IGVC 2018 Code Repository:

Author: Connor Fuhrman

connorfuhrman@(outlook.com)/(email.arizona.edu)

5128970709

All code used in the 2017-2018 year for the 2018 IGVC can be found on github at <https://github.com/cfuhrman-utcr2018/utcr2018-ROS_SRC>

This document will attempt to outline the procedures and code used to accomplish near autonomy with Bender. It will step through the various components and discuss the code used with each. All code has also been commented and extraneous information and files have been removed.

It must be noted that much code is found online through various open source locations. If the code was found and cloned into a local .git repository then it will appear as an empty folder in the github account. Note that much of the essential code was found this way.

Be aware that this document is an attempt at a segmentation of information but there is such an overlap that it will read more of like a narrative – i.e. to understand the information outlined in the GPS section one would have to read the LiDAR section first. Therefore for a full understanding it is recommended that all members of the team read this document in its entirety.

All design reports should be read before reading this document. They can be found easily through Dr. John Peeples, Mr. Bart Knapp, or the 2018 IGVC GDrive folder. At the time of this document Bill Quade is the owner of the Google Drive folder.

The languages used with ROS include C/C++, Python, Julia, XML, and other more obscure and specified languages for simulation and such.

Obstacle Avoidance - LiDAR:

The LiDAR is easy to use with ROS. The [urg node](http://wiki.ros.org/urg_node) reads in the serial data from the Hokuyo LiDAR and publishes a topic called /scan of type LaserScan. This is all that needs to be done for LiDAR data (outside of complex and arguably impossible filtering for solar pollution). This LaserScan topic can be fed directly into the Navigation Stack if that is chosen.

To viualize the LiDAR data in Rviz, a highly useful tool for viewing data within ROS, the frame of references must be compatible. A good introduction to what a reference frame is can be found [here](http://www.pirobot.org/blog/0011/). It is the same concept as a reference frame in basic relativity. Basically, a reference frame is responsible for that feeling when you are in a car and the car in front of you pulls away making it feel as if you are moving backwards. This happens within the robot because the data captured by the LiDAR has a time and a location associated with it. The data must be transformed, with the [ROS tf](http://wiki.ros.org/tf) capabilities, into what it would ‘look like’ to the main computer. One should read and attempt to fully understand the tf library. All that I used were static transforms between the different coordinate frames (GPS, LiDAR, Camera, base\_link, ect.). The nitty gritty of this discussion is that when using just the urg node and Rviz the reference frame in Rviz must be set to whatever the urg node’s published reference frame is. Alternatively you can also launch a statictransformpublisher node so that the data can be visualized in the base\_link frame.

Image Processing for Lane Detection - Camera:

The camera code is extremely complicated and would have taken the IGVC team an eternity to code. Do not attempt to understand – the [code](https://github.com/intel-ros/realsense) was written by Intel software engineers and works great. Essentially one must understand how to [install the correct drivers](https://github.com/IntelRealSense/librealsense/blob/master/doc/installation.md) and launch the prewritten nodes. Then one must understand what topics are being published and how to interact with them. This is left as an exercise to the 2019 IGVC team because what was done in 2018 was rudimentary and pitiful.

*Note that the drivers experience issues with certain GPU’s*. One of such is the one housed in the ASUS laptop. To get this to work I had to install from source an earlier SDK version. Instructions for this are housed under the Intel Realsense SDK install from source website linked above. This is stated to make the reader aware of the pitfalls experienced by previous teams. If the ASUS laptop continues as the main computational center for the vehicle then the drivers are already installed and one would not have to worry about this.

The ideal way to process images is [OpenCV](https://opencv.org/). This works beautifully with ROS and C++. If a member of the team is interested in robotic vision and image processing as a research topic for graduate school OpenCV should be used!!! The 2018 team utilized MATLAB for its easy image processing algorithms. Please be aware that I do NOT recommend using MATALB for anything that is mission critical. The reasoning for this is the [computational speed](https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjM4o74-qvdAhWP_1QKHes5AjMQjRx6BAgBEAU&url=https%3A%2F%2Fraid6.com.au%2F~onlyjob%2Fposts%2Farena%2F&psig=AOvVaw1VColcNskIAyKXBr5t6xgM&ust=1536515067002100) is horrid compared to C, [Julia](https://julialang.org/), or Python. I battled and never conquered image processing because of a lack of time. The trick with the image processing is to get a 3D representation of your environment. This is why the realsense camera is so attractive because it publishes data in 3D, if requested. This comes published as a PointCloud2 message but this is where the MATLAB fails. MATLAB is beautiful for subscribing to the PointCloud2 message and extracting the data into a MATLAB ptcloud data type. Then MATLAB can extract the RGB and XYZ values (RGB for processing and XYZ for reference). Then the processing algorithm is run on the RBG and the XYZ image can be multiplied by the RGB image (once converted to a logical array of data where the pixels containing the line are segmented away). This is then able to be combined in to the MATLAB native ptcloud data type but it is nearly impossible to transmit the data from MATLAB into ROS PointCloud2 format. Generating a PointCloud2 is theoretically possible in MATLAB but takes a greater understanding into the underlying roscpp protocols that the 2018 team did not have time or resources to explore.

For this reason, the MATLAB exclusively subscribed the RGB image within ROS and outputted a segmented image along with the corresponding disparity image. Then the ROS package [depth\_image\_proc](http://wiki.ros.org/depth_image_proc) was used to create a PointCloud2 from the RGB image and the camera info (published by the Realsense node containing information about the focal length and other intrinsic camera information that allows the image to be constructed in a 3D manner (see [image\_to\_xyz.launch](https://github.com/cfuhrman-utcr2018/utcr2018-ROS_SRC/blob/master/bender_2dnav/Launch/image_to_xyz.launch)).

I recommend a full overhaul of the image processing algorithm and structure. This is the hardest part of the competition, and justly so because PhD dissertations are still being written trying to solve the problem of lane detection with autonomous vehicles. There is no easy answer. I suggest trying to reach out to experts in the image processing field seeing if they can give any direction or guidance. There are also many, many good tutorials available online. Someone needs to take the whole year (and I mean the whole year) to learn OpenCV, get data from the Realsense camera, process it, and send it to ROS in a *meaningful* way. This meaningful way means that the team must decide if they are using the Navigation Stack or writing the navigation algorithm themselves.

GPS:

Do not waste your time with the GPS. Get a GPS that can output an NMEA string via USB (the Garmin brand one supplied by Dr. Peeples will work just fine) and plug into the laptop. Fire up the nmea\_navsat\_driver and you get a topic called /fix with the GPS coordinates. This is all you need from the GPS. During the 2018 campaign too much time was wasted on a GPS ‘algorithm’ that used Arduino code to calculate where the robot was and routed a path to a specified GPS point with near nonexistent update rate and horrid integration capability with ROS. What is outlined in the design reports and the PowerPoint presentations should be disregarded completely. In the course of 20 minutes I accomplished getting the /fix topic from the nmea\_navsat\_driver serial node.

There are good and stable prewritten ROS nodes that will take the GPS fix and a given goal and output a different goal in the robot reference frame. This can then be passed to the Navigation Stack as a user defined path goal. Personally I read much literature about this and never experiemtned with it due to a lack of time. If I had my way I would have used the GPS with the Navigation Stack’s path planning capabilities.

Physical Movement – motors:

The motor commands are straightforward. The part the is not straightforward is the PID controller. Note that it is impossible to achieve good performance without PID feedback. The differential\_drive ROS package (this is one of those that was cloned from a git repository and will show as an empty folder) is useful for PID. It is written in Python and states that it is a general purpose PID controller. What I discovered was that this was very difficult to work with. The output of this is undefined – the wiki only states that it is the “motor\_cmd” is “the output of the PID controller, the power going to the motor. Arbitrary units, in the range of out\_min to out\_max.”. I finally decided, through much trial and error, that I would use this as an offset to the command. The motor\_cmd, to me, acted as an additive or subtractive force to the nominal motor command (in m/s). Therefore, if the given velocity was not achieved the commanded velocity would increase (although arbitrarily it would not matter because eventually the commanded velocity would equal the actual velocity and the PID would have done its job). This package works well but relies on the speed calculations from the wheel encoders. These are an area of contention.

As for the wheel encoders I would recommend having multiple ways of measuring the wheel speed so that the vehicle is fault tolerant. Order the encoders early and often because they have to be custom made and do not have a quick turnaround time. One of the problems that we faced was having too much precision in our encoders. This made tuning the PID extremenly hard (it never was actually accomplished correctly just guessed at and parameters were thrown into the code) because when putting the vehicle up on block so the wheels spin freely the slight mechanical play in the gearing causes some play in the wheels. Thus, when the wheel is slowing down its inertia causes it to keep spinning even though the motor command has stopped. This in turn causes the velocity calculations to be erroneous in that the computer thinks the vehicle is still moving much faster than it is, and sends a negative motor\_cmd so that the wheel will slow to the given velocity. This causes the motor to slow down or even move in the other directin and the same process happens but in reverse. The effect of this is that the wheel will achieve a sort of critically damped oscillation and never be able to be tuned. The way that I think is best around this is getting a treadmill setup so that there is some frictional resistance against the wheel to eliminate some of this mechanical play. We were very confused as to why we were experiencing oscillations with such small PID coefficients and discovered that is was nearly impossible to tune on the blocks with the given mechanical play. I would suggest getting encoders with slightly less resolution than what we used (we used the highest resolution US Digital E4T Miniature Optical Encoder with differential ended measurements). This is the top of the line optical encoder and can measure things down to fractions of turns. The vehicle does not need that kind of precision and it really only makes the computations take that much longer. My suggestion is to use the differential\_drive package as I have outlined it in the source code with many sets of optical encoders feading to different microcontrollers.

Arduino microcontrollers are a must for the IGVC. ROS plays well with them and almost nothing else. This makes life easier than trying to use something like a Teensy. The 2018 team used a breakout board for calculating the encoder measurements that did the math and was polled by the Arduino through SPI communication. There are Arduino libraries that use the interrupt pints to poll the encoders and get measurements. I would suggest thinking outside of the box with this.

Final Schematic:

The following is a hand drawn schematic of the software used in the IGVC. In no way is this a complete diagram and one should note that this is only a high-level overview. ../Downloads/IGVC%20Software%20Layout.pdf