Designing an Autonomous Off-Road Vehicle

Justin Poh, Claire Diehl, Ryan Eggert, Caleb Kissel
Fall, 2015

Under the guidance of
Professor David Barrett

Supported by:
Student Academic Grant (SAG)
Franklin W. Olin College of Engineering

Olin College
of Engineering
ROBOTICS

Intelligent Vehicles Laboratory
Franklin W. Olin College of Engineering
Needham, Massachusetts

TABLE OF CONTENTS

TABLE OF CONTENTS 2

Introduction 4

Initial Issues and Improvements to the Vehicle 5

- Adding Version and Source Control 5
- Studying the LIDAR to Robot Data Transform 6
- GPS Data Sometimes Shows Up as NAN 8
- Trouble Reading Encoders and Commanding Actuators 8
- E=Stop System Not Well Understood 9
- Firefly MV Camera Working with NI 9
- Changing the On-Board UPS 9
- New USB Hub Does Not Register Properly in Windows 10
- Designing the Mount for the Camera 10
- Studying and Repairing the Tilt Unit Motors and Encoders 10
- Upgrading LabVIEW 12
- Adding a Safety Beacon Light 12

Initial Work on Data Collection 13

- High-level Algorithm Design 13
- Adding the Hardware 13
- Writing the Data Collection VI 14
- Problems and Next Steps 18

Addition of the Robot Operating System (ROS) 19

- Hardware Additions 19
  - Intel NUC 19

Wireless Router 19
Network Configuration 20
Software Interface Between ROS and LabVIEW 20
Implementing the Large-Scale RGB-D Sensor 22
Hardware Modifications and Settings 22
Setting the Camera 22
Adjusting the Tilt Unit Encoders 22
Software for Merging Sensor Data 23
Controlling the Tilt Unit 23
Calibrating the Tilt Unit 24
Mapping LIDAR Points to Camera Image 25

Introduction

Human drivers alter their navigation and driving techniques to adjust to the terrain that they encounter. For example, upon encountering mud, a driver would limit quick acceleration to avoid digging the tires into the ground. Sand necessitates slow wheel turns to prevent the vehicle from spinning out. Rocks, potentially the most dangerous type of terrain, must be traversed with slow accelerations and turns to prevent a roll.

This type of conditional control is very desirable for an autonomous vehicle. Off-road driving cannot be safely attempted at high-speed without some form of terrain-dependent driving intelligence. However, currently vehicle sensor suites lack the ability to distinguish between different terrain types. LIDAR, automotive RADAR, and other ranging devices cannot provide information about surface materials. Conversely, cameras can guess material composition but are weak for navigation.

The team seeks to implement an RGB-D sensor system. This is a combination of a camera system and a LIDAR system, with data from both sensors combined to produce a camera image that also contains information about the 3D shape of the image that the camera sees. This type of information can allow an intelligent system to make assumptions about the types of terrain that the vehicle is approaching.

In addition to this work on terrain classification, the team also sought to replace the path planning layer of the LabVIEW code with a layer implemented using ROS. This will allow the vehicle to take advantage of existing packages for path planning and autonomy. It will also offer the added benefit of offering the capability for multiple teams working on the vehicle to essentially be able to interface with the vehicle in a plug-and-play fashion.

Over the course of this report, the work carried out over the course of the semester starting with the minor changes we had to make to get the vehicle running up to the technical research work we did for sensor fusion and ROS integration.

Initial Issues and Improvements to the Vehicle

Adding Version and Source Control

Since the vehicle has two computers, one running LabVIEW and the other running ROS, two different version control systems are used to keep all code under version control. In total, the team has 3 different repositories across the two version control systems - 2 in git, 1 in SVN.

To keep the LabVIEW code under version control, the team is using a subversion server hosted by Assembla combined with tortoiseSVN as the SVN client. We chose to use SVN because LabVIEW code is not text-based code and is, instead, binary code. As such, typical conflict resolution tools available using git version control do not work well with LabVIEW. SVN offers the ability to lock files while they are being edited to prevent others from also editing that file and causing a merge conflict. We chose Assembla as the host because Assembla hosts an actual SVN server that knows how to handle the requests to lock files. We explored using github, which claims to also support SVN. However, github merely "pretends" to be an SVN server and so, although SVN typically would support file locking, github's servers do not know how to handle the request to lock a file. To maintain our ability to lock files, we chose to use Assembla.The repository can be found at: https://www.assembla.com/spaces/olin-gator-research/subversion/source.

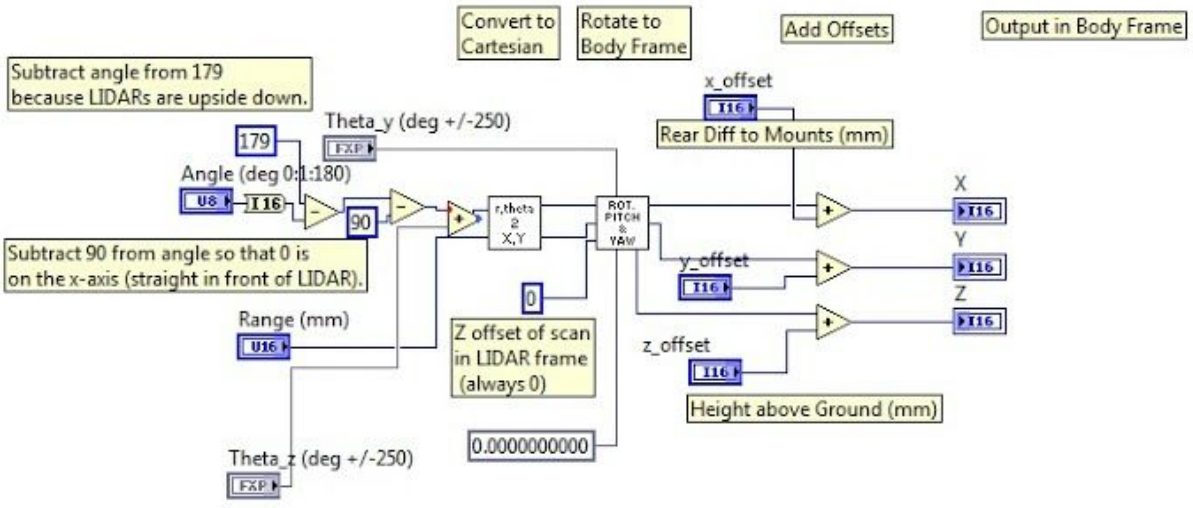
For the ROS code, we chose to use github since the ROS code is text-based and therefore will interface fine with git version control. The reasons we chose github are the same typical reasons that github is used as the version control system of choice for text-based code. The repository can be found at the following link: https://github.com/olinrobotics/GatorResearch.

We also have a repository that we used for the initial work on data exploration in pursuit of terrain recognition. Although we stopped work in this repository mid-way through this semester, the plan is to pick up this work again in the spring semester. This repository is also under git version control and can be found at the following link: https://github.com/olinrobotics/Gator_DataExploration.

Studying the LIDAR to Robot Data Transform

When we first ran the existing code on the vehicle, one of the key pieces we tried to find and understand was where and how the vehicle was transforming distance measurements from the LIDAR into points with a known 3-dimensional position. We eventually discovered that the LIDAR readings were received from the Sick LMS291 LIDARs as a stream of readings with a known range and angle measurement. Each reading in the stream was then passed to the VI named “Lidar2RobotTransform.vi” which transforms the points to be with reference to the ground directly underneath the rear axle of the vehicle.

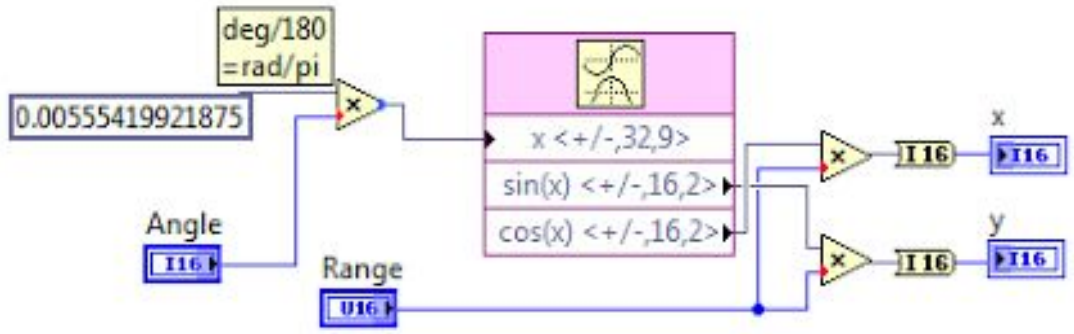
A picture of the block diagram for the VI is shown below:



As part of the transform process, there are two sets of transforms that are performed: one to convert the raw LIDAR data consisting of angle and corresponding distance measurement (i.e. polar coordinates) into cartesian X and Y coordinates. This is performed by the “r,theta 2 X,Y” block shown in the image above, also known as “Polar2Cartesian.vi”. Since this conversion is merely from polar to cartesian coordinates, the X and Y coordinates are still in the reference frame of the LIDAR. However, to be useful, the coordinates should be in the reference frame of the vehicle. Therefore, one more transform is performed by the “ROT, PITCH & YAW” block shown above, also known as “LIDARRoateYZ.vi”, to express each LIDAR measurement in terms of its X, Y and Z position with reference to the vehicle

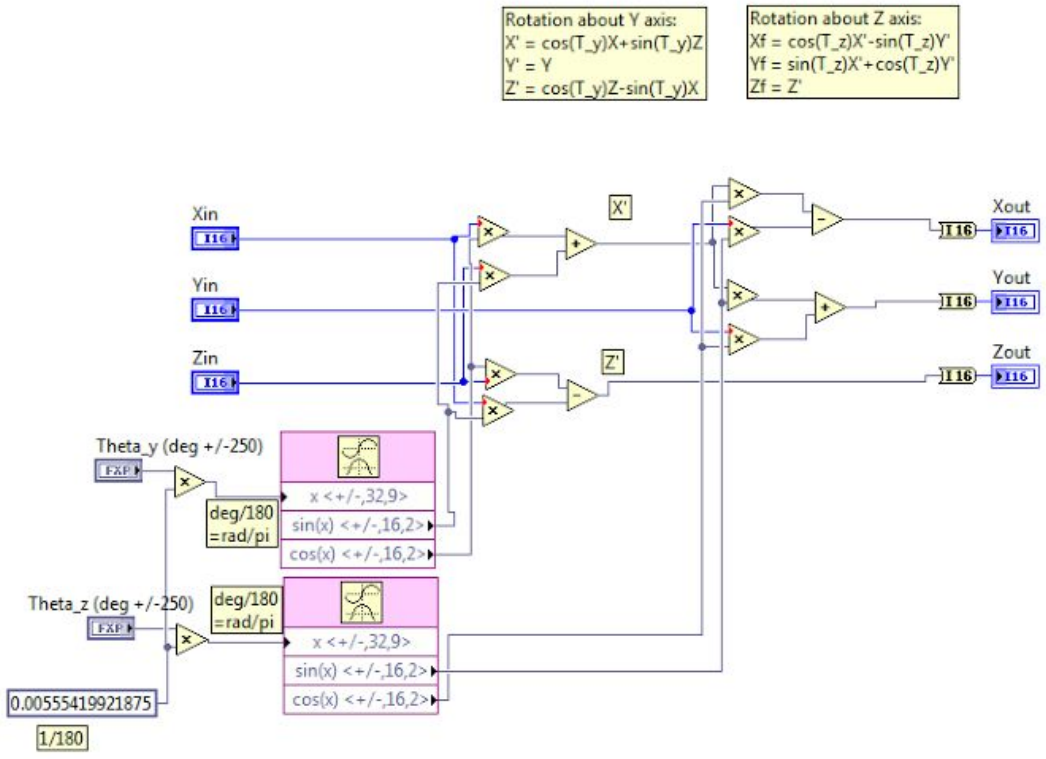
Finally, the last summations simply add on the translational offsets between the location of the LIDAR and the ground immediately underneath the rear axle of the vehicle.

The block diagram for the “Polar2Cartesian.vi” shown below is:



As can be seen, the VI essentially takes the angle and range data from the LIDAR and uses basic trigonometry to convert the polar angle and range data in the X and Y coordinates that are still relative to the LIDAR.

The block diagram for “LIDARRotateYZ.vi” is shown below:



This VI is a little more complicated to read. Essentially, it borrows the principles of rotation matrices that is usually used to rotate between coordinate frames in dynamics problems. As labeled in the documentation to the code shown above,, one set of sine and cosine calculations calculate the transform for the Y axis rotation (i.e. the tilt of the LIDARS that are caused by the tilt unit itself). The other set of sine and cosine calculations compute the transform for the Z axis rotation caused by the way the LIDARS are positioned in their mounts such that they are rotated outward by 45 degrees. The resulting output of this transform is the X, Y and Z position of the point relative to the position of the LIDAR with the tilt already accounted for.

As a reminder, this reading is then modified near the end of “LIDAR2RobotTransform.vi” to add the additional translational offsets in order to get the coordinates with reference to the ground immediatley below the rear axle.

GPS Data Sometimes Shows Up as NAN

When initially powered-on, the Gator read useless GPS information, often giving readings as NaN. However, the NavCom unit in the electronics box indicated that satellites were locked and GPS was reading properly. Using NavCom’s GUI, “StarUtil”, the team was able to correct the problem. The NavCom unit was outputting NCT data to the PC instead of the expected NMEA sentences. The StarUtil GUI allows for switching the COM port setup such that NMEA data was broadcast along the correct line, to the PC. With this problem resolved, GPS data was highly accurate. The team recommends troubleshooting GPS problems with this GUI open, to help isolate the location of the fault. If StarUtil reads GPS information properly, the problem is in reading or otherwise interpreting GPS data.

Another noteworthy issue is that the vehicle is very dependant on GPS signal. This is not an issue during normal operation, but can cause troubleshooting problems while indoors. The vehicle refuses to assume any position without GPS data. While indoors, it is recommended to switch to INS data only; this can be done using the midbrain front panel.

Trouble Reading Encoders and Commanding Actuators

At the beginning of the semester, the robot was nominally drive-by-wire capable. All motors and actuators, as well as all necessary encoders, were installed. However, after three years none of these systems were functional when the gator was brought online. The problem was narrowed down to two issues with cabin wiring. The connectors in the cabin are vulnerable to physical disruption, especially those that run underneath the passenger’s feet. This includes the power and signal wires for the gas, brake, and steering controls. These cables must be exposed to allow disconnection of key components, but these exposed connections proved problematic. Several of these wires appeared connected, but had to be re-seated properly before they became functional. If the actuators or encoders do not respond to commands or behave erratically, it is recommended to check all connections, especially those that are in exposed positions.

Another major issue encountered with cabin encoders involves the vaguely-named “Cabin 5V” cable. This cable is a red-and-black wire pair that runs from the 5V terminal box, and connects to a cable from the bundle of cabling that passes through the hole in the bottom of the electronics box with a power pole connector. This cable splits somewhere inside the vehicle and provides much of the 5V power for the entire cabin, including to many of the encoders, with unlabelled branches. If several seemingly unrelated cabin components fail at once, this cable may be disconnected.

With both of these problems resolved, the vehicle is operationally drive-by-wire, and can be controlled manually. Steering is acceptably fast, and the gas and brake actuators function as designed.

E=Stop System Not Well Understood

In addition to the e-stop system not being understood, at the beginning of the semester it did not appear to be doing anything.

The Gator has two e-stops. One is located in the cab and one on the back of the electrical box. These are connected in a parallel manner to a relay in the electrical box. If either e-stop is pushed, power is cut to all actuators. This means that the gas pedal and brake actuator clutch are released and the vehicle becomes drivable by a human.

There is a boolean value read by LabVIEW regarding the state of the e-stop system. It appears that not all of the higher-level VIs use this boolean correctly but it has been verified to work.

Firefly MV Camera Working with NI

The Greypoint Firefly MV camera is the camera that we chose to be used on the front of the Gator. It is a relatively cheap color firewire camera that was accessible to us.

This camera has an issue connecting to both LabVIEW and NI MAX. The problem causes the image to only be accessible as a black and white image. When attempting to snap an image in MAX, the cameras causes MAX to crash. Similarly, it causes vision acquisition assistant in LabVIEW to hang or crash while taking a test image. Some potential fixes made the camera unable to connect with MAX at all. This is solved when LabVIEW is upgraded to LV2014 SP1.

The camera image originally was displayed as a mirror image of what it was capturing. To fix this the registers of the camera are edited such that the image is correctly displayed.

Changing the On-Board UPS

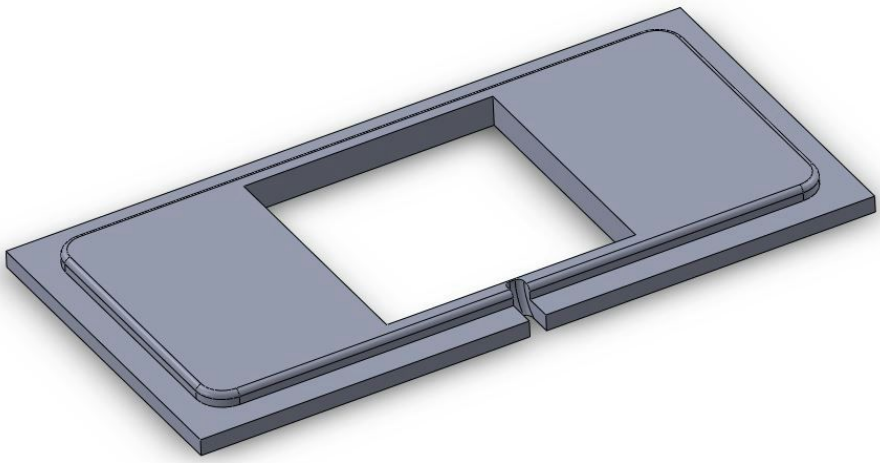
During the semester the On-Board UPS stopped remaining powered on. If the button was held it would remain on but when released would power off. We replaced it with a BR1500G from APC. This model was recommended by APC for a power draw of 750W for 15 minutes with 10% overhead for future power increases. It may not work with newer computer models as it does not have the capability to generate a pure sine wave. It does however work with the Dell R5400 workstation that we are using. When supporting the server and monitor, the UPS is capable of sustaining power for 38 minutes.

New USB Hub Does Not Register Properly in Windows

The USB hub was added to allow more convenient switching between peripherals with two machines installed. However, it suffers from intermittent outages

Designing the Mount for the Camera

The missing original camera was mounted on a threaded rod that had a ball and socket pivot point. By removing the rod and this joint, then mounting the new Firefly camera on the exposed thread, the camera was able to fit under a 12x9 inch fish tank to protect it from damage. The mount for the fish tank is a set of two 3D printed panels with slots for weather stripping. The fish tank is held on with two velcro straps.



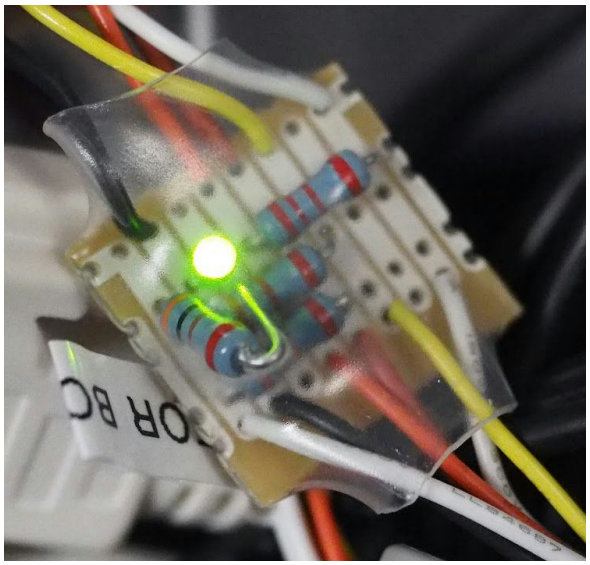
Considering the closeness of the camera to the glass of the fish tank, the size of any lenses could be a problem in the future.

Studying and Repairing the Tilt Unit Motors and Encoders

The tilt units are custom built by a previous SCOPE team, and all hardware for them was already installed. They are driven by a 24 volt brushless DC gearmotor and sealed inside an aluminum housing. Each output shaft is read by a 10,000-tick/revolution quadrature encoder, and two NI 9505 modules have been mounted in the electronics box to handle the two tilt units. However, there was no pre-existing code to drive or read these units, and all of this code had to be written this semester.

In order to make the tilt motors run with the code as written, one last step must be taken. The motor driver interface takes a ‘current limit’ input, determining the maximum current that the 9505 will give to the motor. Through experimentation, it was determined that the user input is scaled by 6.2; that is, increasing the ‘current limit’ integer by 1 increases the maximum output current by 6.2 milliamps. This arbitrary factor may encourage user error, as the motors will not run if the current limit is set too low. This control should be set no higher than 419, which equals a maximum output current of 2.6 amps (the motor’s rated max current).

The encoders, as installed by the SCOPE team, were mis-wired, with input power routed to the shielding. This prevented the encoders from receiving power but, other than energizing the shielding, did no damage to the hardware. The encoders were re-wired and power was confirmed to be routed properly. In addition, the encoders were open-collector encoders that required pull-up resistors in order to output meaningful data. Because the previous SCOPE team did no account for these pull-up resistors in their design, we added small boards in series between the encoders and the NI9505 module. A picture of these boards is shown below:



These boards were constructed using two small proto-boards with cables with appropriate the encoder cables inside the electrical box. These pull-up boards also include a small LED that confirms power. This offers a visual confirmation that the encoders themselves are powered and helps during the debugging process.

With these modifications complete, the encoders read position data properly. The tilt system is considered operational, reading positional data and moving to commanded positions as desired.

Upgrading LabVIEW

We installed LabVIEW 2014 SP1 alongside LabVIEW 2012. This was necessary in order to use the LabVIEW/ROS package [discussed later]. This also made the software on the vehicle compatible with

Adding a Safety Beacon Light

Installed an orange, revolving safety light. It is on whenever the Gator has power so generally while it is in the space we remove the

Initial Work on Data Collection

When the project first began, the original goal was to design an algorithm that would harness machine learning to determine the features needed and then use those features in a classifier to classify terrain. To begin this work, we placed a priority on getting the hardware ready to collect data sets because without datasets, the actual machine learning could not take place. Although we eventually did not have sufficient time to actually write the algorithm to carry out terrain classification, we did begin some initial steps to get the system ready to do that.

High-level Algorithm Design

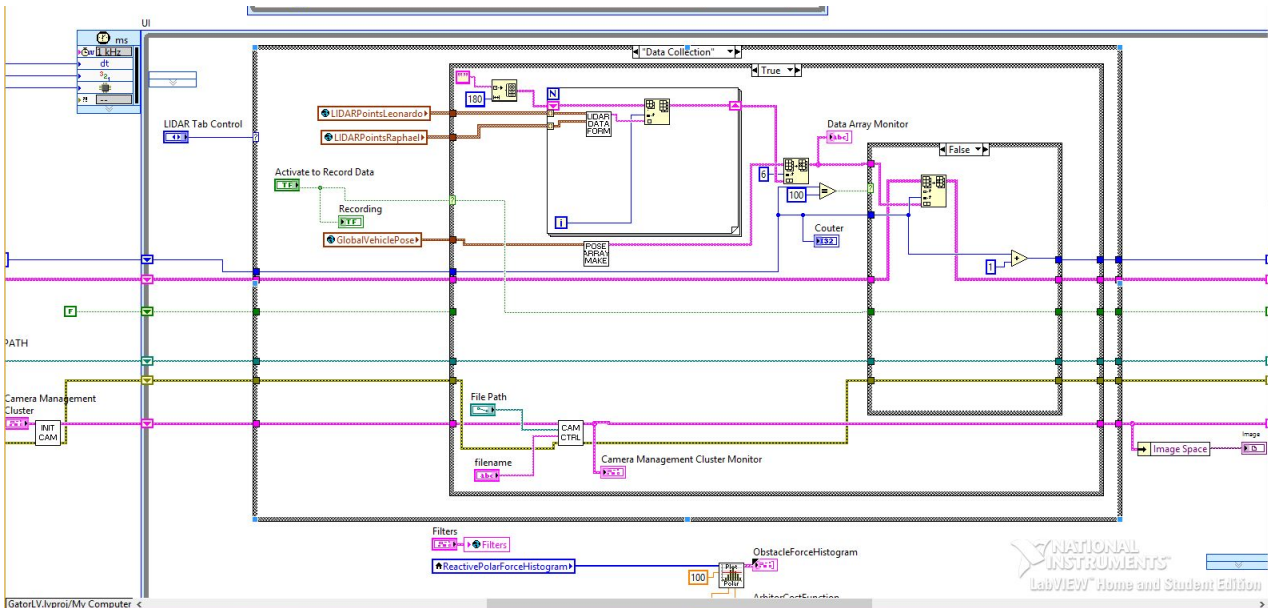
At the start of the project, we decided to start with a simple algorithm and gradually make it more complex as needed. Based on our review of several research papers that have attempted terrain recognition, we decided to go with the paper by [INSERT REFERENCE HERE]. In this paper, they propose that terrain categorization can be done by combining the results of LIDAR data processed by an SVN classifier and textural information obtained off of the camera image. In addition, we also decided for the purposes of training the classifier, we should also store GPS and heading information to better constrain the process of feature identification.

Adding the Hardware

Since the vehicle already had 2 Sick LMS291 LIDARs already mounted, we did not need to add those sensors. The vehicle also already had GPS and heading capabilities already so we did not have to add those sensors either. The only sensor we did need to add was the camera. For this purpose, we chose the Firefly MV camera by Point Grey Research. The reason we chose this camera is because it is a reasonably cheap camera that is meant for carrying out machine vision and communicates via firewire, which is a more secure protocol for data transmission compared to USB or ethernet. Because the camera supports firewire400 and the windows server only has firewire800 ports, we needed to use a bilingual cable to connect the camera to the server. We could not find a sufficiently long bilingual cable and therefore added a firewire hub in order to extend the bilingual cable we had using another, longer firewire400 cable.

Writing the Data Collection VI

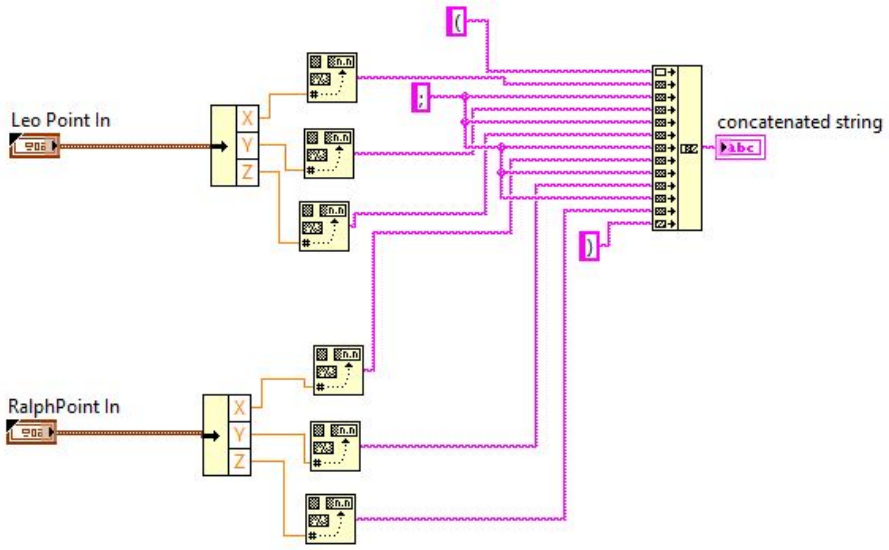
With all the sensors in place, we then wrote the VI to collect and store the data. We decided to locate this data inside “Midbrain_DataCollect.vi” in the case statement of the UI loop:



There are three key tasks that this section of code performs:

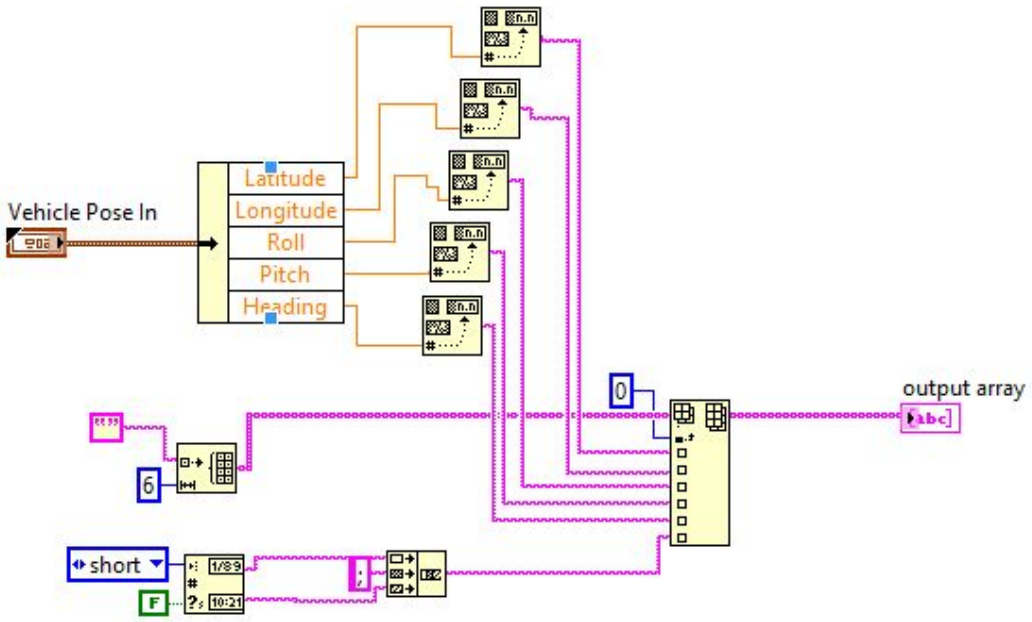
1. Take the X, Y and Z coordinates for each point for each LIDAR (total of 6 numbers) and convert that into a string of the 6 numbers in comma-separated format. We do this for each of the 180 points that make up a complete LIDAR scan to form a 180 element array where each element is a string of 6 comma-separated numbers
2. We also get what is known as the Vehicle Pose Data: GPS latitude and longitude, roll, pitch and yaw in addition to obtaining the timestamp when the data was recorded. All this information is again converted into a comma-separated string to form a single element.
3. Finally, we need to control the camera to record an image at a regular interval using the Grab VI.

The VI that reformats the LIDAR data is shown below:



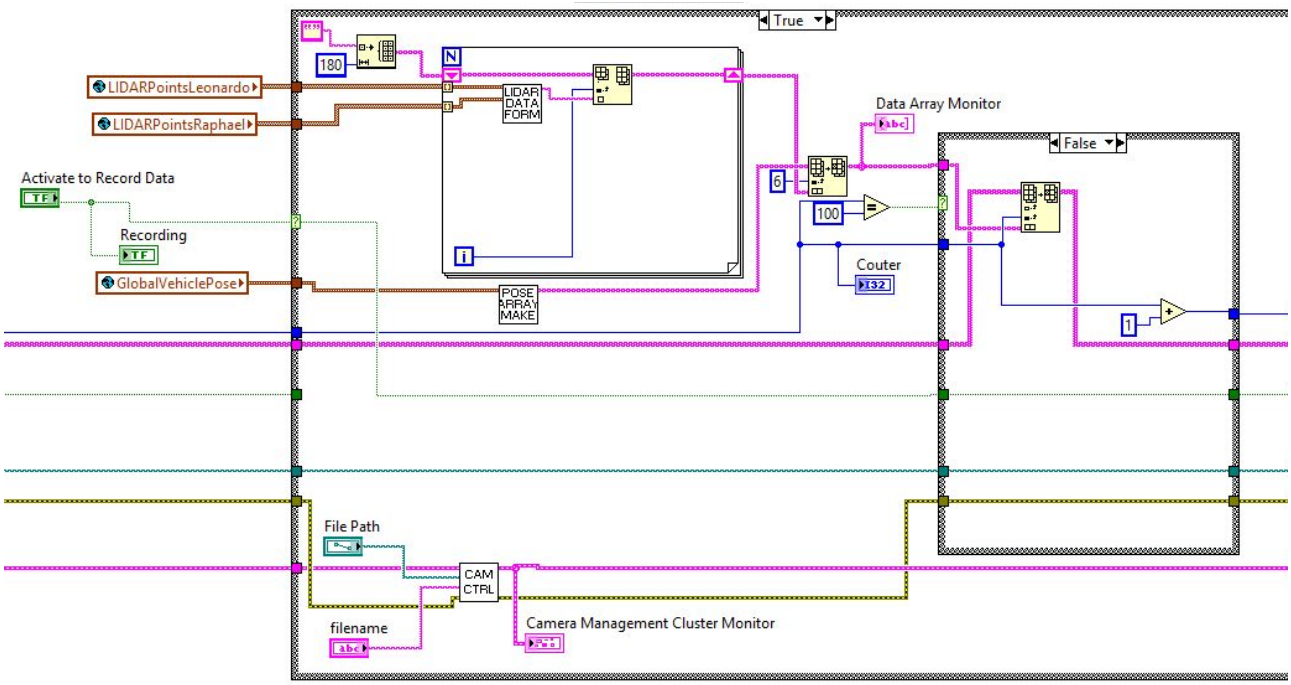
As described earlier, it takes the LIDAR point from both the left and right lidar, converts each coordinate into a string and then combines it all together with commas between the coordinates to make one element of 6 numbers.

Similarly, the VI that reformats the pose information is shown below:



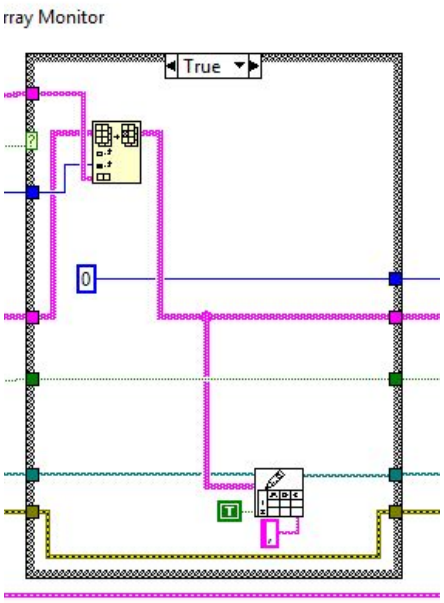
As can be seen, the latitude, longitude, roll, pitch and heading are all recorded along with the timestamp and saved as an array of 6 elements.

Once both arrays are formed, the code running in Midbrain_Datacollect.vi puts them together as shown in the image below:

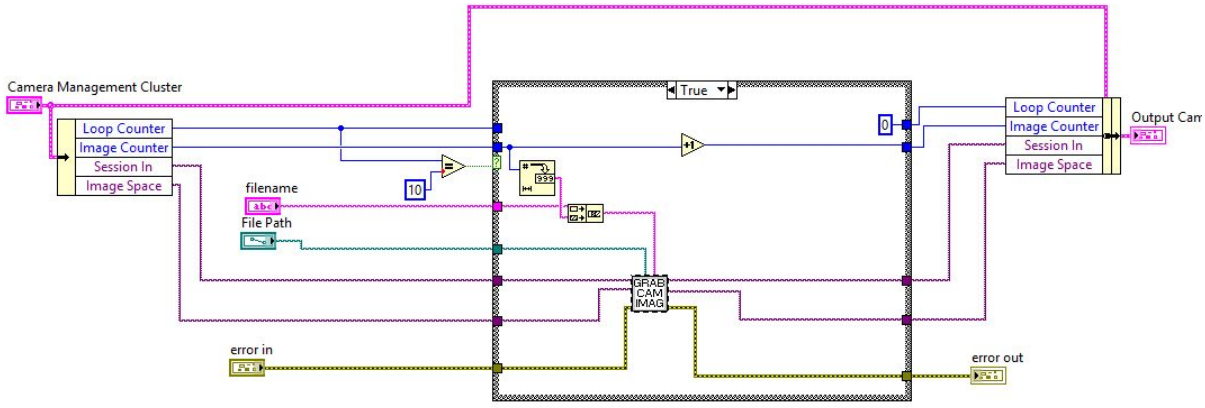


As can be seen in the image above, the Replace Array Subset block adds the LIDAR data to the end of the Vehicle Pose data array, forming one large 186 element array where the first 6 elements are the vehicle pose information and time stamp and the last 180 elements are the LIDAR data points.

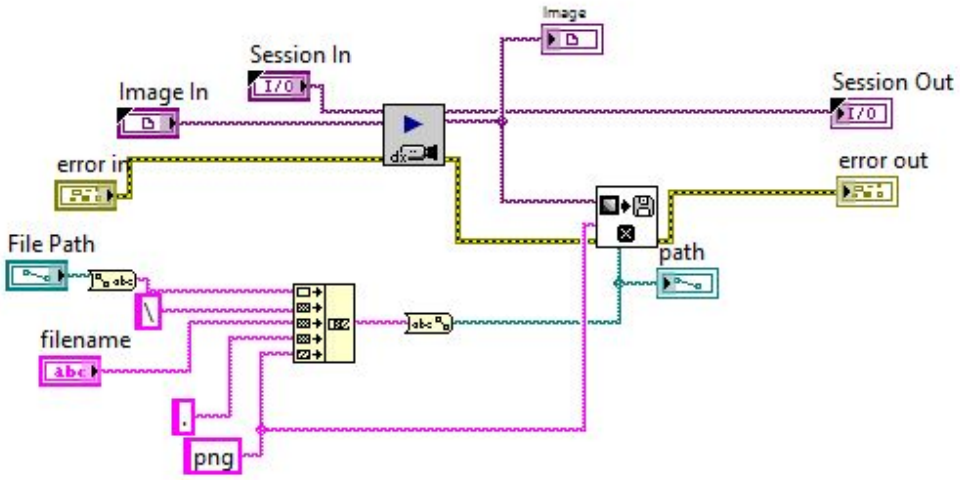
Finally, the case statement to the right of the image above essentially decides when to store the data into a csv. In our case, we have chosen to write data to the CSV every 100 readings. As such, the while loop and case statement will count 100 readings. When the 100 readings have been made, the case statement will save the data to a CSV using the code shown in the image below:



Finally, the camera control is shown in the image below:



As can be seen, the camera is controlled in a similar manner to the way in which the code decides when to write the LIDAR data to CSV. Essentially, for every 10 readings of the LIDAR, one camera image is recorded. Once the controller has decided a camera image does need to be recorded, the VI that actually captures and stores the camera image is shown below:



As can be seen, the image capture is also relatively simple. Essentially, the grab VI is used to grab an image from the camera. A separate block is then used to store the image to the given filepath using the appropriate name.

Problems and Next Steps

This semester, we tried to use this setup to record data. Although we were successfully able to obtain data, we found that the data was unusable because we were unable to restrain the LIDARs enough to keep them from moving without engaging the motors of the tilt units. As such, we decided to put the terrain categorization task on hold in order to properly set up the tilt units to maintain the desired tilt.

In terms of terrain categorization, next steps would include collecting new terrain data using the improved hardware on the vehicle. Once usable datasets have been collected, terrain categorization can then proceed either using the original algorithm we had wanted to try or using some other method.

Addition of the Robot Operating System (ROS)

Hardware Additions

Intel NUC

ROS requires Linux architecture, but the server must remain running Windows in order to interface with the NI real-time layer. For this reason, an Intel NUC running Ubuntu 14.04 was installed inside the electronics box. The NUC is a powerful but compact computer, only about 4.5” inches square and less than 1.5” inches thick. It has four USB 3.0 ports, an RJ45 port, and a reasonably capable i5 processor. This NUC adds relatively little to the power requirements of the vehicle.

Wireless Router

We added a TP-LINK TL-WR841N wireless router to the vehicle’s internal network. This remains connected to the existing 8-port ethernet switch. With the addition of our Intel NUC and ROS’s emphasis on communication between computers on a network, the switch alone would not be a viable solution. The router provides a centralized location to assign network addresses and manage network devices. It also enables additional computers to wirelessly monitor robot processes in the field.

When parked in the garage, the router also enables all computers to be easily connected to the Olin network (and thus the internet). To do this, connect an ethernet cord from an Olin network jack to the blue WAN plug. The router will automatically configure the WAN and route Olin network and internet traffic correctly to and from devices on the gator’s network.

The router has had its firmware updated to DD-WRT. This is a linux-based firmware for routers which makes possible additional advanced network management tools.
The router’s user interface requires a login and a password in order to change most settings. This has been configured to use username `gator` and password `deere850`.

Network Configuration

In order to leverage the networking strengths of ROS, we have created an internal vehicle network.This is managed Currently, there are two computers connected via ethernet to this network--the Windows server and the NUC.

The router is configured to be locally addressable as 192.168.1.1 [subnet mask 255.255.255.0]. It has a LAN domain of "`gator`". This means that network devices should be addressable as `hostname.gator`. The Windows server's hostname is `gator-lv-server`. The NUC's hostname is `gator-ros-master`.

ROS can run into problems if networked computers' respective system times vary. To combat this, the NUC has been configured to act as a Network Time Protocol [NTP] server. The Windows server will reference the NUC to set its own clock. As long as the Windows server can communicate over the network with the NUC, the Windows server should keep its system time in sync with the NUC. With the addition of the router, we plan to shift the timekeeping responsibilities to the router. DD-WRT allows the router to additionally serve as an NTP server. This change has not been performed yet. Once it has, computers connecting to the vehicle's network, especially to publish or subscribe to ROS topics, should point to the router's address [192.168.1.1].

The router's wireless radio is turned off when parked in its garage to prevent interference with Olin's network. A wireless network has been configured for use in the field. When the wireless radio is switched on from router's web interface, a network named "GATORNET" will be broadcast. This has been configured with WPA2 Personal Mixed authentication--the password to the wireless network is `deere850`.

As a disclaimer, work on integrating ROS and LabVIEW is ongoing and specific network configuration details are subject to change. The above information is accurate as of December 21, 2015.

Software Interface Between ROS and LabVIEW

It is crucial to this project that we find a way to interface with ROS from LabVIEW. Our LabVIEW code has the ability to interface with the vehicle's motors and sensors through NI hardware. Much of this sensor data will need to be relayed to ROS topics. The vehicle's motors and actuators should be addressable and controllable through ROS. There may exist a few ways of doing this, but one promising means is a set of LabVIEW files written by Tufts University's Mechanical Engineering Department and Center for Engineering Education and Outreach.[1]

[1] https://github.com/tuftsBaxter/ROS-for-LabVIEW-Software

This package provides VIs for initializing, publishing to, subscribing to, and closing ROS topics. It also provides extensible means for creating ROS messages from LabVIEW data types. Additionally included are VIs for controlling a Rethink Robotics Baxter robot and VIs for emulating a ROS master in LabVIEW.

Our goal is to have parallel loops publishing sensor data and subscribing to motor command topics. These can all be run at different rates and independently turned on and off. Our initial tests of the LabVIEW/ROS package showed a robust ability to close and reopen connections to topics from LabVIEW.

We have faced a number of challenges in achieving stable communications between our LabVIEW and ROS machines. While we have observed successful publishing from LabVIEW and subscription from Ubuntu/ROS, this behavior has been inconsistent, especially when publishing to multiple topics in parallel. We've been troubleshooting network connectivity issues (e.g. changing network configurations, inspecting packets using tools like Wireshark, as well as investigating the structure of LabVIEW/ROS package. We've also collaborated with Whitney Crooks and Dr. Chris Rogers of Tufts University, two authors and users of this package. We have identified a few bugs and some potential workarounds. Current work is targeted towards resolving a TCP timeout error which occurs when a published topic receives a new subscriber.

This area is currently under very active development. The details of our implementation and complete instructions for use will be written once our solution has been shown to be robust

Implementing the Large-Scale RGB-D Sensor

Hardware Modifications and Settings

Since the vehicle already had a camera, LIDARs and tilt units to tilt the LIDARs for scanning, we did not need to add any new hardware to the vehicle. However, we did need to set the camera and make changes to the configuration of the tilt units.

Setting the Camera

In order to match the color data from the camera to the range data from the LIDARs, we need to know the focal length of the lens that the camera is using. We currently have a zoom lens with a focal length range of 3.8 - 13mm on the camera right now to allow for adjustments should they be needed. Since we need to know the exact focal length, we have chosen to set the focal length to the maximum 3.8mm to obtain the widest possible image while still knowing the exact focal length of the lens. We then set the aperture and focus to get the image in focus and tightened down the knobs.

Adjusting the Tilt Unit Encoders

In order to use the tilt units to get the angle of tilt of the LIDARs, we needed to be able to calibrate the position of the tilt units to some known angle to start so that we would know what angle the tilt units were at at all times. Since the encoders are quadrature encoders with an index, we use the index as the known point to calibrate against.

While testing, we discovered that the NI9505 stock code was much better at handling positive values of encoder counts. So, we placed the index of the encoder at the lowest possible tilt position of the tilt unit such that counts of the encoder greater than zero would tilt the LIDAR upward toward being horizontal.

Software for Merging Sensor Data

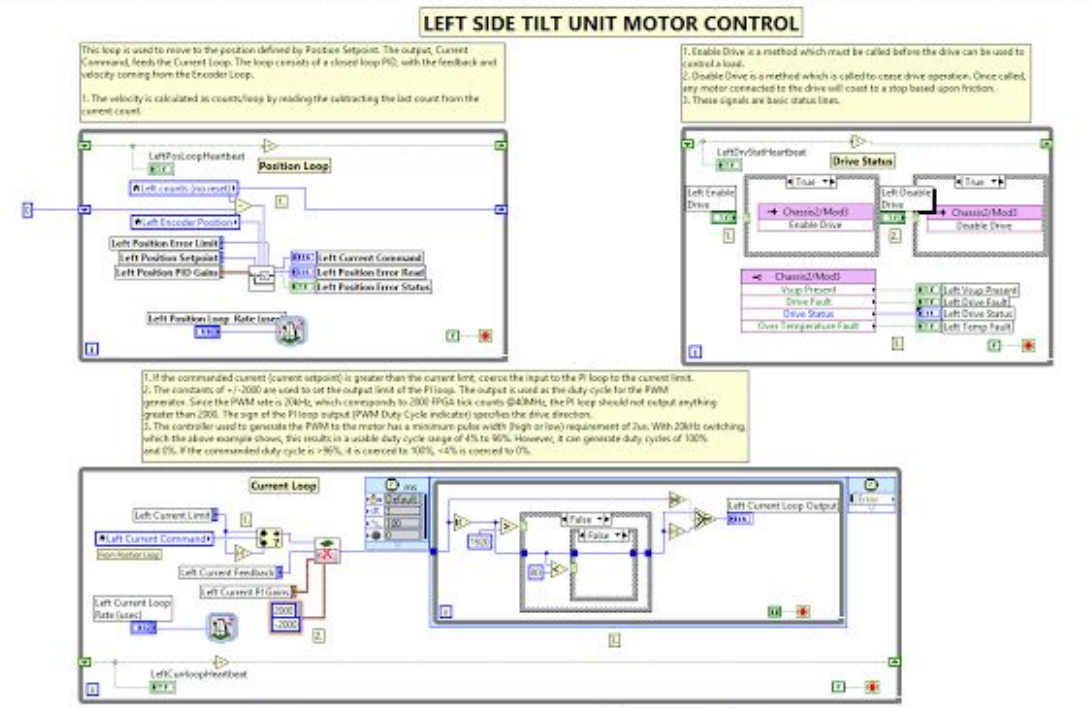
The software for merging sensor data was not fully written by the end of the semester in part because of several complications along the way. We did, however, manage to get the following features written:

1. Code to control the tilt units once this calibration is complete
2. Code to move the tilt unit to the calibrated position prior to starting to run the robot
3. Code to map known LIDAR points in X, Y and Z to points in the camera image

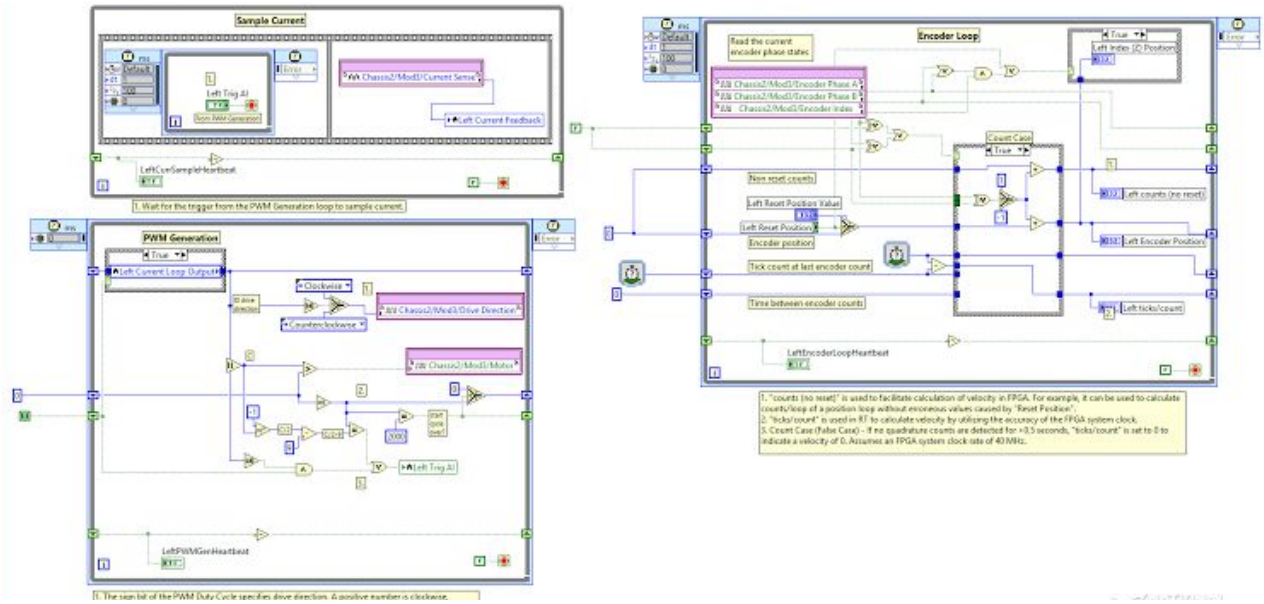
The overall concept behind operating the sensor suite on the vehicle is that upon initialization, the tilt units should both begin to move until they are both at their calibrated start positions as described in the sections below. Once they are at their calibrated positions, they can then be commanded just like any other standard motor controller. The following sections will discuss this code in greater detail.

Controlling the Tilt Unit

Since the tilt units are being controlled using NI9505 modules, the code to control the tilt units is simply the stock motor control code provided by National Instruments. Because of the size of the code, we will show it in two parts. Below is the first part that consists of the Position Loop, the Drive Status loop and the Current Loop:



The second part is the Sample Current loop, Encoder Loop and PWM Generation loop:



Calibrating the Tilt Unit

We now needed a way to command the tilt unit to find the calibrated position and stop as soon as it found that position. In our case, the calibrated position is the position of the index as described in the hardware modifications section above.

Since the control code described in the previous section is designed to move the tilt unit to a specified position, we could not just use that code to calibrate the tilt unit. As such, we decided to use a process that required a change to both the FPGA and real-time layer code.

In the FPGA, we basically added a flat sequence to the tilt-unit control portion of the code. We then added a panel in front of that control portion and duplicated just the current drive, index pulse reader and the enable drive, disable drive sections of the code. These are the only sections we need because the current drive portion allows us to command a steady current to the motor, the index pulse reader tells us when the index is found and the enable drive, disable drive sections of the code will allow us to enable or disable the NI9505 module as needed. It is also worth noting that the stop conditions for all of the while loops in the first panel of the flat sequence are tied whether the index pulse is true or false. That way, when the index pulse is found, all of the other loops should stop execution and the FPGA should advance to the next pane in the flat sequence, which should be the control portion.

Mapping LIDAR Points to Camera Image

The FPGA transforms the LIDAR data to produce points with known X, Y and Z coordinates with respect to the ground below the rear axle of the vehicle. In order to then locate these points in the camera image, we can use a technique known as perspective projection[2]. In essence, perspective projection uses the vanishing point of the image in addition to the properties of the camera, lens and image in order to project the known 3D object onto the image plane and return the pixel locations of that point in the image.

We begin with the X, Y and Z coordinates of the point in “world coordinates” which, in our case, is the coordinates of the LIDAR scan that has been transformed in the FPGA to be with reference to the ground underneath the rear axle of the vehicle.

$$\overrightarrow{P_w} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

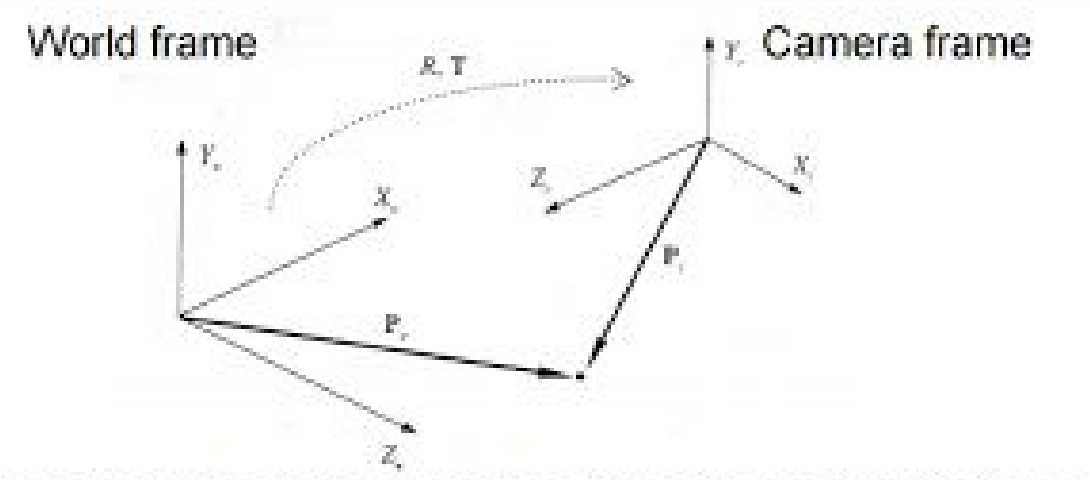
We start by translating the reference frame to the camera reference frame. Since the camera is located on the centerline of the vehicle and the camera and LIDAR are located at approximately the same height and distance along the centerline of the vehicle, the translation of the point in world coordinates to camera coordinates is:

$$\begin{bmatrix} x - T_x \\ y - T_y \\ z - T_z \end{bmatrix}$$

where T_x is 86 inches, T_y is 0 inches and T_z is 37.25 inches for the John Deere XUV 850D vehicle used in this research project.

Now the coordinates have been translated appropriately, we now need to rotate the coordinates appropriately in order to align the world frame with the camera frame.

[2] https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect5.pdf



Based on the coordinate frame of the vehicle as prescribed in LabVIEW and the camera frame shown above, the rotation matrix to align the coordinate frames is:

$$\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \end{bmatrix}$$

Therefore, the coordinates in terms of the camera frame is:

$$\overrightarrow{P_c} = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x - T_x \\ y - T_y \\ z - T_z \end{bmatrix} = \begin{bmatrix} T_y - y \\ z - T_z \\ T_x - x \end{bmatrix}$$

Now that we have the LIDAR readings transformed into the camera reference frame, we now need to carry out the actual projection to project the object onto the image plane. That transform is:

$$\begin{bmatrix} -d\frac{P_{cx}}{P_{cz}} \\ -d\frac{P_{cy}}{P_{cz}} \end{bmatrix} = \begin{bmatrix} -d\frac{T_y - y}{T_x - x} \\ -d\frac{z - T_z}{T_x - x} \end{bmatrix} = \begin{bmatrix} X \\ Y \end{bmatrix}$$

where d is the focal length of the lens (3.8mm in this case).

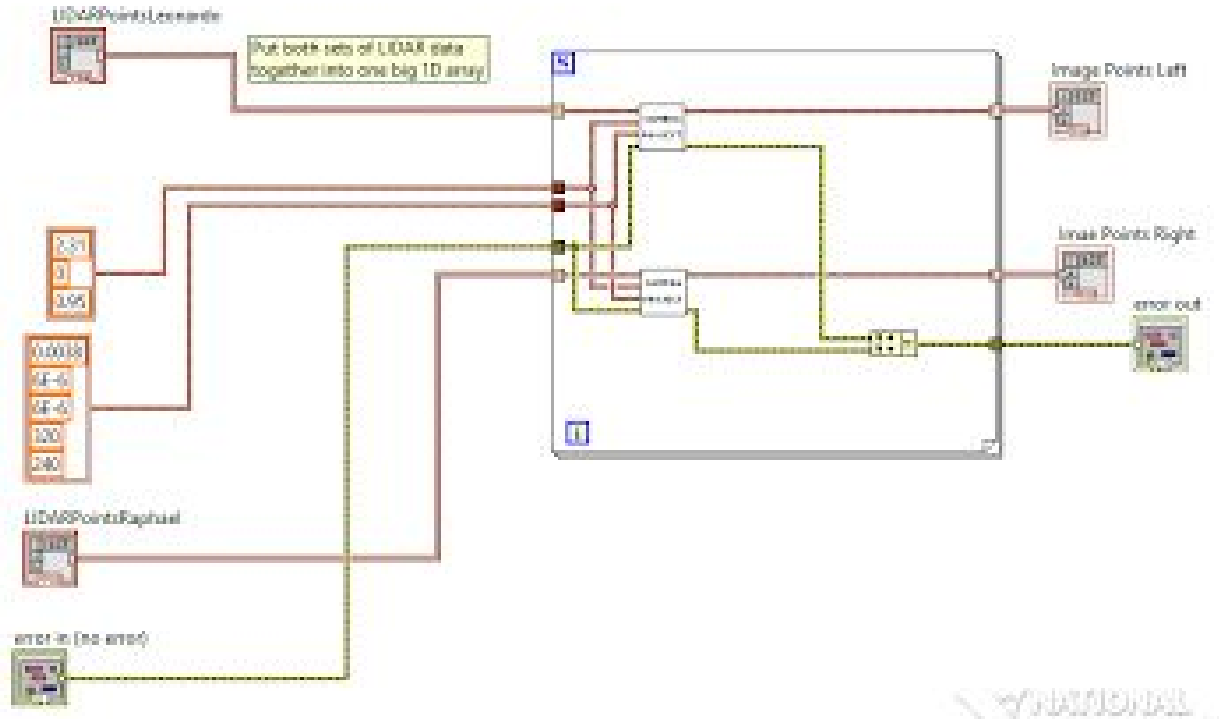
Finally, we can calculate the pixel location of the object in the image using:

$$x_{image} = -\frac{X}{S_x} + O_x = -\frac{-d\frac{T_y - y}{T_x - x}}{S_x} + O_x$$

$$y_{image} = -\frac{Y}{S_y} + O_y = -\frac{-d\frac{z - T_z}{T_x - x}}{S_x} + O_y$$

where S_x is the size of the pixel in the x direction (6 micrometers for the Firefly MV camera), S_Y is the size of the pixel in the Y direction (6 micrometers for the Firefly MV camera), O_x is the location of the center of the image in the x direction (320 pixels since the image in LabVIEW is 640 x 480 pixels) and O_Y is the location of the center of the image in the Y direction (240 pixels).

Based on these calculations, the top-level LabVIEW VI that performs this calculation is “ConvertLIDARtoCam.vi”:



This VI simply takes the the array of LIDAR points as well as the constants needed to perform perspective projection. The upper, 3-element array is known as the Extrinsic Parameters and they are the X, Y and Z translations from the ground beneath the rear axle to the location of the image sensor in the camera. The lower, 5-element array is known as the Intrinsic Parameters and they are properties of the camera and lens. So, first element is focal length of the lens in meters, second and third elements are the pixel size in x and y in meters and the fourth and fifth elements are the location in X and Y of the center of the image in pixels.

The actual VI that performs the perspective projection is “PerspectiveProjection.vi”: