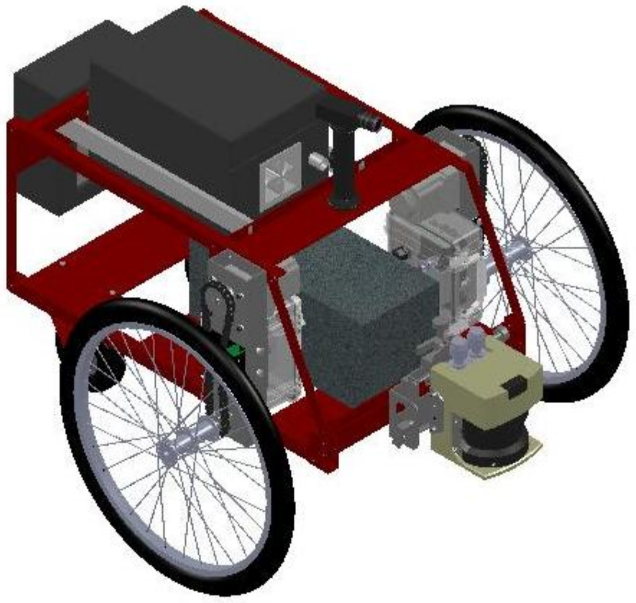
Cornell University
Cornell Minesweeper
Vehicle Design Report for Nero V2.0



Steven Liu, Cameron Salzberger, Barrett Ames, Garrett Bernstein, Matt Christensen, Platon Dickey, Alan Calabrese, Justin Churchill, Sam Weiss, Scott Franklin

Faculty Advisor Statement
I certify that the engineering design of the robot Nero, described in this report has been significant and equivalent to what might be awarded credit in a senior design course.

____
Professor Ephrahim Garcia
Department of Mechanical and Aerospace Engineering, Cornell University

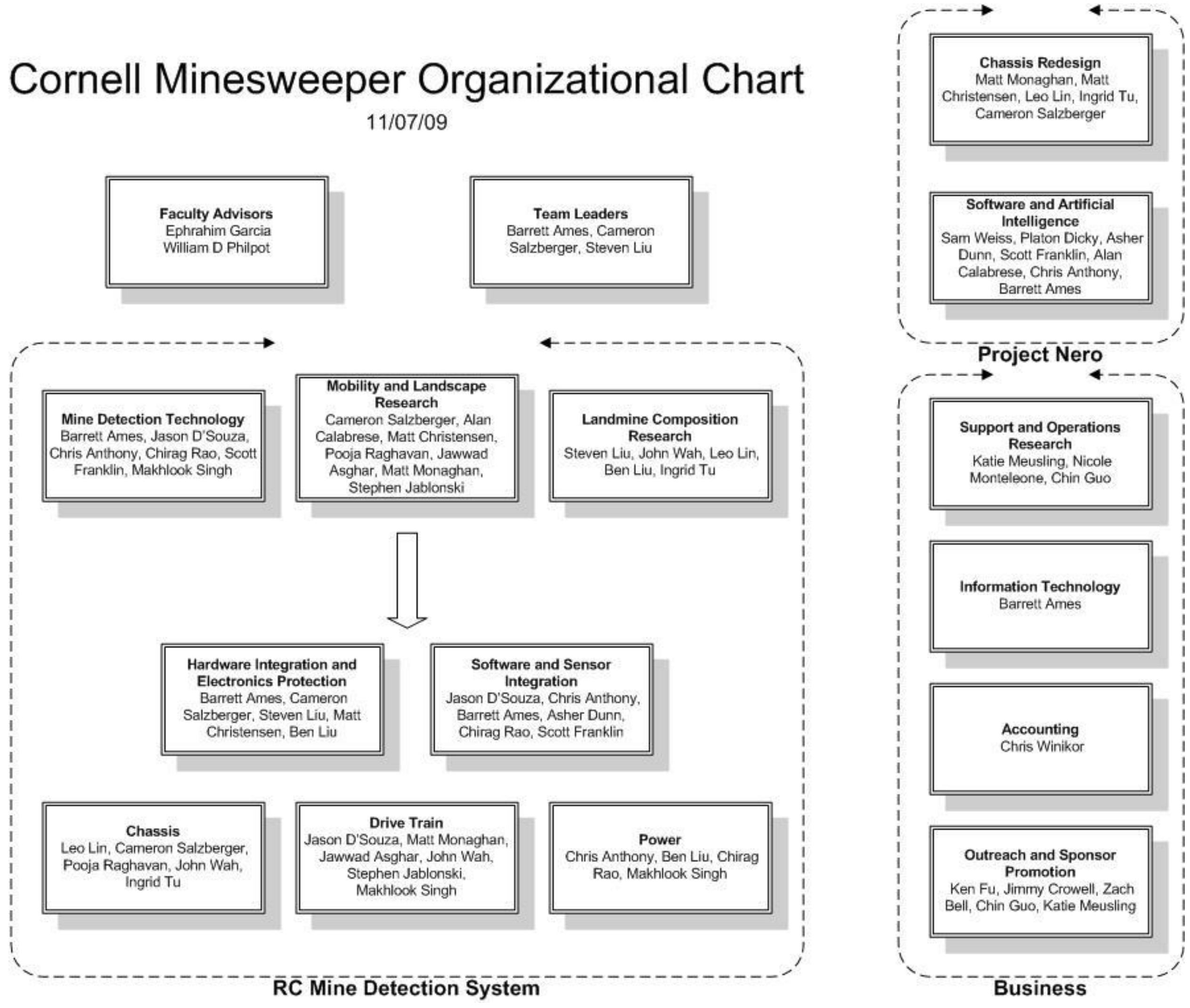
Table of Contents

1. Introduction

Cornell Minesweeper (CMS) project was founded in 2006 to develop an autonomous vehicle that can accurately detect land mines and facilitate their clearance. To continue the design validation of the autonomous platform on which the mine detection sensors will be mounted, we present Nero V2.0, Cornell University's Second entry at the International Ground Vehicle Competition.

The Cornell Minesweeper follows a hybrid team structure in that it is a mixture between a flat organization and tiered structure. This leads to greater responsibility and participation from all team members, regardless of experience. The team consists of 31 undergraduates split into three main divisions RC Mine Detection System, Business, and Project Nero. Project Nero had two phases, hardware and software redesign and software testing.

Nero V2.0 is IP65 rated, lightweight and modular in design and features a more robust software solution. It was designed and built entirely in-house by Cornell engineering undergraduates.



Cornell Minesweeper Organizational Chart
11/07/09
Faculty Advisors Ephrahim Garcia William D Philpot
Team Leaders Barrett Ames, Cameron Salzberger, Steven Liu
Mine Detection Technology Barrett Ames, Jason D'Souza, Chris Anthony, Chirag Rao, Scott Franklin, Makhlook Singh
Mobility and Landscape Research Cameron Salzberger, Alan Calabrese, Matt Christensen, Pooja Raghavan, Jawwad Asghar, Matt Monaghan, Stephen Jablonski
Landmine Composition Research Steven Liu, John Wah, Leo Lin, Ben Liu, Ingrid Tu
Hardware Integration and Electronics Protection Barrett Ames, Cameron Salzberger, Steven Liu, Matt Christensen, Ben Liu
Software and Sensor Integration Jason D'Souza, Chris Anthony, Barrett Ames, Asher Dunn, Chirag Rao, Scott Franklin
Chassis Leo Lin, Cameron Salzberger, Pooja Raghavan, John Wah, Ingrid Tu
Drive Train Jason D'Souza, Matt Monaghan, Jawwad Asghar, John Wah, Stephen Jablonski, Makhlook Singh
Power Chris Anthony, Ben Liu, Chirag Rao, Makhlook Singh
RC Mine Detection System
Chassis Redesign Matt Monaghan, Matt Christensen, Leo Lin, Ingrid Tu, Cameron Salzberger
Software and Artificial Intelligence Sam Weiss, Platon Dicky, Asher Dunn, Scott Franklin, Alan Calabrese, Chris Anthony, Barrett Ames
Project Nero
Support and Operations Research Katie Meusling, Nicole Monteleone, Chin Guo
Information Technology Barrett Ames
Accounting Chris Winikor
Outreach and Sponsor Promotion Ken Fu, Jimmy Crowell, Zach Bell, Chin Guo, Katie Meusling
Business

Figure 1: Organizational Chart - Redesign Phase

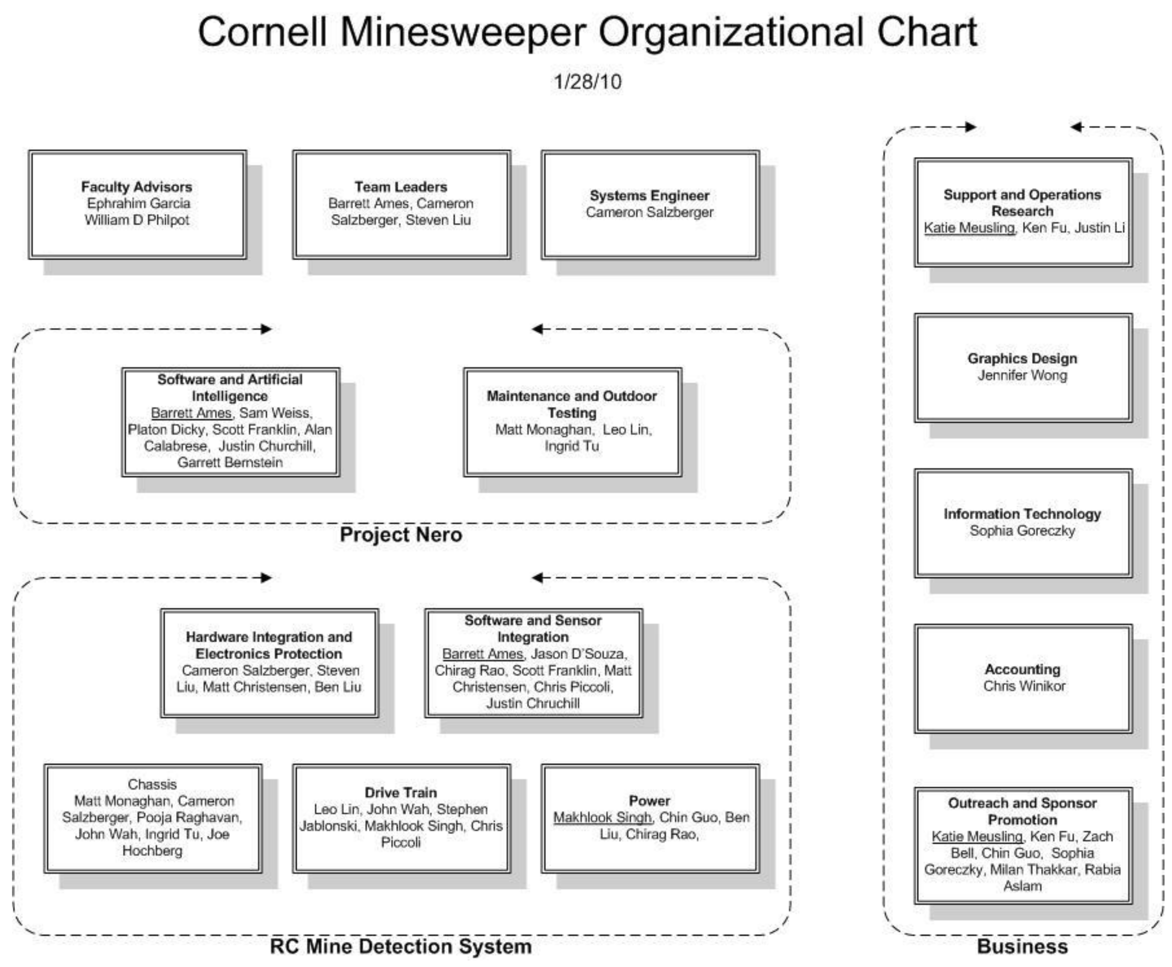

Cornell Minesweeper Organizational Chart
1/28/10
Faculty Advisors
Ephrahim Garcia
William D Philpot
Team Leaders
Barrett Ames, Cameron Salzberger, Steven Liu
Systems Engineer
Cameron Salzberger
Software and Artificial Intelligence
Barrett Ames, Sam Weiss, Platon Dicky, Scott Franklin, Alan Calabrese, Justin Churchill, Garrett Bernstein
Maintenance and Outdoor Testing
Matt Monaghan, Leo Lin, Ingrid Tu
Project Nero
Hardware Integration and Electronics Protection
Cameron Salzberger, Steven Liu, Matt Christensen, Ben Liu
Software and Sensor Integration
Barrett Ames, Jason D'Souza, Chirag Rao, Scott Franklin, Matt Christensen, Chris Piccoli, Justin Chruchill
Chassis
Matt Monaghan, Cameron Salzberger, Pooja Raghavan, John Wah, Ingrid Tu, Joe Hochberg
Drive Train
Leo Lin, John Wah, Stephen Jablonski, Makhlook Singh, Chris Piccoli
Power
Makhlook Singh, Chin Guo, Ben Liu, Chirag Rao,
RC Mine Detection System
Support and Operations Research
Katie Meusling, Ken Fu, Justin Li
Graphics Design
Jennifer Wong
Information Technology
Sophia Goreczky
Accounting
Chris Winikor
Outreach and Sponsor Promotion
Katie Meusling, Ken Fu, Zach Bell, Chin Guo, Sophia Goreczky, Milan Thakkar, Rabia Aslam
Business

Figure 2: Organizational Chart - Software Testing

2. Design Process

Design requirements are the most important part of the design process because they define what shape the design must take. The requirements for Nero V2.0 came from an evaluation of the first version of Nero. Problems with the old system were turned into additional requirements for the new system.

The design requirements for the redesign of the physical system included: more balanced weight distribution and increased usability. The design requirements for the software system included: increased modularity, code reuse, and increased documentation. The software objectives were achieved by the introduction of a bug list, and the use of ROS (ROS will be explained in more depth later). The hardware requirements were met through the remanufacture of the chassis to better fit these new requirements.

3. Hardware

As highlighted before by the organizational chart, the production division of Minesweeper was broken up into 5 main groups: Chassis, Drive Train and Controls, Power and Quality Control, Electronics Protection and Safety, and Software and Hardware Processing. The goal of these subgroups was to be cross-disciplinary, drawing expertise from different fields to create the best design. While the different subsystems were developed independently, communication and integration between sub-teams was a top priority, ensuring an effective overall design.

Beyond the fundamental requirements, the design emphasizes endurance, safety and reliability, organization, and modularity that separate Nero from other robots.

Throughout the design process, extensive use of the *SolidWorks™* Computer Aided Drafting and Design tool was utilized in product visualization and troubleshooting. Every component from design to fabrication was drafted: aiding in fabrication, assembly and redesign every step of the way.

Table 1: Summary of Costs

Item	Value	Cost to Team
Microbotics MIDG-II INS	$7,060.00	$0.00
Sick LMS291 LIDAR	$5,137.00	$5,137.00
Venus634FLPx	$49.95	$49.95
MC-433 Camera	$350.00	$350.00
Drive Train	$1,109.48	$1,109.48
Chassis	$300.00	$285.00
Mini ITX	$529.93	$529.93
Wireless E-Stop	$85.00	$85.00
Mechanical E-Stop	$50.00	$50.00
Amphenol Connectors	$962.00	$0.00
Waterproof Connectors	$75.00	$75.00
Waterproof Cases	$140.00	$140.00
Battery	$869.95	$869.95
Power System Hardware	$560.00	$560.00
Total	$17,278.31	$9,241.31

3.1 Innovations

While Nero V2.0 is a redesign of the older version there are significant changes that set Nero V2.0 apart in design and quality. Innovations for the new version of Nero include an improved center of mass to decrease tipping. This involved the changing the placement of the payload as well as rotating the drive train blocks so that they are vertical. Also a new GPS with a higher update rate was implemented to increase localization accuracy. For maintenance purposes, the electronics box has been moved to the top of the chassis to provide easy access in-situ, rather than requiring sliding on rails to open.

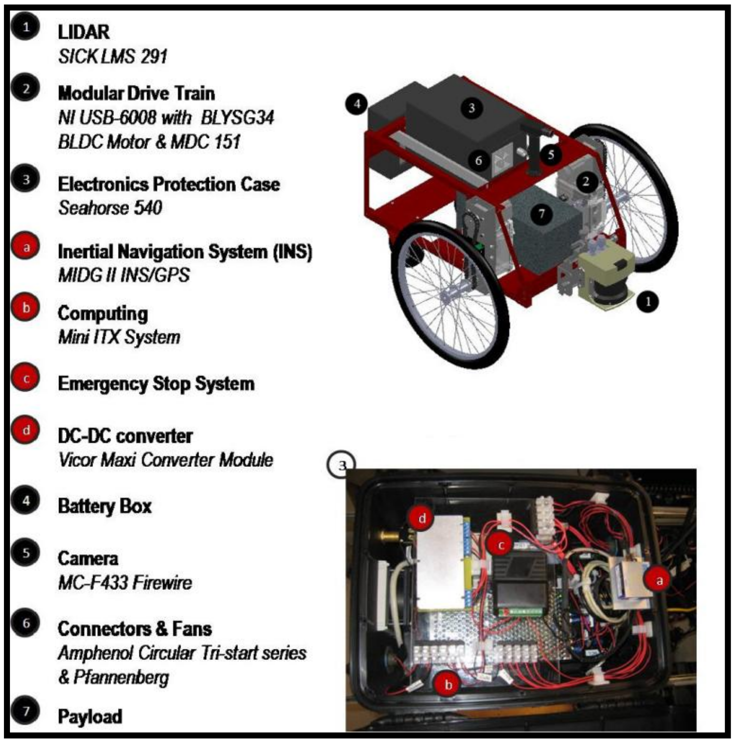


Figure 5: Overview of Nero and Sensors

3.2 Software and Hardware Processing

The following sensors are selected and utilized in data processing: computer, LIDAR, camera, INS, and motor encoders and GPS. The function of each sensor is described in Table 2, and the location of each sensor is depicted in Figure 5.

Table 2: Sensors Functionality

Part	Description
Computer [Kontron 986LCD-M/mITX]	Processes the image, combines all available data to compute, and stores old data.
LIDAR [Sick LMS291]	Determines the distance from the nearest obstacles for a 180 degree range.
Camera [MC-F433 Firewire]	Takes pictures and performs basic preprocessing such as white balancing and scaling to detect lane lines, edges, and possibly other obstacles.
INS [Microbotics MIDG-II]	Obtains the current orientation, acceleration and velocity of the robot.
Motor Encoder	Hall effect sensors are used to calculate the current speed of each wheel in revolutions per second.
GPS [Venus634FLPx]	Obtains the current latitude and longitude of the robot.

3.3 Chassis

Nero has a main rectangular chassis with two large bike wheels up front and two smaller caster wheels on the back. A short projection in front of the bike wheels holds the LIDAR away from the rest of the chassis to allow for proper sensing. The main chassis holds two drive train blocks, two IP65 cases to house the electronics, a forward facing camera, and two plates to mount a payload.

This year’s redesigned chassis boasts several advantages over last year’s chassis. To counteract tipping problems from last year, the chassis was designed to move the center of mass to between the two sets of wheels. The placement of the payload, sensors, and drive train blocks puts the center of mass on the midline of the chassis 3 inches back from the front bike wheels. Nero’s wide stance, relatively long wheelbase, and placement of the payload low on the chassis further enhance Nero’s stability. Another improvement is in the overall structure of the chassis. Last year, a pod was connected to the chassis, but after encountering problems securing the pod to the chassis, the system was completely revamped. Now the entire chassis is one large, rectangular frame that is anchored by two aluminum square bars running along the bottom. Besides the obvious improvement in stability, Nero’s large frame also leaves plenty of room for carrying a large range of payload types.

Table 3: Mechanical Properties of Nero

Property	Specification
Material	6061 Aluminum and 1018 Steel
Length	48.25 in
Width	32.50 in
Height	25.50 in (without camera mount) 34.40 in (with camera mount)
Weight	45 kg
Ground Clearance	9 in
Payload Capacity	20 kg
Wheel Diameter	24 in
Drive Gear Reduction	75:1
Maximum Speed	4.86 mph
Speed on 15 ramp	2.5 mph
Torque required	43 N-m
Sprocket Reduction	10:17

3.4 Drive Train and Controls

3.4.1 Overview

The drive system consists of two identical units, referred to as “drive blocks.” Each one turns a 24-inch bicycle wheel. The large diameter of the wheel achieves smooth performance over rough terrain, yet is lightweight. The drive blocks were designed to be simple, robust, and easily replaceable. In the event of a drive train failure, a new block can be installed with simple tools in only a few minutes. The wheels can also be quickly interchanged.

Currently there is only one set, but in the future, various wheels may be adapted for different conditions, and it will be easy to install the most applicable pair for the scenario.

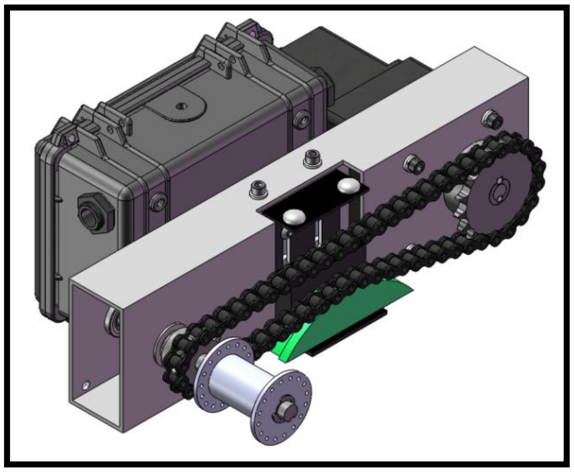


Figure 6: CAD Model of a Drive Block

Each block has a 220 W motor and gearbox combination. A pair of bearings supports the shaft of the bicycle wheel. The shafts of the motor and wheel are connected with sprockets and heavy chain. A tensioner unit keeps the chain tight, and provides some shock absorption to protect the motor from abrupt loading. The motor driver and USB interface are housed in a waterproof case; there are connection points for power and for signal from the main computer. This is the same system that was used in previous years.

3.5 Power and Electronics Protection

To provide for a total power consumption of 310W at a peak of 30A with a factor of safety of 2 (Table 5), a high power supply system that could handle high power and amperage was required. As a result, with a minimum run-cycle of 30 minutes and up to 10 cycles expected per day, 51.8V, 518Wh polymer Li-Ion rechargeable batteries were selected for their high power source and peak current tolerance of 40 Amps. Nominal 48V systems are common in professional applications in the industry and are the military-specification norm.

Table 5: Electronics Power

Part	**Power (W)**
Camera	1
Computer	40
INS	1
LIDAR	20
Motors (x2)	246
Wireless E-stop	1
Total	**309**

This system was preserved from the previous year and reused as there were no failures in the system and has proven to be robust after two continuous years of use. More can be learned about it in the 2009 IGVC tech report.

4. Software

4.1 General Framework

4.1.1 Platform

Nero runs on an Ubuntu Linux platform. Linux was chosen because it supports all of the libraries utilized by our software. In addition, Ubuntu is open source, well documented, familiar to the team, and has less processing overhead than Windows. A middleware called ROS (Robot Operating System) was used for the framework. Lastly, programs can be written in any of the languages supported by ROS, however we chose to rely on C++ to achieve maximum execution speed.

4.1.2 ROS

ROS is a communication framework developed by Willow Garage and is an open source project freely available for many different operating systems. ROS middleware is best described as a system of interconnected nodes. Each node is a program that connections to the ROS core via one of the communication paradigms. The two communication paradigms are publisher and subscriber, and server and requester. Independent of the communication paradigm is the messages that are sent. ROS defines many messages that allow many different types of data to be passed in between nodes and if the message is not defined yet, it can be very easily implemented. The actual connection between each node is made using TCP.

In addition to the communication framework provided, ROS also provides an open source professional level robotics community that is willing to discuss problems as well as share the solutions to their problems. Many of the projects that use ROS as a base for their platform share the nodes they write with Willow Garage, whom then distributes all of these packages in each new version of ROS.

Another great facet of ROS is the modularity that it forces between software nodes. Since the interfaces between nodes are well defined by the messages then a driver for a motor controller to receive commands from many different AI algorithms, without being changed. This is also important in to the simulator, as all of the AI algorithms do not have to be changed to interface with the simulator.

Lastly, ROS is also tightly integrated with OpenCV, which Willow Garage also maintains. This allows for a quick an easy interface between the communication framework and the OpenCV processing functions.

ROS functions as a versatile and powerful communication platform that can handle many different programming languages. It also provides the team with an invaluable source of knowledge on how others have performed similar actions. For these reasons it has been instrumental in the production of a robust software solution.

4.2 Simulator

Simulators are invaluable to allow us to test AI and path-planning code for our robot without taking the robot outside, preparing a course for it, and making it run. Instead, code testing can first be done in the comfort of our own home – as code is written and compiled, it can simply be run with ROS and Stage or Gazebo to see if it runs the way we expect it to. Testing various situations becomes trivial as well, as different worlds can be set up relatively easily to allow us to test special cases where we think our navigation system may have trouble. Without a simulator, we would have to find a clear field in the area where we can set up a test course, and from doing this in the past; we know that it is not a simple thing to organize. However, we cannot rely entirely on our simulator when testing. Everything about the simulator is idealized; it receives perfect data from its sensors, has ideal environments, and does not allow us to perfectly ensure that the code will run on the real robot as expected, given these differences between it and the simulated robot. Thus, the simulator is a great place to begin when writing code and testing, and as soon as we are confident in our code, it is necessary to test it on the real robot. Regardless of the limitations, the simulator makes the entire design process much more convenient.

ROS comes with two packages that allow for simulation of a robot's operation: Stage and Gazebo. Both Stage and Gazebo have many attractive features that suit our needs well. Stage is appealing because it is simple and easy to understand. The robot is a small box that navigates in an environment of stationary walls and boxes. It may not be sophisticated in this regard, but often sophistication is not needed when we only wish to test waypoint or obstacle avoidance code. Additionally, all the design parameters about the simulator environment and the robot in it go into one single file. The simplicity of this part of Stage is extremely valuable.

While Stage's simplicity is what makes it appealing, it is also what limits it. Simulation in Stage can never look like reality, because there is little you can do to customize the environment beyond the shapes of the walls and obstacles. Physically, it is very limited as well; the only mechanical things Stage really checks for are collisions between the robot and its environment. However, since our autonomous robot currently is essentially a box with sensors, this suits most our needs as of right now.

Gazebo is very similar to Stage, but has many more features. While objects in Stage can be given a height and can be viewed in an isometric 3D view, that is the extent of its capabilities with the third dimension. Gazebo, on the other hand, can fully take advantage of all three dimensions both visually and physically. Additionally, worlds in Gazebo contain a physics engine whose properties can be manipulated as desired. This can certainly be useful to test mechanical properties of the robot, such as the challenges it may face when climbing steep slopes or running over ditches. Since our purpose for using a simulator was only to test autonomous movement code, we hardly used this feature. However, it certainly has potential for future development. Additionally, Gazebo allows for much greater detail in the design of the world and the robot. A robot can have many parts, joints, and sensors, attached in any sort of structure, whereas all of this functionality in Stage is crammed into one little box that represents the robot. Also, objects in Gazebo can be much more complex than those in Stage; for example, one sample world that comes prepackaged in ROS contains a table with a coffee cup on it. Gazebo also allows for meshes to be overlaid on objects and environments to make things look more appealing. In Stage, the ground can only be white, but in Gazebo, the ground can look like grass or dirt! Of course, this feature would only be for show in some sort of demonstration of our robot's capabilities.

Of course, with more features comes greater difficulty in understanding how to use these features. Because it was so difficult to determine how to take advantage of many of Gazebo's features, our world in Gazebo looks a lot messier and less impressive than the simple, clean one that Stage provides. Also, in Gazebo, different parts of the robot and components of the world can be defined in many different files, and putting them all into one world requires one launch file to run them all. Not only does Gazebo require a solid understanding of the Linux file system, but it also requires lots of coding in xml with unusual syntax that is very poorly documented. In the future we intend to continue making a more realistic simulated environment for our robot, using Gazebo, but as of now Stage takes care of our needs.

4.3 Localization

The GPS is used to place the robot in a reference frame that is created by orbiting satellites, detected by the Venus GPS chip which allows for an update rate of 10Hz. This high update rate allows for increased accuracy in absolute positioning. The Venus GPS chip was paired with the GPSD library. This library allows for a GPS device to stream position data over any port and the data will be gathered, and parsed. Lastly, a compact and easy to use API is provided with C and C++.

Wheel odometry is obtained by measuring the difference in time between Hall effect sensors going high at several different locations on the circumference of the motor shaft. The RPM of the shaft is determined, then converted into wheel RPM by multiplying through by the gear ratio. Lastly the RPM is multiplied by the wheel circumference to get the forward speed of each side of the robot.

A complete localization solution is obtained with an Extended Kalman Filter. The Kalman Filter is fed data from the wheel odometry, the Inertial Measurement Unit, and the GPS. All of these different data types are used to compensate for the other data types' weaknesses. The IMU and the Wheel Odometry compliment the sudden spikes of the GPS. The GPS gives the IMU and the Wheel Odometry an absolute reference frame to work in.

4.4 Path Planning

For the navigation challenge, a text file containing all of the GPS coordinates is read in. Then GPS data is used to determine which starting point the robot is currently at and, then because there is no initial information about obstacles on the field, we do a brute force solution of the traveling salesman problem to determine the shortest route through all of the points. This algorithm pre-computes the distance between each pair of waypoints, then finds the shortest distance through all the points with a recursive depth-first traversal. It runs in factorial time, but the small number of waypoints negates any benefit to be gained from a more efficient but far more complicated heuristic algorithm.

For the autonomous challenge the center of the lane is determined and the robot is given a waypoint in front of it that is in free space for it to follow.

To determine how the robot behaves at any given moment, the obstacle avoidance code simply makes use of the angle between the robot's current heading and the direction from the current position to the next waypoint.

4.5 Obstacle Avoidance

We use a method of obstacle avoidance called a 'free space follower'. Essentially, the system scans the surrounding area in front of the robot, finds areas between obstacles that can fit the width of the robot, and chooses the free block closest to its desired direction. We chose this system because of the relative simplicity of implementation. It allows us to leave previously written goal point navigation code in place and make use of the LIDAR array in a straightforward manner. It also integrates well with lane detection code simply by marking lines as 'obstacles'.

Scanning for Obstacles

To locate free spaces, the LIDAR array mounted on the front of the chassis takes 180 distance measurements in one-degree increments. We take these measurements and filter them into a list of Boolean values that denote whether the vector at the denoted angle is free for a defined distance proportional to the velocity of the robot. We also mark vectors as non-free if they intersect painted lines within that distance.

Calculating Free Space

We next use the resulting array to find free space blocks that are wide enough to fit the robot chassis. We do this by counting the number of continuous of degrees of free space. If the block is wide enough to fit the robot chassis, we store the start and end points of the free space block. The spaces that we consider free blocks have a perpendicular width greater than that of the robot, plus a buffer of several degrees on either side. That is, a block is only free if the distance between the closer obstacle and where the other obstacle would be if brought forward to the same distance away is at least 50% greater than the width of the robot. This is to ensure that the robot does not damage itself or become stuck due to a bad reading from the LIDAR. We are then left with a series of start and end points of free spaces that could potentially fit the robot.

Selecting the Free Space Block

The path planning aspect of the code has already selected a desired direction using the positions of the robot and goal. We must then merge this desired path with the realistically possible paths found by the obstacle avoidance code. If the desired direction falls inside of a free space block, we allow the robot to continue along the desired path. Otherwise, we find the free space block closest to the desired angle and push the desired angle into this free space block with a buffer of 5 degrees from the edge of the block.

Areas of Future Research

In the coming semester, we would like to update the obstacle avoidance system to use a potential field method. A potential field system of navigation treats the robot as a particle with a negative charge, obstacles as negative charges, and the goal point as a positive charge. The robot is then repulsed from obstacles and attracted to the goal point. The advantage of this system is that it incorporates obstacle avoidance and goal point navigation into one system, rather than combining the results of two calculations. However the downside of the potential field approach is that the robot can get stuck in a local minimum and never reach the goal.

4.6 Lane Following

Lane detection in real time is a challenging and complicated task, requiring many stages of processing. To this end, the lane detection code consists of two main components that were developed throughout the year. The first component is responsible for actually detecting the lanes (if any) that are present in the image. The second component is responsible for, once given a lane, determining how far away the lane currently is from the robot and ensuring that the robot avoids the lane. These two pieces interact to allow Nero to navigate its terrain while successfully staying in the lane.

First Component (Lane Detection)

Canny/Hough Method

As far as detecting the lanes is concerned, our original solution implemented was Canny Edge Detection followed by a Hough Transform. The Canny algorithm works by differentiating the image, meaning that it looks for pixels where there is a large change in brightness. Once it has found such pixels, it marks them as an edge. Given an image, the Canny code would first preprocess the image, turning it into a binary image where edges were marked white and everything else was marked black. After this was completed, this binary image would be fed to the Hough transformation code. It would then process the image and output where it determined the lines to be. While this method sounds good in theory, when we tested the technique live we found that it was riddled with problems. First the Canny code was too sensitive to noise. Leaves, flowers, and shadows from trees would all show up as valid edges. It was impossible to tweak the parameters of the Canny code to get it to ignore the environmental noise but still detect valid lanes. Secondly, because the output of the Canny was too noisy, the performance of the Hough transformation suffered as well. There were too many edges for it to deal with, resulting in many detected lines that did not correspond to the lane lines in the image.

Canny/Hough Method Improvement

To address these problems, we recognized a fundamental flaw in the design of our algorithm. The Canny/Hough approach looked for straight edges only; it could not distinguish between a line from the shadow of a tree and a line from a valid lane. Thus, to rectify these issues, we scrapped the method in favor of one that was tailored specifically to lanes. The method we implemented, based on "An Effective and Fast Lane Detection Algorithm" by Chung-Yen Su and Gen-Hau Fan, takes advantage of the changing width of the lanes due to perspective. First the algorithm applies a binary threshold to the image, turning to black pixels those that are below the threshold value and setting to white pixels those that are above the threshold value. Once this is done, the algorithm iterates through each row of the image. At each new row, the algorithm calculates how large it expects the lane to be. It then sweeps through each row, marking contiguous white points that are the expected width (within a threshold of error) as points in a real world lane. Once it has done this for every point, the algorithm fits a line of least squares error to the points it found; this line is then output as a lane. The algorithm as it is described above would work fine if there were not the possibility of two lanes in the image. To deal with this, the images coming in from the camera are first split in two and the algorithm is run independently on each half. While this sounds like an easy fix, due to environmental noise or a line that cuts across the image, the algorithm would often find two lanes where it should have only located one. To pick the correct lane (from the possible two), the algorithm also computes an uncertainty level for each lane. The uncertainty is a weighted average of the error between each point that composed a lane and the least squares line that defines the line and the number of points that were used to find the least squares line.

Thus, if there were a lot of error between the points and the least squares line we would decrease our uncertainty in the lane. However, if there were many points that defined the least squares line we would increase our uncertainty in the lane (so we are more confident in a lane that resulted from a least squares line through 100 points as opposed to 10 points). The final uncertainty is then revealed to be a weighting of these two metrics.

This algorithm appears to consistently outperform the original Canny/Hough based method, as it is better at dealing with ground foliage and the shadows of trees. While it works well in shade, the algorithm does appear to have some difficulty finding lanes when the sun is shining very brightly. This is because the image coming from the camera has large patches of white in it, which make it very difficult to distinguish between lanes or just light. At the time of this writing, we are trying to boost the algorithm’s performance by making the threshold value a function of the image brightness. Also, it is worth mentioning that the many parameters input into the algorithm, such as the expected lane width and the correct threshold value, were determined by inspection. That is to say, we wrote software to display what the robot saw and where the lines were being drawn and tweaked the various inputs to the algorithm until we saw acceptable performance.

Second Component (Lane Integration)

Once the lanes have been detected, we must then integrate the information regarding the location of the lanes with the rest of the system. Next, given that the lane is in the form of a line, we check if it intersects a circle centered at the robot. This step is only possible because we derived an equation that relates every pixel in the camera to a real world distance in front of the robot (assuming that everything the robot is seeing with the camera is on the ground). If the lane does not intersect our imaginary circle, there is nothing to do and we proceed to wait for more potential lanes to be given to us. If, though, we have determined that a lane intersects the circle, we must notify other ROS nodes so that the robot does not come in contact with the lane. This is achieved by marking all degrees between the two points (we assume the lane will never be perfectly tangent to the circle) of intersection with circle as blocked, meaning that there is an obstacle there.

4.7 Actuation Control

The speed of the robot is dictated by how far away the robot is from an obstacle and by how close it is to the end goal. The closer it gets to its goal or to an object the slower the robot gets. Then a PID loop is used to ensure that each wheel is turning the appropriate speed as set by the intelligence. This helps reduce errors in position by making the robot follow its path more correctly.

5. Conclusion

Cornell Minesweeper is proud to present Nero V2.0 as the result of the two semesters of dedicated effort. The redesign of Nero has produced a robot that has increased software capabilities and fewer mechanical deficiencies. Nero V2.0 is a reliable and effective autonomous platform that will perform well at IGVC. We look forward to testing Nero’s capabilities in competition and continuing to improve upon our design in years to come.