Capstone Design Document

GFR Group

CS461 Senior Software Engineering Project

Fall 2018

December 3, 2018

Autonomous vehicles require robust internal representations of the world in order to make correct driving decisions. Information from multiple sensor modalities including camera, lidar, and odometry must be utilized in order to create a map of the world and localize the vehicle within that map. This document covers the design methodology used to develop a driverless formula-style racing car. The racing car will be entered into driverless racing competitions starting in 2019.

CONTENTS

1 INTRODUCTION

1.1 Terminology

- **Lidar** A visual sensor that fundamentally works like a radar but uses lasers instead of radio waves
- **GPS** Global Positioning System. Device used to a estimate longitudinal and latitudinal coordinates of the vehicle.
- **DGPS** Differential GPS. Device used to estimate relative change of position over time.
- **IMU** Inertial Measurement Unit. Device used to estimate the velocity of the vehicle.
- **ROS** Robot Operating System. Software used to manager subsystems of an autonomous agent.
- **Faster RCNN** Faster Region-based Convolutional Neural Network. Machine learning model used to localize objects in an image.
- **YOLO** You Only Look Once. Machine learning model used to localize objects in an image.
- **SSD** Single Shot Multibox Detector. Machine learning model used to localize objects in an image.
- **SLAM** Simultaneously Localization And Mapping. Algorithm for mapping the surroundings of an agent and localizing that agent within the map.
- **GFR** Global Formula Racing. International formula racing cooperation between OSU and DHBW.
- **DHBW** Duale Hochschule Baden-Wurttemberg. University located in Stuttgart, Germany
- **OSU** Oregon State University. University located in Corvallis, Oregon
- **EKF** Extended Kalman Filter. Algorithm for approximating the behavior of a function.
- **RANSAC** Random Sample Consensus. Algorithm for approximating the behavior of a function.

1.2 Purpose

The driverless vehicle must utilize sensor input from camera, lidar, GPS, and IMU in order to construct a virtual representation of the world around it as well as localize itself within that world. The virtual representation of the world is then used to compute the optimal path and trajectories needed maneuver around a track. The driverless vehicle is expected to compete in the Formula SAE (FSAE) competition in August 2019.

1.3 Scope

Our team is tasked with constructing the vision, odometry, mapping, and localization systems for a driverless formula style racing car for the Global Formula Racing (GFR) team. The purpose of our system is to create a robust virtual representation of the vehicle's environment. Algorithms will be developed for detecting the location and color of the cones using lidar and camera. The cone locations and colors will be used to construct a dynamic map of the world, updating the map as the vehicle moves and new cones are seen. As the vehicle moves, our system will need to use positional data from the GPS and IMUs in order to update the position of the vehicle within the map.

1.4 Stakeholders

The stakeholder of this system is the Global Formula Racing team. GFR will use the system described in this document to participate in international driverless racing competitions.

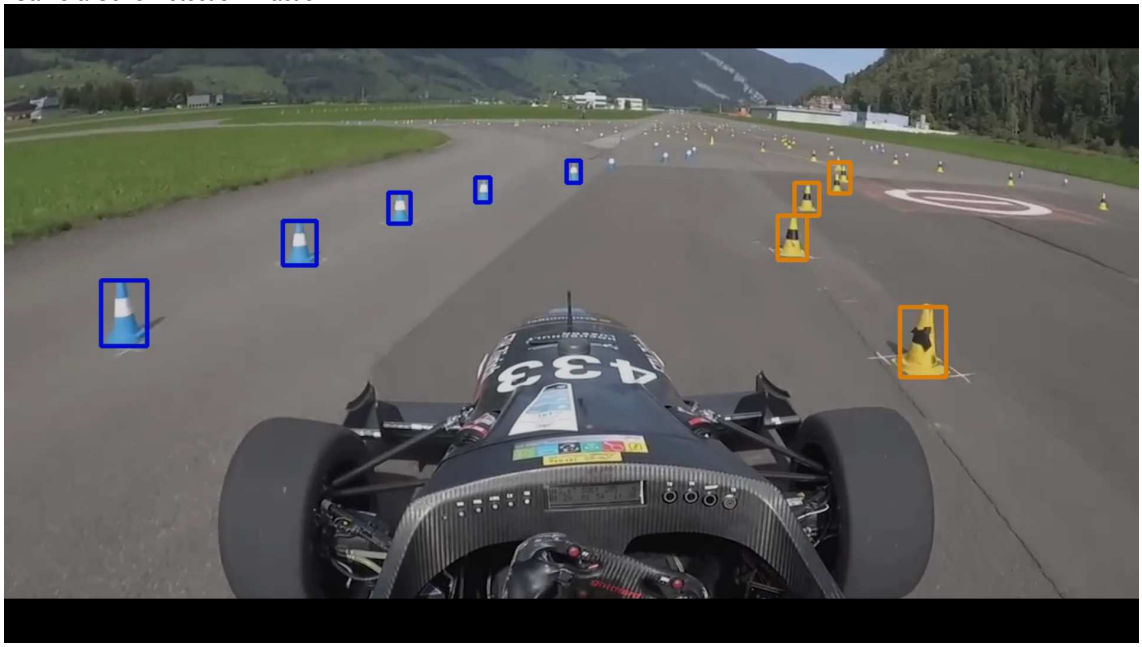
2 DESIGN COMPONENTS

2.1 Camera

2.1.1 Design View

The camera vision system should be able to accurately detect all cones within a sufficient distance from the camera.

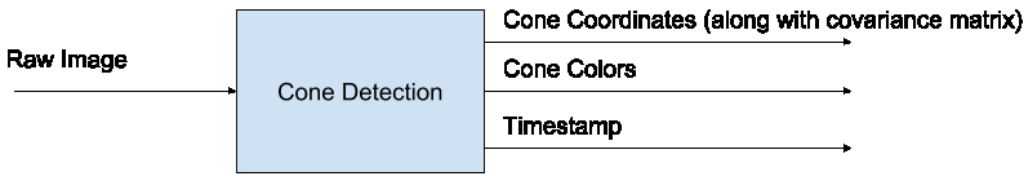
Fig. 1. Camera Cone Detection in action



2.1.2 Design Viewpoints

The vision system will be contained within a ROS node which takes in individual frames and outputs the color of each cone, the mean x and y locations of all cones, as well as the standard deviation of the cone's location in the x and y direction. The x and y locations returned will represent the position of each cone relative to the vehicle. Covariance of x and y coordinates will also be computed. Once cones have been detected, they will be published to the camera_stream topic for use by other parts of the system.

Fig. 2. Camera Functional Overview



2.1.3 Design Elements

The cone detection model will consist of a pair of Convolutional Neural Networks, a large, accurate network, and a small, fast network. The networks will operate in tandem. The larger network will operate at a slower prediction rate, generating more accurate predictions at longer distances, while the smaller network operate at a faster prediction rate, generating less accurate predictions at longer distances but which does well detecting cones that are closer to the vehicle. Details of the algorithm are shown in algorithm 1.

The neural networks used for cone detection will be Faster RCNN with a ResNet-style backbone and Feature Pyramid Network. ResNet-style backbone will consist of several stages of the original ResNet. The smaller network will have less convolutional filters per layer as well as using less stages instead of than what is typically recommended.

In order to utilize a Faster RCNN model, the model must be trained on a dataset of blue, yellow, and orange traffic cones. The dataset should contain a minimum of 100 ground truth images containing at least 250 blue and yellow cones and 60 orange cones.

2.1.4 Design Rationale

The dual network structure for cone detection allows the vehicle to balance short-term and long-term planning of the vehicle's route. While the large network can determine twists and turns in the track from farther away, the small gives the system quick updates about what is directly in front of the vehicle. The choice to use Faster RCNN was made due to the accuracy of Faster RCNN compared to other object detection models. While models such as YOLO and SSD perform faster, the bounding box predictions of these models are less accurate. These models also perform poorly on objects at longer distances, which is crucial to accurately planning the vehicle's route from farther away.

2.1.5 Design Tools

The latest versions of Pytorch and Caffe2 will be used as our machine learning framework. Facebook Research Group's Detectron, an open source implementation of Faster RCNN, will be used as our object detection model. ROS nodes will be used for data transfer between the cone detection node and other parts of the driverless system.

Algorithm 1 Cone Detector

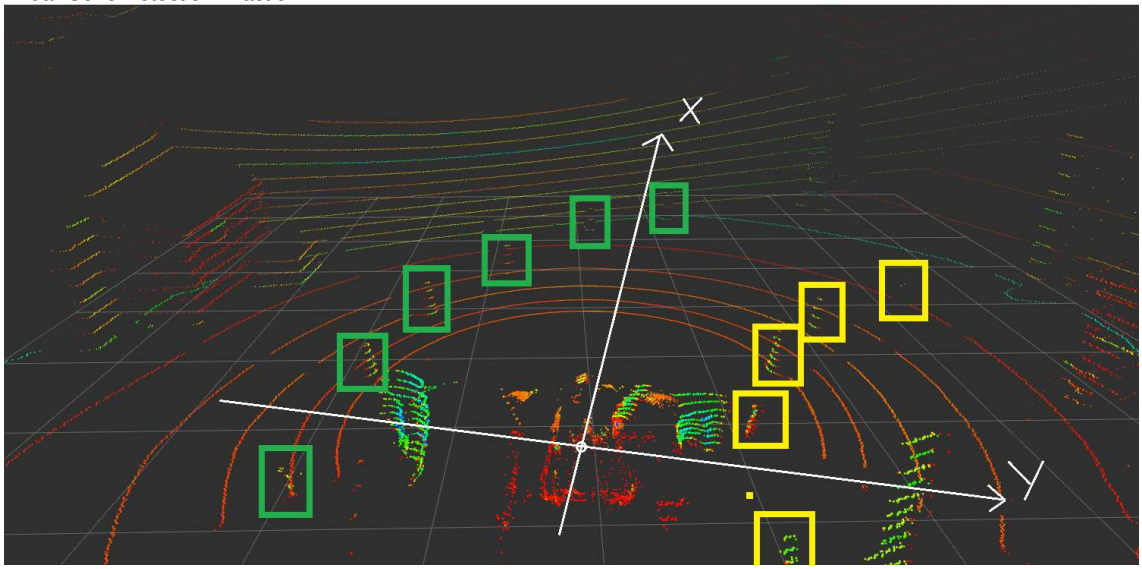
procedure DETECTIONLOOP
    Input
        $fast\_model$: faster detection model that operates at higher frequency
        $slow\_model$: slower detection model that operates at higher frequency
        $fast\_refresh\_rate$: time in seconds between fast model detections
        $slow\_refresh\_rate$: time in seconds between slower model detections
        $length\_of\_history$: max number of frame predictions to store
    $fast\_model\_last\_run \leftarrow 0$
    $fast\_model\_last\_run \leftarrow 0$
    $prediction\_history \leftarrow$ list to store the last $length\_of\_history$ predictions
    while true **do**
        $image \leftarrow$ waitForImage()
        $time\_since\_fast\_model\_ran \leftarrow fast\_model\_last\_run -$ time.now()
        $time\_since\_slow\_model\_ran \leftarrow slow\_model\_last\_run -$ time.now()
        if $time\_since\_fast\_model\_ran > fast\_refresh\_rate$ **and** $fast\_model$.isAvailable() **then**
            $preprocessed\_image \leftarrow preprocessImage(image)$
            $cone\_colors, bounding\_boxes \leftarrow fast\_model$.getPredictionsInSeparateThread($preprocessed\_image$)
            $fast\_model\_last\_run \leftarrow$ time.now()
        else if $time\_since\_slow\_model\_ran > slow\_refresh\_rate$ **and** $slow\_model$.isAvailable() **then**
            $cone\_colors, bounding\_boxes \leftarrow slow\_model$.getPredictionsInSeparateThread($preprocessed\_image$)
            $slow\_model\_last\_run \leftarrow$ time.now()
        else
            continue
        $cone\_positions \leftarrow$ computeConePosition($bounding\_boxes$)
        publish($cone\_colors, cone\_positions$)
        $prediction\_history$.add($cone\_colors, cone\_positions$)
        if $length\_of\_history > prediction\_history$.getLength() **then**
            $prediction\_history$.removeOldestPrediction()

2.2 Lidar

2.2.1 Design View

Mounted on the chassis of the car, the lidar scans the surrounding physical environment of the car. From the scanned data, the ROS node detects and calculates the location of cones relative to the car as well as estimates the color of cones based on intensity values of the laser scans corresponding to those cones.

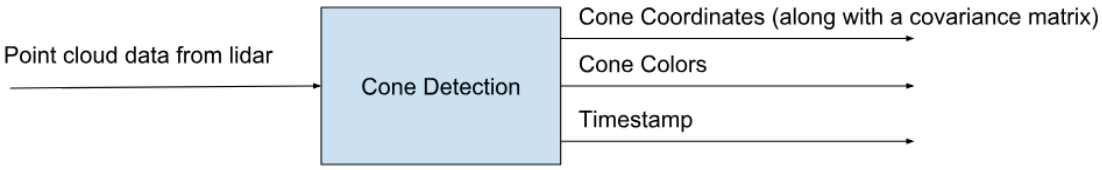
Fig. 3. Lidar Cone Detection in action



2.2.2 Design Viewpoints

In overview, our lidar ROS node receives 3D point cloud data from the lidar laser sensor and gives 3 outputs: first, the 2D location of the cones relative to the car; second, a covariance matrix describing how certain our 2D location is; and third, a timestamp on when the lidar point cloud was originally captured.

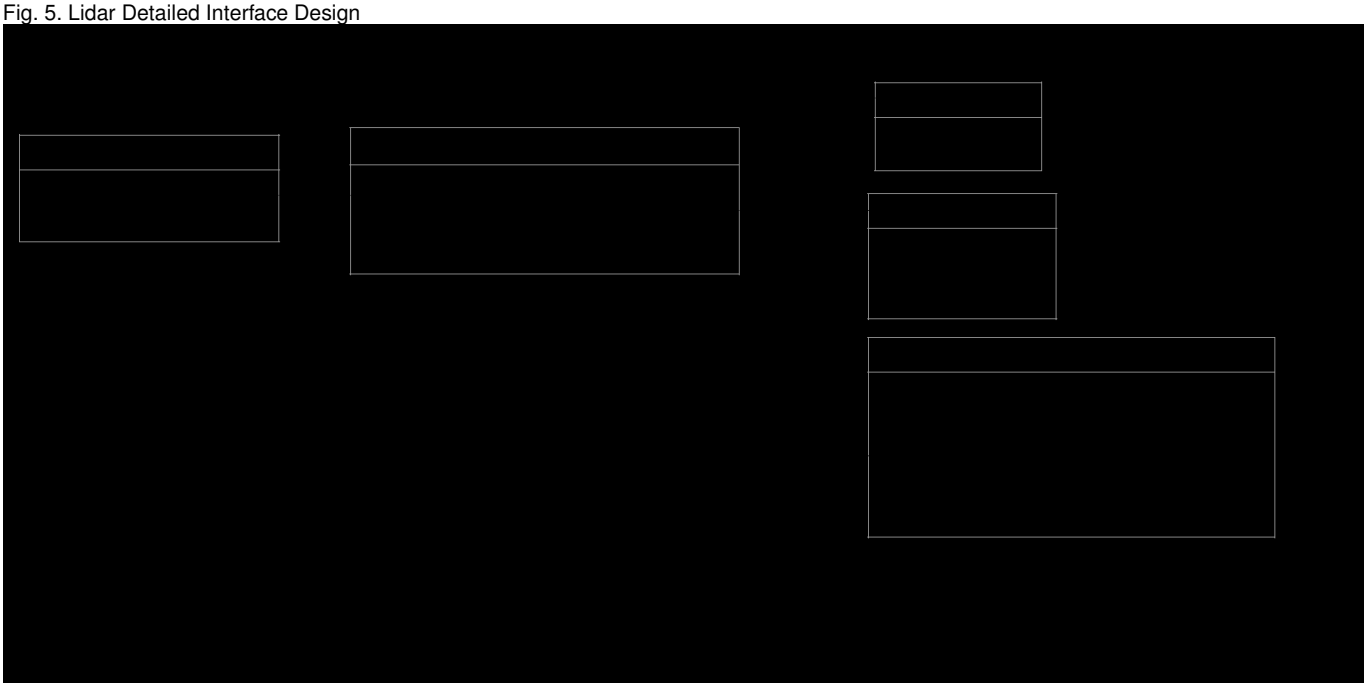
Fig. 4. Lidar Functional Overview



The lidar sensor continuously scans the physical environment around the car as the car is moving at a speed of at most 60 miles per hour until being turned off. Point cloud data from the lidar sensor is then published to lidar processing node in ROS. In real time, the lidar processing node differentiates between points that are cones and points that are not cones. The lidar processing node then computes the 2D coordinate of the cones relative to the physical location of the lidar sensor mounted on the car. And finally, the lidar processing node

predicts and differentiates between the different colors of detected cones based on their position relative to the lidar sensor and the intensity values of the points in the point cloud corresponding to those cones.

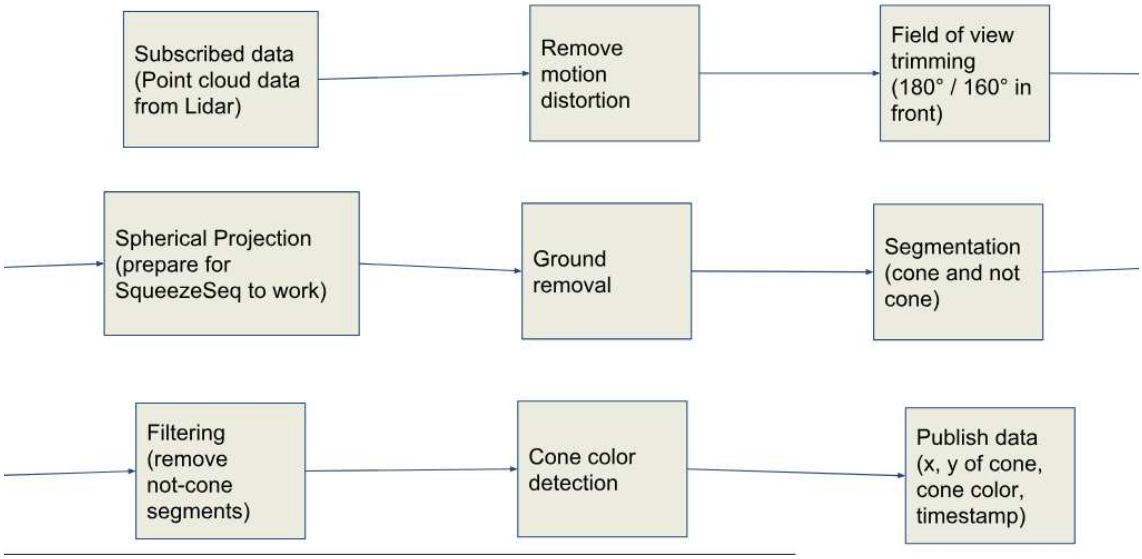
Fig. 5. Lidar Detailed Interface Design



2.2.3 Design Elements

We use 3D hough transform to find the ground plane. After removing the ground plane, RANSAC is used to fit a cone model to cones located in the point cloud.

Fig. 6. Lidar Processing Pipeline Diagram



2.2.4 Design Tools

We use ROS for our robotics platform and use C++ to implement our algorithms except our machine learning algorithms which we implement in Python. We use rviz to visualize our lidar scanning output overlayed with the cone detection output which are boxes surrounding the 3D model of cones.

2.2.5 Design Rationale

We remove the ground plane because it does not give us any valuable information about our surrounding and also to the computational cost needed to detect cones. Because the shape of the cones is static, using a cone model to fit cones in the point cloud using RANSAC will be highly efficient.

We decided not to use the state-of-the-art, neural-network-based algorithm, SqueezeSeg, for real-time 3D point cloud object clustering because SqueezeSeg is designed and implemented for general object detection on the real life street traffic problem. Our problem is much less complicated. Thus, SqueezeSeg is probably too complicated for what we need in our problem, which is just fixed-size, stationary, small number of cones ahead of the car.

For our programming language decision, there are two major options: Python and C++. Because the scanning data from the lidar (hardware) must be processed by a lidar processing algorithm in real time, a major concern of our system is efficiency. ROS can work with both Python and C++, however most existing implementations of relevant algorithms in the ROS ecosystem are written in C++. Python will help us get the prototype up and running quickly, and also most of machine learning model implementations and deployment pipelines are written in Python. Therefore, we believe that the best course of action will be to do initial development in Python and rewriting all performance critical elements in C++ after our solution and been evaluated.

2.3 Odometry

2.3.1 Design View

The odometry system will use information from three IMUs and one GPS present on the car to determine where the car is and how fast it is moving.

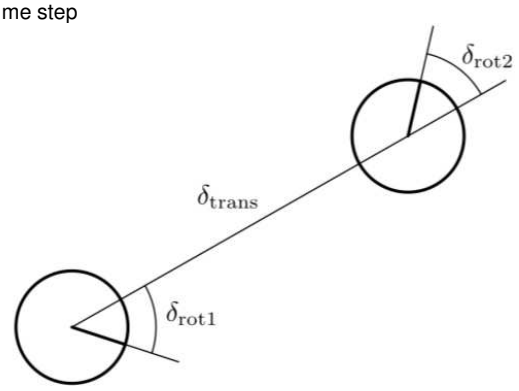
2.3.2 Design Viewpoints

The filtered sensor data from the car will be used to determine how the car has moved over the course of the previous time step. The motion of the car will then be put through a particle filter to account for kinematics, commands, and uncertainty in sensor measurements. The change in location data will be sent to SLAM, and velocity will be sent to the dynamic controllers system.

2.3.3 Design Elements

Each movement through a time step will be modeled by a rotation from the start, a translation along the new heading, and a second rotation, as shown in figure 2.3.3. These three parameters are all that is needed to describe an arbitrary translation and rotation through space. A particle filter will be used to fuse the IMU and GPS sensor data into a single model, where it will be sent to the SLAM system for use in their localization algorithm. A high level algorithm of a particle filter is shown in algorithm 2.

Fig. 7. Model of a movement across a time step



2.3.4 Design Rationale

Because SLAM will be implementing localization, it is redundant to determine absolute position in odometry as well. Therefore the minimum amount of information needed by the SLAM is how one location changes to become the next measured location, resulting in the decided upon model. The particle filter implementation was chosen because of the nonlinear nature of the sensors and model at work, and because it tends to be the most robust statistical filter implementation available.

2.3.5 Design Tools

ROS includes an odometry message which contains most of what is needed in our description of how the robot has moved. The final iteration of the message type passed to SLAM will include additional fields. The

implementation will be done in Python, because this is one of the less computationally intensive components of the driverless system. NumPy will be needed for developing the system propagation and control model.

Algorithm 2 Particle Filter

procedure FILTERLOOP
    Input
        $weight\_calculation$: determines how likely a possible state is given an external measurement
        $system\_noise$: gives a probability distribution of possible states given the last state and a control
    $particles \leftarrow$ randomly assigned states
    while true **do**
        $new\_particles \leftarrow []$
        $sum\_weights \leftarrow 0$
        for each particle in particles **do**
            $random\_particle \leftarrow$ random particle from particles by weight
            $random\_particle \leftarrow system\_noise(random\_particle, control)$
            $new\_weight \leftarrow weight\_calculation(random\_particle)$
            $sum\_weights \leftarrow sum\_weights + new\_weight$
            $new\_particles.append(random\_particle)$
        $particles \leftarrow new\_particles$

2.4 SLAM

2.4.1 Design View

The overall SLAM system will be able to map the surroundings of the car using cones as landmarks. It will also localize the car within the map.

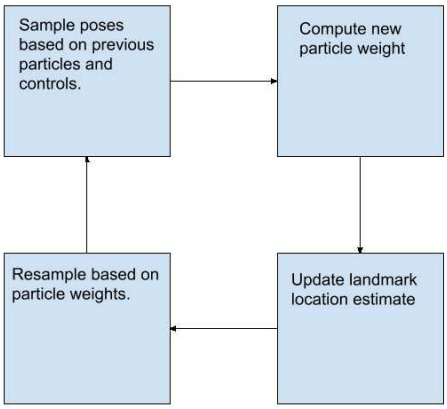
2.4.2 Design Viewpoints

The first step is open loop SLAM, followed by loop closure detection, and finally closed loop localization. Open loop SLAM and closed loop localization will run as different threads. Finally the OSU SLAM team will improve the fastSLAM system built by the DHBW team. The input for SLAM comes from odometry, camera, and Lidar. These inputs consist of data representing the car's movement, as well as perceived landmarks. The output, which describes the car and cone location, is required in order for trajectory planning to function.

2.4.3 Design Elements

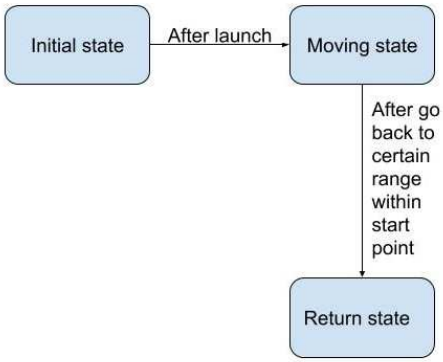
2.4.3.1 Open Loop SLAM: Open loop SLAM is used on the first lap around a track. The first lap is the slowest in order to collect the most accurate data to represent the track. Our initial open loop SLAM algorithm is FastSLAM1.0. Our open loop program will focus on creating the most accurate map and localization possible. The FastSLAM algorithm uses a particle filter to approximate the pose of the car as well as the location of the cones. FastSlam works by sampling N different poses based on the previous pose, odometry controls, and uncertainties. These samples, or particles, are assigned a weight that represents how reliable their estimates are. The landmark locations are then updated represented by the mean and covariance. Next the particles are resampled. Particles with low weights will be replaced by particles with high weights. Then the new set of particles are propagated forward based on odometry controls. This process repeats until the vehicle approaches its original position, wherein loop closure occurs. Every time the algorithm repeats, new cones are found, and old cones positions are improved. This process flow is displayed in figure 8.

Fig. 8. FastSLAM Flow Chart



2.4.3.2 Loop Closure and Localization: Loop closure will occurred as the vehicle finishes the first lap on the track. The purpose of loop closure is to minimize the accumulated error. Because of the high accuracy of DGPS, the vehicle will be able to accurate estimate when it has come back to its starting point. Additionally, because the starting point will be marked with different colored and sized cones, we will receive a signal once this particular cone is detected by the camera or lidar. In order to avoid a false positive case, (assuming loop closure immediately after the vehicle takes off) we will implement a state machine, the state machine will contain 3 states, the initial state would be set up when the car is ready to launch, after the car starts moving, it will transfer to second state that indicates the car is moving. Once the car is located within a certain range of the start point, it will transfer to its terminated state and a loop closure would be occurred only at terminated state. After the loop closure is detected, the map of the highest weighted particle will be selected and will no longer update. We will only take input from odometry and observations from camera and lidar, then we will implement an EKF on our observations in order to update the location of the vehicle.

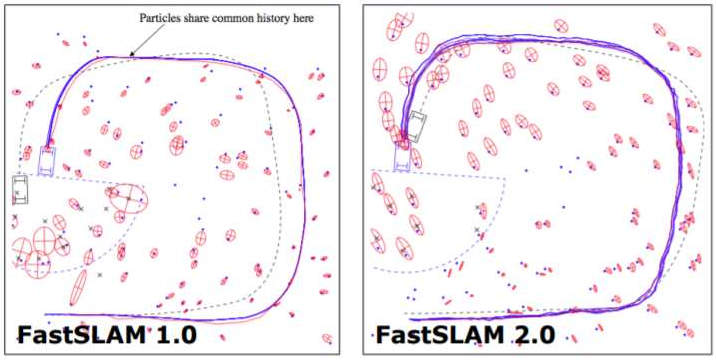
Fig. 9. State Machine



2.4.3.3 FastSlam Improvement: For the first implementation of our SLAM algorithm we will be using FastSLAM1.0. The purpose being to produce a simple system quickly in order to begin testing and create a working model. We will then upgrade to FastSLAM2.0 after the initial implementation of FastSLAM1.0 has been evaluated. This upgrade is difficult since the implementation is much more complex, however, the end result will allow us to create more accurate and robust map and localization with fewer particles. Speed and accuracy are the primary criteria we are basing the SLAM design decisions upon and FastSLAM2.0 can produce higher accuracy with less data while decreasing the time spent updating sensor inputs.

Loop closure's accuracy also improves; since FastSLAM2.0 data has a smaller error distribution, the ability to recognize a loop closure (i.e. data points it has already processed and mapped) is much higher. This difference becomes more evident as the loop increases in size.

Fig. 10. FastSLAM1.0 v FastSLAM2.0



2.4.4 Design Rationale

2.4.4.1 Open Loop SLAM: FastSLAM was chosen for open loop SLAM because of its ability to handle non-linear distributions. The extended Kalman filter was considered because of how commonly it's used, but was not chosen because its accuracy relies on linear distributions and Gaussian noise. FastSLAM, while not as commonly used as the Kalman filter, still provides plenty of sources to draw from to help implement our own SLAM system.

2.4.4.2 Loop Closure and Localization: An extended Kalman filter was chosen for the closed loop localization. An EKF was chosen because of its computational simplicity. In closed loop, the track is already recorded, so mapping cones is not the purpose of this system. Instead, EKF works to localize the car within the map. A particle filter may have been more robust for non-liner models, but keeping track of a high number of particles becomes computationally expensive.

2.4.5 Design Tools

The SLAM system will be built in ROS along with the rest of the project components. ROS was chosen because of its ability to send data between nodes easily. The code for SLAM will be written in C++. ROS supports C++ and Python, but C++ was chosen because it has less overhead than Python. The Eigen C++ library will be used in the SLAM system to simplify matrix operations.

REFERENCES