [31] was chosen for this project for its ease of use, rigorous verification against reality, and flexibility. This package is an open source project [39] created to test robotic algorithms without having to build expensive and complex physical hardware. The result is a simulation engine built on the powerful Unreal Tournament 3D [38] gaming graphic and physics engine and verified with real-world experiments.

Researchers used the Unreal Tournament 2004 game engine to model the environment, models, and physics between objects because it provides high fidelity at relatively low cost. Unreal Tournament uses a proprietary communication protocol, thus USARSim relies on the Gamebots [2] [18] modification to handle communication between the Unreal engine and other applications. When run, the Unreal engine creates a server that simulates the world and handles all graphics and physics interactions. Individual instances of robots are spawned by clients that connect to the server to create and control instances of individual robots.

USARSim supports several controller architectures, most notably Player [19] [37] and MOAST [5]. In addition, it incorporates several maps and a variety of robot models, including a Pioneer 2-DX, ATRVJr, Humvee, blimp, submarine, and Talon. USARSim also implements a number of common sensors, including inertial navigation units, laser range-finding devices, and cameras [12]. Additional sensors, mission packages, effectors, and robots can be created using 3D modeling programs such as Blender [6] along with some UnrealScript, the code native to Unreal Tournament.
Verification of USARSim accuracy to reality has been verified at several levels by repeated evaluations. Several research groups have verified the accuracy of simulated platform navigation and motion over various surfaces and obstacles for multiple robot models [11] [21] [26] [35] [47]. Visual sensing using the simulated camera and laser range-finder has also demonstrated the accuracy of sensor models, including realistic noise models [9] [10].

**MOAST Controller Framework**

In recent years, a number of controller architectures have been proposed to simultaneously handle the wide array of subtasks needed to successfully drive an autonomous vehicle. This particular research builds the MOAST environment [5]. This implementation uses a hierarchy and concurrency to subdivide the tasks needed for successful robotic navigation and manipulation into separate processes according to scope and data resolution.

MOAST utilizes the 4D/RCS reference model architecture developed by Albus [3] specifically for the control of unmanned ground vehicles. The system is divided into five levels: the servo level, primitive level, subsystem level, vehicle level, and section level. These levels run as concurrent, independent processes, and each implements its own sensor processing, world modeling, behavior generation, and planning related to an increasing scope, decreasing update frequency, and decreasing sensor resolution. Plans generated by the higher echelons of the controller are passed down to lower levels as messages over shared buffers for translation into actions leading to the final emergent behavior.

The MOAST system is an open source project, and for the purposes of this research, only the first three levels of the architecture were used. However, MOAST features automatic obstacle avoidance, mapping, and sensor blending. In addition, it supports a wide array of vehicle types, from skid-steer robots to Ackermann-steered
vehicles to submarines and blimps. While flexible to a number of standard robotic simulators, MOAST was developed to integrate specifically with USARSim.

**Implementation**

A number of additions were made to the USARSim environment in order to support the requirements for this research; however, none of these additions affect the accuracy of the simulation or the validity of the results produced. These changes included the addition of new classes of objects to the simulation environment, the modification of maps, and changes to the mobile platform. All USARSim modifications can be found in Appendix C.

In order to simulate the action of the vacuum end effectors, two new classes were added that simulate the desired result by creating a virtual link between the manipulator and the target. A new SuctionEffecter class was defined, inheriting from the USARSim Effecter class, and the SuctionEffecter type was added to the list of effecters in USARSim. Effecters in USARSim are subsystems that directly affect or alter the environment, and are assigned six opcodes to govern their behavior: Activate, Animate, Fire, Release, Reset, and NOP. The Fire and Release opcodes were defined for the SuctionEffecter to find the nearest object of type Cargo and call its Grabbed method. Similarly, a new Cargo class inheriting from the Unreal Actor class was created with a Grabbed method. This method sets the base of the Cargo object to that of the Actor that called it, essentially creating a virtual link between the two objects, holding them at the same relative position to each other as when the function was called.

For all testing simulations in this research, the DM_Tutorial map included with USARSim was used with some modification. Specifically, two Actors of type USARContainer (a class inheriting from Cargo) were added to the environment. These Actors were assigned static meshes consistent with the definition of a featureless panel
established for this research. The static mesh was created using Blender [6] and exported as a type ASE static mesh using the goofos Python script. These panels were positioned at the desired joining angles relative to one another using UnrealEd, the map editor included with Unreal Tournament. Additionally, player start positions were added as spawning points for the robot at points along an arc 6.4 meters (1600 Unreal Units) from the joint location. Start locations were added at 15° intervals to both sides of the joint bisector up to 75°.

In order to construct the manipulator, the iRobot ATRV Jr. mobile platform included with USARSim was used as a base. For simplicity, all sonar sensors were removed, as well as the camera and associated pan-tilt mission package included for moving the camera. The manipulator parts were modeled using Blender and exported using the goofos script as was done for the panels, with the exception of the prismatic link. The collision model in Unreal does not allow static meshes to overlap or to be concave, therefore to maintain both aesthetics and proper motion, the static mesh can not match the visual model. Blender was unable to create a static mesh independent of the visual model, so the model for a prismatic link from the TeleMax, another USARSim robot, was added as a link to the mission package. Additionally, Blender does not support the use of dimensioning in such a way that it accurately translates to the simulation environment. Therefore, all static meshes are representational models and not accurate for dimensioning. The end result is shown in Figure 8 from four different perspectives, and all supporting Unreal/USARSim files are included in Appendix C.

In order to avoid the need for changing compiler files, the existing subsystem control shell program (amShell) was modified to add the functionality needed for the controller described in Chapter III. In order to support the functionality of this system, a number of additions were made to the control shell that slightly modified the structure of the MOAST hierarchy.
Figure 8: Manipulator design as represented in simulation.
First, it was found that the autonomous waypoint navigation set the wheel speed to zero once the target was reached within some specified error. However, instead of stopping the vehicle, the wheels were instead left in a neutral position and the vehicle continued to roll for a short distance. The finite state machine controlling the speed of the wheels was altered to move to a state with a very small negative wheel speed before transitioning to zero in order to remedy this problem. This was found to be effective means when stopping for low travel speeds.

In order to access all data and control of the robot without modifying multiple levels of the MOAST hierarchy, direct control was given to the subsystem control shell of both the servo and primitive control level by the addition of RCS control buffers to each. Access to the primitive level allowed acquisition of raw laser range-finder data for panel identification. Control of the mobility and mission package servo buffers was added in order to allow for proportional control of wheel speed without a target waypoint in the Approach behavior and because the subsystem echelon does not at all implement control of mission packages. The addition of these RCS buffers required changes to the buffer definition files in the MOAST source code, which are included as a part of Appendix D.

The remaining modifications to the code affected only the amShell program, and include the addition of functions implementing and supporting the hybrid automaton design described in Chapter III. The code added to amShell.cc is included in Appendix D.
Experimental validation of the manipulator design was carried out through a series of three experiments in order to determine its capabilities and robustness. First, a panel joining operation was completed for a pair of panels in a specified configuration. Next, the ability of the manipulator to join panels at multiple angles was tested using pairs of panels in different initial configurations. Finally, the ability of the system to identify the panels as well as accurately approach and join the panels was tested from various approach angles for different initial panel positions.

**Experiment 1: Panel Joining**

Initial testing was conducted using a 90 degree angle between the panels. This was chosen since it is in the middle of the manipulator arm’s capable range and was anticipated to be the most common joining angle required.

Figure 9 is an example of the results from the Approach behavior for a single trial. The dots indicate laser-rangefinder data. The lines are calculated by the panel identification algorithm, indicating that the line detection algorithm successfully identified the panels. Endpoints of the panels are identified by larger marks, and the vehicle position is indicated using the ‘x’ mark. This plot indicates that the controller correctly identified both panels and their endpoints, calculated the approach location (shown by the circle not on a panel) based on the bisector, and drove the vehicle to it.

After aligning the robot body to the joint and the manipulator arms to the panels, the robot approaches the joint in increasingly small movements for greater accuracy. While this was done at a cost of time, experiments indicate that the additional time
Figure 9: Data from the Approach behavior, indicating successful identification of both panels, calculation of the approach point, and accurate navigation to the desired position.

is not prohibitive. Average simulation time was on the order of seven minutes for the simple controller with the limitations of MOAST. A real controller system designed for practical applications will likely be able to improve on this time significantly.

Figure 10 shows an example of the laser range-finder data following the successful execution of the Join behavior. A successful join was defined as having no detectable gap between the panels, which can be seen as a break in either line representing the panels. Ten trials at an angle between the panels of 90 degrees show a successful join rate of 100%, indicating that the system works as anticipated.

**Experiment 2: Joining Angle Range**

While the initial evaluation verified the ability to join panels when they are positioned at a 90 degree angle to each other, it is necessary to evaluate the manipulator
for varying panel angles. An objective was to determine the minimum and maximum panel angles that the manipulator can join successfully. Evaluations were conducted in the same manner as for the 90 degree case for joining angles between 45 degrees and 225 degrees at 45 degree intervals. Additional configurations beyond these limits were attempted in order to determine the maximum operating range for the manipulator. Ten trials were evaluated for each configuration, excluding trials in which USARSim did not generate the robot in a working configuration, a known problem with USARSim.

Table 1 provides the experimental results from the simulation. The Angle represents the initial angle made by the two panels with respect to each other. For each panel angle, we present the Success percentage over 10 trials, the mean ($\bar{X}$) and standard deviation ($\sigma$) of the Time in minutes to execute a trial, and the mean and standard deviation of the final Alignment Error in degrees. This alignment error is
Table 1: Experimental Results: Joining Angle Range

<table>
<thead>
<tr>
<th>Angle</th>
<th>Success</th>
<th>Time (min.)</th>
<th>Alignment Error (°)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\bar{X}$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td>40</td>
<td>70%</td>
<td>4.96</td>
<td>0.48</td>
</tr>
<tr>
<td>45</td>
<td>80%</td>
<td>6.92</td>
<td>0.45</td>
</tr>
<tr>
<td>90</td>
<td>100%</td>
<td>6.95</td>
<td>0.50</td>
</tr>
<tr>
<td>135</td>
<td>100%</td>
<td>6.17</td>
<td>0.34</td>
</tr>
<tr>
<td>180</td>
<td>100%</td>
<td>6.11</td>
<td>0.46</td>
</tr>
<tr>
<td>225</td>
<td>90%</td>
<td>7.01</td>
<td>0.22</td>
</tr>
<tr>
<td>235</td>
<td>80%</td>
<td>6.50</td>
<td>1.06</td>
</tr>
<tr>
<td>240</td>
<td>40%</td>
<td>5.66</td>
<td>0.41</td>
</tr>
<tr>
<td>250</td>
<td>40%</td>
<td>6.14</td>
<td>0.26</td>
</tr>
</tbody>
</table>

calculated as the difference between the vehicle yaw and the joint bisector. These results demonstrate that the manipulator can compensate for alignment errors up to 23 degrees in some cases and still successfully join two panels. Overall, the data indicates that the manipulator can successfully and consistently join two panels at a wide range of angles, in both convex and concave configurations. A lower success rate was anticipated for smaller joining angles because the panels make a steeper angle with respect to the vehicle, thus decreasing the resolution and accuracy of the range-finder sensor data. Despite this difficulty, the success rate was still high, with a success rate of 80% for the 45 degree joints. In most trials, there was a point at which the range-finder sensor was no longer able to obtain sufficient data to identify the panels. In these cases, the last recorded data from successful panel identification was used, and the experimental data supports this method as an effective solution.

The only observed failure was a premature collision between the manipulator and the panels, most often during the Acquire behavior and sometimes during the Approach behavior. In the former case, the failures appear to be a product of errors in the estimation of both the panel positions and the manipulator arm positions, causing them to be closer together than estimated. This problem can be solved in a
real system by adding small distance sensors mounted on the arms to directly measure this distance or the use of improved estimates. When a premature collision occurred in the Approach behavior, it was the result of the MOAST navigation algorithms overshooting the Approach target. This failure can be protected against with slower driving speeds and more robust navigation algorithms.

Determination of the minimum possible joining angle proved difficult for a number of reasons. Experiments suggest that this limitation is determined by the depth of the panels and by the length of the manipulator arms, as shown in Figure 11. Panel depth will vary according to choice of construction material, and the lack of dimensional accuracy mentioned in Section IV.3 makes it difficult to accurately predict a minimum joining angle. Despite these difficulties, we estimate the minimum joining angle to

Figure 11: Overhead view of a joining operation for two panels at 40 degrees.
be about 40 degrees given the results in Table 1. The success rate is still high for joining at 40 degrees and the manipulator is still relatively close to the joint during the joining operation. However, the data suggests that the robot requires closer alignment to achieve a successful join than for other joining angles. Attempts to join panels at 35 degrees suffered from repeated failure to detect both panels, and the distance between the joint and manipulator center is large enough that the right arm cannot reliably grasp the right panel. Note that the utility of angles under 45 degrees is very slim for practical robotic construction; therefore, this minimum range is more than sufficient for the purpose of the design.

The defined joint limits for manipulator arm rotation allow each arm to pivot 135 degrees from the center, suggesting that the arms can open to join panels at up to 270 degrees. In practice, the arms are limited by interference with the mobile platform and a need for extra range of motion beyond the panel angle to compensate for alignment variations. By experimentation, the manipulator was found to join panels reliably at angles up to 235 degrees as indicated by the data in Table 1. Attempts to join larger angles failed due to premature collision between the manipulator arms and the panels. This appeared to be caused by a lack of sufficient range of motion to compensate for alignment errors and alignment errors due to attempts to open the manipulator arms such that they collided with the static mesh for the mobile platform. While this latter method of failure is not exactly applicable to a real system, it is indicative of the limits on the manipulator’s range of motion and still pertinent to the results.

**Experiment 3: Panel Approach Range**

Based on the results from the path planning of the hybrid automaton, we assert that this system is able to join panels from any arbitrary robot start location at which both panels are visible. In order to test this hypothesis and determine the range at which the system can identify the panels, the robot’s starting location was positioned
at each of the eleven starting locations. Note that each test of a convex joining angle also tests the abilities of the manipulator to join panels at the corresponding concave angle from the other side of the joint. These start locations are at a distance of 6.4 meters from the joint location and at 15° intervals from the joint bisector up to 75° at either side. Five trials were run for each combination of the previous panel joining angles (Section V.2) and each of the robot start locations, providing a wide range of data to validate the design. Failure indicates a trial for which the detected panels were falsely identified or the manipulator prematurely collided with the panels. Success indicates a trial for which the robot successfully identified, approached, and joined the panels. In the remainder of trials, the robot was unable to identify the panels either correctly or incorrectly. This generally happens for all trials for some approach point and join angle, but occasionally will occur when other trials have successfully identified panels due to variations caused by sensor noise.

Figure 12: Percentage of successful joins and failures for five trials each of multiple approach angles for a 45 degree joint angle.

Figure 12 shows the results for the manipulator attempting to join two panels at a 45 degree angle at eleven approach angles for five trials each. For a straight approach at 0 degrees, the panels were successfully joined for 100 percent of trials.
Due to the steepness of the panel angles relative to the robot, the only other successful identification of the panels was in one trial for an approach angle of -15 degrees, but in general off-axis approach angles provided too little data of the far panel for successful identification. In two trials, one at a 75 degree and the other at a 30 degree approach angle, part of the environment was falsely identified as a panel, leading to failure. In the rest of the trials, the steepness of the angle of the far panel relative to the robot leaves the system with too few sensor points to successfully identify both panels.

Figure 13: Percentage of successful joins and failures for five trials each of multiple approach angles for a 90 degree joint angle.

Figure 13 shows the results for the manipulator attempting to join two panels at a 90 degree angle at eleven approach angles for five trials each. The wider joint angle enabled the robot to successfully identify and join the panels in 100% of trials in the range ±15°. The rate of success dropped beyond this range to 60% and 20% at the 30° and −30° approach angles, respectively. Again, this was caused by too few data points to successfully identify the far panel. Single failures at approach angles 75° and 45° were caused by false identification of panels. Similar to the 45 degree case, all other trials ended with the robot unable to identify both panels. However,
this happened in fewer cases because the larger joint angle creates a shallower angle between the panels and the robot for a greater range of approach angles.

Figure 14: Percentage of successful joins and failures for five trials each of multiple approach angles for a 135 degree joint angle.

Figure 14 shows the results for the manipulator attempting to join two panels at a 135 degree angle at eleven approach angles for five trials each. There is a clear trend showing that the success rate is highest for a direct approach and drops off for higher-degree approach angles. In this case, single failures were caused for approach angles of $\pm 15^\circ$ and $\pm 30^\circ$ by premature collision, and failures at $\pm 75^\circ$ caused by false panel detections. For approach angles $\pm 60^\circ$ and $\pm 75^\circ$, the angle between the far panel and the robot is too steep for the panel detection algorithm to detect.

Figure 15 shows the results for the manipulator attempting to join two panels at a 180 degree angle at eleven approach angles for five trials each. This configuration accommodated the widest range of approach angles, successfully identifying panels in all trials up to $\pm 60^\circ$. One trial at the $30^\circ$ approach angle was caused by premature collision between the manipulator and the panels. The only approach angles for which the panels could not be detected were $\pm 75^\circ$.

Figure 16 shows the results for the manipulator attempting to join two panels at
Figure 15: Percentage of successful joins and failures for five trials each of multiple approach angles for a 180 degree joint angle.

Figure 16: Percentage of successful joins and failures for five trials each of multiple approach angles for a 225 degree joint angle.
a 225 degree angle at eleven approach angles for five trials each. The results suggest that concave angles can be more easily detected than convex, likely because the panel at the steepest angle with respect to the vehicle is the near panel, and the far panel faces the vehicle more directly, compensating for being farther away and having lower-resolution data. The two failures were caused by premature collisions between the manipulator and the panels.

The data confirms the expected results - the robot equipped with the manipulator is able to detect the panels from a wide range of approach angles. Specifically, the range of approach positions from which the robot can identify the panels increases as the panel join angle approaches 180 degrees. The robot is never able to join panels from an approach angle of 75 degrees, suggesting that a 15 degree relative angle between the line of the panel and the vehicle yaw is too steep for the current algorithm to detect. These results are consistent with the earlier analysis of the panel detection method, as a steep angle between the panel and vehicle result in few data points using a laser range-finder. However, for a panel join angle of 180 degrees, the panels can consistently be joined from an initial approach at 60 degrees to the bisector. Therefore, the panel detection algorithm works for an angle as steep as 30 degrees. In a real system, other robots placing the panels will provide a priori information to the joining robot about the position of the panels, making this range more than acceptable. As is expected, a direct approach (0 degrees, on the bisector) is nearly always successful in joining the panels.

The first case that results in failure is a navigation error approaching the panels that causes an error in position or alignment of such magnitude that the panels cannot be successfully acquired or joined. The incidence of this failure mode increases with the approach angle, and the results suggest that one major cause of this error is the identification of the panel using low resolution data from the Approach behavior and a failure to identify the panel again from the approach position. In this case,
the low resolution data is used for the Align, Acquire, and Join behaviors, and in some cases does not include accurate estimates of the endpoint locations. Inaccurate determination of endpoint locations is a particular problem because the joint location is defined as the left endpoint of the right panel, and an error in the identification of the endpoint causes the robot to align to the wrong portion of the panel. The most promising remedy to this problem is the creation of an improved panel identification algorithm that can identify the panels when both ends are not visible. An improved algorithm will be able to identify endpoints from a closer perspective with higher-resolution data, giving greater accuracy.

The second cause of failure is the misidentification of another part of the environment as a panel. This failure occurs when the laser range-finder senses both one full panel and a portion of a wall such that the length of the sensor’s field of view incident on the wall is nearly the same length as a panel. Because the length of a straight line is the identification criteria for a panel, the robot identifies that portion of sensor data as a panel and the robot attempts to join it. One potential solution to this problem that does not change the panel identification criteria is to add a requirement to the identification of the second panel that it be within some reasonable distance of the first panel. This method assumes that the panels are greater than this distance from the walls or other straight-line features of the environment. The other solution to the misidentification problem is the creation of a more robust panel identification algorithm that has stronger criteria than straight-line length.

Over the entire range of trials, we define a panel configuration as being visible to a start position if at least one trial in that configuration resulted in a correct identification of the panels by a robot at that start position. In every case, this criteria correlated to having at least one successful join operation for a particular panel configuration by a robot starting from that position. Using this definition, the panels were visible in 60% of the trials, and of those trials in which the panels were
visible, the successful join rate was 82%. A failure to detect the panels account for
6% of failures when the panels were visible, a misidentification as described above
isolated to the Search behavior account for 7% of failures, and the remaining 4% of
trials were joining failures such as premature collision between the manipulator and
panels. In 13% of trials for which the panels were not visible, the panel identification
algorithm caused a misidentification as described above.
CHAPTER VI

CONCLUSIONS

The research presented in this thesis establishes a mechanical design and hybrid controller for a manipulator to join featureless panels for construction. The design is intended to be part of a larger heterogeneous robotic construction team in which other agents are responsible for tasks of panel placement, alignment, and connection (e.g. welding). Verification of the design and its controller was accomplished using three simulation-based evaluations. Experiment 1 demonstrated the ability of the manipulator to join two panels at an angle of 90 degrees to one another. Results from each of the controller behaviors indicated that the robot behaves as expected. Experiment 2 demonstrated the flexibility of the manipulator by joining panels at angles from (minimum) degrees to (maximum) degrees. The time for the system to execute a joining operation was also evaluated, and the results were found to be reasonable given the limitations of the controller. Experiment 3 demonstrated the robustness of the controller to complete the joining operation when the mobile base is initially located at various starting positions. This experiment confirmed the hypothesis that the mobile robot equipped with the designed manipulator and using the hybrid controller presented in this research can join two panels reliably from any start position from which both panels are visible.

The greatest weakness of the current design is the panel detection method applied to the problem. The solution is functional for the limited cases presented in this research, but for greater reliability and application to more complex environments, a more sophisticated method ought to be developed. In particular, an improved panel detection method is able to identify features of the panels when the whole panel is not within the field of view of the sensor and reject false identifications caused by
only portions of larger objects being visible, both weaknesses of the current system
that have been demonstrated to produce failures.

A potential improvement to the hybrid automaton is the addition of “bailout
states” for recovery in case of failure. The current implementation only provides
transitions in the case of successful completion of state goals. It would be useful to
implement states to handle alternative cases. For instance, a failure in the Search
state could transition to a Wander state in case of failure, which drives the robot
around the world using a more sophisticated mobile search algorithm to find the
panels in case they are not visible from the robot’s initial location. A failure in the
Approach behavior could recalculate the approach position and correct errors caused
by overshooting the target position.

Overall, this thesis presents a strong design for a robot manipulator to autonomously
join panels. The next step in the development of this manipulator is the realization
of the design on a physical robot. Future research ought to also focus on the develop-
ment of the rest of the heterogeneous robotic construction team, including robots
to place, align, and bond the panels.
APPENDIX A

DESIGN AND CONTROL OF A MANIPULATOR FOR AUTONOMOUS JOINING OF FEATURELESS PANELS

Abstract—The use of robots for autonomous assembly has traditionally been reserved for the constrained environments of factory floors, where the positions of the end effector and parts to be assembled can be completely controlled. However, as the use of autonomous robotic vehicles becomes more widespread, more versatile robots create opportunities for robotic assembly in harsher, less constrained environments. This paper presents the design of a manipulator that allows a robot to join two panels without relying on special features for sensing, acquisition, or alignment. The paper presents control system development and experimental results based on simulation.

I. INTRODUCTION

Research into autonomous vehicles over the past several years has provided robust solutions to many of the basic problems involved in navigation and basic transportation. However, many technological innovations must be made to combine mobile platforms with manipulators in order to create flexible, reliable, and practical robots not confined to traditional working envelopes. One potential application of these mobile manipulators is construction, especially in hazardous environments such as space or extreme terrain on earth. Such applications are relevant to NASA, which has announced an intent to establish a presence on the moon by 2020 [1]; to the military, which often must work in hostile territory; and international aid organizations, which also work in unfavorable environments and prefer to focus on the human element rather than constructing buildings.

While some research has been conducted in autonomous construction using mobile manipulators, the most advanced current technology relies heavily on specialized features as part of the construction materials [7]. Such allowances have produced solutions with some degree of success, but the need for such special features decreases the practical utility of the solution by increasing the cost and decreasing the flexibility of the building design. The goal of this research is to overcome the need for special features in construction materials for autonomous assembly through the development of a manipulator and control system for mobile platforms that can identify, acquire, and join featureless panels.

Section II of this paper presents existing related research and describes previous progress in autonomous robotic construction using mobile manipulators. Section III describes the mechanical design of the manipulator. Section IV follows the development of a hybrid automaton to control the vehicle and manipulator. Finally, Section V experimentally validates the system in a simulated domain.

II. RELATED WORK

Mobile manipulation has been a rich field of study for nearly twenty years. As early as 1990, researchers proposed the use of autonomous robots in the construction of a lunar base [2], and in 2004 a plan was presented for a brick-laying mobile robot for use in construction industries [3]. Early system development often simplified the problem by pushing construction elements rather than picking them up [6]. In 2000, researchers demonstrated the use of a multi-robot team for the completion of multiple component mating [4].

The Robotic Construction Crew (RCC), developed at NASA’s Jet Propulsion Laboratory (JPL), is an autonomous robotic construction system [7]. The JPL solution consists of a heterogeneous robotic team with two different types of robots, the Sample Return Rover (SRR) and the SRR2K [12]. Both robots feature a four degree-of-freedom robotic arm, stereo camera pair, and a 3-axis force-torque sensor at the base of the gripper. Construction is accomplished using rigid beams and flexible panels, both of which have a length that is four times the width of the robot. The construction elements each have two grasping points where the robots grasp them, several interlocking cones for alignment and locking, and a set of three fiducials to provide position and orientation information [7]. The robots are controlled using a behavior-based architecture to compensate for uncertainties due to sensor noise and limited processing capabilities. Cooperative manipulation and positioning of parts is accomplished in iterative steps, and the system relies heavily upon relative position information provided by the stereo camera system to correct errors [7].

End effectors using suction to hold a target object are common throughout robotics, from office work [8] to harvesting [9]. Two patents indicate the applicability of suction-based effectors to the problem addressed in this research. The first, a 1986 patent [10], is for a suction-cup-based end effector for loading and unloading sheet metal from a molding machine. The second, a 1995 patent [11], describes the design of a robotic end effector to pick up and transport rigid sheets of glass using suction.

III. MANIPULATOR DESIGN

A. Considerations

The goal of this research is to design and implement, in simulation, a joining manipulator that aligns and pulls
together any pair of featureless panels in close alignment. Figure 1 shows a panel as represented in USARSim [14]. Unlike the panels used in the RCC, our panels use no special markings, pins, or holes to aid in robot manipulation. The intention is that this design will support a system in which the construction materials can be unaltered panels, such as drywall sheets, plywood boards, or sheet metal.

![Featureless panel](image1.png)

**Fig. 1.** Featureless panel used for joining.

In order to be sufficiently flexible as to be practically useful, the design must accommodate a range of panel placements and joining angles, be able to grasp a variety of construction materials, be simple to incorporate into an existing robot, and not rely on expensive or complicated supporting technologies. The presented manipulator design meets these criteria with an implementation featuring pivoting joining arms and suction grasping. This design assumes that a separate system handles the bonding of the panels after they are assembled.

### B. Manipulator Design

The general manipulator design consists of two pivoting arms attached to a common support, which extends away from the mobile robot base. Each of the two arms defines one side of the joint, and the arms are designed to pivot up to 135 degrees away from center. In order to avoid planning difficulties involved with moving both panels together to form the joint, the design is intended to hold the right panel in place and pull the left panel toward it to create the joint. The left arm uses a prismatic joint to extend and retract the grasping component of the effector to pull the left panel towards the right.

The central support section extends far enough away from the robot body in order to provide clearance for the manipulator arms to open all the way. This support also houses the drive motors that pivot the manipulator arms, as shown in Figure 2. The motors are secured via a face mount to a mounting plate built into the support section. The pivoting arms are actuated by the motors using a geartrain of conical gears attached to a vertical shaft, which drives the arms. These shafts are also connected to rotary encoders, which provide the angle of the arms for the controller. Both of these features can be seen in Figure 3.

The prismatic joint is driven along a pair of guide shafts using a linear actuator. The robot must already have both pneumatic and electric systems, to drive the suction and drive systems, respectively, therefore either is acceptable for use. However, pneumatic actuators provide a particular benefit in that the force can be easily controlled by the pressure of the air used for actuation. Such a feature is useful in halting the joining action once the panels are in contact.

![Side view of the manipulator design showing the drive system for the arms.](image2.png)

**Fig. 2.** Top view of the manipulator design, with the top plate of the support transparent.

**Fig. 3.** Top view of the manipulator design showing the drive system for the arms.

Gripping by the manipulator is accomplished via commercially available pliable suction cups (Figure 2) connected to a vacuum feed. When the manipulator is close to the panels, the feed turns on and the panel is held by the suction cups.

### C. Robot Base Requirements

For purposes of the simulation, the robot base is an iRobot ATRV Jr., shown in Figure 4. However, the manipulator is designed to be independent of any particular mobile robot base and to impose as few requirements on that base as possible. In order to support the manipulator, the mobile robot base must be able to supply electrical energy for the motors and a vacuum source for the suction grasping. A laser range-finding sensor mounted such that it will detect panels within range of the robot and not be occluded by the manipulator is also required. In addition, the robot should
not have any features that interfere with the manipulator’s range of motion.

Fig. 4. The robot joining two panels at a 135 degree joint angle.

IV. BEHAVIORAL CONTROL FOR PANEL JOINING

In order to control the robot during panel joining, a hybrid automaton was developed to identify, acquire, and connect two adjacent panels to form a joint. A hybrid automata is a model of computation that is uniquely able to represent both continuous and discrete state information. In the most straightforward implementation, the discrete state information is encoded as the discrete states of the system, while the continuous state information can be described as a differential equation. Egerstedt and Hu have shown that hybrid automata can be used for behavioral control of autonomous mobile robots [13]. Their work involved the development of a regularized hybrid automaton to handle simple obstacle avoidance by a mobile robot through the use of attractive and repulsive fields. The full automaton developed for this research is shown in Figure 5.

Fig. 5. Hybrid automaton for the autonomous robot panel-joining system.

A. Search

The automaton Search behavior (Figure 5) assumes that the panels are within range of the robot’s laser range-finder sensor, and assumes that they are the only two panels immediately identifiable by the robot. Therefore, the robot spins in place in order to locate the intended joint and exits this state when the two adjacent panels are identified. As in all other states, the robot identifies the panels by executing a simple Hough transform on the laser range-finder data, then iterates through the data points in order to find the points that define the panel and remove outliers. The remaining points describe the continuous line representing the panel. The panel is identified by its width, requiring the assumption that the panels are the only linear objects of that width in the environment. It is important to note, however, that once the endpoints of the panels are identified, all other algorithms are independent of the laser range-finding data, making the system as a whole compatible with any panel detection method that returns the endpoints of those panels.

B. Approach

The search behavior leaves the robot in an orientation from which the joint is visible. However, in order to join the panels, the robot must be in a position with the intersection of the arms at the desired joint location and in such an orientation that both arms can attach to their respective panel. The Approach behavior (Figure 5) detects the panels again, calculates the bisector of the two panels, and identifies an approach position on a line parallel to the bisector and passing through the intended joint location. This position allows for a straight-line, centered approach to the joint. Before moving to the location, the robot opens both arms of the manipulator so that they do not protrude and potentially collide with the panels prematurely. Using the obstacle avoidance and point-to-point navigation included in the autonomous mobility echelon of the MOAST controller [17], the robot moves to this approach location, re-planning along the way as closer sensor data provides greater accuracy for panel detection.

C. Align

It is likely that the Approach behavior will place the robot in the correct position, but with an orientation that is not conducive to joining the panels. The Align behavior (Figure 5) detects the panels, this time with greater resolution from a closer perspective, and orients the robot such that its central axis is pointed toward the intended joint location. This alignment centers the robot on the joint, giving an equal angular offset from each arm to its respective panel. Because the laser range-finder’s field of view is limited to 180 degrees and has a finite angular resolution, it is possible that the sensor’s perspective will leave it unable to detect the panels. In the Search and Approach behaviors, this situation simply results in a failure. However, in the Align behavior and all subsequent behaviors, the system will attempt to recover by using the panel information recorded from the last successful detection.

D. Acquire

In order to pull the panels together, the Acquire behavior (Figure 5) first moves the manipulator close enough to the intersection that the suction effectors can effectively grasp
the panels and pull them together. Knowing the robot’s orientation and the manipulator’s physical parameters, the line describing the position of an arm can be described by the angle $\theta$ passing through the intersection of the two arms $(x, y)$,

\[
\begin{align*}
x &= l \cos \psi + x_{veh} \\
y &= l \sin \psi + y_{veh} \\
\theta &= \psi + \phi
\end{align*}
\]

where $(x_{veh}, y_{veh})$ is the location of the vehicle center, $\psi$ is the vehicle yaw, $l$ is the distance from the vehicle center to the intersection of the joining arms, and $\phi$ is the angle of the arm with respect to the vehicle. Using this estimate, the arms are opened using a simple proportional controller with $\dot{\phi} = 0.5(\theta - \theta_{\text{panel}})$ until they are parallel to their corresponding panel.

Actual panel acquisition is achieved via a control loop that iteratively handles corrections to vehicle alignment and arm alignment while driving the vehicle forward. In each case, a proportional controller was found to be sufficient for reliable success. Therefore, in each iteration of the control loop, the following control laws are executed, in order:

Vehicle: $\dot{\psi} = 0.5(\psi - \tan^{-1}(y_{\text{int}} - y_{veh}, x_{\text{int}} - x_{veh}))$

Arms: $\dot{\phi} = 0.5(\theta - \theta_{\text{panel}})$

Drive: $v = 0.25 \sqrt{(x_{\text{int}} - x)^2 + (y_{\text{int}} - y)^2}$

where $(x_{\text{int}}, y_{\text{int}})$ is the intersection point of the panels. The tolerances on each control are 0.005 radians, 0.01 radians, and half the initial distance from the panel intersection to the manipulator arms’ intersection for that iteration, respectively. Therefore, the control loop causes the robot to take smaller and smaller movements closer and closer to the panels, providing accuracy at a cost of time. The control loop exits when both of the manipulator arms are close enough to grasp the panels, 2 cm in this case.

E. Join

The Join behavior (Figure 5) simply initiates a suction event, causing the effector arms to “grasp” the panels, and then retracts the prismatic link to pull them together. For the purposes of simulation, the suction is approximated by the creation of a virtual link between the arm and the panel. Once a successful join is completed, defined by the elimination of the gap between panels, the system returns an appropriate success or failure code.

V. EVALUATION AND VERIFICATION

Due to cost and time considerations, the verification of the system was completed in simulation. The system was initially evaluated for its ability to simply join panels. However, follow-up evaluations were also conducted to determine the ability of the effector to join panels at a range of angles.

A. Simulation Environment

The Unified System for Automation and Robotics Simulation (USARSim) [14] was chosen for this project due to its ease of use and flexibility. In addition, the USARSim package has been verified to effectively simulate reality in a number of tests to determine the accuracy of its sensors and movement [15]. Simulations were run on a MacBook Pro under Windows XP in BootCamp. The manipulator was modeled as a static mesh using Blender [16], see Figure 6. The manipulator was attached to the front of the standard ATRV Jr. robot model that is a part of USARSim.

The controller was the Mobility Open Architecture Simulation and Tools (MOAST) environment [17]. The MOAST system is an open source project that provides automatic obstacle avoidance, mapping, and sensor blending. In addition, it supports a wide array of vehicle types, including skid-steer robots (such as the ATRV Jr.). Additionally, MOAST was designed to integrate directly with USARSim. One significant drawback to the MOAST system is that, because it uses a number of different parallel processes operating concurrently and uses message buffers to communicate between them, it was only possible to do one action at a time rather than coordinating separate movements in parallel (i.e. moving both manipulator arms at once).

For each testing configuration, the panels are placed at the desired angle and within sensor range (8 meters) from the robot start location via the Unreal editor, a tool for building maps in Unreal. Therefore, they are generated in the same location for each trial at a given panel angle. On the other hand, an instance of the robot must be generated in the simulation environment before it can be used. This process results in a slightly different initial position for the robot at the beginning of each trial. However, because the path to the panels is planned by the Approach, Align, and Acquire behaviors, the system is robust to these small disturbances. No obstacles exist in the environment other than the panels.

B. Simple Panel Joining

Initial testing was conducted using a 90 degree angle between the panels. This was chosen since it is in the middle of the manipulator arm’s capable range and was anticipated to be the most common joining angle required.
Figure 7 is an example of the results from the Approach behavior for a single trial. The dots indicate laser-rangefinder data. The lines are calculated by the panel identification algorithm, indicating that the line detection algorithm successfully identified the panels. Endpoints of the panels are identified by larger marks, and the vehicle position is indicated using the x mark. This plot indicates that the controller correctly identified both panels and their endpoints, calculated the approach location (shown by the circle not on a panel) based on the bisector, and drove the vehicle to it.

After aligning the robot body to the joint and the manipulator arms to the panels, the robot approaches the joint in increasingly small movements for greater accuracy. While this was done at a cost of time, experiments indicate that the additional time is not prohibitive. Average simulation time was on the order of seven minutes for the simple controller with the limitations of MOAST. A real controller system designed for practical applications will likely be able to improve on this time significantly.

Figure 8 shows an example of the laser range-finder data following the successful execution of the Join behavior. A successful join was defined as having no detectable gap between the panels, which could be seen as a break in either line representing the panels. Ten trials at an angle between the panels of 90 degrees show a successful join rate of 100%, indicating that the system works as anticipated.

### C. Range of Joint Angles

While the initial evaluation verified the ability to join panels when they are positioned at a 90 degree angle to each other, it is necessary to evaluate the manipulator for varying panel angles. An objective was to determine the minimum and maximum panel angles that the manipulator can join successfully. Evaluations were conducted in the same manner as for the 90 degree case for joining angles between 45 degrees and 225 degrees at 45 degree intervals. Ten trials were evaluated for each configuration, excluding trials in which USARSim did not generate the robot in a working configuration, a known problem with USARSim.

#### Table I

<table>
<thead>
<tr>
<th>Angle</th>
<th>Success</th>
<th>Time (min.)</th>
<th>Alignment Error (°)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Standard Deviation</td>
<td>Mean</td>
</tr>
<tr>
<td>45</td>
<td>80%</td>
<td>6.92</td>
<td>0.45</td>
</tr>
<tr>
<td>90</td>
<td>100%</td>
<td>6.95</td>
<td>0.50</td>
</tr>
<tr>
<td>135</td>
<td>100%</td>
<td>6.17</td>
<td>0.34</td>
</tr>
<tr>
<td>180</td>
<td>100%</td>
<td>6.11</td>
<td>0.46</td>
</tr>
<tr>
<td>225</td>
<td>90%</td>
<td>7.01</td>
<td>0.22</td>
</tr>
</tbody>
</table>

Table I provides the experimental results. The Angle represents the angle made by the two panels with respect to each other. For each panel angle, we present the Success percentage over 10 trials, the mean (X̄) and standard deviation (σ) of the Time in minutes to execute a trial, and the mean and standard deviation of the final Alignment Error in degrees. This alignment error is calculated as the difference between the vehicle yaw and the joint bisector. These results demonstrate that the manipulator can compensate for alignment errors up to 23 degrees and still successfully join two panels. Overall, the data indicates that the manipulator can successfully and consistently join two panels at a wide range of angles, in both convex and concave configurations. A lower success rate was anticipated for smaller joining angles because the panels make a steeper angle with respect to the vehicle, thus decreasing the resolution and accuracy of the laser range-finder sensor data. Despite this difficulty, the success rate was still high, with the lowest success rate of 80% for the 45 degree joints. In most trials, there was a point at which the range-finder sensor was no longer able to obtain sufficient data to identify the panels. In these cases,
the last recorded data from successful panel identification was used, and the experimental data supports this method as an effective solution.

The only observed failure was a premature collision between the manipulator and the panels, most often during the Acquire behavior and sometimes during the Approach behavior. In the former case, the failure appears to be a product of errors in the estimation of both the panel positions and the manipulator arm positions, causing them to be closer together than estimated. This problem can be solved in a real system by adding small distance sensors mounted on the arms to directly measure this distance or the use of improved estimates. When a premature collision occurred in the Approach behavior, it was the result of the MOAST navigation algorithms overshooting the Approach target. This failure can be protected against with slower driving speeds and more robust algorithms. Based on the results from the path planning of the hybrid automaton, we assert that this system should be able to join panels from any arbitrary robot start location at which both panels are visible. Verification of this hypothesis is ongoing.

Although the joint limits on the manipulator arms allow them to pivot up to 270 degrees, this angle does not accommodate joining two panels positioned at an angle of 270 degrees to one another. This is the case because the arms require some flexibility in order to compensate for small errors in vehicle alignment. Operating the manipulator at its maximum or minimum limits does not allow for this type of flexibility. Despite this limitation, an attempt was made to test the manipulator’s ability to join panels positioned at 270 degrees. It was found that the static meshes used to simulate the manipulator collided before reaching 270 degrees, and if all the elements were moved out far enough so as not to collide, the arm position estimates were no longer accurate, leading to a failure on every attempt.

VI. CONCLUSIONS AND FUTURE WORKS

This research has developed a design for a robot manipulator and associated control system to join featureless panels for autonomous construction. Simulation results indicate that the design is successful in attaining this goal consistently and accurately when the angle between the panels is reachable by the manipulator arms and both panels are initially visible to the laser range-finding sensor. Research to fully characterize the manipulator by experimentally determining the limits of its joining capability is ongoing. In addition, test are being conducted to confirm the ability of the hybrid automaton to identify and navigate to a joint from any location from which both panels are visible. The next step is to build the manipulator and test it on a robot in reality. Future work should focus on the development of robots to collect and place the panels in the configuration to be joined. This design is intended to be part of a larger heterogeneous team of robots that autonomously assemble walls for a structure by repeatedly placing and joining panels together.

REFERENCES

APPENDIX B

MANIPULATOR DETAIL MACHINE DRAWINGS
MOTOR PLATE

DIMENSIONS ARE IN INCHES
TOLERANCES: FRACTIONAL
ANGULAR: MACH

THREE PLACE DECIMAL
MATERIAL

TWO PLACE DECIMAL
PROHIBITED.

THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF [INSERT COMPANY NAME HERE]. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF [INSERT COMPANY NAME HERE] IS PROHIBITED.

SB-2

UNLESS OTHERWISE SPECIFIED:
SCALE: 1:2
WEIGHT:

REV DWG. NO.
A

NAME DATE

Q.A.
MFG APPR.
ENG APPR.
CHECKED
DRAWN

DO NOT SCALE DRAWING

5.47
0
.83
2.77
1.45
0
.53
1.50
2.47
3.53
4.50
5.25
.75
3.00
0
5.0
3.00
0
5.25
.25
.75
1/4-20 UNC
.50

8X Ø .18 THRU
3X Ø .20 ± .75
2X Ø 1.50
1/4-20 UNC ± .50

6.0
5.0
2.77
1.45
0
.83
0
.53
1.50
2.47
3.53
4.50
5.47
.25

FINISH

BEND

PROPRIETARY AND CONFIDENTIAL

NOTES:

APPLICATION

HOLE ASY

USED ON

NOMINAL

TOLERANCE

MATERIAL

FINISH

COMM.1

SCALE: 12
WEIGHT:

SHEET 1 OF 1
<table>
<thead>
<tr>
<th>ITEM NO.</th>
<th>PartNo</th>
<th>DESCRIPTION</th>
<th>Vendor</th>
<th>QTY.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HA-1</td>
<td>RIGHT ARM</td>
<td>CUSTOM</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>VC35</td>
<td>FLAT BONDED VACUUM CUP, 3.5&quot; DIAMETER</td>
<td>ANVER VACUUM</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6673T271</td>
<td>STEEL ROD 1&quot; DIAMETER, 8&quot; LENGTH</td>
<td>MCMCMASTER-CARR</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>6529K16</td>
<td>STEEL 20 DEG ANGLE GEAR 16 PITCH, 1.5&quot; PITCH DIA.</td>
<td>MCMCMASTER-CARR</td>
<td>1</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Title:** HOLD ARM
- **Drawing Scale:** 1:6
- **Material:** STEEL
- **Notes:**
  - Dimensions are in inches.
  - Tolerances: Fractional.
  - Angular tolerances: Mach 2.
  - Three place decimal interpretation of geometric tolerances.

**Disclaimer:**

The information contained in this drawing is the sole property of [insert company name here]. Any reproduction in part or as a whole without the written permission of [insert company name here] is prohibited.
<table>
<thead>
<tr>
<th>ITEM NO.</th>
<th>PART NUMBER</th>
<th>DESCRIPTION</th>
<th>MANUFACTURER</th>
<th>QTY.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JA-1</td>
<td>LEFT ARM 1</td>
<td>CUSTOM</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>JA-2</td>
<td>LEFT ARM 2</td>
<td>CUSTOM</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>JA-3</td>
<td>BRACKET 1</td>
<td>CUSTOM</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>JA-4</td>
<td>BRACKET 2</td>
<td>CUSTOM</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>JA-5</td>
<td>ALIGN SHAFT</td>
<td>CUSTOM</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>JA-6</td>
<td>ALIGN SHAFT STOP CAP</td>
<td>CUSTOM</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>VC35</td>
<td>FLAT BONDED VACUUM CUP, 3.5&quot; DIAMETER</td>
<td>ANVER VACUUM</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>LCEN1NS11P118 82-003</td>
<td>PNEUMATIC CYLINDER</td>
<td>COWAN DYNAMICS INC</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>6673T271</td>
<td>STEEL ROD 1&quot; DIAMETER, 8&quot; LENGTH</td>
<td>MCMASTER-CARR</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>98381A546</td>
<td>STEEL DOWEL PIN 1/4&quot; DIAMETER, 1-1/2&quot; LENGTH</td>
<td>MCMASTER-CARR</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>90145A496</td>
<td>STEEL DOWEL PIN 5/32&quot; DIAMETER, 1-1/2&quot; LENGTH</td>
<td>MCMASTER-CARR</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>933671</td>
<td>PILLOW-BLOCK LINEAR BEARING CLOSED, 1/2&quot; ID</td>
<td>MCMASTER-CARR</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>6529K16</td>
<td>STEEL 20 DEG ANGLE GEAR 16 PITCH, 1.5&quot; PITCH DIA.</td>
<td>MCMASTER-CARR</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>94912A455</td>
<td>18-8 SS SHCS W/LOCK WASHER 1/4&quot;-20 THD, 5/8&quot; LGTH</td>
<td>MCMASTER-CARR</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>94912A110</td>
<td>18-8 SS SHCS W/LOCK WASHER 6-32 THD, 1/2&quot; LGTH</td>
<td>MCMASTER-CARR</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>98398A120</td>
<td>BOWED E-STYLE RETAINING RING FOR 1/4&quot; SHAFT DIA</td>
<td>MCMASTER-CARR</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>98408A124</td>
<td>SS E-STYLE RETAINING RING FOR 5/32&quot; SHAFT DIA</td>
<td>MCMASTER-CARR</td>
<td>2</td>
</tr>
</tbody>
</table>
LEFT ARM 2


UNLESS OTHERWISE SPECIFIED

SCALE: 1:4

THREE PLACE DECIMAL INTERPRET GEOMETRIC TOLERANCING PER:

THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF [INSERT COMPANY NAME HERE]. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF [INSERT COMPANY NAME HERE] IS PROHIBITED.
; USARBot.ini
; Additions by Andrew Bouchard
; The following code was added to USARBot.ini in the UT2004/System
; folder to define the robot used for research

[USARBot.Joiner]
bDebug=False
Weight=50
Payload=25
ChassisMass=2.000000
MotorTorque=100.0
MaxTorque=150.0
bMountByUU=False

JointParts=(
    PartName="RightFWheel", PartClass=class'USARModels.AtrvRTire',
    DrawScale3D=(X=1.0, Y=1.0, Z=1.0), bSteeringLocked=True, bSuspensionLocked=True,
    Parent="", JointClass=class'KCarWheelJoint', ParentPos=(Y=0.2559997, X=0.1939998, Z=0.1919998),
    ParentAxis=(Z=1.0), ParentAxis2=(Y=1.0), SelfPos=(Z=0.0), SelfAxis=(Z=1.0), SelfAxis2=(Y=1.0)
)
JointParts=(
    PartName="LeftFWheel", PartClass=class'USARModels.AtrvLTire',
    DrawScale3D=(X=1.0, Y=1.0, Z=1.0), bSteeringLocked=True, bSuspensionLocked=True,
    Parent="", JointClass=class'KCarWheelJoint', ParentPos=(Y=-0.2559997, X=0.1939998, Z=0.1919998),
    ParentAxis=(Z=1.0), ParentAxis2=(Y=1.0), SelfPos=(Z=0.0), SelfAxis=(Z=1.0), SelfAxis2=(Y=1.0)
)
JointParts=(
    PartName="RightRWheel", PartClass=class'USARModels.AtrvRTire',
    DrawScale3D=(X=1.0, Y=1.0, Z=1.0), bSteeringLocked=True, bSuspensionLocked=True,
    Parent="", JointClass=class'KCarWheelJoint', ParentPos=(Y=0.2559997, X=-0.1939998, Z=0.1919998),
    ParentAxis=(Z=1.0), ParentAxis2=(Y=1.0), SelfPos=(Z=0.0), SelfAxis=(Z=1.0), SelfAxis2=(Y=1.0)
)
JointParts=(
    PartName="LeftRWheel", PartClass=class'USARModels.AtrvLTire',
    DrawScale3D=(X=1.0, Y=1.0, Z=1.0), bSteeringLocked=True, bSuspensionLocked=True,
    Parent="", JointClass=class'KCarWheelJoint', ParentPos=(Y=-0.2559997, X=-0.1939998, Z=0.1919998),
    ParentAxis=(Z=1.0), ParentAxis2=(Y=1.0), SelfPos=(Z=0.0), SelfAxis=(Z=1.0), SelfAxis2=(Y=1.0)
)

MisPkgs=(
    PkgName="MagnetGripper", Location=(Y=0.0, X=0.25, Z=0.05), PkgClass=Class'USARMisPkg.MagnetGripper'
)

Sensors=(
    ItemClass=class'USARModels.SICKLMSr', ItemName="Scanner1", Position=(X=0.1, Y=0.0, Z=-0.20),
    Direction=(Y=0.0, Z=0.0, X=0.0)
)
Sensors=(
    ItemClass=class'USARBot.OdometrySensor', ItemName="Odometry", Position=(X=0.0, Y=0.0, Z=-0.0),
    Direction=(Y=0.0, Z=0.0, X=0.0)
)
Sensors=(
    ItemClass=class'USARBot.INSSensor', ItemName="Compass", Position=(X=0.0, Y=0.0, Z=0.0),
    Direction=(X=1.5664821, Y=0.0, Z=0.0)
)
Sensors=(
    ItemClass=class'USARBot.GroundTruth', ItemName="GroundTruth", Position=(X=0.0, Y=0.0, Z=0.0),
    Direction=(Y=0.0, Z=0.0, X=0.0)
)

Effecters=(
    ItemClass=class'USARBot.ResearchSuctionEffecter', ItemName="HoldStill", Parent="MagnetGripper_Link1", Position=(Y=0.0, X=0.15, Z=0.0),
    Direction=(Y=0.0, Z=0.0, X=0.0)
)
Effecters=(
    ItemClass=class'USARBot.ResearchSuctionEffecter', ItemName="Suction", Parent="MagnetGripper_Link5", Position=(Y=0.0, X=0.15, Z=0.0),
    Direction=(Y=0.0, Z=0.0, X=0.0)
)
class Joiner extends SkidSteeredRobot config(USARBot);

defaultproperties
{
    // Note that the "Joiner robot" is built on top of the ATRVJr, 
    // therefore has all the same parameters, but features 
    // an additional mission package

    //---Wheel information. Since the ATRVJr has four wheels, we have 
    // The next four definitions will create a 4-wheel skid steer ATRVJr 
    // where all the wheels are locked (cannot be steered)

    // Number=0 indicates that this wheel is the first JointPart 
    // defined in USARBot.ini (ORDER MATTERS!) 
    // Power=Right_Powered tells USARSim to spin this wheel using the 
    // right throttle 
    // Note that since the variables SteerType and MaxSteerAngle are 
    // not defined, this wheel is not steered (steering is locked) 
    Wheels(0)=(Number=0,PowerType=Right_Powered);

    // Number=1 indicates that this wheel is the second JointPart 
    // defined in USARBot.ini (ORDER MATTERS!) 
    // Power=Left_Powered tells USARSim to spin this wheel using the 
    // left throttle 
    // Note that since the variables SteerType and MaxSteerAngle are 
    // not defined, this wheel is not steered (steering is locked) 
    Wheels(1)=(Number=1,PowerType=Left_Powered);

    // Number=2 indicates that this wheel is the third JointPart 
    // defined in USARBot.ini (ORDER MATTERS!) 
    // Power=Left_Powered tells USARSim to spin this wheel using the 
    // right throttle 
    // Note that since the variables SteerType and MaxSteerAngle are 
    // not defined, this wheel is not steered (steering is locked) 
    Wheels(2)=(Number=2,PowerType=Right_Powered);

    // Number=3 indicates that this wheel is the fourth JointPart 
    // defined in USARBot.ini (ORDER MATTERS!) 
    // Power=Left_Powered tells USARSim to spin this wheel using the 
    // left throttle 
    // Note that since the variables SteerType and MaxSteerAngle are 
    // not defined, this wheel is not steered (steering is locked) 
    Wheels(3)=(Number=3,PowerType=Left_Powered);

    //---Other Robot's Properties 
    bDebug=false 
    StaticMesh=StaticMesh'USARSim_Vehicles_Meshes.ATRVJr.ATRVJrBody' 
    DrawScale=4.762
DrawScale3D=(X=1.0,Y=1.0,Z=1.0)

ChassisMass=1.000000

// Configuration Parameters. Note: these variables need to be
correspond to your robot's model.
    WheelRadius=0.1922 // Wheel Radius, in unreal
units. Note: Value is in meters
    Dimensions=(X=0.7744,Y=0.6318,Z=0.5754) // X=Length=0.7744m,
Y=Width=0.6318m, Z=Height=0.5754m
    maxSpinSpeed=5.2 // Maximum wheel's spin speed
        is 1/0.1922 = 5.2 rad/sec
    TireRollFriction=15.000000
    TireLateralFriction=15.0
    TireRollSlip=0.0600
    TireLateralSlip=0.0600000
    TireMinSlip=0.005000
    TireSlipRate=0.00500000
    TireSoftness=0.000020
    TireAdhesion=0.000000
    TireRestitution=0.000000

Begin Object Class=KarmaParamsRBFull Name=KParams0
    KActorGravScale=2.58
    bKNonSphericalInertia=True
    KInertiaTensor(0)=0.061
    KInertiaTensor(3)=0.078
    KInertiaTensor(5)=0.083
    KCOMOffset=(X=0.0,Y=0.0,Z=0.0)
    KLinearDamping=0.0
    KAngularDamping=0.0
    KMaxAngularSpeed=100
    KMaxSpeed=25000
    KStartEnabled=True
    bHighDetailOnly=False
    bClientOnly=False
    bKDoubleTickRate=True
    KFriction=0.9
    Name="KParams0"
End Object
    KParams=KarmaParamsRBFull'USARBot.ATRVJr.KParams0'
  //--
}
The following code was added to the USARMisPkg.ini file in UT2004/System to define the manipulator used in research

; Magnet Panel Gripper mission package used for the Joiner robot

[USARMisPkg.MagnetGripper]
Link=(LinkNumber=0,LinkClass=Class'USARMisPkg.MagnetGripperBase',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),ParentLinkNumber=-1,SelfMount="A")
Link=(LinkNumber=1,LinkClass=Class'USARMisPkg.MagnetArmRight',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),ParentLinkNumber=0,ParentMount="B",SelfMount="A")
Link=(LinkNumber=2,LinkClass=Class'USARMisPkg.MagnetArm_LinkTwo',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),ParentLinkNumber=0,ParentMount="C",SelfMount="A")
Link=(LinkNumber=3,LinkClass=Class'USARMisPkg.MagnetArm_LinkThreeA',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),ParentLinkNumber=2,ParentMount="B",SelfMount="A")
Link=(LinkNumber=4,LinkClass=Class'USARMisPkg.MagnetArm_LinkFourA',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),ParentLinkNumber=3,ParentMount="B",SelfMount="A")
Link=(LinkNumber=5,LinkClass=Class'USARMisPkg.MagnetArm_LinkFive',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),ParentLinkNumber=4,ParentMount="B",SelfMount="A")

[USARMisPkg.MagnetGripperBase]
MountPoints=(Name="A",JointType="Revolute",Location=(X=0.0,Y=0.0,Z=0.0),Orientation=(X=0,Y=0,Z=0))
MountPoints=(Name="B",JointType="Revolute",Location=(X=0.37,Y=0.02,Z=0.0),Orientation=(X=0,Y=1.5707963267948966192313216916398,Z=0))
MountPoints=(Name="C",JointType="Revolute",Location=(X=0.37,Y=-0.02,Z=0.0),Orientation=(X=0,Y=1.5707963267948966192313216916398,Z=0))
MaxSpeed=0.0
MaxTorque=200000
MinRange=0.0
MaxRange=0.0

[USARMisPkg.MagnetArmRight]
MountPoints=(Name="A",JointType="Revolute",Location=(X=0.0,Y=0.0,Z=0.0),Orientation=(X=0,Y=0,Z=0))
MaxSpeed=0.1745
MaxTorque=20
MinRange=0
MaxRange=2.356

[USARMisPkg.MagnetArm_LinkTwo]
MountPoints=(Name="A",JointType="Revolute",Location=(X=0.0,Y=0.0,Z=0.0),Orientation=(X=0,Y=0,Z=0))
MountPoints=(Name="B",JointType="Revolute",Location=(X=0.20,Y=-0.045,Z=0.0),Orientation=(X=0,Y=0,Z=0))
MaxSpeed=0.1745
MaxTorque=20
MinRange=-2.356
MaxRange=0

[USARMisPkg.MagnetArm_LinkThreeA]
MountPoints=(Name="A",JointType="Revolute",Location=(X=0.0,Y=0.0,Z=0.0),Orientation=(X=0,Y=0,Z=0))
MountPoints=(Name="B",JointType="Revolute",Location=(X=-0.05028,Y=0.0,Z=0.0),Orientation=(X=-1.5707963267948966192313216916398,Y=0,Z=0))
MaxSpeed=0.5
MaxTorque=20
MinRange=0
MaxRange=0

[USARMisPkg.MagnetArm_LinkFourA]
MountPoints=(Name="A",JointType="Prismatic",Location=(X=-0.05028,Y=0.0,Z=0.0),Orientation=(X=0.0,Y=1.5707963267948966192313216916398,Z=0.0))
MountPoints=(Name="B",JointType="Prismatic",Location=(X=0.10,Y=0.045,Z=0.0),Orientation=(X=0.0,Y=1.5707963267948966192313216916398,Z=0.0))
; max translation speed in m/s
MaxSpeed=0.5
MaxTorque=300
; MinRange should always be zero - for prismatic joints
; MaxRange defines how far along the Z-Axis the part can move - for prismatic joints
MinRange=0
MaxRange=0.3

[USARMisPkg.MagnetArm_LinkFive]
MountPoints=(Name="A",JointType="Revolute",Location=(X=0.0,Y=0.0,Z=0.0),Orientation=(X=0.0,Y=1.5707963267948966192313216916398,Z=0.0))
MaxSpeed=0.0
MaxTorque=300
MinRange=0
MaxRange=0
// MagnetGripper.uc
// Written by Andrew Bouchard
// Defines the MagnetGripper mission package,
// corresponding to part A1 in the CAD package
// From folder UT2004\USARMisPkg\Classes

class MagnetGripper extends MisPkgInfo config(USARMisPkg);
// MagnetGripperBase.uc
// Written by Andrew Bouchard
// Defines the base of the MagnetGripper mission package,
// corresponding to part M1 in the CAD package
// From folder UT2004\USARMisPkg\Classes

class MagnetGripperBase extends MisPkgLinkInfo config(USARMisPkg);

defaultproperties
{
    ModelClass=class'USARModels.MagnetGripperBase'
}
// MagnetGripperBase.uc
// Written by Andrew Bouchard
// Defines the static mesh for the base of the MagnetGripper mission package
// From folder UT2004\USARModels\Classes

class MagnetGripperBase extends KDPart;

defaultproperties
{
// Scaled with 4.762 at Mon Sep 25 14:21:51 EDT 2006
   Type='Part'

   StaticMesh=StaticMesh'USARSim_VehicleParts_Meshes.ATRVJr.MagnetGripperB
tem' 
   DrawScale=4.762
   Mass=0.1

   Begin Object Class=KarmaParamsRBFull Name=KarmaParams1
      KMass=0.007
      bKNonSphericalInertia=True
      KInertiaTensor(0)=0.0008
      KInertiaTensor(3)=0.0008
      KInertiaTensor(5)=0.0008
      KLinearDamping=0
      KAngularDamping=0
      KMaxAngularSpeed=100
      KMaxSpeed=25000
      KActorGravScale=2.58
      KRestitution=0
      bHighDetailOnly=False
      bClientOnly=false
      bKDoubleTickRate=True
      KFriction=0.5
      Name="KarmaParams1"
   End Object

   KParams=KarmaParamsRBFull'USARModels.MagnetGripperBase.KarmaParams1'
}

KParams=KarmaParamsRBFull'USARModels.MagnetGripperBase.KarmaParams1'
// MagnetArmRight.uc
// Written by Andrew Bouchard
// Defines the right link of the MagnetGripper mission package,
// corresponding to part HA-1 in the CAD package
// From folder UT2004\USARMisPkg\Classes

class MagnetArmRight extends MisPkgLinkInfo config(USARMisPkg);

defaultproperties
{
    ModelClass=class'USARModels.MagnetArmRight'
}
// MagnetArmRight.uc
// Written by Andrew Bouchard
// Defines the static mesh for the right link of the MagnetGripper mission package
// From folder UT2004\USARModels\Classes

class MagnetArmRight extends KDPart;

defaultproperties
{
    //Scaled with 4.762 at Mon Sep 25 14:21:51 EDT 2006
    Type='Part'

    StaticMesh=StaticMesh'USARSim_VehicleParts_Meshes.ATRVJr.MagnetArmRight'
        DrawScale=4.762
        Mass=0.1

    Begin Object Class=KarmaParamsRBFull Name=KarmaParams1
        KMass=0.007
        bKNonSphericalInertia=True
        KInertiaTensor(0)=0.0008
        KInertiaTensor(3)=0.0008
        KInertiaTensor(5)=0.0008
        KLinearDamping=0
        KAngularDamping=0
        KMaxAngularSpeed=100
        KMaxSpeed=25000
        KActorGravScale=2.58
        KRestitution=0
        bHighDetailOnly=False
        bClientOnly=false
        bKDoubleTickRate=True
        KFriction=0.5
        Name="KarmaParams1"
    End Object

    KParams=KarmaParamsRBFull'USARModels.MagnetArmRight.KarmaParams1'
}

class MagnetArm_LinkTwo extends MisPkgLinkInfo config(USARMisPkg);

defaultproperties
{
    ModelClass=class'USARModels.MagnetArm_LinkTwo'
}
// MagnetArm_LinkTwo.uc
// Written by Andrew Bouchard
// Defines the static mesh for the second link of the MagnetGripper mission package
// From folder UT2004\USARModels\Classes

class MagnetArm_LinkTwo extends KDPart;

defaultproperties
{
    // Scaled with 4.762 at Mon Sep 25 14:21:51 EDT 2006
    Type='Part'
    StaticMesh=StaticMesh'USARSim_VehicleParts_Meshes.ATRVJr.MagnetArm_LinkTwo'
    DrawScale=4.762
    Mass=0.1

    Begin Object Class=KarmaParamsRBFull Name=KarmaParams1
        KMass=0.007
        bKNonSphericalInertia=True
        KInertiaTensor(0)=0.0008
        KInertiaTensor(3)=0.0008
        KInertiaTensor(5)=0.0008
        KLinearDamping=0
        KAngularDamping=0
        KMaxAngularSpeed=100
        KMaxSpeed=25000
        KActorGravScale=2.58
        KRestitution=0
        bHighDetailOnly=False
        bClientOnly=false
        bKDoubleTickRate=True
        KFriction=0.5
        Name="KarmaParams1"
    End Object

    KParams=KarmaParamsRBFull'USARModels.MagnetArm_LinkTwo.KarmaParams1' }
}
// MagnetArm_LinkThreeA.uc
// Written by Andrew Bouchard
// Defines the third link of the MagnetGripper mission package,
// corresponding to part of the pneumatic actuator
// From folder UT2004\USARMisPkg\Classes

class MagnetArm_LinkThreeA extends MisPkgLinkInfo config(USARMisPkg);

defaultproperties
{
    ModelClass=class'USARModels.MagnetArm_LinkThreeA'
}
// MagnetArm_LinkThreeA.uc  
// Written by Andrew Bouchard  
// Defines the static mesh for the third link of the  
// MagnetGripper mission package  
// From folder UT2004\USARModels\Classes  

class MagnetArm_LinkThreeA extends KDPart;

defaultproperties
{
    Type='Part'
    StaticMesh=StaticMesh'USARSim_VehicleParts_Meshes.TelMax.TelMax Shoulder'
    DrawScale=0.5
    Mass=0.005

    Begin Object Class=KarmaParamsRBFull Name=KarmaParams1
        KLinearDamping=0.700000
        KAngularDamping=0.000000
        KRestitution=0.0
        bHighDetailOnly=False
        bClientOnly=false
        bKDoubleTickRate=True
        KFriction=0.500000
        Name="KarmaParams1"
        KMass=0.005
    End Object

    KParams=KarmaParamsRBFull'USARModels.MagnetArm_LinkThreeA.KarmaParams1'
}


// MagnetArm_LinkFourA.uc
// Written by Andrew Bouchard
// Defines the fourth link of the MagnetGripper mission package,
// corresponding to part of the pneumatic actuator
// From folder UT2004\USARMisPkg\Classes

class MagnetArm_LinkFourA extends MisPkgLinkInfo config(USARMisPkg);

defaultproperties
{
    ModelClass=class'USARModels.MagnetArm_LinkFourA'
}
// MagnetArm_LinkFourA.uc  
// Written by Andrew Bouchard  
// Defines the static mesh for the fourth link of the  
// MagnetGripper mission package  
// From folder UT2004\USARModels\Classes  

class MagnetArm_LinkFourA extends KDPart;  

defaultproperties  
{  
    Type='Part'  
    StaticMesh=StaticMesh'USARSim_VehicleParts_Meshes.TelMax.TelMax Telescope'  
    DrawScale=0.5  
    Mass=0.005  

    Begin Object Class=KarmaParamsRBFull Name=KarmaParams1  
    KLinearDamping=0.700000  
    KAngularDamping=0.000000  
    KRestitution=0.0  
    bHighDetailOnly=False  
    bClientOnly=false  
    bKDoubleTickRate=True  
    KFriction=0.500000  
    Name="KarmaParams1"  
    KMass=0.005  
    End Object  

    KParams=KarmaParamsRBFull'USARModels.MagnetArm_LinkFourA.KarmaParams1'  
}
// MagnetArm_LinkFive.uc
// Written by Andrew Bouchard
// Defines the fifth link of the MagnetGripper mission package,
// corresponding to part JA-2 in the CAD package
// From folder UT2004\USARMisPkg\Classes

class MagnetArm_LinkFive extends MisPkgLinkInfo config(USARMisPkg);

defaultproperties
{
    ModelClass=class'USARModels.MagnetArm_LinkFive'
}
class MagnetArm_LinkFive extends KDPart;

defaultproperties
{
    // Scaled with 4.762 at Mon Sep 25 14:21:51 EDT 2006
    Type='Part'

    StaticMesh=StaticMesh'USARSim_VehicleParts_Meshes.ATRVJr.MagnetArm_LinkFive'
        DrawScale=4.762
        Mass=0.1

    Begin Object
        Class=KarmaParamsRBFull
        Name=KarmaParams1
            KMass=0.007
            bKNonSphericalInertia=True
            KInertiaTensor(0)=0.0008
            KInertiaTensor(3)=0.0008
            KInertiaTensor(5)=0.0008
            KLinearDamping=0
            KAngularDamping=0
            KMaxAngularSpeed=100
            KMaxSpeed=25000
            KActorGravScale=2.58
            KRestitution=0
            bHighDetailOnly=False
            bClientOnly=false
            bKDoubleTickRate=True
            KFriction=0.5
            Name="KarmaParams1"
        End Object

    KParams=KarmaParamsRBFull'USARModels.MagnetArm_LinkFive.KarmaParams1'
}
// SuctionEffecter.uc
// Written by Andrew Bouchard
// Defines an Effecter type that will attach to an Actor
// of type Cargo by calling its Grabbed function on a
// Fire opcode.
// From folder UT2004\USARBot\Classes

class SuctionEffecter extends Effecter config(USARBot);

var(SuctionEffecter) config StaticMesh TableMesh;
var KDHinge hinge;
var KActor table;
var Cargo CarriedCargo; // The actor that the mission package is
carrying

simulated function Init(String SName, Actor parent, vector position,
rotator direction, KVehicle veh, name mount)
{
    local int num;
    local vector X,Y,Z,parentLocation;
    local Rotator parentRotation;
    super.Init(SName,parent, position, direction, veh,mount);
    if(parent!=None)
    {
        parentLocation=Parent.Location;
        parentRotation=Parent.Rotation;
    }
    else
    {
        parentLocation=vect(0,0,0);
        parentRotation=Rotator(vect(0,0,0));
    }
    num=0;
    GetAxes(parentRotation,X,Y,Z);
    if(TableMesh!=None)
    {
        table=spawn(class'USARBot.DummyActor',parent,,parentLocation+position,parentRotation+direction);
        table.SetStaticMesh(TableMesh);
        table.setDrawScale(DrawScale);
        attachTable(table,position);
    }
}

function Bool getFireConf(out EffectorOpcodeConfig opcodeConf)
{
    opcodeConf.opcode=EFFECTOR_OPCODE_FIRE_TYPE;
    opcodeConf.maxVal=0;
}
function Bool getReleaseConf(out EffectorOpcodeConfig opcodeConf) {
    opcodeConf.opcode=EFFECTOR_OPCODE_RELEASE_TYPE;
    opcodeConf.maxVal=0;
    opcodeConf.minVal=0;
    return true;
}

function Bool DoFire(float val) {
    local Actor HitActor;
    if ( CarriedCargo == None ) {
        HitActor = GetNearestActor();
        Log("Cargo Location is "@HitActor.Location);
        if ( HitActor.IsA ('Cargo') ) {
            GrabItem ( self, Cargo(HitActor) );
            Log("Cargo Location is "@CarriedCargo.Location);
            return true;
        } else {
            CarriedCargo = None;
            return false;
        }
    } else {
        return false;
    }
}

function Bool DoRelease(float val) {
    Log("SuctionEffecter: In release - CarriedCargo is "@CarriedCargo);
    if ( CarriedCargo != none) {
        Log("SuctionEffecter: In release - Calling DropItem...");
        DropItem ( self );
    } else {
        return false;
    }
}
{  
  Log( "SuctionEffecter: No Cargo carried - exiting" );  
  return false;  
}

function actor GetNearestActor()
{
  local Actor test, closest;

  foreach VisibleCollidingActors(class 'Actor', test, 20000, location)
  {
    // Only update closest if it is currently nothing, or is  
    // closer than the previously tested actor
    if(closest == none || (VSize(location - test.location) <  
                            VSize(location - closest.location)))
    {
      if ( test.IsA('Cargo') )
      {
        closest=test;
      }
    }

  return closest;
}

// Sets the holding item to be our object and calls the grabbed  
// function of our object.
function GrabItem ( Actor PC, Cargo deco )
{
  deco.Grabbed (PC);
  CarriedCargo = deco;
}

// Sets the holding item to be none and calls the dropped  
// function of our object.
function DropItem (Actor PC)
{
  Log( "SuctionEffecter: In DropItem - CarriedCargo is  
        "@CarriedCargo );
  Log( "Calling Dropped function of "@CarriedCargo );
  Log( "PC location is "@PC.Location );
  Log( "PC rotation is "@PC.Rotation );
  Log( "Cargo location is "@CarriedCargo.Location );
  Log( "Cargo rotation is "@CarriedCargo.Rotation );
  CarriedCargo.Dropped (PC.Location, PC.Rotation);
  CarriedCargo = none;
}

function  attachTable(KActor a,Vector position)
{
  local vector X,Y,Z;
  hinge=spawn(class'KDHinge',Owner,,a.Location);
  hinge.KConstraintActor1=a;
  hinge.KPos1=vect(0,0,0);
hinge.KPriAxis1=vect(1,0,0);
hinge.KSecAxis1=vect(0,1,0);

if(Owner==None || Owner.Physics!=PHYS_Karma ||
   InStr(String(Owner.KParams),"KarmaParamsCollision")!=-1)
{
   GetAxes(a.Rotation,X,Y,Z);
   hinge.KConstraintActor2 = None;
   hinge.KPos2=hinge.Location/50;
   hinge.KPriAxis2=X;
   hinge.KSecAxis2=Y;
}
else
{
   GetAxes(a.Rotation-Owner.Rotation,X,Y,Z);
   hinge.KConstraintActor2 = Owner;
   hinge.KPos2=(position)/50;
   hinge.KPriAxis2=X;
   hinge.KSecAxis2=Y;
}
hinge.KMaxTorque=10000000;
hinge.KHingeType=HT_Controlled;
hinge.KUpdateConstraintParams();
hinge.SetPhysics(PHYS_Karma);
}
simulated event Destroyed()
{
   if(hinge!=None)hinge.Destroy();
   if(table!=None)table.Destroy();
   Super.Destroyed();
}
defaultproperties
{
   DrawScale=1
   bHidden=True
   StaticMesh=StaticMesh'USARSim_Manufacturing_Meshes.conveyor.conveyorrollers'
   DrawType=DT_StaticMesh
   TableMesh=None
}
// ResearchSuctionEffecter.uc
// Written by Andrew Bouchard
// Defines a particular instance of the SuctionEffecter
// class with its own static mesh
// From folder UT2004\USARBot\Classes

class ResearchSuctionEffecter extends SuctionEffecter;

defaultproperties
{
    TableMesh=StaticMesh'USARSim_Objects_Meshes.VictCollision.StandBo

// Cargo.uc
// Written by Andrew Bouchard
// Defines a class Cargo inheriting from KActor that can be
// picked up and manipulated using an Effecter of type
// SuctionEffecter
// From folder UT2004\USARBot\Classes

class Cargo extends KActor;

var int id;
var Actor m_oldOwner, m_oldBase;
var bool m_pickedUp;

simulated event PostBeginPlay()
{
    local int i;

    if(USARDeathMatch(Level.Game).Cargo.length>1 &&
    {
        for(i=1;i<USARDeathMatch(Level.Game).Cargo.length;i++)
        {
            if(USARDeathMatch(Level.Game).Cargo[i].id-
            USARDeathMatch(Level.Game).Cargo[i-1].id>1)
            {
                id=USARDeathMatch(Level.Game).Cargo[i-1].id+1;
                USARDeathMatch(Level.Game).Cargo.Insert(i,1);
                USARDeathMatch(Level.Game).Cargo[i]=self;
                return;
            }
        }
    }
    if(USARDeathMatch(Level.Game).Cargo.length>0)
    {
        id=USARDeathMatch(Level.Game).Cargo[USARDeathMatch(Level.Game).Cargo.length-1].id+1;
        else
        {
            id=0;
            log(USARDeathMatch(Level.Game).Cargo[USARDeathMatch(Level.Game).Cargo.length-1].id@USARDeathMatch(Level.Game).Cargo.length-1);
        }
}

// Records the old owner, removes collisions
event Grabbed ( Actor newOwner )
{
    setphysics( PHYS_None );
    SetCollision ( false, false, false );
    SetOwner ( newOwner );
    SetBase ( newOwner );
    SetCollision ( true, true, true );
}
event Dropped (vector newLoc, rotator newRot)
{
    local vector newlocation;
    // Set the new location to be the players but in front of him.
    //newlocation = newLoc + vector(newRot)*50;
    //Log( "Cargo/Dropped: newlocation = "@newlocation );
    //SetLocation ( self.Location );
    //SetRotation ( self.Rotation );
    SetCollision (true, true, true);
    Log( "SetLocation and SetCollision successful" );
    //setphysics(PHYS_None);
    //Set the Owner and Base to those prior to being grabbed.
    Log( "Cargo/Dropped: Setting owner to "@m_oldOwner );
    SetOwner ( m_oldOwner );
    Log( "Cargo/Dropped: Setting base to "@m_oldBase );
    SetBase ( m_oldBase );
    bHidden = false;
    // set falling physics so when the item is dropped it falls to
    // the floor
    //setphysics(PHYS_Karma);
    m_pickedUp = false;
}

simulated event Destroyed()
{
    Super.Destroyed();
}

// slightly modified from actor.uc to add log statement
simulated event FellOutOfWorld(eKillZType KillType)
{
    log(self"fell out of world at"@location);
    SetPhysics(PHYS_None);
    Destroy();
}

// maybe its colliding with something else?
event EncroachedBy( actor Other )
{
    log(self"encroached by"@ other @"at"@location);
    super.EncroachedBy(Other);
}

/* DisplayDebug()
list important actor variable on canvas. HUD will call DisplayDebug()
on the current ViewTarget when
the ShowDebug exec is used */
simulated function DisplayDebug(Canvas Canvas, out float YL, out float YPos) 
{
    local string T;
    local float XL;
    local int i;
    local Actor A;
    local name anim;
    local float frame, rate;

    Canvas.Style = ERenderStyle.STY_Normal;
    Canvas.StrLen("TEST", XL, YL);
    YPos = YPos + YL;
    Canvas.SetPos(4, YPos);
    Canvas.SetDrawColor(255, 0, 0);
    T = GetDebugName();
    if (bDeleteMe)
        T = T" DELETED (bDeleteMe == true)";

    Canvas.DrawString(T, false);
    YPos += YL;
    Canvas.SetPos(4, YPos);
}

if (Level.NetMode != NM_Standalone)
{
    // networking attributes
    T = "ROLE ";
    Switch(Role)
    {
        case ROLE_None: T = T"None"; break;
        case ROLE_DumbProxy: T = T"DumbProxy"; break;
        case ROLE_SimulatedProxy: T = T"SimulatedProxy";
        break;
        case ROLE_AutonomousProxy: T = T"AutonomousProxy";
        break;
        case ROLE_Authority: T = T"Authority"; break;
    }
    T = T" REMOTE ROLE ";
    Switch(RemoteRole)
    {
        case ROLE_None: T = T"None"; break;
        case ROLE_DumbProxy: T = T"DumbProxy"; break;
        case ROLE_SimulatedProxy: T = T"SimulatedProxy";
        break;
        case ROLE_AutonomousProxy: T = T"AutonomousProxy";
        break;
        case ROLE_Authority: T = T"Authority"; break;
    }
    if (bTearOff)
        T = T" Tear Off";
    Canvas.DrawString(T, false);
    YPos += YL;
    Canvas.SetPos(4, YPos);
}

T = "Physics ";
Switch(PHYSICS)
case PHYS_None: T=T"None"; break;
case PHYS_Walking: T=T"Walking"; break;
case PHYS_Falling: T=T"Falling"; break;
case PHYS_Swimming: T=T"Swimming"; break;
case PHYS_Flying: T=T"Flying"; break;
case PHYS_Rotating: T=T"Rotating"; break;
case PHYS_Projectile: T=T"Projectile"; break;
case PHYS_Interpolating: T=T"Interpolating"; break;
case PHYS_MovingBrush: T=T"MovingBrush"; break;
case PHYS_Spider: T=T"Spider"; break;
case PHYS_Trailer: T=T"Trailer"; break;
case PHYS_Ladder: T=T"Ladder"; break;
case PHYS_Karma: T=T"Karma"; break;

T = T" in physicsvolume "$GetItemName(string(PhysicsVolume))$"
on base "$GetItemName(string(Base));
if ( bBounce )
  T = T" - will bounce";
Canvas.DrawText(T, false);
YPos += YL;
Canvas.SetPos(4,YPos);
Canvas.DrawText("Location: "$Location$ Rotation "$Rotation,
false);
YPos += YL;
Canvas.SetPos(4,YPos);
Canvas.DrawText("Velocity: "$Velocity$ Speed "$VSize(Velocity)$" Speed2D "$VSize(Velocity-Velocity.Z*vec(0,0,1)), false);
YPos += YL;
Canvas.SetPos(4,YPos);
Canvas.DrawText("Acceleration: "$Acceleration, false);
YPos += YL;
Canvas.SetPos(4,YPos);

Canvas.DrawColor.B = 0;
Canvas.DrawText("Collision Radius "$CollisionRadius$" Height "$CollisionHeight);
YPos += YL;
Canvas.SetPos(4,YPos);

Canvas.DrawText("Collides with Actors "$bCollideActors$, world "$bCollideWorld$", proj. target "$bProjTarget);
YPos += YL;
Canvas.SetPos(4,YPos);
Canvas.DrawText("Blocks Actors "$bBlockActors);
YPos += YL;
Canvas.SetPos(4,YPos);

T = "Touching ";
ForEach TouchingActors(class'Actor', A)
  T = T$GetItemName(string(A))$ " ";
if ( T == "Touching ")
  T = "Touching nothing";
Canvas.DrawText(T, false);
YPos += YL;
Canvas.SetPos(4,YPos);
Canvas.DrawColor.R = 0;
T = "Rendered: ";
Switch(Style)
{
    case STY_None: T=T; break;
    case STY_Normal: T=T$"Normal"; break;
    case STY_Masked: T=T$"Masked"; break;
    case STY_Translucent: T=T$"Translucent"; break;
    case STY_Modulated: T=T$"Modulated"; break;
    case STY_Apha: T=T$"Alpha"; break;
}

Switch(DrawType)
{
    case DT_None: T=T$" None"; break;
    case DT_Sprite: T=T$" Sprite "; break;
    case DT_Mesh: T=T$" Mesh "; break;
    case DT_Brush: T=T$" Brush "; break;
    case DT_RopeSprite: T=T$" RopeSprite "; break;
    case DT_VerticalSprite: T=T$" VerticalSprite "; break;
    case DT_Terraform: T=T$" Terraform "; break;
    case DT_SpriteAnimOnce: T=T$" SpriteAnimOnce "; break;
    case DT_StaticMesh: T=T$" StaticMesh "; break;
}

if ( DrawType == DT_Mesh )
{
    T = T$GetItemName(string(Mesh));
    if ( Skins.length > 0 )
    {
        T = T$" skins: ";
        for ( i=0; i<Skins.length; i++ )
        {
            if ( skins[i] == None )
                break;
            else
                T = T$GetItemName(string(skins[i]))$", ";
        }
    }
    Canvas.DrawText(T, false);
    YPos += YL;
    Canvas.SetPos(4,YPos);
    // mesh animation
    GetAnimParams(0,Anim,frame,rate);
    T = "AnimSequence $Anim$ Frame $frame$ Rate $rate;
    if ( bAnimByOwner )
        T = T$" Anim by Owner";
    }
else if ( (DrawType == DT_Sprite) || (DrawType == DT_SpriteAnimOnce) )
    T = T$Texture;
else if ( DrawType == DT_Brush )
    T = T$Brush;
Canvas.DrawText(T, false);
YPos += YL;
Canvas.SetPos(4,YPos);

Canvas.DrawColor.B = 255;
Canvas.DrawText("Tag: "$Tag" Event: "$Event" STATE: "$GetStateName(), false);
YPos += YL;
Canvas.SetPos(4,YPos);

Canvas.DrawText("Instigator "$GetItemName(string(Instigator))$" Owner "$GetItemName(string(Owner))$" Base "$GetItemName(string(Base))");
YPos += YL;
Canvas.SetPos(4,YPos);

Canvas.DrawText("Location relative to owner: "$RelativeLocation");
YPos += YL;
Canvas.SetPos(4,YPos);

Canvas.DrawText("Timer: "$TimerCounter" LifeSpan "$LifeSpan" AmbientSound "$AmbientSound" volume "$SoundVolume");
YPos += YL;
Canvas.SetPos(4,YPos);

}
defaultproperties
{
    bNoDelete=false
    bStasis=True
    Begin Object Class=KarmaParamsRBFull Name=KParams0
        KStartEnabled=True
        bKDoubleTickRate=True
        bHighDetailOnly=False
        bClientOnly=False
        KActorGravScale=2.58
        KMaxAngularSpeed=100
        KMaxSpeed=25000
        KMass=0.1
        bKNonSphericalInertia=True
        //7x7x7 cm cube inertia tensor
        KInertiaTensor(0)=0.0008
        KInertiaTensor(1)=0
        KInertiaTensor(2)=0
        KInertiaTensor(3)=0.0008
        KInertiaTensor(4)=0
        KInertiaTensor(5)=0.0008
        KLinearDamping=0.0
        KAngularDamping=0.0
        KFriction=0.8
        KRestitution=0.3
        Name="KParams0"
    End Object
    KParams=KarmaParamsRBFull'KParams0'
}
// USARContainer.uc
// Written by Andrew Bouchard
// Defines a particular instance of the Cargo class that
// uses a panel for its static mesh
// From folder UT2004\USARBot\Classes

class USARContainer extends Cargo;

defaultproperties
{
    StaticMesh=StaticMesh'USARSim_Objects_Meshes.Objects.Panel'
    DrawScale3D=(X=1.0,Y=1.0,Z=1.0)
    Begin Object Class=KarmaParams Name=KarmaParams0
        KMass=1.000000
        bHighDetailOnly=False
        bClientOnly=False
        bKDoubleTickRate=True
        KStartEnabled=True
        KAngularDamping=0.3
        KLinearDamping=0.1
        KFriction=0.150000
        KScale=0.5
        KRestitution=0.200000
        KImpactThreshold=100000.000000
    End Object
    KParams=KarmaParams'USARBot.USARContainer.KarmaParams0'
}
APPENDIX D

CONTROLLER CODE
/* amShell.cc
This code was modified by Andrew Bouchard for use in Masters research at Vanderbilt University. It implements control via a hybrid automaton and all supporting functions including panel detection.
*/

/**************************************************************************************************/

// Added for sensor data acquisition
#include "primSP.hh"
#include "sensorData.hh" // defines sensor data types
#include "LinkedList.h"

// Added for servo level control
#include <string>
#include <iostream>
#include <vector>
#include <sstream>
using namespace std;
#include "servoMobJA.hh" // servoMobJA_format()
#include "servoMisJA.hh" // servoMisJA_format()
#include "servoEffJA.hh" // servoEffJA_format()
#include "ServoNodeNmlBuffers.h"

static NML *navDataExtBuf = NULL;
// Added for sensor data acquisition
static NML *sensorDataListBuf = NULL;
//static NML *sensorData1DBuf = NULL;
static RCS_CMD_CHANNEL * amMobJACmdBuf = NULL;
static RCS_STAT_CHANNEL * amMobJAStatBuf = NULL;
static RCS_CMD_CHANNEL * amMobJACfgBuf = NULL;
static RCS_STAT_CHANNEL * amMobJASetBuf = NULL;
static RCS_CMD_CHANNEL * amSPCmdBuf = NULL;
static RCS_STAT_CHANNEL * amSPStatBuf = NULL;
static RCS_CMD_CHANNEL * amSPCfgBuf = NULL;
static RCS_STAT_CHANNEL * amSPSetBuf = NULL;

// Servo NML buffers for servo-level control when needed
static RCS_CMD_CHANNEL *servoMobJACmdBuf = NULL;
static RCS_STAT_CHANNEL *servoMobJAStatBuf = NULL;
static RCS_STAT_CHANNEL *servoMobJACfgBuf = NULL;
static RCS_STAT_CHANNEL *servoMobJASetBuf = NULL;

static AmMobJAStat * amMobJAStat;
static AmMobJASet * amMobJASet;

static AmMobJACmdInit amMobJACmdInit;
static AmMobJACmdHalt amMobJACmdHalt;
static AmMobJACmdAbort amMobJACmdAbort;
static AmMobJACmdShutdown amMobJACmdShutdown;
static AmMobJACmdNop amMobJACmdNop;
static AmMobJACmdSpin amMobJACmdSpin;
static AmMobJACmdPrepareMove amMobJACmdPrepareMove;
static AmMobJACmdMoveWaypoint amMobJACmdMoveWaypoint;
static AmMobJACfgCellResolution amMobJACfgCellResolution;
static AmMobJACfgCycleTime amMobJACfgCycleTime;
static AmMobJACfgDebug amMobJACfgDebug;
static AmMobJACfgDumpWM amMobJACfgDumpWM;
static AmMobJACfgPlanHorizon amMobJACfgPlanHorizon;
static AmMobJACfgVehicleMinTurnRad amMobJACfgVehicleMinTurnRad;
static AmMobJACfgVehicleWidth amMobJACfgVehicleWidth;
static AmMobJACfgNop amMobJACfgNop;

static AmSPStat * amSPStat;
static AmSPSet * amSPSet;

static AmSPCmdInit amSPCmdInit;
static AmSPCmdHalt amSPCmdHalt;
static AmSPCmdAbort amSPCmdAbort;
static AmSPCmdShutdown amSPCmdShutdown;
static AmSPCmdGo amSPCmdGo;
static AmSPCmdNop amSPCmdNop;
static AmSPCfgCycleTime amSPCfgCycleTime;
static AmSPCfgDebug amSPCfgDebug;
static AmSPCfgDumpWM amSPCfgDumpWM;
static AmSPCfgNop amSPCfgNop;

static char *INI_FILE = NULL;
static char *NML_FILE = NULL;

static AmMobJASet set; // config command status
message
double vehSpeed; // m/s XY speed of vehicle
double vehYaw; // rad/s rate of change of vehYaw
static PM_CARTESIAN vehVelocity; // m/s vector in local coordinates
double vehX; // vehicle local X coordinate
double vehY; // vehicle local Y coordinate
double vehYaw; // rad angle CCW (seen from +Z) from local +X
static PM_POSE localToGlobal; // transform local to global coords
static PM_POSE globalToLocal; // transform global to local coords
double gPanel[6] = {0, 0, 0, 0, 0, 0};
double gSensorPos[2] = {0, 0};
double gP = 0;
double gTheta = 0;
double gRightAngle;
double gRightDist;
double gEndAngle;
double gEndDist;
double PanelAngle;
double RightPanelM, RightPanelB, RightPanelTheta, RightPanelP;
double LeftPanelM, LeftPanelB, LeftPanelTheta, LeftPanelP;
double IntersectionX, IntersectionY;
double RightPanel[4], LeftPanel[4];
char * cfgFile;
int serial_number_sent = 0;
int eff_serial_number = 0;
ServoEffJACmdOpcode effCmdOpcode;
ServoEffJACmdInit effCmdInit;
ServoEffJACmdHalt effCmdHalt;
ServoEffJACmdAbort effCmdAbort;
ServoEffJACmdShutdown effCmdShutdown;

ServoNodeNmlBuffers * eff = NULL;

int mis_serial_number = 0;
ServoMisJACmdInit misCmdInit;
ServoMisJACmdHalt misCmdHalt;
ServoMisJACmdAbort misCmdAbort;
ServoMisJACmdShutdown misCmdShutdown;
ServoMisJACmdMove misCmdMove;

vector<ServoNodeNmlBuffers> mis;

// Supporting functions
static void open270( );
static double absVal( double num );
static void doBrake();
static void findPanelIntersection( double &x, double &y );
static double getLidarHits( list &hits );
static void doHough( list &hits, double angleRes, double distRes,
                    double &bestTheta, double &bestP, double maxDist,
                    double excludeTheta = 360, double excludeP = 1000 );
static int removeOutliers( list &onLine, int doX, int doY);
static void processVertical( list &hits, list &panel, double theta,
                            double p, double distRes, double end1[2], double end2[2]);
static void processHorizontal( list &hits, list &panel, double theta,
                             double p, double distRes, double end1[2], double end2[2]);
static void processSlanted( list &hits, list &panel, double theta,
                           double p, double distRes, double end1[2], double end2[2]);
static int getPanel( list &hits, double &maxDist );
static int getTwoPanels( list &hits, double &maxDist, double left[4],
                        double &leftTheta, double right[4], double &rightTheta);
static void calcJoinerApproach( double x, double y, double &final_x,
                                double &final_y );
static double goTo( double x, double y, double speed);
static int updatePosition();
static double minusPiToPi( double angle );
static void spinToAngle( double angle, double error);
static void mergeSort( list &sortMe, int doX );
static void mergeX( list &left, list &right, list &result );
static void mergeY( list &left, list &right, list &result );

// For joiner hybrid automaton
static void HAJoiner();
static int exJoinerWait();
static int exJoinerGetClose();
static int exJoinerSearch();
static int exJoinerApproach();
static int exJoinerAlign();
static int exAlignPanel();
static int exMovePanel();
static int exCheckAlign();
static int exCheckMove();
static int exJoinerAcquire();
static int exJoinerTogether();
static int exReleasePanels();
static int exJoinerCalcAndSend();

/***************************************************************************/
static int updatePosition() /* NO ARGUMENTS */
{
    int dataType;
    NavDataExt * navData;
    int retVal;
    PMPOSE pose;
    RCS_TIMER * timer;

    // Initialize the timer
    timer = new RCS_TIMER(0.1);

    dataType = (int)navDataExtBuf->read();
    while ( dataType != NAV_DATA_EXT_TYPE )
    {
        timer->wait();
        dataType = (int)navDataExtBuf->read();
    }

    navData = (NavDataExt *)navDataExtBuf->get_address();
    pose.tran = navData->tranRel;
    pose.rot = navData->rpyRel;
    vehX = pose.tran.x;
    vehY = pose.tran.y;
    vehYaw = minusPiToPi(navData->rpyRel.y);
    vehSpin = navData->rpyRelRate.y;
    vehVelocity.x = navData->tranRelRate.x;
    vehVelocity.y = navData->tranRelRate.y;
    vehSpeed = hypot(vehVelocity.x, vehVelocity.y);
    globalToLocal = navData->absToRel;
    localToGlobal = inv(navData->absToRel);
    retVal = NAV_DATA_EXT_TYPE;

    delete timer;

    return retVal;
}

/***************************************************************************/

/*! minusPiToPi takes an angle and returns an angle in the range [-Pi, Pi)
that is equal to the original angle plus or minus a multiple of 2Pi. */
static double minusPiToPi( double angle ) /*!< the angle to get in range */
{
    return ((angle < -M_PI) ? minusPiToPi(angle + (2 * M_PI)) :
(angle >  M_PI) ? minusPiToPi(angle - (2 * M_PI)) :
   (angle == M_PI) ? -M_PI : angle);
}  

/*********************************************************************/
/*! absVal returns the absolute value of a double passed to it */

static double absVal( double num )
{
   if ( num < 0 )
      return -num;
   else
      return num;
}
/*********************************************************************/

/*/! doBrake brakes the vehicle from the servo level*/

static void doBrake( ) /* NO ARGUMENTS */
{
    // Declare variables
    ServoMobJACmdSkid skidMsg;
    ServoMobJAStat * stat;
    int serial_number, command_type_sent;
    RCS_TIMER * timer;

    // Initialize the timer and state buffer
    timer = new RCS_TIMER(0.1);
    stat = (ServoMobJAStat *) (servoMobJAStatBuf->get_address());
    servoMobJAStatBuf->read();

    // Just to be sure, do the reverse-stop thing again
    skidMsg.wLeft = -0.001;
    skidMsg.wRight = -0.001;
    command_type_sent = skidMsg.type;
    setOddSerial(stat->echo_serial_number, &serial_number);
    skidMsg.serial_number = serial_number;
    servoMobJACmdBuf->write(&skidMsg);

    // Wait for five seconds
    fflush(stdout);
    for ( int i = 0; i < 50; i++ )
       timer->wait();

    // Send zero velocity command
    fflush(stdout);
    servoMobJAStatBuf->read();
    skidMsg.wLeft = 0;
    skidMsg.wRight = 0;
    command_type_sent = skidMsg.type;
    setOddSerial(stat->echo_serial_number, &serial_number);
    skidMsg.serial_number = serial_number;
    servoMobJACmdBuf->write(&skidMsg);

    // Clean up

    // Clean up
static void doFlex() /* NO ARGUMENTS */
{
    // Declare variables
    ServoMobJACmdSkid skidMsg;
    ServoMobJAStat * stat;
    RCS_TIMER * timer;
    int count = 0;
    int command_type_sent, serial_number;
    double misPos, last_misPos;

    // Initialize the timer and state buffer
    timer = new RCS_TIMER(0.1);
    stat = (ServoMobJAStat *) (servoMobJAStatBuf->get_address());
    servoMobJAStatBuf->read();

    // Turn on the "brakes", since the prismatic link can kick the robot a little
    skidMsg.wLeft = -0.001;
    skidMsg.wRight = -0.001;
    command_type_sent = skidMsg.type;
    setOddSerial(stat->echo_serial_number, &serial_number);
    skidMsg.serial_number = serial_number;
    servoMobJACmdBuf->write(&skidMsg);

    // Extend the prismatic link
    command_type_sent = misCmdMove.type;
    misCmdMove.serial_number = mis_serial_number++;
    misCmdMove.linkCmd_length = 1;
    misCmdMove.linkCmd[0].linkID = 4;
    misCmdMove.linkCmd[0].value = 1.0;
    misCmdMove.linkCmd[0].type=SERVO_MIS_JA_LINK_CMD_ABS_VALUE_TYPE;
    mis[0].cmdBuf->write(misCmdMove);

    // Wait for the command to finish executing before continuing
    while (1)
    {
        timer->wait();
        mis[0].statBuf->read();
        misPos = ((ServoMisJAStat *)mis[0].statMsg)->linkStat[3].jointVal;
        if (mis[0].statMsg->command_type == command_type_sent &&
            misPos == last_misPos &&
            mis[0].statMsg->echo_serial_number == (mis_serial_number - 1))
        {
            count++;
            if (count > 5)
                break;
        }
    }
}
else {
    count = 0;
}
last_misPos = misPos;

// Retract the prismatic link
command_type_sent = misCmdMove.type;
misCmdMove.serial_number = mis_serial_number++;
misCmdMove.linkCmd_length = 1;
misCmdMove.linkCmd[0].linkID = 4;
misCmdMove.linkCmd[0].value = 0;
misCmdMove.linkCmd[0].type = SERVO_MIS_JA_LINK_CMD_ABS_VALUE_TYPE;
mis[0].cmdBuf->write(misCmdMove);

// Wait for the command to finish executing before continuing
for (int i = 0; i < 100; i++) {
    timer->wait();
}
while (1) {
    timer->wait();
mis[0].statBuf->read();
misPos = ((ServoMisJAStat *)mis[0].statMsg)->linkStat[3].jointVal;
    if (mis[0].statMsg->command_type == command_type_sent &&
        misPos == last_misPos && mis[0].statMsg->echo_serial_number == (mis_serial_number - 1)) {
        count++;
        if (count > 5)
            break;
    }
else {
    count = 0;
}
last_misPos = misPos;

// Send zero velocity command
servoMobJAStatBuf->read();
skidMsg.wLeft = 0;
skidMsg.wRight = 0;
command_type_sent = skidMsg.type;
setOddSerial(stat->echo_serial_number, &serial_number);
skidMsg.serial_number = serial_number;
servoMobJACmdBuf->write(&skidMsg);
return;
}

/******************************************************************************/
/!* open270 opens the Joiner manipulator to its fully open position*/

static void open270( ) /* NO ARGUMENTS */
{
  // Declarations
  ServoMobJAStat * stat;
  RCS_TIMER * timer;
  int count = 0;
  int command_type_sent;
  double misPos, last_misPos;
  
timer = new RCS_TIMER(0.1);

  // Open the mission package
  command_type_sent = misCmdMove.type;
  misCmdMove.serial_number = mis_serial_number++;
  misCmdMove.linkCmd_length = 1;
  misCmdMove.linkCmd[0].linkID = 1;
  misCmdMove.linkCmd[0].value = (5 * M_PI) / 8;
  misCmdMove.linkCmd[0].type = SERVO_MIS_JA_LINK_CMD_ABS_VALUE_TYPE;
  mis[0].cmdBuf->write(misCmdMove);

  // Wait for the command to finish executing before continuing
  while (1)
  {
    timer->wait();
    mis[0].statBuf->read();
    misPos = ((ServoMisJAStat *)mis[0].statMsg)->linkStat[0].jointVal;
    if (mis[0].statMsg->command_type == command_type_sent &&
        misPos == last_misPos && mis[0].statMsg->echo_serial_number == (mis_serial_number - 1))
    {
      count++;
      if (count > 5)
        break;
    }
    else
    {
      count = 0;
    }
  
  last_misPos = misPos;
  
  command_type_sent = misCmdMove.type;
  misCmdMove.serial_number = mis_serial_number++;
  misCmdMove.linkCmd_length = 1;
  misCmdMove.linkCmd[0].linkID = 2;
  misCmdMove.linkCmd[0].value = -(5 * M_PI) / 8;
  misCmdMove.linkCmd[0].type = SERVO_MIS_JA_LINK_CMD_ABS_VALUE_TYPE;
  mis[0].cmdBuf->write(misCmdMove);

  // Wait for the command to finish executing before continuing
  for (int i = 0; i < 100; i++)
  {
    timer->wait();
  }
}
while (1) {
    timer->wait();
    mis[0].statBuf->read();
    misPos = ((ServoMisJAStat *)mis[0].statMsg)->linkStat[1].jointVal;
    if (mis[0].statMsg->command_type == command_type_sent &&
        misPos == last_misPos && mis[0].statMsg->echo_serial_number == (mis_serial_number - 1)) {
        count++;
        if (count > 5)
            break;
    } else {
        count = 0;
    }
    last_misPos = misPos;
}

delete timer;

/**************************************************************************/

/*! findPanelIntersection finds the intersection of the two most recently identified panels and returns it as x and y coordinates*/

static void findPanelIntersection( double &x, double &y ) /* NO ARGUMENTS */
{
    // Left panel is horizontal
    if ( absVal( LeftPanelTheta ) < 0.02 ) {
        y = LeftPanelP;
        // Right panel is horizontal
        if ( absVal( RightPanelTheta ) < 0.02 ) {
            x = RightPanel[0];
            y = RightPanel[1];
        } else if ( absVal( absVal( RightPanelTheta ) - ( M_PI / 2 ) ) < 0.02 ) {
            x = RightPanelP;
        } else { // Right panel is slanted
            x = ( y - RightPanelB ) / RightPanelM;
        }
    } else { // Right panel is vertical
        x = RightPanelP;
    }
    return;
}
// Left panel is vertical
if ( absVal( absVal( LeftPanelTheta ) - ( M_PI / 2 ) ) < 0.02 ) {
    x = LeftPanelP;
}

// Right panel is vertical
if ( absVal( absVal( RightPanelTheta ) - ( M_PI / 2 ) ) < 0.02 ) {
    x = RightPanel[0];
    y = RightPanel[1];
}

// Right panel is horizontal
else if ( absVal( RightPanelTheta ) < 0.02 ) {
    y = RightPanelP;
}

// Right panel is slanted
else {
    y = RightPanelM * x + RightPanelB;
}
return;

// Right panel is horizontal, left is slanted
if ( absVal( RightPanelTheta ) < 0.02 ) {
    y = RightPanelP;
    x = ( y - LeftPanelB ) / LeftPanelM;
    return;
}

// Right panel is vertical, left is slanted
if ( absVal( absVal( RightPanelTheta ) - ( M_PI / 2 ) ) < 0.02 ) {
    x = RightPanelP;
    y = LeftPanelM * x + LeftPanelB;
    return;
}

// Both panels are slanted
x = ( RightPanelB - LeftPanelB ) / ( LeftPanelM - RightPanelM );
y = RightPanelM * x + RightPanelB;
return;

/*******************************************************************************/

/*! getLidarHits reads the sensor data buffer to get the lidar data and
returns a linked list of all the sensor points that are
hits on
some object*/

static double getLidarHits( list &hits ) /* NO ARGUMENTS */ {
    // Variable declarations
int go;
double x, y; // coordinates of data point
double range; // distance sensor point to data

double maxDist = 0;
double hit[2];
SensorDataList *input;
SensorPoint * item;
RCS_TIMER * timer;

// Clear existing lidar data
for (int index = 0; 0 < hits.count();)
{
    hits.access(0, hit);
    hits.del(hit[0], hit[1]);
}

// Initialize the timer
timer = new RCS_TIMER(0.1);

while (1)
{
    // Check for good buffer data
    go = sensorDataListBuf->read();
    if (go == SENSOR_DATA_LIST_TYPE)
        break;

    // If the data isn't good, wait a tick and try again
    timer->wait();
}

// Clean up
delete timer;

// Initialize the maximum distance
maxDist = 0;

// Read data from buffers
input = (SensorDataList *) sensorDataListBuf->get_address();
for (int step = 0; step < input->featList_length; step++)
{
    // Look at each element in the incoming buffer
    item = (input->featList + step);
    x = item->dataLoc.x;
    y = item->dataLoc.y;
    gSensorPos[0] = item->sensorLoc.x;
    gSensorPos[1] = item->sensorLoc.y;
    range = sqrt((gSensorPos[0] - x) * (gSensorPos[0] - x) +

    if (range < (0.95 * input->maxRange))
    {
        hits.append(x, y);
        if (range > maxDist)
        {
            maxDist = range;
        }
    }
static void doHough( list &hits, double angleRes, double distRes, double &bestTheta, double &bestP, double maxDist, double excludeTheta, double excludeP )
{
    // Variable declaration
    double theta_rad;
    double hit[2];
    double p, m, b;
    double radicand;
    double xc, yc;
    int sign = 1;
    int theta_len, r_len;
    int thetaIndex, rIndex;
    int bestVal = 0;

    // Calculate the size of the accumulator array and declare the array
    theta_len = static_cast<int> (180/angleRes) + 1;
    r_len = static_cast<int> ((maxDist*2)/distRes) + 1;
    int A[theta_len][r_len];

    // Initialize the accumulator array to all zeros
    for (int g = 0; g<theta_len; g++)
    {
        for (int h = 0; h<r_len; h++)
        {
            A[g][h] = 0;
        }
    }

    // Iterate through all the data points and for each calculate the possible lines through them
    for (int i = 0; i < hits.count(); i++)
    {
        hits.access(i, hit);

        // Re-center the data to the sensor
        hit[0] = hit[0] - gSensorPos[0];
        hit[1] = hit[1] - gSensorPos[1];

        for (double theta = 0; theta < 180; theta = theta + angleRes)
        {
            // Assign values for line description
            theta_rad = theta * (3.14159/180);
            }
// Calculate the distance p for the line described as going through point (x,y)
// with a normal that crosses the origin with angle theta off the positive x-axis
// and round to the nearest distRes
p = hit[0]*cos(theta_rad) + hit[1]*sin(theta_rad);
p = static_cast<int>(p/distRes);
p = p*distRes;

// Convert values to array indices
thetaIndex = static_cast<int>(theta/angleRes);
rIndex = static_cast<int>(((p+maxDist)/distRes);

// Increment the associated accumulator entry, keeping track of the maximum
A[thetaIndex][rIndex] = A[thetaIndex][rIndex] + 1;
if ( ( A[thetaIndex][rIndex] > bestVal ) && ( p != excludeP ) && ( theta != excludeTheta ) )
{
    bestVal = A[thetaIndex][rIndex];
    bestTheta = theta;
    bestP = p;
}

// Case for a vertical line
if ( bestTheta == 90 )
{
    bestP = bestP + gSensorPos[1];
}
// Case for a horizontal line
else if ( bestTheta == 0 )
{
    bestP = bestP + gSensorPos[0];
}
// Case for a slanted line
else
{
    xc = gSensorPos[0];
    yc = gSensorPos[1];
    theta_rad = (90-bestTheta) * (3.14159/180);
    m = sin(theta_rad)/cos(theta_rad);
    if ( ( m>0 ) && ( bestTheta>0 ) && ( bestTheta < 90 ) )
    {
        m = -m;
    }
    if ( ( m<0 ) && ( bestTheta>90 ) && ( bestTheta < 180 ) )
    {
        m = -m;
    }
    radicand = bestP*bestP + m*m*bestP*bestP + ( yc - m*xc )*( yc - m*xc ) - m*m*xc*xc + 2*m*xc*yc - yc*yc;
    if ( radicand < 0 )
    {
        radicand = -radicand;
    }
}
if ( bestP < 0 )
{
   sign = -1;
}

b = yc - m*xc + sign*sqrt( radicand );
bestP = b * cos(theta_rad);
}

return;
}

/****************************************************************************

/*! removeOutliers takes the linked list of hits on the line and
removes outliers
using a standard deviation analysis*/
static int removeOutliers( list &onLine, int doX, int doY)
{
   // Declare variables
   list temp, errors;
   double transfer[2];
   double pt1[2], pt2[2], pt3[2];
   double x, y, error;
   double m, b, avg;
   double sum = 0;
   double mean, stddev;
   double distRight, distLeft;
   int breakpt = onLine.count();
   int offset;
   int isBreak = 0;

   // Sort the list
   mergeSort( onLine, doX );

   // Find the first break in the line based on the distances
   between adjacent points
   for ( int i = 1; i < onLine.count() - 1; i++ )
   {
      // Get the point and both its neighbors
      onLine.access( i - 1, pt1 );
      onLine.access( i , pt2 );
      onLine.access( i + 1, pt3 );

      // Calculate the distances to either side
      distLeft = sqrt( ( pt2[0] - pt1[0] ) * ( pt2[0] - pt1[0] )
      distRight = sqrt( ( pt3[0] - pt2[0] ) * ( pt3[0] - pt2[0] )
      fflush(stdout);

      // Identify first break in the line
      if( ( distLeft > 0.01 ) && ( distRight > 0.01 ) )
      {
         fflush(stdout);
         if( distLeft > ( 1.5 * distRight ) )
         {
            breakpt = i;
         } 
   }
```cpp
    isBreak = 1;
    break;
}

if( distRight > ( 1.5 * distLeft ) )
{
    breakpt = i + 1;
    isBreak = 1;
    break;
}
}

breakpt = onLine.count();

// Copy the portion of the list up to the break to the temporary list
for ( int j = 0; j < breakpt; j++ )
{
    onLine.access( j, transfer );
    temp.append( transfer[0], transfer[1] );
}

// Use a standard deviation analysis to remove those points with too much error off the predicted line
// Note that this will only work with some number of points
sum = 0;
if ( temp.count() > 5 )
{
    // Calculate a new predicted line with from entries inside the line
    offset = static_cast<int> ( ceil( temp.count() / 10 ) );
    temp.access( offset, pt1 );
    temp.access( ( temp.count() - offset - 1 ), pt2 );

    // Slanted line case
    if ( doX && doY )
    {
        m = ( pt2[1] - pt1[1] ) / ( pt2[0] - pt1[0] );
        b = pt1[1] - m * pt1[0];
        // Make a list that contains all the error entries
        for ( int j = 0; j < temp.count(); j++ )
        {
            temp.access( j, pt1 );
            x = pt1[0];
            y = pt1[1];
            error = absVal( y-(m*x+b) );
            errors.append( error, error );
            sum = sum + error;
        }
    }
    // Horizontal line case
    else if ( doX )
    {
        // Take the average of the y values
        for ( int j = 0; j < temp.count(); j++ )
        {
            temp.access( j, pt1 );
        }
    }
```
y = pt1[1];
sum = sum + y;
}
avg = sum / temp.count();
sum = 0;
for ( int j = 0; j < temp.count(); j++ )
{
    temp.access( j, pt1 );
x = pt1[0];
y = pt1[1];
error = absVal( y - avg );
extors.append( error, error );
sum = sum + error;
}

} // Vertical line case
else
{
    // Take the average of the x values
    for ( int j = 0; j < temp.count(); j++ )
    {
        temp.access( j, pt1 );
x = pt1[0];
sum = sum + x;
}
avg = sum / temp.count();
sum = 0;
for ( int j = 0; j < temp.count(); j++ )
{
    temp.access( j, pt1 );
x = pt1[0];
y = pt1[1];
error = absVal( x - avg );
extors.append( error, error );
sum = sum + error;
}

} // Calculate the mean and standard deviation
mean = sum / errors.count();
error = 0;
for ( int j = 0; j < temp.count(); j++ )
{
    errors.access( j, pt1 );
error = error + ( pt1[0] - mean ) * ( pt1[0] - mean
);
}
stddev = sqrt( error / errors.count() );

} // Clear onLine
for ( int k = 0; onLine.count() > 0; )
{
onLine.access( 0, transfer );
onLine.del( transfer[0], transfer[1] );
}

} // Write temp to onLine for all elements less than two
standard deviations from the mean
for ( int n = 0; n < temp.count(); n++ )
{
  temp.access( n, transfer );
  x = transfer[0];
  y = transfer[1];
  if ( doX && doY )
    error = absVal( y-(m*x+b) );
  else if ( doX )
    error = absVal( y - avg );
  else
    error = absVal( x - avg );
  if ( ( error < ( mean + 2 * stddev ) ) && ( error > ( mean - 2 * stddev ) ) )
    onLine.append( transfer[0], transfer[1] );
}
else
{
  // Clear onLine
  for ( int k = 0; onLine.count() > 0; )
  {
    onLine.access( 0, transfer );
    onLine.del( transfer[0], transfer[1] );
  }
  // Write temp to onLine
  for ( int n = 0; n < temp.count(); n++ )
  {
    temp.access( n, transfer );
    onLine.append( transfer[0], transfer[1] );
  }
}
return isBreak;
}
/********************++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

/*! mergeSortX sorts a variable of type list by the x components of its data using a standard merge sort*/
static void mergeSort( list &sortMe, int doX )
{
  // Declare variables
  list left, right, result;
  int middle;
  double contents[2];
  double temp[2];

  // Return if the length is one or less
  if ( sortMe.count() <= 1 )
  {
    return;
  }

  // Find the middle index of the list
  middle = sortMe.count() / 2;
// Take elements 0 to middle and put them in left
for( int i = 0; i < middle; i++ )
{
    sortMe.access( i, contents );
    left.append( contents[0], contents[1] );
}

// Take elements middle to end and put them in right
for( int j = middle; j < sortMe.count(); j++ )
{
    sortMe.access( j, contents );
    right.append( contents[0], contents[1] );
}

// Empty sortMe
for ( int i = 0; sortMe.count() > 0; )
{
    sortMe.access( 0, temp );
    sortMe.del( temp[0], temp[1] );
}

// Sort the left and right lists recursively
mergeSort( left, doX );
mergeSort( right, doX );

// Merge the result into one list
if ( doX )
    mergeX( left, right, result );
else
    mergeY( left, right, result );

// Exchange the contents of sortMe for results
for ( int j = 0; j < result.count(); j++ )
{
    result.access( j, temp );
    sortMe.append( temp[0], temp[1] );
}

/*********************************************************************/
/*! mergeX merges the left and right lists in order*/
static void mergeX( list &left, list &right, list &result )
{
    // Declare variables
    double firstLeft[2];
    double firstRight[2];

    // Merge the lists, maintaining order
    while ( ( left.count() > 0 ) && ( right.count() > 0 ) )
    {
        left.access( 0, firstLeft );
        right.access( 0, firstRight );
        if ( firstLeft[0] <= firstRight[0] )
        {
            // Append the first element of left to the result and
            remove from left
result.append( firstLeft[0], firstLeft[1] );
left.del( firstLeft[0], firstLeft[1] );
}
else
{
    // Append the first element of right to the result
    result.append( firstRight[0], firstRight[1] );
    right.del( firstRight[0], firstRight[1] );
}

while ( left.count() > 0 )
{
    left.access( 0, firstLeft );
    result.append( firstLeft[0], firstLeft[1] );
    left.del( firstLeft[0], firstLeft[1] );
}

while ( right.count() > 0 )
{
    right.access( 0, firstRight );
    result.append( firstRight[0], firstRight[1] );
    right.del( firstRight[0], firstRight[1] );
}

/***************************************************************************/
/*! mergeY merges the left and right lists in order by y-coordinate*/
static void mergeY( list &left, list &right, list &result )
{
    // Declare variables
    double firstLeft[2];
    double firstRight[2];

    // Merge the lists, maintaining order
    while ( ( left.count() > 0 ) && ( right.count() > 0 ) )
    {
        left.access( 0, firstLeft );
        right.access( 0, firstRight );
        {
            // Append the first element of left to the result and
            remove from left
            result.append( firstLeft[0], firstLeft[1] );
            left.del( firstLeft[0], firstLeft[1] );
        }
        else
        {
            // Append the first element of right to the result
            and remove from right
            result.append( firstRight[0], firstRight[1] );
            right.del( firstRight[0], firstRight[1] );
        }
    }
}
while ( left.count() > 0 )
{
    left.access( 0, firstLeft );
    result.append( firstLeft[0], firstLeft[1] );
    left.del( firstLeft[0], firstLeft[1] );
}

while ( right.count() > 0 )
{
    right.access( 0, firstRight );
    result.append( firstRight[0], firstRight[1] );
    right.del( firstRight[0], firstRight[1] );
}

/***************************************************************************/
/*! processVertical uses the hits linked list and the found line           
   parameters                                                        */
static void processVertical( list &hits, list &panel, double theta, 
    double p, double distRes, double end1[2], double end2[2])
{
    // Declare variables
    double hit[2];
    double temp[2];
    double error, dist;
    double theta_rad;
    double x, y, m, b;

    // Clear any other data that might be in panel
    for (int index = 0; 0 < panel.count();)
    {
        panel.access(0, hit);
        panel.del(hit[0], hit[1]);
    }

    // Initialize the endpoints
    end1[0] = -1000000;
    end1[1] = -1000000;
    end2[0] = 1000000;
    end2[1] = 1000000;

    // Find all points on the line
    for (int index = 0; index < hits.count(); index++)
    {
        // Get data point and round according to the resolution
        hits.access( index, hit );
        x = hit[0];
        y = hit[1];

        // Check if the point is on the line
        error = absVal( x - p );
        if ( error < 0.05 )
        {
            // Process point

            // Further processing logic...
        }
    }
}
// If so, add it to the onLine list
panel.append( x, y );
}

// Remove the outliers
if ( panel.count() > 5 )
    removeOutliers( panel, 0, 1 );

// Remove all the points on the line from hits, keeping track of
// the extrema
for ( int index = 0; index<panel.count(); index++)
{
    // Remove from hits
    panel.access(index, hit);
    hits.del(hit[0], hit[1]);

    // Check for extrema
    if ( hit[1] > end1[1] )
    {
        end1[0] = hit[0];
        end1[1] = hit[1];
    }
    {
        end2[0] = hit[0];
        end2[1] = hit[1];
    }
}

/*********************************************************************/
/*!
 processHorizontal uses the hits linked list and the found line
 parameters to identify the panel. It returns the list of hits after
 removing all the points on the line and the endpoints of the line.*/
static void processHorizontal( list &hits, list &panel,
    double theta, double p, double distRes,
    double end1[2], double end2[2])
{
    // Declare variables
    double hit[2];
    double temp[2];
    double error, dist;
    double x, y, m, b;

    // Clear any other data that might be in panel
    for ( int index = 0; 0 < panel.count(); )
    {
        panel.access(0, hit);
        panel.del(hit[0], hit[1]);
    }

    // Initialize the endpoints
    end1[0] = -1000000;
    end1[1] = -1000000;
end2[0] = 1000000;
end2[1] = 1000000;

// Find all points on the line
for (int index = 0; index<hits.count(); index++)
{
    // Get data point and round according to the resolution
    hits.access(index, hit);
    x = hit[0];
    y = hit[1];

    // Check if the point is on the line
    error = absVal( y - p );
    if ( error < 0.05 )
    {
        // If so, add it to the onLine list
        panel.append( x, y );
    }
}

// Remove the outliers
if ( panel.count() > 5 )
    removeOutliers( panel, 1, 0 );

// Remove all the points on the line from hits, keeping track of the extrema
for (int index = 0; index<panel.count(); index++)
{
    // Remove from hits
    panel.access(index, hit);
    hits.del(hit[0], hit[1]);

    // Check for extrema
    if ( hit[0] > end1[0] )
    {
        end1[0] = hit[0];
        end1[1] = hit[1];
    }
    if ( hit[0] < end2[0] )
    {
        end2[0] = hit[0];
        end2[1] = hit[1];
    }
}

/*******************************************************************************/
/*! processSlanted uses the hits linked list and the found line parameters to identify the panel. It returns the list of hits after removing all the points on the line and the endpoints of the line.*/
static void processSlanted( list &hits, list &panel, double theta, double p, double distRes, double end1[2], double end2[2])
{
    // Declare variables
double hit[2];
double temp[2];
double error, dist;
double theta_rad;
double x, y, m, b;

// Clear any other data that might be in panel
for (int index = 0; 0 < panel.count();)
{
    panel.access(0, hit);
    panel.del(hit[0], hit[1]);
}

// Initialize the endpoints
end1[0] = -1000000;
end1[1] = -1000000;
end2[0] = 1000000;
end2[1] = 1000000;

// Convert to slope-intercept form
theta_rad = (90-theta) * (3.14159/180);
m = sin(theta_rad)/cos(theta_rad);
if ((m>0) && (theta>0) && (theta<90))
{
    m = -m;
}
if ((m<0) && (theta>90) && (theta<180))
{
    m = -m;
}
b = p / cos(theta_rad);

// Find all points on the line
for (int index = 0; index < hits.count(); index++)
{
    hits.access(index, hit);
    x = hit[0];
y = hit[1];

    error = absVal( y-(m*x+b) );
    if ( error < 0.05 )
    {
        panel.append( hit[0], hit[1] );
    }
}

// Remove the outliers
if ( panel.count() > 5 )
    removeOutliers( panel, 1, 1);

// Remove all the points on the line from hits, keeping track of the extrema
for (int index = 0; index < panel.count(); index++)
{
    // Remove from hits
    panel.access(index, hit);
hits.del(hit[0], hit[1]);

// Check for extrema
if ( hit[0] > end1[0] )
{
    end1[0] = hit[0];
    end1[1] = hit[1];
}
if ( hit[0] < end2[0] )
{
    end2[0] = hit[0];
    end2[1] = hit[1];
}
return;
}

/*********************************************************************
/*! getPanel reads from the sensor data buffer to get the
Lidar data, processes it using a Hough transform to identify the
line that the panel lies on, and returns the sensor points that
are hits and are not part of the panel. It also uses a simple
standard deviation analysis to identify and remove outliers.*/
static int getPanel( list &hits, double &maxDist ) /* NO ARGUMENTS */
{
    // Declare variables
    list panel, temp;
    double end1[2], midp[2], end2[2];
    double theta, p;
    double hit[2];
    int theta_len, r_len;
    double angleRes = 2.0;
    double distRes = 0.02;
    double lastTheta, lastP;
    double lineLength, lenError;
    double PanelWidth = 1.67;
    int RepeatCount = 0;

    // Copy hits to temp so as to return it with only the found panel removed
    for ( int index = 0; index < hits.count(); index++ )
    {
        hits.access(index, hit);
        temp.append(hit[0], hit[1]);
    }

    while (1)
    {
        // If there are no hits to process, exit
        if ( temp.count() == 0 )
        {
            gPanel[0] = -1;
            gPanel[1] = -1;
            gPanel[2] = -1;
        }
    }
}
gPanel[3] = -1;
gPanel[4] = -1;
gPanel[5] = -1;
gP = -1;
gTheta = 1;
return 0;
}

// Perform a Hough transform to find the lines in the data
doHough( temp, angleRes, distRes, theta, p, maxDist );

// Handle the case for a vertical line
if (theta == 0)
{
    processVertical( temp, panel, theta, p, distRes, end1, end2 );
}

// Handle the case for a horizontal line
else if (theta == 90)
{
    processHorizontal( temp, panel, theta, p, distRes, end1, end2 );
}
else
{
    processSlanted( temp, panel, theta, p, distRes, end1, end2 );
}

// Calculate the distance between the line endpoints
lineLength = sqrt(((end1[0] - end2[0]) * (end1[0] -
    end2[0])) + ((end1[1] - end2[1]) * (end1[1] - end2[1])));
lenError = (PanelWidth-lineLength)/PanelWidth;

// Determine whether the found line is the panel
if ((lenError < 0.25) && (lenError > -0.25))
{
    // Calculate the midpoint
    midp[0] = (end1[0] + end2[0]) / 2;
    midp[1] = (end1[1] + end2[1]) / 2;

    // This is the panel
    gPanel[0] = end1[0];
gPanel[1] = end1[1];
gPanel[2] = midp[0];
gPanel[3] = midp[1];
gPanel[4] = end2[0];
gPanel[5] = end2[1];
gP = p;
gTheta = theta;

    // Remove the points on the panel from hits
    for (int index = 0; index < panel.count(); index++)
    {
        panel.access(index, hit);
        hits.del(hit[0], hit[1]);
    }
if ((theta == lastTheta) && (p == lastP)) {
    RepeatCount++;
    if (RepeatCount > 5) {
        gPanel[0] = -1;
        gPanel[1] = -1;
        gPanel[2] = -1;
        gPanel[3] = -1;
        gPanel[4] = -1;
        gPanel[5] = -1;
        gP = -1;
        gTheta = 1;
        return 0;
    }
} else {
    RepeatCount = 0;
}

// Used to check for infinite loop
lastTheta = theta;
lastP = p;

/*****************************************************************************
/*!
getTwoPanels calls getPanel twice to find two panels in the available lidar data. Returns 1 if successful, 0 otherwise.*/
static int getTwoPanels( list &hits, double &maxDist, double left[4], double &leftTheta, double right[4], double &rightTheta) /* NO ARGUMENTS */
{
    // Declare variables
    int found = 0;
    int found1 = 0;
    int found2 = 0;
    float temp;
    float panel1[4], panel2[4], tempan[4];
    float theta1, theta2;
    float p1, p2;
    float angleTo1, angleTo2;
    float diff1, diff2;

    // If there are less than two hits left, return a failure
    if ( hits.count() < 2 ) {
        return 0;
    }
// Find the two panels
found1 = getPanel( hits, maxDist );
panel1[0] = gPanel[0];
panel1[1] = gPanel[1];
panel1[2] = gPanel[4];
panel1[3] = gPanel[5];
theta1 = gTheta;
p1 = gP;

if ( !found1 )
{
    return 0;
}

// If there are less than two hits left, return a failure
if ( hits.count() < 2 )
{
    return 0;
}

found2 = getPanel( hits, maxDist );
panel2[0] = gPanel[0];
panel2[1] = gPanel[1];
panel2[2] = gPanel[4];
panel2[3] = gPanel[5];
theta2 = gTheta;
p2 = gP;

if ( !found2 )
{
    return 0;
}

// Convert angles to radians
theta1 = theta1 * ( M_PI / 180 );
theta2 = theta2 * ( M_PI / 180 );

// Find relative angles to the panels, and make the lesser one
angleTo1 = atan2(panel1[1]-vehY, panel1[0]-vehX);
angleTo2 = atan2(panel2[1]-vehY, panel2[0]-vehX);
if ( angleTo2 < angleTo1 )
{
    temp = angleTo1;
    angleTo1 = angleTo2;
    angleTo2 = temp;
    temp = theta1;
    theta1 = theta2;
    theta2 = temp;
    temp = p1;
    p1 = p2;
    p2 = temp;
    for ( int i = 0; i < 4; i++ )
        tempan[i] = panel1[i];
    for ( int i = 0; i < 4; i++ )
        panel1[i] = panel2[i];
}
for ( int i = 0; i < 4; i++ )
    panel2[i] = tempan[i];

// Assign which are right and left
// Handle the case for panels on either side of the pi/-pi split
if ( ( angleTo1 - angleTo2 ) > 3.2 )
{
    for ( int i = 0; i < 4; i++ )
        right[i] = panel1[i];
    rightTheta = theta1;
    RightPanelP = p1;
    for ( int i = 0; i < 4; i++ )
        left[i] = panel2[i];
    leftTheta = theta2;
    LeftPanelP = p2;
}
else
{
    for ( int i = 0; i < 4; i++ )
        right[i] = panel2[i];
    rightTheta = theta2;
    RightPanelP = p2;
    for ( int i = 0; i < 4; i++ )
        left[i] = panel1[i];
    leftTheta = theta1;
    LeftPanelP = p1;
}

// Get the right and left edges of the panels and order them
[leftx, lefty, rightx, righty]
// Left panel
// Find relative angles to the panels, and make the lesser one
angleTo1 = atan2(left[1]-vehY, left[0]-vehX);
angleTo2 = atan2(left[3]-vehY, left[2]-vehX);
if ( angleTo2 < angleTo1 )
{
    temp = angleTo1;
    angleTo1 = angleTo2;
    angleTo2 = temp;
    temp = left[0];
    left[0] = left[2];
    left[2] = temp;
    temp = left[1];
    left[1] = left[3];
    left[3] = temp;
}

// Assign which are right and left
// Handle the case for panels that straddle the pi/-pi split
if ( ( angleTo1 - angleTo2 ) > 3.2 )
{
    temp = angleTo1;
    angleTo1 = angleTo2;
    angleTo2 = temp;
    temp = left[0];
left[0] = left[2];
left[2] = temp;
temp = left[1];
left[1] = left[3];
left[3] = temp;
}

// Right panel
// Find relative angles to the panels, and make the lesser one
angleTo1 = atan2(right[1]-vehY, right[0]-vehX);
angleTo2 = atan2(right[3]-vehY, right[2]-vehX);
if ( angleTo2 < angleTo1 )
{
    temp = angleTo1;
    angleTo1 = angleTo2;
    angleTo2 = temp;
    right[0] = right[2];
    right[2] = right[0];
    right[1] = right[3];
    right[3] = right[1];
}

// Assign which are right and left
// Handle the case for panels that straddle the pi/-pi split
if ( ( angleTo1 - angleTo2 ) > 3.2 )
{
    temp = angleTo1;
    angleTo1 = angleTo2;
    angleTo2 = temp;
    left[0] = left[2];
    left[2] = left[0];
    left[1] = left[3];
    left[3] = left[1];
}

// Get panel angles
rightTheta = atan2( right[3] - right[1], right[2] - right[0] );
leftTheta = atan2( left[1] - left[3], left[0] - left[2] );

// Assign the values to the global variables
RightPanelB = right[1] - RightPanelM * right[0];
RightPanelTheta = rightTheta;
RightPanel[0] = right[0];
RightPanel[1] = right[1];
RightPanel[2] = right[2];
RightPanel[3] = right[3];
LeftPanelB = left[1] - LeftPanelM * left[0];
LeftPanelTheta = leftTheta;
LeftPanel[0] = left[0];
LeftPanel[1] = left[1];
LeftPanel[2] = left[2];
LeftPanel[3] = left[3];

129
LeftPanel[1] = left[1];
LeftPanel[2] = left[2];
LeftPanel[3] = left[3];

    return 1;
}
/

/**
 * calcJoinerApproach calculates the approach position for a joiner robot
 * from the last pair of panels found*/
static void calcJoinerApproach( double x, double y, double &target_x, double &target_y )
{
    // Declare variables
    double approachDist = 1.0;
    double thetad, thetab;
    double diff1, diff2;
    double x1, y1, x2, y2;
    double m, b;
    double d1, d2;
    double inter_x, inter_y;
    double final_x, final_y;

    // Find the angular bisector of the two panels
    thetad = absVal( RightPanelTheta - LeftPanelTheta );
    thetad = thetad / 2;
    // Find thetab by trying both plus and minus and seeing which falls between rightTheta and leftTheta
    diff1 = minusPiToPi( LeftPanelTheta + thetad );
    diff2 = minusPiToPi( LeftPanelTheta - thetad );
    if ( ( ( LeftPanelTheta < diff1 ) && ( diff1 < RightPanelTheta ) ) || ( ( LeftPanelTheta > diff1 ) && ( diff1 > RightPanelTheta ) ) )
        thetab = diff1;
    else if ( ( ( LeftPanelTheta < diff2 ) && ( diff2 < RightPanelTheta ) ) || ( ( LeftPanelTheta > diff2 ) && ( diff2 > RightPanelTheta ) ) )
        thetab = diff2;
    else
        thetab = -10;

    // The bisector is vertical
    if ( absVal( thetab - ( M_PI / 2 ) ) < 0.02 )
    {
        // Find the two approach points
        x1 = x;
        y1 = y + approachDist;
        x2 = x;
        y2 = y - approachDist;
    }

    // The bisector is horizontal
    else if ( absVal( thetab ) < 0.02 )
    {
        // Find the two approach points
        x1 = x + approachDist;
        y1 = y;
\[
x_2 = x - \text{approachDist}; \\
y_2 = y;
\]

} // The bisector is slanted
else
{
  m = \tan(\ \text{thetab} );
  b = y - m * x;

  // Find the two approach points
  x_1 = (-b*m+y*m+x + \sqrt{(b*m-y*m-x)*(b*m-y*m-x)-(1+m*m)*(x*x+b*b-2*b*y+y*y-\text{approachDist}*\text{approachDist}))/((1 + m*m); \\
y_1 = m*x_1+b;
  x_2 = (-b*m+y*m+x - \sqrt{(b*m-y*m-x)*(b*m-y*m-x)-(1+m*m)*(x*x+b*b-2*b*y+y*y-\text{approachDist}*\text{approachDist}))/((1 + m*m); \\
y_2 = m*x_2+b;
}

// Find the distance to the two points from the vehicle
updatePosition();
\text{d}_1 = \sqrt{((\text{vehX} - x_1) * (\text{vehX} - x_1)) + ((\text{vehY} - y_1) * (\text{vehY} - y_1))}; \\
\text{d}_2 = \sqrt{((\text{vehX} - x_2) * (\text{vehX} - x_2)) + ((\text{vehY} - y_2) * (\text{vehY} - y_2))};

// Choose the closest
if ( \text{d}_1 < \text{d}_2 )
{
  \text{final}_x = x_1; \\
  \text{final}_y = y_1;
}
else
{
  \text{final}_x = x_2; \\
  \text{final}_y = y_2;
}

// Find the line between the current location and the final position
m = ( \text{final}_y - \text{vehY} ) / ( \text{final}_x - \text{vehX} );
\text{b} = \text{final}_y - m * \text{final}_x;

// Get the two positions along this line that are distance 2 from the current location
x_1 = (-b*m+vehY*m+vehX + \sqrt{(b*m-vehY*m-vehX)-(1+m*m)*(vehX*vehX+b*b-2*b*vehY+vehY*vehY-2.0*2.0)))/(1 + m*m); \\
y_1 = m*x_1+b;
  x_2 = (-b*m+vehY*m+vehX - \sqrt{(b*m-vehY*m-vehX)-(1+m*m)*(vehX*vehX+b*b-2*b*vehY+vehY*vehY-2.0*2.0)))/(1 + m*m); \\
y_2 = m*x_2+b;

// Set the intermediate target to the one of these closest to the final position
\text{d}_1 = \sqrt{((\text{final}_x - x_1) * (\text{final}_x - x_1)) + ((\text{final}_y - y_1) * (\text{final}_y - y_1))}; \\
\text{d}_2 = \sqrt{((\text{final}_x - x_2) * (\text{final}_x - x_2)) + ((\text{final}_y - y_2) * (\text{final}_y - y_2))};
if ( d1 < d2 )
{
  inter_x = x1;
  inter_y = y1;
}
else
{
  inter_x = x2;
  inter_y = y2;
}

// Pick the closest point to the vehicle between the final and
intermediate point and go to it

d1 = sqrt(((final_x - vehX) * (final_x - vehX)) + ((final_y -
vehY) * (final_y - vehY)));
d2 = sqrt(((inter_x - vehX) * (inter_x - vehX)) + ((inter_y -
vehY) * (inter_y - vehY)));

// Exception: if the difference is less than 0.5 meters, go to
the final point
if ( ( d1 < d2 ) || ( absVal( d1 - d2 ) < 0.5 ) )
{
  target_x = final_x;
  target_y = final_y;
}
else
{
  target_x = inter_x;
  target_y = inter_y;
}

return;

/******************************************************************************/

/*! spinToAngle takes an angle in degrees as input and issues a prim
command to spin the robot to that angle*/
static void spinToAngle(double angle, double error)
{
  // Declare variables
  AmMobJAStat * stat = amMobJAStat;
  static int command_type_sent;
  static int serial_number = 1;
  RCS_TIMER * timer;
  double calc_error = 1.0;

  // Initialize the timer
  timer = new RCS_TIMER(0.1);

  // Bring the angle in range and convert to radians
  angle = minusPiToPi ( RAD ( angle ) );

  while ( calc_error > RAD( error ) )
  {
    // Construct and send the spin command
command_type_sent = amMobJACmdSpin.type;
amMobJACmdSpin.direction = MOAST_ROTATE_SHORTEST;
amMobJACmdSpin.absAngle = angle;
amMobJACmdSpin.tolerance = RAD(error);
setOddSerial(amMobJAStat->echo_serial_number, &serial_number);
amMobJACmdSpin.serial_number = serial_number;
amMobJAcmdbuf->write(amMobJACmdSpin);

// Wait for the command to complete
while (1)
{
    timer->wait();
amMobJAStatBuf->read();
    if (stat->command_type == command_type_sent && stat->echo_serial_number == serial_number && stat->status != RCS_EXEC)
        break;
    if (error < 1)
        { updatePosition();
        }
}

// Calculate the error
updatePosition();
calc_error = sqrt((vehYaw - angle) * (vehYaw - angle));
calc_error = minusPiToPi(calc_error);
}
delete timer;

/***************************************************************************/

/*! goTo takes a location (x,y) in global coordinates and a speed and
   sends the necessary command to the AM echelon to go there, then
   stops once it has arrived within some accuracy*/
static double goTo(double x, double y, double speed)
{
    // Declare variables
    AmMobJAStrat * stat = amMobJAStrat;
    static int command_type_sent;
    static int serial_number = 1;
    RCS_TIMER * timer;
double direction;
double error = 1.0;

    // Initialize the timer
timer = new RCS_TIMER(0.1);

    // Start by turning to point in the direction of travel
updatePosition();
direction = atan2(y - vehY, x - vehX);
direction = direction * 57.296;
spinToAngle(direction, 1);
// Stop the vehicle and wait for a second, just in case
doBrake();
for ( int i = 0; i < 10; i++ )
    timer->wait();

// Construct and send the move command
command_type_sent = amMobJACmdMoveWaypoint.type;
amMobJACmdMoveWaypoint.waypoint[0].p.x = x;
amMobJACmdMoveWaypoint.waypoint[0].p.y = y;
amMobJACmdMoveWaypoint.waypoint[0].p.z = 0;
amMobJACmdMoveWaypoint.waypoint[0].neighborhood = 0.1;
amMobJACmdMoveWaypoint.waypoint[0].speed = speed;
amMobJACmdMoveWaypoint.waypoint_length = 1;
amMobJACmdMoveWaypoint.computeHorizonCosts = false;
setOddSerial(amMobJAStat->echo_serial_number, &serial_number);
amMobJACmdMoveWaypoint.serial_number = serial_number;
amMobJACmdBuf->write(amMobJACmdMoveWaypoint);

// Wait for the command to complete
fflush(stdout);
while (1) {
    updatePosition();
    timer->wait();
amMobJAStatBuf->read();
    if (stat->command_type == command_type_sent && stat->echo_serial_number == serial_number && stat->status != RCS_EXEC)
        break;
}

// Calculate the error
updatePosition();
error = sqrt ((vehX - x) * (vehX - x) + (vehY - y) * (vehY - y));
return error;

/*!
 * HAJoiner is the automaton to control the joining robot*/
static void HAJoiner()
{
    // Declare local constants for readability
    int SUCCESS = 1;    // State succeeded in its goal
    int FAILURE = 0;    // State failed to achieve its goal

    // States
    const int JOINER_SEARCH = 12;    // Search for the joint
    const int JOINER_APPROACH = 13;  // Approach the joint
    such that both panels can be detected and the joiner can control their alignment
    const int JOINER_ALIGN = 14;     // Align the joiner robot to the joint

const int JOINER_ACQUIRE = 19; // Get up next to the panels so that they can be grabbed
const int JOINER_TOGETHER = 20; // Pull the panels together, effectively joining them
const int COMPLETE_FAILURE = 99; // No possible recovery - print an error message to indicate what the trouble is
const int DONE = 100;

// Declare variables
int result;
int State = JOINER_SEARCH;
int PrevState, CurrState;

// Extend and retract the prismatic joint, since it kicks the first time
doFlex();

while (1)
{
    CurrState = State;

    // Determine the state of the hybrid automaton and execute
switch(State)
{
    case JOINER_SEARCH:
    {
        result = exJoinerSearch();
        if (result == SUCCESS)
            State = JOINER_APPROACH;
        else
            State = COMPLETE_FAILURE;
        break;
    }
    case JOINER_APPROACH:
    {
        result = exJoinerApproach();
        if (result == SUCCESS)
            State = JOINER_ALIGN;
        else
            State = COMPLETE_FAILURE;
        break;
    }
    case JOINER_ALIGN:
    {
        result = exJoinerAlign();
        if (result == SUCCESS)
            State = JOINER_ACQUIRE;
        else
            State = COMPLETE_FAILURE;
        break;
    }
    case JOINER_ACQUIRE:
    {

result = exJoinerAcquire();
if (result == SUCCESS)
    State = JOINER_TOGETHER;
else
    State = COMPLETE_FAILURE;
break;
}
case JOINER_TOGETHER:
{
    result = exJoinerTogether();
    if (result == SUCCESS)
        State = DONE;
    else
        State = COMPLETE_FAILURE;
    break;
}
default:
{
    // The automaton entered an illegal state -
    throw an error and break
    State = COMPLETE_FAILURE;
    break;
}
}
PrevState = CurrState;
// Update the global variables keeping track of the robot's
tableau
setPosition(updatePosition());
if ((State == COMPLETE_FAILURE) || (State == DONE))
    break;
}

/***************************************************************************/
/*! exJoinerSearch executes a search for two panels by spinning in
place at
intervals of 25 degrees. We assume that the robot is within
sight
of the panels*/
static int exJoinerSearch()
{
    // Declare variables
    list hits;
    int found = 0;
    int iterCount = 0;
    double hit[2];
    double angle, initAngle;
    double maxDist;
    double right[4], left[4];
    double right_theta, left_theta;

/** Find the current position so as to start from it **/
updatePosition();
angle = vehYaw * 57.295;

while ( !found )
{
    spinToAngle(angle, 1);

    // Find the panels
    maxDist = getLidarHits( hits );
    found = getTwoPanels( hits, maxDist, right, right_theta,
                         left, left_theta);

    iterCount++;
    angle = angle+22.5;

    // Correct for overflow
    if ( angle > 360 )
        angle = angle - 360;

    // Limit the search
    if ( iterCount > 16 )
        break;
}

// Success or failure determined by whether panel found
return found;

/***********************************************************************/

/*! exJoinerApproach drives the robot to a place where it has a good
    approach angle on the joint and can see both panels*/
static int exJoinerApproach()
{
    // Declarations
    list hits;
    int found = 0;
    int count = 0;
    double maxDist, x, y;
    double right[4], left[4];
    double rightTheta, leftTheta, thetad, thetab;
    double approachDist = 1.0;
    double cornerAngle;
    double m, b;
    double m1, b1, m2, b2;
    double x1, y1, x2, y2, d1, d2;
    double target_x, target_y;
    double temp, diff1, diff2;
    double error, startYaw;

    updatePosition();
    startYaw = vehYaw;

    // Open the manipulator arms to 270 degrees to support angles
greater than 180
open270();

while (1) {
    // Find the two panels
    while (!found) {
        // Timeout
        if (count > 10) {
            return 0;
        }
        maxDist = getLidarHits(hits);
        found = getTwoPanels(hits, maxDist, left, leftTheta,
                             right, rightTheta);
        count++;
    }
    // Set to approach the leftmost end of the right panel
    findPanelIntersection(x, y);
    temp = sqrt((x - vehX) * (x - vehX) + (y - vehY) * (y - vehY));
    // Get the approach points
    calcJoinerApproach(x, y, target_x, target_y);
    // Go to the target location
    error = goTo(target_x, target_y, 0.25);
    doBrake();
    // Finished if less than approachDist from intersection
    error = sqrt(((x - vehX) * (x - vehX)) + ((y - vehY) * (y - vehY)));
    if (error < (approachDist + 0.5)) {
        spinToAngle(atan2(IntersectionY - vehY,
                          IntersectionX - vehX) * 57.296, 1.0);
        return 1;
    }
}

/*********************************************************************/
/*! exJoinerAlign aligns the robot to the joint, intended to be along a 
line bisecting the angle made by the panels and pointed 
along it*/
static int exJoinerAlign() {
    // Declarations
    ServoMobJACmdSkid skidMsg;
    ServoMobJAStat * stat;
    RCS_TIMER * timer;
    list hits;
    int found = 0;
    int count = 0;
    int found1, found2;
    int command_type_sent, serial_number;
double maxDist, dist, x, y, x_mid, y_mid;
double right[4], left[4];
double rightTheta, leftTheta;
double angleTo1, angleTo2;
double approachDist = 1.5;
double cornerAngle, theta, error, lastError;
double tangentTheta, tangentP, m, b;
double m1, b1, m2, b2;
double x1, y1, x2, y2, d1, d2;
double misPos, last_misPos;

// Initialize the timer and state buffer
timer = new RCS_TIMER(0.1);
stat = (ServoMobJAStat *) (servoMobJAStatBuf->get_address());
servoMobJAStatBuf->read();

// Find the two panels
while ( !found )
{
    // Timeout
    if ( count > 10 )
    {
        break;
    }
    maxDist = getLidarHits( hits );
    found = getTwoPanels( hits, maxDist, left, leftTheta, right, rightTheta);
    count++;
}
if ( !found )
{
    for ( int i = 0; i < 4; i++ )
    {
        left[i] = LeftPanel[i];
        right[i] = RightPanel[i];
    }
    leftTheta = LeftPanelTheta;
    rightTheta = RightPanelTheta;
}

// Turn on the "brakes", since the prismatic link can kick the robot a little
skidMsg.wLeft = -0.001;
skidMsg.wRight = -0.001;
command_type_sent = skidMsg.type;
setOddSerial(stat->echo_serial_number, &serial_number);
skidMsg.serial_number = serial_number;
servoMobJACmdBuf->write(&skidMsg);

// Extend the prismatic link
command_type_sent = misCmdMove.type;
misCmdMove.serial_number = mis_serial_number++;
misCmdMove.linkCmd_length = 1;
misCmdMove.linkCmd[0].linkID = 4;
misCmdMove.linkCmd[0].value = 2;
misCmdMove.linkCmd[0].type=SERVO_MIS_JA_LINK_CMD_ABS_VALUE_TYPE;
mis[0].cmdBuf->write(misCmdMove);
// Wait for the command to finish executing before continuing
while (1)
{
    timer->wait();
    mis[0].statBuf->read();
    misPos = ((ServoMisJAStat *)mis[0].statMsg)-
    >linkStat[3].jointVal;
    if (mis[0].statMsg->command_type == command_type_sent &&
        misPos == last_misPos && mis[0].statMsg->echo_serial_number == (mis_serial_number - 1))
    {
        count++;
        if (count > 5)
            break;
    } else
    {
        count = 0;
    }
    last_misPos = misPos;
}

// Send zero velocity command
servoMobJAStatBuf->read();
skidMsg.wLeft = 0;
skidMsg.wRight = 0;
command_type_sent = skidMsg.type;
setOddSerial(stat->echo_serial_number, &serial_number);
skidMsg.serial_number = serial_number;
servoMobJACmdBuf->write(&skidMsg);

// The intended joint location is the intersection of the two lines
findPanelIntersection( x, y );

// We just want to point to the intended joint location and center on it
setPosition();
tangentTheta = atan2( y - vehY, x - vehX );
spinToAngle( tangentTheta * 57.2958, 0.5);

return 1;

*******************************************************************************/

/*! exJoinerAcquire drives the robot forward to where it can connect to the panels that make the joint*/
static int exJoinerAcquire()
{
    // Declare variables
    ServoMobJACmdSkid skidMsg;
    ServoMobJAStat * stat;
    list hits;
    RCS_TIMER * timer;
int count = 0;
int found = 0;
int direction = 1;
int repeatCount = 0;
int command_type_sent, serial_number;
double maxDist, leftTheta, rightTheta;
double lastError = 10;
double error = 1.0;
double control, temp;
double bodyError = 1.0;
double rightArmError = 1.0;
double leftArmError = 1.0;
double leftAngle, rightAngle;
double leftError = 10;
double rightError = 10;
double left[4], right[4];
double target_x, target_y, x, y;
double manipTheta, targetTheta;
double mpan_l, bpan_l, mpan_r, bpan_r;
double meff_l, beff_l, meff_r, beff_r;
double m_body, b_body;
double panelP, armP, theta;
double leftArmLength = 1.5;
double rightArmLength = 1.0;
double armEndX, armEndY;

// Initialize the timer and state buffer
RCS_TIMER timer = new RCS_TIMER(0.1);
ServoMobJAStat *stat = (ServoMobJAStat *) (servoMobJAStatBuf->get_address());
servoMobJAStatBuf->read();

// Find the two panels
while ( !found )
{
  // Timeout
  if ( count > 10 )
  {
    break;
  }
  maxDist = getLidarHits( hits );
  found = getTwoPanels( hits, maxDist, left, leftTheta, right, rightTheta);
  count++;
}
if ( !found )
{
  for ( int i = 0; i < 4; i++ )
  {
    left[i] = LeftPanel[i];
    right[i] = RightPanel[i];
  }
  leftTheta = LeftPanelTheta;
  rightTheta = RightPanelTheta;
}

// Save the angle of the panels
PanelAngle = rightTheta - leftTheta;
/ Lines describing the panels
bnan_l = left[1] - mpan_l * left[0];
bnan_r = right[1] - mpan_r * right[0];

/ Intersection point of those lines
findPanelIntersection( target_x, target_y );

/ Repeat until the error is less than 2 cm
while ( ( leftError > 0.02 ) || ( rightError > 0.02 ) )
{
    bodyError = 1.0;
    rightArmError = 1.0;
    leftArmError = 1.0;

    / Find the two panels
    found = 0;
    while ( !found )
    {
        / Timeout
        if ( count > 3 )
        {
            break;
        }
        maxDist = getLidarHits( hits );
        found = getTwoPanels( hits, maxDist, left, leftTheta,
right, rightTheta);
        count++;
    }

    / Calculate new values if the panels were found
    if ( found )
    {
        / Lines describing the panels
    }
    bpan_l = left[1] - mpan_l * left[0];
    bpan_r = right[1] - mpan_r * right[0];

        / Intersection point of those lines
        findPanelIntersection( target_x, target_y );
    }

    / Fix the alignment as best as possible
    while ( absVal( bodyError ) > 0.005 )
    {
        / Get the line describing the body and orientation
        updatePosition();
        x = 0.7 * cos( vehYaw ) + vehX;
        y = 0.7 * sin( vehYaw ) + vehY;
        m_body = ( y - vehY ) / ( x - vehX );
        b_body = y - m_body * x;
// Calculate the angle to the manipulator center and
to the target
manipTheta = atan2( y - vehY, x - vehX );
targetTheta = atan2( target_y - vehY, target_x - vehX );

// Calculate the error and reverse direction if
// needed
bodyError = manipTheta - targetTheta;
if ( bodyError == lastError )
    repeatCount++;
else
    repeatCount = 0;
if ( ( absVal( bodyError ) > absVal( lastError ) ) ||
( repeatCount > 50 ) )
{
    direction = -direction;
    repeatCount = 0;
}

// Send a manual spin proportional to the error
skidMsg.wLeft = direction * bodyError * 0.5;
skidMsg.wRight = -direction * bodyError * 0.5;
command_type_sent = skidMsg.type;
setOddSerial(stat->echo_serial_number,
&serial_number);
skidMsg.serial_number = serial_number;
servoMobJACmdBuf->write(&skidMsg);

// Let a few moments pass
for ( int t = 0; t < 5; t++ )
timer->wait();
lastError = bodyError;
}
doBrake();

// Update vehicle position information
updatePosition();
x = 0.7 * cos( vehYaw ) + vehX;
y = 0.7 * sin( vehYaw ) + vehY;

// Fix the alignment of the manipulator arms to make them
// parallel
repeatCount = 0;
while ( absVal( rightArmError ) > 0.01 )
{
    // Read the mission package buffer to get the angle
    of the arm
    mis[0].statBuf->read();
    rightAngle = ((ServoMisJAStat *)mis[0].statMsg)->
>linkStat[0].jointVal;

    // Calculate the line equation describing the right
    arm
    meff_r = tan( vehYaw + absVal(rightAngle) );
    beff_r = y - meff_r * x;
// Calculate the error and reverse direction if needed
rightArmError = rightTheta - (vehYaw + absVal(rightAngle));
if (rightArmError == lastError)
    repeatCount++;
else
    repeatCount = 0;
if ((absVal(rightArmError) > absVal(lastError)) ||
    (repeatCount > 10))
{
    direction = -direction;
    repeatCount = 0;
}

// Send a command to move the arm with a velocity proportional to the error
command_type_sent = misCmdMove.type;
misCmdMove.serial_number = mis_serial_number++;
misCmdMove.linkCmd_length = 1;
misCmdMove.linkCmd[0].linkID = 1;
misCmdMove.linkCmd[0].value = 0.5 * rightArmError * direction;

misCmdMove.linkCmd[0].type = SERVO_MIS_JA_LINK_CMD_VELOCITY_TYPE;
 mis[0].cmdBuf->write(misCmdMove);

// Let a few moments pass
for (int t = 0; t < 5; t++)
    timer->wait();
lastError = rightArmError;

// Send a command to stop the movement of the arm
command_type_sent = misCmdMove.type;
misCmdMove.serial_number = mis_serial_number++;
misCmdMove.linkCmd_length = 1;
misCmdMove.linkCmd[0].linkID = 1;
misCmdMove.linkCmd[0].value = 0;

misCmdMove.linkCmd[0].type = SERVO_MIS_JA_LINK_CMD_VELOCITY_TYPE;
 mis[0].cmdBuf->write(misCmdMove);

// Let a moment pass
 timer->wait();

direction = -direction;       // Probably opposite right arm
repeatCount = 0;
while (absVal(leftArmError) > 0.01)
{
    // Read the mission package buffer to get the angle of the arm
    mis[0].statBuf->read();
    leftAngle = ((ServoMisJAStat *)mis[0].statMsg)->linkStat[1].jointVal;
// Calculate the line equation describing the left arm
meff_l = tan( vehYaw - absVal(leftAngle) );
beff_l = y - meff_l * x;

// Calculate the error and reverse direction if needed
leftArmError = leftTheta - ( vehYaw - absVal(leftAngle) );
if ( leftArmError == lastError )
    repeatCount++;
else
    repeatCount = 0;
if ( ( absVal( leftArmError ) > absVal( lastError ) )
|| ( repeatCount > 10 ) )
{
    direction = -direction;
    repeatCount = 0;
}

// Send a command to move the arm with a velocity proportional to the error
command_type_sent = misCmdMove.type;
misCmdMove.serial_number = mis_serial_number++;
misCmdMove.linkCmd_length = 1;
misCmdMove.linkCmd[0].linkID = 2;
misCmdMove.linkCmd[0].value = 0.5 * leftArmError * direction;

misCmdMove.linkCmd[0].type=SERVO_MIS_JA_LINK_CMD_VELOCITY_TYPE;
mis[0].cmdBuf->write(misCmdMove);

// Let a few moments pass
for ( int t = 0; t < 5; t++ )
    timer->wait();
lastError = leftArmError;
}

// Send a command to stop the movement of the arm
command_type_sent = misCmdMove.type;
misCmdMove.serial_number = mis_serial_number++;
misCmdMove.linkCmd_length = 1;
misCmdMove.linkCmd[0].linkID = 2;
misCmdMove.linkCmd[0].value = 0;

misCmdMove.linkCmd[0].type=SERVO_MIS_JA_LINK_CMD_VELOCITY_TYPE;
mis[0].cmdBuf->write(misCmdMove);

// Let a moment pass
timer->wait();

// Calculate the error between the left panel and the left arm
if ( absVal( LeftPanelTheta ) < 0.02 )
    // Left panel is horizontal
    leftError = absVal( y - target_y );
else if ( absVal( absVal( LeftPanelTheta ) - ( M_PI / 2 ) ) < 0.02 ) // Left panel is vertical
    leftError = absVal( x - target_x );
else // Left panel is slanted
{
    temp = y - mpan_l * x;
    leftError = absVal( ( bpan_l - temp ) * cos( -leftTheta ) );
}

// Calculate the error between the right panel and the rightarm
if ( absVal( RightPanelTheta ) < 0.02 ) // Right panel is horizontal
{
    rightError = absVal( y - target_y );
} else if ( absVal( absVal( RightPanelTheta ) - ( M_PI / 2 ) ) < 0.02 ) // Right panel is vertical
{
    rightError = absVal( x - target_x );
} else // Right panel is slanted
{
    temp = y - mpan_r * x;
    rightError = absVal( ( bpan_r - temp ) * cos( rightTheta ) );
}

temp = y - mpan_l * x;
leftTheta = absVal( ( bpan_l - temp ) * cos( -leftTheta ) );

error = sqrt( ( x - target_x ) * ( x - target_x ) + ( y - target_y ) * ( y - target_y ) );

// Drive forward until the error is cut in half
control = error;
while ( error > ( control * 0.5 ) )
{
    // Send a drive forward message proportional to the error
    skidMsg.wLeft = absVal( error * 0.25 );
    skidMsg.wRight = absVal( error * 0.25 );
    command_type_sent = skidMsg.type;
    setOddSerial(stat->echo_serial_number, &serial_number);
    skidMsg.serial_number = serial_number;
    servoMobJAcmdbuf->write(&skidMsg);

    // Let a moment pass
    timer->wait();

    // Recalculate the position of the manipulator arms
    updatePosition();
    x = 0.7 * cos( vehYaw ) + vehX;
    y = 0.7 * sin( vehYaw ) + vehY;
meff_l = tan( vehYaw - absVal(leftAngle) );
beff_l = y - meff_l * x;
meff_r = tan( vehYaw + absVal(rightAngle) );
beff_r = y - meff_r * x;

left arm
if ( absVal( LeftPanelTheta ) < 0.02 )
  // Left panel is horizontal
  leftError = absVal( y - target_y );
else if ( absVal( absVal( LeftPanelTheta ) - ( M_PI / 2 ) ) < 0.02 )
  // Left panel is vertical
  leftError = absVal( x - target_x );
else
  // Left panel is slanted
  temp = y - mpan_l * x;
  leftError = absVal( ( bpan_l - temp ) * cos( -leftTheta ) );

the right arm
if ( absVal( RightPanelTheta ) < 0.02 )
  // Right panel is horizontal
  { rightError = absVal( y - target_y );
  }
else if ( absVal( absVal( RightPanelTheta ) - ( M_PI / 2 ) ) < 0.02 )
  // Right panel is vertical
  { rightError = absVal( x - target_x );
  }
else
  // Right panel is slanted
  { temp = y - mpan_r * x;
    rightError = absVal( ( bpan_r - temp ) * cos( rightTheta ) );
  }

error = sqrt( ( x - target_x ) * ( x - target_x ) + ( y - target_y ) * ( y - target_y ) );

// Stop the vehicle
// Send a drive forward message proportional to the error
skidMsg.wLeft = -0.00001;
skidMsg.wRight = -0.00001;
command_type_sent = skidMsg.type;
setOddSerial(stat->echo_serial_number, &serial_number);
skidMsg.serial_number = serial_number;
servoMobJACmdBuf->write(&skidMsg);
timer->wait();
skidMsg.wLeft = 0;
skidMsg.wRight = 0;
command_type_sent = skidMsg.type;
setOddSerial(stat->echo_serial_number, &serial_number);
skidMsg.serial_number = serial_number;
servoMobJACmdBuf->write(&skidMsg);

// Check to see if everything is within the bounds for connecting to the panels
if (leftError > 0.02)
{
    return 0;
}
if (rightError > 0.02)
{
    return 0;
}
return 1;

RIENDC_TIMER(0.1);
updatePosition();

// Fire the effector to attach to the right panel
effCmdOpcode.serial_number = eff_serial_number++;
effCmdOpcode.effID = 1;
effCmdOpcode.opcode = SERVO_EFF_JA_FIRE_OPCODE_TYPE;
effCmdOpcode.param = 0;
eff->cmdBuf->write(effCmdOpcode);

// Wait for a few ticks to let it take
for (int i = 0; i < 10; i++)
    timer->wait();
// Fire the effecter to attach to the left panel
effCmdOpcode.serial_number = eff_serial_number++;
effCmdOpcode.effID = 2;
effCmdOpcode.opcode = SERVO_EFF_JA_FIRE_OPCODE_TYPE;
effCmdOpcode.param = 0;
eff->cmdBuf->write(effCmdOpcode);

// Wait for a few ticks to let it take
for (int i = 0; i < 10; i++)
timer->wait();

// Lock the wheels because the mission package can kick
skidMsg.wLeft = -0.001;
skidMsg.wRight = 0.001;
servoMobJAStatBuf->read();
command_type_sent = skidMsg.type;
setOddSerial(stat->echo_serial_number, &serial_number);
skidMsg.serial_number = serial_number;
servoMobJACmdBuf->write(&skidMsg);

while (misPos > 0)
{
    // Pull the panels together by 0.1
    mis[0].statBuf->read();
    misPos = ((ServoMisJAStat *)mis[0].statMsg)-
>linkStat[3].jointVal;
    command_type_sent = misCmdMove.type;
    misCmdMove.serial_number = mis_serial_number++;
    misCmdMove.linkCmd_length = 1;
    misCmdMove.linkCmd[0].linkID = 4;
    misCmdMove.linkCmd[0].value = misPos - 0.05;

    misCmdMove.linkCmd[0].type = SERVO_MIS_JA_LINK_CMD_ABS_VALUE_TYPE;
    mis[0].cmdBuf->write(misCmdMove);

    // Wait for the command to finish executing before
    continuing
    while (1)
    {
        for (int t = 0; t < 10; t++)
timer->wait();
        mis[0].statBuf->read();
        misPos = ((ServoMisJAStat *)mis[0].statMsg)-
>linkStat[3].jointVal;
        if (mis[0].statMsg->command_type == command_type_sent
           && misPos == last_misPos && mis[0].statMsg->echo_serial_number ==
             (mis_serial_number - 1))
        {
            count++;
            if (count > 5)
            break;
        }
        else
        {
            count = 0;
        }
}
last_misPos = misPos;
mis[0].statBuf->read();
misPos = ((ServoMisJAStat *)mis[0].statMsg)-
>linkStat[3].jointVal;
}

// Wait for the pull together to complete
for ( int i = 0; i < 100; i++ )
  timer->wait();

// Release wheel lock
skidMsg.wLeft = 0;
skidMsg.wRight = 0;
servoMobJAStatBuf->read();
command_type_sent = skidMsg.type;
setOddSerial(stat->echo_serial_number, &serial_number);
skidMsg.serial_number = serial_number;
servoMobJACmdBuf->write(&skidMsg);

// Clean up
delete timer;

  return ( 1 );
}
REFERENCES


