Rosie Clearpath

An Entry from ***Rose-Hulman Institute of Technology*** in the 2013 Intelligent Ground Vehicle Competition (IGVC 2013)

Derek Heeger, Chad Jones, Dongyang Li, Chenqi Cao, Tayler Burns, Rain Dartt, Tayler Berns, Aaron Golliver

Faculty Advisor:
David Mutchler, Professor of Computer Science and Software Engineering

Faculty statement: I hereby certify that the design and development of the robot discussed in this technical report has involved significant contributions by the aforementioned team members, consistent with the effort required in a Senior Design course.

_____

David Mutchler, Professor of Computer Science and Software Engineering

Table of Contents

Note: The outline of this report was taken from the Princeton entry in the 2008 IGVC [1]. We are grateful for their example of an award-winning design report.

1 Team Overview

In this problem-based course, students design and develop software to solve challenges in navigation faced by autonomous vehicles (robots). The problems are real problems faced by real robots in a real competition. The software to be developed is for challenges faced by a robot that will be entered in the 2012 Ground Vehicles Competition (IGVC), although the software will be designed to apply to other robots as well. Challenges include location, vision, planning and control:

- Where am I, and where will I be in X seconds?
- What obstacles are ahead of me?
- What sensors should I use to learn the above? How do I deal with uncertain and conflicting information?
- How will I avoid the obstacles ahead of me? Where do I want to go next?
- At each step, what speed should I be driving, to accomplish my overall goal of winning the competition? What sub-goals should I set to accomplish that overall goal?
- What commands do I give to motors to accomplish what I want to do next?
- What information do I need from sensors to accomplish what I want to do next?

Figure 1. A portion of the course description of CSSE 290 *Software Challenges in Autonomous Vehicle Navigation*, Rose-Hulman Institute of Technology.

The Husky Rose team is organized via a new course offered this year at Rose-Hulman: CSSE 290 *Software Challenges in Autonomous Vehicle Navigation*. Twenty-five students took the course over 3 terms, for a total of student 50 credit-hours over the year. Eight students are in the course for the spring term and authored this report.

2 Design Process

In the fall term, we decided which hardware, software, and sensors we were going to use. We knew that we wanted to use the ***Robot Operating System (ROS)*** [2] platform developed by Willow Garage [3]. We chose our robot frame, a ***Husky A200*** [4] that we bought from Clearpath Robotics [5]. We also concluded on using a Lidar, a Logitech camera, an IMU from Yost Engineering Inc, and a GPS from GlobalsSat.

In the spring term, we used a simplified version of the ***Agile*** [6] software development process called ***Scrum*** [7] in which we set out deliverables due each week and re-planned each week based on what we did (and did not) accomplish. The diagram to the right summarizes the Scrum development process.

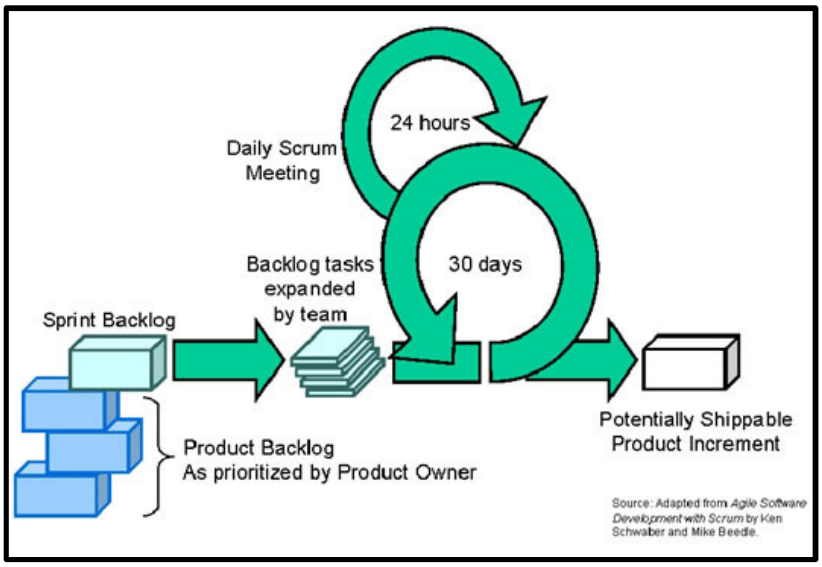

Figure 2. The Scrum development process **[15]**

3 Hardware

3.1 The Husky Robot from Clearpath Robotics

Our team wanted to focus on software challenges, per the course driving the team. For that reason, we bought a Husky A200 robot from Clearpath Robotics that provided a very strong, solid base – but only the base: wheels, motors, chassis, battery etc. No sensors and no brain.

- Dimensions: About 3 feet long, 2 feet wide, 15 inches tall, with a 5 inch clearance. It weighs about 100 pounds.
- Speed: It runs in rugged terrain (snow, etc) quite nicely, with a max speed of about 2.3 mph. It turns in a tight radius with a wheel base of about 2 feet.
- Power: Battery powered. It runs for several hours on a charge.

More details of the robot are at http://www.clearpathrobotics.com/husky.

3.2 Hardware We Added to the Basic Husky Robot

- A mounting system made from 80/20 aluminum t-slotted tube framing [8].
- One cameras (mounted front and sides): Logitech HD Pro Webcam C920 [9].
- LIDAR: A used LIDAR loaned to use from an alumni.
- GPS: A BU-353-S4 Weather-proof USB GPS Receiver by GlobalSat [10].
- Computer: HP/Compaq Tablet computer, model TC 4400 (several years old).

- Inertial Measurement Unit (IMU)
- Safety system as required for IGVC.

3.3 Kill Switch

A 12V line from the Husky will always power a solid light on when the Husky is on. The button device mounted on the Husky will connect a 5V source from the Husky to a Spektrum AR6210 6-CHANNEL DSMX Receiver SPMAR6210 and also run a parallel line from the 12V source to the relay. A DoubleSwitch Radio Controlled Dual 8A Relay connected to the receiver connects the 12V to the channel on the LED that toggles flashing for autonomous mode and also splices the serial cord connecting the laptop to the Husky. When the Killswitch button is in the 'off' position, the receiver defaults to open its circuits; if the receiver is defaulted or triggered to open the circuits, the heartbeat signal from the laptop is disrupted, and the Husky stops moving while the LED returns to solid mode instead of flashing. A diagram of the circuit is shown below.

3.4 Costs of Equipment

Item	Retail cost	Cost to team
Husky A200 robot, with 1 extra battery (shipped, with tax)	$ 14,700	$ 7,600
Camera – Logitech HD Pro Webcam C920	$ 80	$ 80
LIDAR – SICK LMS291-S14	$ 5,000	$ 0
GPS – BU-354-S4 Weather-proof USB GPS Receiver.	$ 60	$ 60
Computer (HP/Compaq Tablet computer, model TC 4400)	$ 2,500	$ 0
Mounting hardware	$ 500	$ 100
Safety system (light, remote control, etc)	$ 800	$ 400
TOTAL	**$ 23,640**	**$ 8,240**

4 Software

This section describes the software we developed for our robot. ***Our development is still in progress – we will give a more complete report at the oral presentation at the contest.***

4.1 Platform – Robot Operating System (ROS)

The software platform chosen for use on our robot is ***Robot Operating System (ROS)*** [2], an open source software framework for use with robotics. It is a mostly self-contained system, which according to the ROS Wiki, provides "the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. [2]" More simply put, it is similar to an operating system (although it has to run on a host OS; Ubuntu Linux is the only one officially supported), and uses a message-passing graph architecture to facilitate communication between various inputs, processing nodes, and outputs.

A more detailed overview of ROS is as follows (most of this is summarized from the ROS concepts page [11] located on the ROS Wiki): ROS has three layers of concepts and functionality: ***Filesystem Level***, ***ROS Graph***, and ***Community***.

The most important parts of the ***Filesystem Level*** are ***Packages*** and ***Stacks***. ***Packages*** are large units that may contain ROS nodes, ROS libraries, ROS configuration, or anything else that needs to go together. ***Stacks*** are collections of related packages. An example would be the Clearpath Husky stack; it contains packages for Husky initialization, Husky remote teleoperation, and Husky simulation.

The ***Graph Level*** of ROS is where most of the work is done. The graph level consists of ***nodes***: modular processes that, like Unix processes (and that similarly adhere to the philosophy "do one thing, and do it well"), typically do one specific task (control GPS, transform coordinate frames, control LIDAR, etc). Nodes communicate to each other by sending messages to each other over a routing system that utilizes named ***topics***. Nodes can either ***publish*** messages to a specific topic or ***subscribe*** to a topic, receiving (and handling) any messages that are published to it. Nodes may publish and subscribe to several different topics. There are also ***services***, which allow for synchronous communication (request/reply), but these are used relatively infrequently. "***Bags***" are another useful part of the Graph Level of ROS. They are a format for saving and replaying ROS message data, allowing one to work on algorithms without having to collect new data each time.

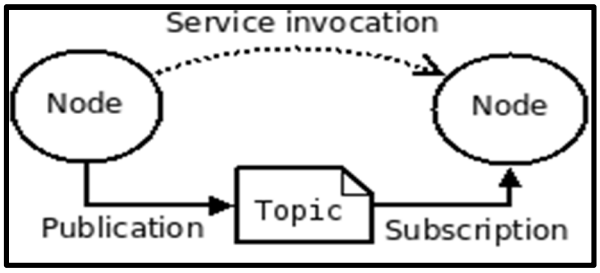

Figure 3. The ROS message-passing structure (from **[11]**)

The third level of ROS is the ***Community*** level, which consists of the online distributions and repositories of ROS. Our Husky robot is currently running the 4th version of ROS, ***Electric Emys***.

We chose ROS due to the large amount of available software for it, the large amount of compatibility, and its ease of development, due to being open-source. Our production/deployment system is running on Ubuntu 11.10, and our development systems are all running either Ubuntu 11.10 or Ubuntu 12.04. ROS supports code written in Python and in C++. Most of our high-level code is currently written in ***Python***, with calls to ***libraries implemented in C++***.

4.2 Vision for white line detection

The detection algorithm for extracting the coordinates of white lines is simple but works well. It uses the open source vision library OpenCV, which is well-supported by ROS and has a powerful library that contains a variety of classical and most efficient algorithms for image processing. To access these positions of the pixels that represent white lines, an easy way is to transform the image to whiten the pixels on white lines at the same time black anything else. The transformation of a colored image to black and white is performed on a gray scaled image with some threshold constant. However, the threshold to transform the gray image is hard to fix, because the gray level of the white lines are always similar to its background. Therefore we converted the RGB image to **HSV image**, and split it into Hue, Saturation and Volume layer. Among these three layers, the ***Saturation layer*** is the most important to us.

Saturation layers represent each pixel's saturation with an integer value within 0 to 255; therefore the saturation layer can be converted into a binary image directly. The threshold should vary from different images that are taken under varying external illumination, which poses a challenging obstacle. At first we arbitrarily chose a value as our threshold. Later, to make our algorithm more robust to different images of different daylight intensities, we implemented an adaptive algorithm to determine the threshold constant by using the histogram of the image. In the saturation image we easily noticed that the white line part now becomes the blackest part, whose pixels have the lowest value among 0 to 255. And these pixels actually compose only a small percentage of the whole image. Based on this priori knowledge, we decided to choose the value at which the histogram has a largest derivative as our threshold. We compared these adaptive thresholds with our former experimental threshold constants on different images. We

found that we need to minimize the adaptive threshold at some percentage (i.e. 30% or more) thus we have these two similar values.

After the correct threshold was chosen, there was one final step, and that was to remove the speckled points that were not part of the line. The ***median blur*** algorithm was used and effectively removed all the speckles.

In order to effectively find lines on the image, the Hough ***Transform*** was implemented to find the straight lines on the image effectively. It's a method that finds straight lines of some length within a pre-set range in an image. The value returned by the Hough Method is an array of lines defined by two points on the original image. The images below show the line recognition code finding a white line.

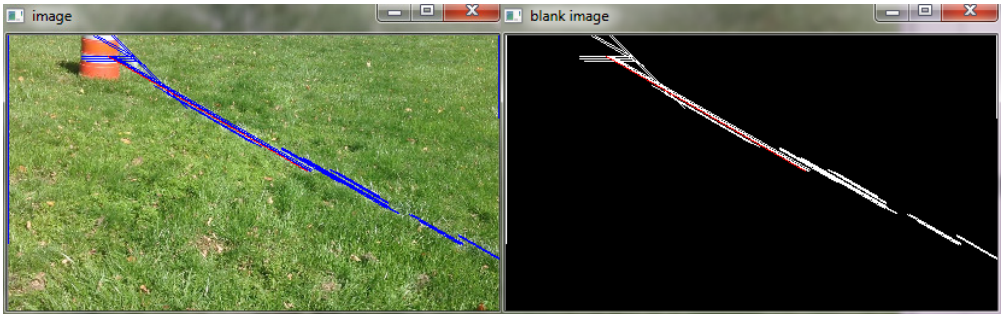

Figure 4 shows the line code finding a line.

4.3 Vision for detecting barrels

So far all the barrels we have seen in photos from IGVC are orange. Therefore we decided to take color segmentation into consideration as the orange color is unique in the whole photo. We first resize the image, then convert the resized RGB image into HSV color space. We then use cv2.inRange to convert the image to a black and white image. The white lines of the image are the orange parts of the barrels. Median blur was also used to remove the noises left in the background. The Canny method is then called to outline the barrel out as follows. This is all prepared for the Hough Line method to recognize a line that describes the bottom position of the barrel in the picture. The lowest horizontal line is a different color from all the other lines in order to mark it as the bottom of the barrel for later coordinate transformation use. The image shows the barrel code

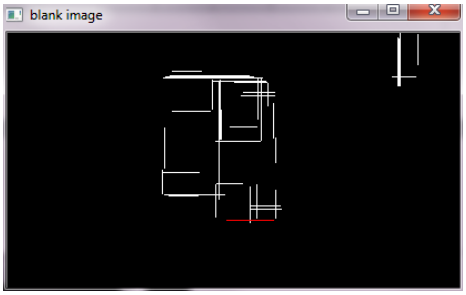

Figure 5 shows the barrel recognition code working.

finding the edges of the barrel.

4.4 Image Location and Object Location Transformation

The image location to object space transformation was based on trigonometry that calculated the distance relative to the base of the object's location on the screen. By breaking down the location of the object into two right triangles, we were able to convert the pixel location to an angle, and then add that angle to the base angles that were relative to the camera. With these calculated angles, and the given height measurement, the formulas and code calculates the distance of the object using the camera within an acceptable tolerance. The tolerance is less at a certain distance away from the camera. We suspect this is because of the optics within the camera itself, not due to the formula or algorithms used to find the distance. The distance is most accurate at about 8-10 feet away from the camera.

4.5 LIDAR

There was a ROS driver for the LIDAR. It would publish the LIDAR data as a topic in ROS as a function of distance and an angle. We then created a transform to turn the location of the object that we found to a real location in space.

4.6 GPS

We have written code that takes in GPS data. It then publishes the data to the GPS topic. We tested the GPS and found it reports consistent data within 1 meter but the actual error can vary up to 3 meters.

4.7 IMU

In order to measure and record inertial measurement unit (IMU) data, we used Yost Engineering, Inc.'s 3-Space Sensor which measures accelerometer, compass, and gyroscopic data. The device is used by making a USB serial port connection and sending packs of command bytes to the device then reading the requested data sent back from the device. Python scripts were used to continually request and read data from the IMU sensor, correctly transform the data and publish it to the "imu/data" ROS topic, which was passed into the Kaulman Filter. With this transformed data from the IMU, the robot can determine where it's current location is as well as its current orientation.

4.8 *Extended Kalman Filter*

ROS has a built in extended Kalman Filter which inputs the data from the GPS, IMU and Encoders. It uses probability to smoothly incorporate all the sensor data which helps the robot know its exact location in space.

4.9 *Gmapping*

Gmapping is the combination of all the sensor and odometry data into one file so that the robot can create a picture of the world around it. When Gmapping is accomplished, the robot can easily use premade navigation stacks to go where it needs to. The image below shows the summary of what is necessary for Gmapping to work.

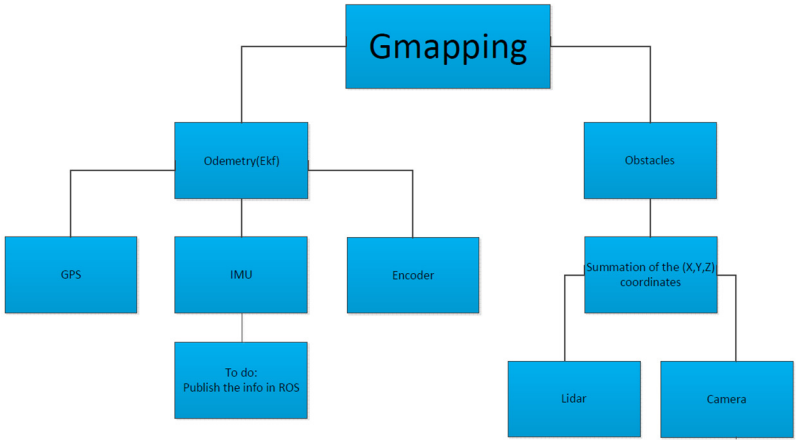

Figure 6 shows the steps required for successful Gmapping.

4.10 *Navigation*

For navigation functionality, the ROS ***Navigation Stack*** is used. It contains a variety of components that work together with other nodes as part of a larger system. The simplest explanation of it is that it takes sensor data and localization data, then signals the robot's control rfaces to take it to a given destination.

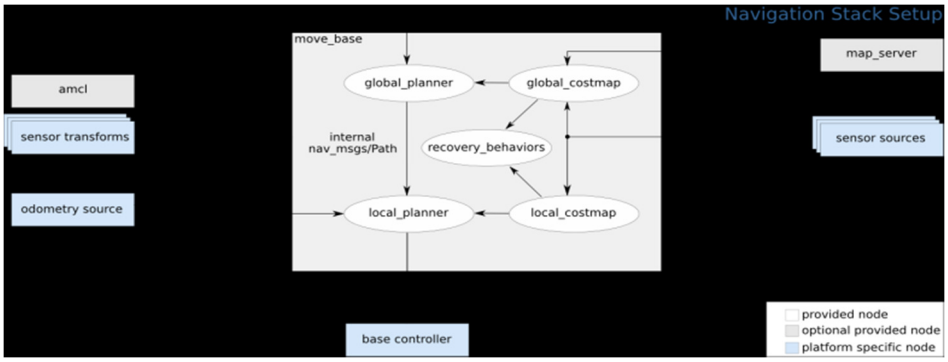

Figure 7 shows how the Navigation stack works.

A more complex

explanation of the ROS Navigation stack is as follows: The navigation stack is a series of nodes that work together to give the robot navigation functionality. The internals of the navigation stack consist of a ***global costmap***, a ***local costmap***, a ***global planne***r, a ***local planner***, and a ***recovery behaviors*** node.

Costmaps are the primary way by the robot knows where it can and cannot go, and what paths are best to take. They are built by taking in sensor data (whether from a laser scan [usually output by LIDAR] or point cloud) and using this to produce an occupancy grid (a 2D or 3D grid of places the robot can and cannot go). The "cost" part of the costmap is that each grid space has a "cost" associated with it. The robot avoids spaces with high cost (corresponding to high probability of collision), and prefers to go through spaces with low cost (corresponding to low or no probability of collision). These are calculated by giving the robot an "inscribed radius" (corresponding to a circle drawn inside the robot's body) and a "circumscribed radius" (corresponding to a circle drawn outside the robot's body). Obstacles that would fall within the robot's inscribed radius have a very high cost, while obstacles that fall within the robot's circumscribed radius have a slightly lower cost, and obstacles that fall within neither have practically no cost. The use of costmaps allows the ROS navigation stack to deal with many types of obstacles in an elegant manner.

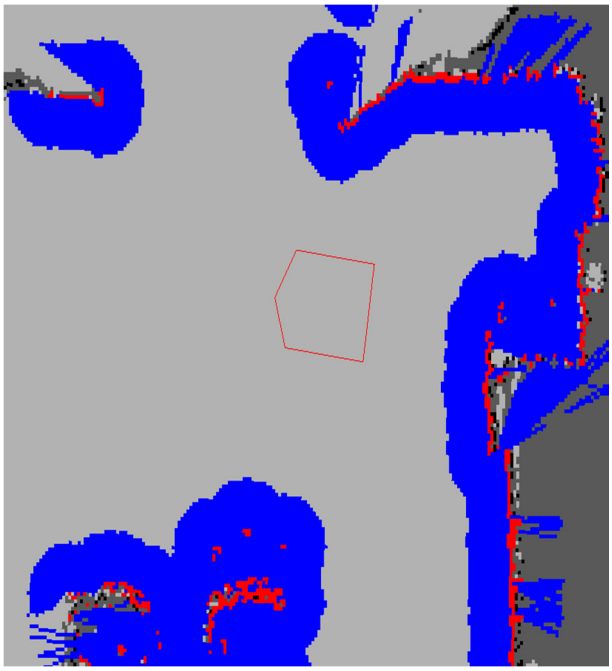

Figure 8. A Costmap Visualization

The second part of the navigation stack is the ***planner*** node. These use the costmap to attempt to find the "cheapest" route from the robot to a specified goal. After such a route is found, it sends differential and angular velocities to the robot. The algorithm used by the planner node is as follows (quoted from [14]):

1. Discretely sample in the robot's control space (dx, dy, dtheta)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.

3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
5. Rinse and repeat.

Finally, the recovery node rotates the robot 360 degrees when the robot appears to be "stuck".

To make the navigation stack functional, a few things are required: sensor inputs, robot controller output, odometry inputs, and a map server (for sharing the map with other nodes, as well as saving it to the disk). At this point, the navigation stack simply needs to receive a "goal message", specifying a desired destination location. To do this, given GPS information, a "gps_common" node is used, which converts the spherical projection of GPS coordinates into UTM coordinates, which are rectangular coordinates that the navigation stack can understand.

When all of this data is fed into the navigation stack, the robot can navigate from waypoint to waypoint while avoiding obstacles and automatically adjusting its speed. The challenge inherent in this is combining the sensor data and tuning the navigation stack's settings to give a balance of processing speed and accuracy (by adjusting the grid size of the costmap, making it 2D vs 3D, etc), as well as adjusting the acceleration, speed, and radius parameters of the robot to allow it to go from point to point as quickly as possible while not coming into contact with any obstacles.

5 Conclusion

This is our first year in IGVC. We are still implementing ideas. We will update this report at the oral presentation at IGVC. We think that these will prove to be winning design decisions:

- Buying the Husky A200 as our robot base. This allows us to focus on our interests – software and sensors – instead of grappling with mechanical issues.
- Using ROS as our platform. The learning curve is steep, but we believe that ROS will let us focus on applying known algorithms in novel ways, instead of re-inventing the wheel.
- Our vision algorithms for line and barrel detection
- Our use of an IMU, GPS, LIDAR and Camera

6 References

[1] "IGVC Design Reports, 2008, Princeton University," 2008. [Online]. Available: http://www.igvc.org/design/reports/dr218.pdf. [Accessed 8 May 2012].

[2] "Robot Operating System (ROS)," [Online]. Available: http://www.ros.org/wiki/. [Accessed 8 May 2012].

[3] W. Garage, "Willow Garage home page," [Online]. Available: http://www.willowgarage.com/. [Accessed 8 May 2012].

[4] "Husky A200 Unmanned Ground Vehicle," [Online]. Available: http://www.clearpathrobotics.com/husky. [Accessed 8 May 2012].

[5] "Clearpath Robotics," [Online]. Available: http://www.clearpathrobotics.com. [Accessed 8 May 2012].

[6] "Agile Software Development Processes," [Online]. Available: http://www.agile-process.org/. [Accessed 8 May 2012].

[7] K. Schwaber and M. Beedle, Agile Software Development with Scrum, Prentice-Hall, 2001.

[8] "80/20 T-Slotted Aluminum Tube Framing," [Online]. Available: http://www.8020.net/T-Slot-1.asp. [Accessed 8 May 2012].

[9] "Logitech HD Pro Webcam C920," [Online]. Available: http://www.logitech.com/en-us/webcam-communications/webcams/devices/hd-pro-webcam-c920. [Accessed 8 May 2012].

[10] "BU-353-S4 Weather-proof USB CPS Receiver by GlobalSat," [Online]. Available: http://www.usglobalsat.com/p-688-bu-353-s4.aspx#images/product/large/688.jpg.

[11] "ROS Concepts Page," [Online]. Available: http://www.ros.org/wiki/ROS/Concepts. [Accessed 8 May 2012].

[12] "Open CV Vision Library, Wiki," [Online]. Available: http://opencv.willowgarage.com/wiki/. [Accessed 8 May 2012].

[13] "Histogram of Oriented Gradients (HOG)," [Online]. Available: http://en.wikipedia.org/wiki/Histogram_of_oriented_gradients. [Accessed 8 May 2012].

[14] "ROS Dynamic Window Approach (DWA) Planner," [Online]. Available: http://www.ros.org/wiki/dwa_local_planner.

[15] "Scrum Services from Icon ATG: diagram adapted from "Agile Software Development with Scrum" by Schwaber and Beedle," [Online]. Available: http://www.iconatg.com/services/process/scrum.php. [Accessed 8 May 2012].