Object detection, Kalman filtering and Collision detection in a collision warning system with LIDAR and Camera

Escuela Técnica Superior de Ingeniería (ICAI)

Author: Roberto Rioja García
Director: Dr. Joydeep Ghosh

Madrid
Junio 2018

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. ROBERTO RIOJA GARCÍA.

DECLARA ser el titular de los derechos de propiedad intelectual de la obra: DETECCIÓN DE OBJETOS, FILTRADO DE KALMAN Y DETECCION DE COLISIONES, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente)*.

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e

intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a ..20..... de ....Julio................. de ..2018..

ACEPTA

Fdo...

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Detección de objetos, filtrado de Kalman y detección de colisiones en un sistema de detección de colisiones con LIDAR y cámara

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2017/18 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Roberto Rioja García Fecha: 15/ 05/ 2018

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Joydeep Ghosh Fecha: 21/ 05/ 2018

Digitally signed by Joydeep Ghosh
DN: cn=Joydeep Ghosh, o=ECE, ou, email=jghosh@utexas.edu, c=US
Date: 2018.05.21 12:55:41 -05'00'

CONTENTS

TABLES iv

FIGURES vi

EXECUTIVE SUMMARY 5

1.0 INTRODUCTION 8

2.0 DESIGN PROBLEM STATEMENT 9

2.1 Design Objective

2.2 Specifications

3.0 DESIGN PROBLEM SOLUTION 11

3.1 Overview 11

3.2 Sensors 12

3.3 Object Detection 12

3.4 Sensor Data Syncing 15

3.5 LIDAR-Camera Transformation 15

3.6 Object Locator 16

3.7 Kalman Filter 17

3.8 Collision Detection and Alert 19

4.0 DESIGN IMPLEMENTATION 20

4.1 Failed Design Components 21

4.1.1 Apollo 21

4.1.2 ROS 21

4.1.3 KITTI 21

4.2 Design Choices 23

4.2.1 YOLO 24

4.2.2 LIDAR-Camera Calibration 24

4.2.3 Object Locator 27

4.2.4 Kalman Filter 27

4.2.5 Collision Detection 28

4.2.6 Alert 29
4.3 Cost/Benefit Analysis 29
5.0 TEST AND EVALUATION 30
5.1 Sensors 31
5.1.1 LIDAR 31
5.1.2 GPS 33
5.2 Object Detection 34
5.3 Data Syncing 38
5.4 Object Locator 39
5.5 Kalman Filter 42
5.6 Collision Detection 44
5.7 Overall Time 45
5.8 System Testing 46
6.0 TIME AND COST CONSIDERATIONS 48
7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN 48
8.0 RECOMMENDATIONS 49
9.0 CONCLUSIONS 49
REFERENCES 52
APPENDIX A – Object Locator Distance Test A-1
APPENDIX B – Collision Detection Code A-2
APPENDIX C – Kalman Filter Code A-3

TABLES

1 *Ideal Testing Conditions* 10

2 *Performance Specifications* *10*

3 *YOLO Data Packet Structure* *14*

4 *Final LCT Calibration Measurements* *26*

5 *Time to Receive LIDAR Data* *31*

6 *Results of GPS Speed Test* *34*

7 *Accuracy and Speed Test Results* *37*

8 *Results of Data Sync Tests* *38*

9 *Time Spent per Module* *46*

10 *Collision Detection Test Summary* *47*

FIGURES

1 *System Block Diagram* 12

2 *Overview of YOLO System* 13

3 Illustration of 3D-2D Projection 16

4 Location Selection of Truck using Our LCT 17

5 Collision Detection 20

6 LIDAR-Camera KITTI Calibration Checkerboard 22

7 Ponale Mapping (Left) and LIDAR Mapping (Right) 23

8 Point Cloud Left and Photo of Truck LCT Setup 25

9 Image of Point Cloud Overlaid on Truck Using LCT 26

10 LCT Used on Two Trucks, a Pillar, and Person 26

11 LIDAR Point Cloud 3D Plots 33

12 YOLO Output on a Daytime Image 35

13 YOLO Output on a Nighttime Image 36

14 Object Locator Distance Test Setup 40

15 Average Percent Accuracy vs. Position 41

16 Total Accuracy vs. Distance 42

17 Comparing Inputs and Results for Kalman Filtering with 1 Object 43

18 Comparing Inputs and Results for Kalman Filtering with 2 Objects 44

19 Visualizing Objects and Collisions 45

EXECUTIVE SUMMARY

This project was done by Roberto Rioja García at the University of Texas at Austin with the support of Dr. Ghosh, the CARSTOP research team and the Texas Department of Transportation. The project was focused on developing a vehicular collision detection and warning system. This system should constitute an affordable way to help cut down on the number of car crashes. The project was focused on creating a system that works in real time, can accurately predict collisions, and alert the driver in a timely fashion. It is only designed to detect possible collisions and alert the driver; this system does not take action to prevent these collisions from occurring. The goals were to have an object location accuracy higher than the 46% of the previous year´s team, to give the driver 3 seconds of warning for a collision, and to have a false positive rate below 30%, and process each frame in less than 250 milliseconds.

This system uses several sensors to find the location of objects relative to the car, track the direction that they are travelling, compare that path to the movement of our car, detect if these two paths ever intersect and provide a sound alert in the case of a potential collision. The final system design utilizes three sensors: Light detection and ranging (LIDAR), camera and global positioning system (GPS). The LIDAR is used to provide measurements about the distance of objects to the car, the camera provides visual data about objects in front of the car and the GPS provides the speed that our car is travelling at.

The initial project idea was to work using an open source autonomous vehicle platform called Apollo, created by the technology company Baidu. This previous system was rejected due to compatibility problems with the current sensors owned by the CARSTOP research team.

The final system is divided in 8 distinct and independent subsystems: sensors, object detection, LIDAR-camera synchronization, LIDAR-camera transformation, object location, Kalman filtering, collision detection and alert. The sensors LIDAR and camera collect data on potential obstacles in front of the vehicle while the GPS is used to determine the velocity of our own vehicle. The object detection module uses camera data to identify potential obstacles and assign bounding boxes to them, which is sent to the LIDAR-camera synchronization module. This synchronization module collects LIDAR point cloud data and bounding box information from camera data at the same instant in time, which is used in the LIDAR-camera transformation module. This module projects the 3D LIDAR points onto the 2D image captured by the camera, making it possible to assign a distance collected from the LIDAR point cloud to the objects within the bounding boxes. The bounding box and LIDAR data is used by the object locator module to determine how far away each object is. Next, the Kalman filtering section creates a model used to predict the path of objects based on their current position and velocity. This model is used in the collision detection section to predict whether our vehicle will collide with any of the objects 1 second in the future. If a collision is detected, the alert module will notify the driver.

My individual function in the project was to design and implement the object detection, Kalman filtering and collision detection subsystems which includes the sound alarm. The object detection submodule uses an open source code called "You Only Look Once" (YOLO), this code receives images from the webcam and uses neural networks to create a bounding box for each detected

object and classify it into one of 9000 possible object categories. The Kalman filter predicts the velocity and position when the filter receives the next input. For this approximation the Kalman filter uses the positions calculated by the object locator as input and compares these with the predicted ones. It actualizes itself changing the weights of the variables as a function of the errors that it gets while comparing the approximated positions to the real ones. This Kalman filter was done under the supposition that within the time that it takes to actualize, objects movement can be approximated as a straight line and its velocity is constant. Last, the collision detection submodule describes the path of the user's vehicle during the next second using GPS data and if this path ever intersects with the ones that we get from the outputs of the Kalman filter, that are the ones from the obstacles, it sends a signal to beep the alarm. The testing of each submodule was done by the person responsible of it, while the rest was done by all of the members together as it needed the use of multiple subsystems at the same time.

The systems were first tested separately, once all of them worked they were combined into two subcategories: object location and collision prediction. Object location had LIDAR and camera sensors, object detection, LIDAR-camera synchronization, LIDAR-camera transformation and object location. This submodule had the inputs of the two sensors and used them to return the position relative to the car of possible obstacles. Collision prediction had the GPS input data, Kalman filtering, collision detection and alert. With the owner´s vehicle velocity and object locator´s output of obstacle´s relative velocity and position this system returned a noise alarm once it detected one possible collision. The last step was to test the entire system. Most of the testing for each submodule consisted of debugging during our implementation phase. The two most important tests were the system tests that the accuracy of our system's ability to measure distance and the accuracy of our system's ability to predict collisions. The idea was to set up a test to measure the distance of objects at multiple distances with objects in the left, center, and right of our image with both cars and people. The result was that the object locator had approximately 60%, 90%, and 75% accuracy for the left, center, and right, respectively. While looking for the cause it was noticed that as objects became closer to the edge of the image that the calibration projects them further to the edge than they actually were. This issue is someone alleviated by using closest point to the center of the bounding box, however at a certain point the LIDAR projection and image diverge too far, and the system picks a point on the ground instead of the actual object. It was also noticed that the system had above 88% accuracy up to 10 meters away but at 15 meters away it dropped down to 59%. The results were similar for cars and people, but cars had a slightly higher accuracy due to their larger size. This is because further objects have less LIDAR points on them. The next test was the ability of the system to detect collisions. People walked and ran at the car and drove another car towards it. During this testing the alert time was lowered to 1 second because the system provided too many false alerts at 3 seconds. Once this change was done, the system had a 90% true positive alert rate and a 33% false negative alert rate. The final system did not meet the initial requirements for alert time or false positive rates.

The system is capable of detecting collisions with a 90% true positive rate. However, due to time constraints it was necessary to limit the system to only tracking one object at a time. It was also reduced the alert time to only one second to improve the accuracy of the system. Because the purpose of the project is to prevent possible collisions, an ethical standpoint was taken and therefore accuracy was prioritized. Further work on this project should be aimed at adding

multiple object functionality, further increasing accuracy, and increasing alert time. In order to add multiple object functionality, it is needed to create a module that can determine if objects between two different time frames are the same object. Once this system is completed the object detector and locator, Kalman filter, and collision detection module are already formatted to handle multiple objects. In order to increase the accuracy of the system it is needed to increase the accuracy of the system's calibration. This could be done by focusing on the edges of the frame. Once the calibration is more accurate the system would be able to increase the alert time to meet the initial target of 3 seconds.

In this project, a system that detects collisions with a high level of accuracy was successfully created. Also achieving the goal of increasing our location accuracy over last year's team and running time of less than 250 ms and came close to meeting the goal for false positives. However, it failed to meet the goal of providing a 3 second alert time, which is the average reaction time of a driver, because accuracy was considered a priority during the testing.

1.0 INTRODUCTION

This report discusses the problem, the goals, the design solution, the design implementation, testing, and results of the CARSTOP vehicular collision detection and warning system focusing in the creation of the submodules individually created by me. This system is primarily for research in the field of autonomous driving as a continuation on last year's project group. With discussion on difficulties faced in the process as well as various safety and ethical concerns. Finally, recommendations for potential improvements given that demanded more time and resources, and for groups interested in implementing their own collision detection system.

This design used various sensors: LIDAR, camera, and GPS. It passed the input from the camera through an object detection system to sense pedestrians and vehicles using the open-source YOLO (You Only Look Once) object detection software. Then with our LIDAR-Camera transform it gets the distances of these objects from the LIDAR. Using those distances, our system has the ability to track objects in the field of view and predict their paths using Kalman filtering in order to predict potential collision and alert the driver.

In order to test the system, the set-up experiments were needed, to ensure the hardware sensors, object detection, LIDAR-camera calibration, and collision detection worked as expected. During this process, LIDAR and GPS tests to ensure the accuracy in real time were done. There were also tests on object detection and calculated the accuracy with the goal of improving on last year's accuracy. To test calibration, the code was visualized and tested the distances of objects to the owner´s car at different angles and distances. Finally, the tests on the collision detection system at different angles to see the accuracy of our system in predicting collisions.

The cost and ethical considerations for the project by discussing the required hardware and safety goals needed to meet were analyzed. This is a research project so the considerations for this project will differ from a project for commercial use. For this project, the focus was to test the feasibility of standardization in a collision detection system. In the end, there are recommendations for future work such as upgrade hardware, focus on improving calibration, and improve the advance time before collision.

2.0 DESIGN PROBLEM STATEMENT

This whole idea of the project was to create a system capable of detecting collisions and of alerting the driver with reasonable warning time. This report is focused on the creation of an object detection system using a webcam, a Kalman filter to predict positions and velocities of different objects through the time and an alert system that will trigger a sound alarm in the case of detecting a possible collision with the data obtained from the Kalman filter. The system needed to perform in reasonable testing environments, which were defined as situations with moderate temperature, no precipitation, and reasonable visibility. It also needed to meet performance specifications such as surpassing the object detection accuracy reported by prior senior design teams and keeping processing time low enough to function in real-time.

2.1 Design Objective

The goal was to create a collision prediction system that can warn drivers of impending collisions, thus helping prevent some of the 94% of accidents that are caused by human error [1]. The aim was to use LIDAR, camera, and GPS information to predict vehicle collisions with objects, pedestrians, and other motor vehicles and notify the driver if a collision is detected. The design will have to accurately detect objects and predict collisions, perform correctly in ideal testing conditions, and do so in real-time.

2.2 Specifications

The primary stakeholders, the CARSTOP research group and the Texas Department of Transportation (TxDOT) will use this project to guide further research and to shape future public policy in transportation safety. To satisfy the stakeholders, the system will have to meet certain operating environment and performance specifications. CARSTOP and TxDOT do not expect this system to be exposed to extreme environments, therefore it only needs to be tested for fair weather conditions. To meet performance requirements, the system will have to achieve a sufficient level of accuracy while being able to process data in real-time.

Ideally the system would work in any condition that a driver might regularly face. However, because the system is going to be used for research purposes, operating environments are

restricted to reasonable testing conditions, which is clear weather, temperature between 10℃ to 35℃, and a driving speed of less than 45 miles per hour. Table 1 describes the conditions that the system will operate in.

Table 1. Ideal Testing Conditions

Condition	**Operating conditions**
Precipitation	0%
Temperature Range	10℃ to 35℃
Fog	0% (perfect visibility)
Maximum Speed	45 mph

In terms of its performance, the system needs to prioritize detection accuracy and speed. The goal is to improve upon the object detection accuracy achieved by the previous senior design team, which was 46.39% [2]. The system should also take less than 250 milliseconds on average to process each chosen frame from the camera footage, as that is close to the fastest reaction times possible for a human [3]. Since preventing a collision is more important than distracting the driver, outputting a false positive is preferable to outputting a false negative. However, false positives can distract the driver, so they should be no more than 30% of alerts. In addition to accuracy, the driver needs to receive enough notice to take corrective measures. Therefore, the system should provide 3 seconds of notice before any collision. Table 2 lists the performance specifications required for our design.

Table 2. Performance Specifications

Measurement	**Quality**
Detection Accuracy	Improve upon previous group's accuracy of 46.39% [2]

Alerting Driver in Timely Manner	Should provide the driver 3 seconds notice of possible collisions
Processing Time	Should take less than 250 milliseconds to process each selected frame
Collision Alert	No more than 30% false positives

3.0 DESIGN PROBLEM SOLUTION

3.1 Overview

The objective of the project is to create a collision prediction system that can warn drivers of impending collisions, thus preventing a substantial portion of accidents that are caused by human error. The design will have to perform accurately in ideal testing conditions, be accurate in detecting objects and predicting collisions, and do so in real-time.

This collision detection system design has 8 distinct modules: sensors, object detection, LIDAR-camera synchronization, LIDAR-camera transformation, object location, Kalman filtering, collision detection, and alert. See Figure 1 for a block diagram illustrating how all these modules work together.

Here is a high-level description of the design: Starting with the sensor modules, the LIDAR and camera collect data on potential obstacles in front of the vehicle while the GPS is used to determine the velocity of the own vehicle. The object detection module uses camera data to identify potential obstacles and assign bounding boxes to them, which it sends to the LIDAR-camera synchronization module. This synchronization modules collects LIDAR point cloud data and bounding box information from camera data at the same instant in time, which is used in the LIDAR-Camera transformation module. This module projects the 3D LIDAR points onto the 2D image captured by the camera, making it possible to assign a distance collected from the LIDAR point cloud to the objects within the bounding boxes. The bounding box and LIDAR data is used by the object locator module to determine how far away each object is. Next, the Kalman filtering section creates a model used to predict the path of the objects based on their current position and velocity. This model is used in the collision detection section to predict whether the

vehicle will collide with any of the objects 1 second in the future. If a collision is detected, the alert module will notify the driver.

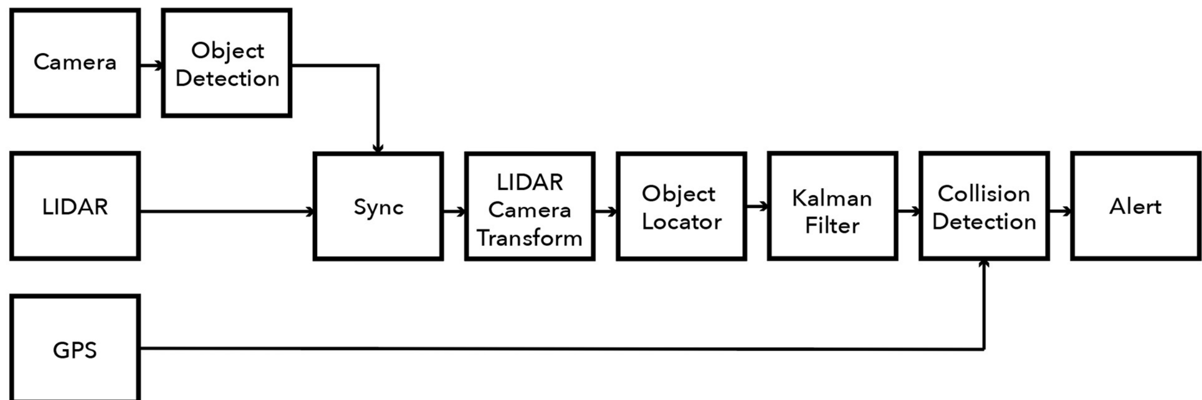

Figure 1: System Block Diagram

3.2 Sensors

The three sensors used in this system are camera, LIDAR, and GPS. This project is using a Logitech C920 1080p HD webcam, the Velodyne PUCK VLP 16 LIDAR, and a modified Garmin GPS that is specifically tuned to interact with the VLP 16 LIDAR. The webcam is used with the object detection algorithm to detect obstacles. The system uses LIDAR point cloud data to measure how far away obstacles are from the vehicle. GPS is used to determine the vehicle's velocity.

3.3 Object Detection

The object detection module receives frames from the camera and uses those to determine whether there are objects present in the frame and creates bounding boxes around detected objects. This looks for objects in front of the car and the bounding boxes of those are going to be considered the possible obstacles that the system will have to follow to predict whether there is a possible future collision or not. After obtaining the bounding box coordinates of each detected object, the module sends the bounding box data to the sync module, which matches the LIDAR point clouds to corresponding object detection data. This will allow the system to obtain distance information for each detected object, which will be used to track the objects and predict collisions.

In order to accomplish object detection, an algorithm called You Only Look Once (YOLO) was used, which is capable of detecting over 9000 object categories in real-time using a webcam [4]. YOLO is different from other object detection methods in that it is essentially a regression problem: it goes straight from image pixels to bounding box coordinates and class probabilities. This is where the name comes from - you only look at the image once to predict the objects in it and where they are. Many other object detection algorithms have much more complex pipelines which are slow and hard to optimize, whereas YOLO's simplicity allows it to run extremely fast and still achieve a high detection accuracy. YOLO works by unifying the separate components of object detection into a single neural network while other object detection algorithms go through multiple neural networks [5]. It uses features from the entire image to predict bounding boxes and predicts all bounding boxes across every class simultaneously. An overview of the system is shown in Figure 2 below.

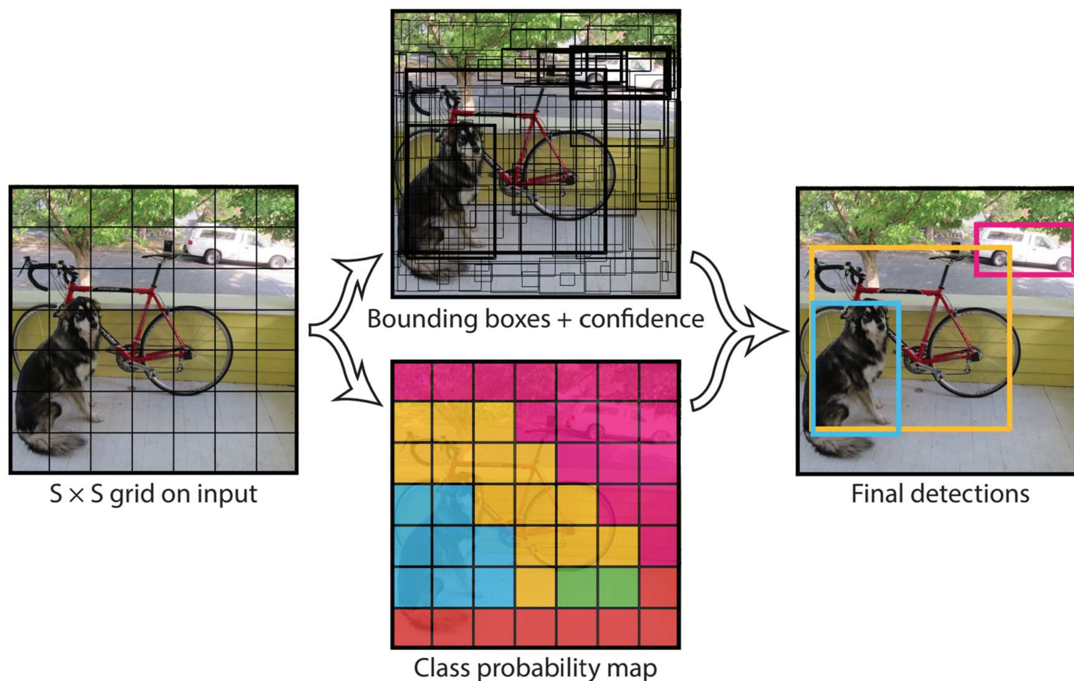

Figure 2. Overview of YOLO system

This algorithm was used by the previous year´s senior design project team so another advanteage was the knowledge of how to work with it, however it was necessary to modify some of the settings to process each frame in less than 250 ms. These modifications consisted of reducing

the number of divisions on the grid. The main problem of this was to find an equilibrium between time and accuracy. At last the selected number of divisions for the grid that worked with the lab´s computer at a reasonable speed and accuracy, that will be mentioned in the testing section, was 288x288.

YOLO divides the input image into an S x S grid of cells and if the center of an object falls into a grid cell, that cell is responsible for detecting that object. Each grid cell predicts B bounding boxes and confidence scores for those boxes. Each bounding box has predictions for the (x, y) coordinates representing the center of the box, the width and height of the box, and the confidence score. Additionally, each grid cell predicts C conditional class probabilities, Pr ($Class_i$ | Object), and uses those to create a class probability map. Finally, the bounding box predictions and the class probability map are combined to produce the final detections and the corresponding classes. Note that S, B, and C are tunable parameters that can be adjusted to improve detection performance. This model is implemented as a convolutional neural network consisting of 24 convolutional layers followed by 2 fully connected layers, that gives us a width and height of 288. This setting allows us to work at a fast speed getting a sufficient accuracy. The initial convolutional layers of the network extract features from the image while the fully connected layers predict the output probabilities and coordinates.

After obtaining the bounding boxes, this module sends a data packet to the sync module. The structure of the packet is shown in Table 3 below.

Table 3. YOLO Data Packet Structure

Location in Packet	**Description**
Bytes 0-3	A header (0x12345678)
Bytes 4-7	Timestamp of packet (in milliseconds since Jan 1 1970)
Bytes 8-11	Number of objects detected by YOLO in that frame
Each subsequent batch of 16 bytes	The left, right, top, and bottom pixel coordinates of each object’s bounding box

This data packets were sent then to the LIDAR-camera synchronization module only once a LIDAR point cloud is received as the velocity of YOLO was higher than LIDAR velocity.

3.4 Sensor Data Syncing

The LIDAR point clouds and YOLO packets are sent at different rates, so this system is responsible for synchronizing them by matching YOLO packets with the point cloud whose timestamp is closest in time to the received YOLO packet's timestamp. Since the point clouds are sent at a slower rate than the YOLO packets, this module first waits to receive a point cloud, then after it has received one, it receives a YOLO packet, which would correspond to the object detections nearest in time to the received point cloud. After the module has received both the LIDAR and object detection data, it sends it to the LIDAR-Camera transformation module.

3.5 LIDAR-Camera Transformation

Although the object detection system works on 2D images, our LIDAR provides distances in 3D point clouds. Therefore, this system helps to correlate points in the point cloud to points on the images so that we can determine the distance to objects detected by this system. This is done by calibrating the LIDAR and camera in the position that they will be held by their mounts. Equation 1 shows the equations that will be used to help with the projections. The 3D coordinates x, y, and z are converted to the 2D coordinates u and v through matrix multiplication [6]. The r matrix contains coefficients determined during the calibration of the two sensors. The variables c_x and c_y represent the center of the photo while f_x and f_y represent the focal lengths. Figure 3 shows an illustration of how our 3D point cloud would be projected onto our 2D image.

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{1}$$

Equation 1. 3D-2D Point Projection Equations [6]

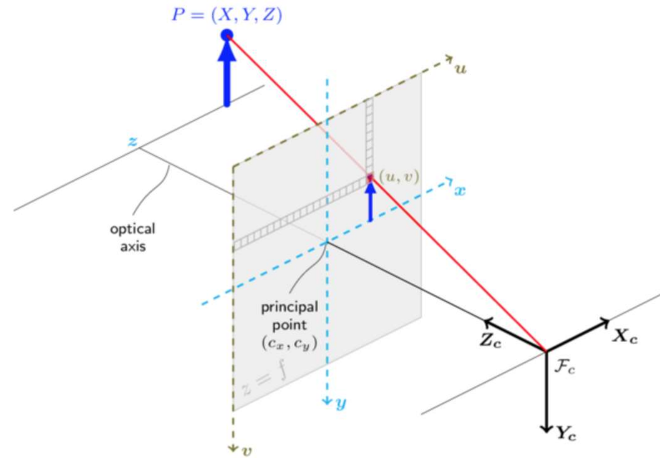

Figure 3. Illustration of 3D-2D Projection [6]

3.6 Object Locator

Now that points in the point cloud and pixels on the image are correlated we need to create a system to identify the locations of objects detected by the object detection system. It was decided to select the closest LIDAR point to the center of the bounding box of a detected object. The formula can be seen below in equation 2. X and Y are the relative location of the object compared to the car, xp and yp are the pixel coordinates of our LIDAR points after passing through the LCT, and x_{center} and y_{center} are the center location of the bounding box.

$$(x, y) = minimum((x_p - x_{center})^2 + (y_p - y_{center})^2) \quad (2)$$

This system was initially tested on the initial truck photo that was used to calibrate our LCT. Minimizing this error picked a LIDAR location at the center of the bounding box created by the object detection system. The point selected can be seen located in red in Figure 4.

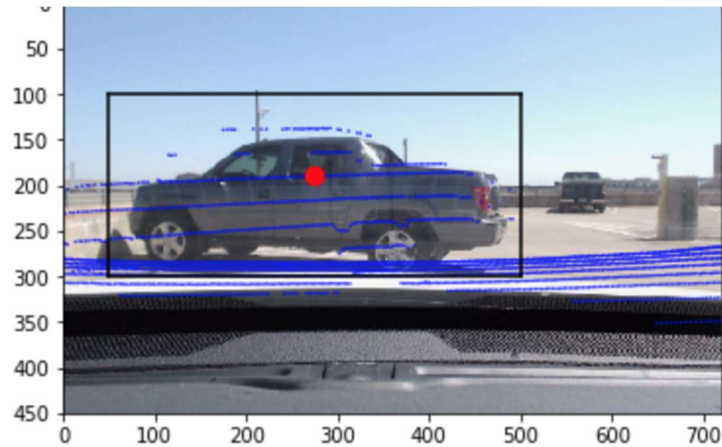

Figure 4: Location Selection of Truck Using Our LCT

3.7 Kalman Filter

Kalman filtering is an algorithm that allows us to estimate accurately unknown variables using single measurements of the input and output along with measurements of the statistical noise and inaccuracies. We can divide this algorithm in two steps. The first one that can be called "Prediction-step" gets an uncertain estimation of the current state values, this will be an output of our Kalman filter. The second step is the "Correction-step", in this step the algorithm updates itself with weighted averages giving more weight to those more accurate estimations. We measure this by comparing the "Prediction-step" state values with the ones measured during this step.

The Kalman filter, developed by Rudolf E. Kalman on 1960, is a recursive algorithm that can work in real time using the actual inputs of the system and the previous ones. It has many different applications in technology and now is being really used in autonomous driving platforms.

In order to detect if there is going to be a collision it is needed to track the objects around the car and be able to predict where they will be at a later point in time. The system is going to track objects around the car using an Extended Kalman filter. This approach helps to account for variations in measurement and create a more accurate tracking system. These formulas can be

seen in Equations 3-9 below [8]. The input to our tracking system are the x and y coordinates and velocities of an object relative to the location of the car. The Kalman filter will then use this data to create a prediction model for where it expects the object to be located and how fast it will be going the next time it receives an input. For this we consider that the object is going to move in a straight line and at the same speed the whole time between one prediction and the next one. This predicted model location (x') can be seen in equation 3 below. Equation 4 shows how much error (P') we expect to have in our predicted model. This is based off of our predicted model (F) and our covariance matrix (Q). The error we predict to have is going to be the acceleration of the objects because our inputs do not consider it, so it will be considered noise. The next time that we receive a measurement we calculate the error (y) between our predicted model (x') and in our input (z) this can be seen in equation 5. Equations 6 and 7 show how we use the expected error in our prediction to find our Kalman filter gain (K) where our error in measurement is (R) and H is a matrix to shape our matrices. Our next state (x) is then calculated as our predicted location (x') plus the difference in our predicted location (x') and the measured location (z) multiplied by our gain(K). This is shown in equation 8. Equation 9 then updates the expected error (P) in our predicted system for the next round. The Kalman Filter then updates our location using both our predicted and measured locations giving a greater weight to the method with lower error.

$$x' = Fx + u \tag{3}$$

$$P' = FPF^T + Q \tag{4}$$

$$y = z - Hx' \tag{5}$$

$$S = HP'H^T + R \tag{6}$$

$$K = P'H^T S^{-1} \tag{7}$$

$$x = x' + ky \tag{8}$$

$$P = (I - KH)P' \tag{9}$$

Equations 3-9. Kalman Filter Equations [8]

As mentioned before, the Kalman filter actualizes itself continuously comparing previous inputs with the actual ones and changing the weights of the variables giving more importance to the more accurate ones. The predictions of it get more accurate as more updates are done being the first prediction a blind pick. This submodule sends the next predicted positions to the collision

detection system which will calculate if during this path the position of the detected objects ever intersects with the position of the car.

3.8 Collision Detection and Alert

The collision prediction module uses data from the GPS along with data about objects surrounding the user on the road from the object locator and the movement predictions previously made by the Kalman filter to detect a potential collision and tell the alert system to warn the driver.

The collision detection module was created on the basis that within the one second time period that our overall system attempts to predict into the future, the user's car will continue to go forward at a constant velocity, this is the same basis as the used in the Kalman filter. This approximation simplifies the calculations and the error are acceleration and possible changes on the direction of the vehicle. The module uses GPS information to determine the car's velocity and generates a straight path in front of the car, in the direction that it is going, based on the velocity, multiplied by the time increment (one second in this case). The module will then use information about potential objects (size, location, angle of approach, and velocity) from the object locator and object tracking modules to generates paths for these objects. Lastly, the module will check for any intersections between any paths. If any are found, a signal is sent to the alert system to warn the driver. This is all meant to occur in real-time, with minimal processing delay such that the alert can be sent within a three second timeframe.

Figure 5 graphically demonstrates the collision detection module. The dark red box represents the user's car, and the dark blue box represents some other car/object. The lighter colored long boxes represent the object's' paths. As shown, the paths intersect, and a crash was detected.

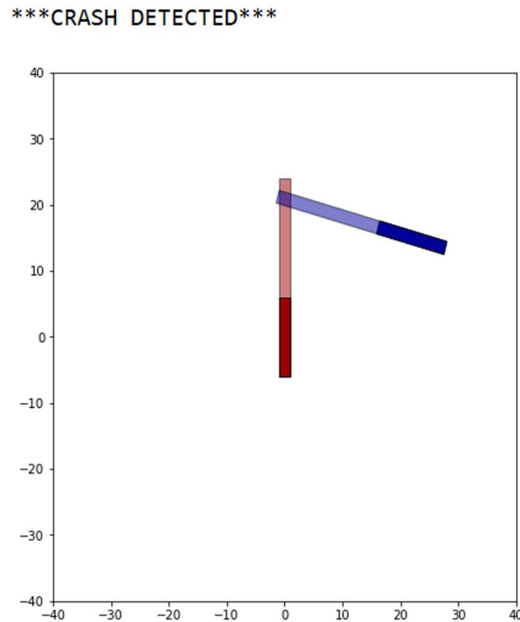

Figure 5. Collision Detection

Once this system detects a possible collision it makes the user computer sound using a python library. The alarm system is presented as a distinct module, but it was all included in the collision detection system because of its simplicity. The python library creates a “beep” using a square wave.

4.0 DESIGN IMPLEMENTATION

The final system design evolved substantially from the original design, which had included platforms such as Apollo and the Robot Operating System (ROS) along with other resources like the Karlsruhe Institute of Technology (KITTI). Compatibility and accuracy issues forced to change the design and use modules from the internet and self-designed. For the final design, YOLO was chosen for object detection primarily for its speed, after this the lidar-camera transformation aligns point clouds and images using rotation and translation matrices and uss this transformation to locate objects which were tracked using Kalman filters.

4.1 Failed Design Components

Several designs were tried before finding they were unsuccessful and settling on the final design solution. Some of the components that were tried but ended up not been used were basing the solution on platforms like Apollo and ROS and using KITTI to calibrate our LIDAR and camera.

4.1.1 Apollo

The first design used an open-source autonomous driving platform called Apollo from Baidu, but after a few months of working with it, it was decided not to use it because of time and resource constraints. While working with Apollo, it was found that the Apollo platform is more difficult to modify and less robust than what was anticipated at first. While running the Apollo simulation demos successfully, it was impossible to modify the software to be compatible with the sensors even after reaching out to the Apollo developers on their online forum. After weeks of working on integrating one sensor with no success, the determination was that in the interest of time and to avoid buying new expensive sensors it would be best to move on to other platforms for the design.

4.1.2 ROS

Another platform tried was ROS (Robot Operating Systems), which ended up not using because of software issues. ROS was considered because Apollo is based on ROS software and ROS is a commonly used platform in industry and research projects. Also, ROS is compatible with many premade software packages related to the sensors. It was successfully installed the latest version of ROS and the software packages related to the sensors, but it also was impossible to install the software needed to sync the camera and LIDAR together. Since syncing the camera and LIDAR data is essential to this project, it was decided to not use ROS as a platform and create the software modules and architecture seen in the final design without using any premade open-source code.

4.1.3 KITTI

The first attempt at a LIDAR-Camera Transformation (LCT) involved an online program from the Karlsruhe Institute of Technology (KITTI). This program uses photos of black and white checkerboards in junction with a LIDAR point cloud to calculate a rotation matrix. In order to

get the transformation from this program the first step was to set up our LIDAR and camera on the car in the position that they will be used. The LIDAR was placed on the mount on the top of the car and taped the camera on the top of the dashboard facing the front of the car. Next, it was necessary to setup the black and white checkerboards in front of the car facing different angles. Because this system calculates the rotation matrix using the normal of the boards in the point cloud, it is important that these face multiple different angles. A photo of this setup can be seen in Figure 6 below. After setting up the test we then took a photo of the boards and collected a LIDAR point cloud. These photos and LIDAR point cloud are then uploaded to their website where they calculate the rotation matrix.

Figure 6: LIDAR-Camera KITTI Calibration Checker Board Setup

The program then tries to find the most accurate matrix for the setup. If it finds a good solution, then it outputs one transformation. If it does not, then it sends multiple for you to choose from. It gave 23 separate LCTs to choose one from these. This means that the results are not accurate and most likely would not meet the previous expectations. The next step was to look through the options and found the transformation shown in Figure 7 to be the most accurate. The photo on the left shows the photo that we submitted and the identified boards which are each assigned a color. The photo on the right shows these color mapped boards inside the plane of the LIDAR. It is easy to see that each of the boards is approximately in the correct location of the point cloud

and leaning in the right direction. However, it appears that the boards were translated further to the right then they actually were. Because of this translation it was decided that this transformation was not accurate enough for the purposes. This means that it would be necessary to create a calibration in particular using matrix for the position of the sensors. It was done with code in python based off equation 1 in the LIDAR-camera transformation section of the design problem solution. The process used is described in detail later on in the implementation section.

Figure 7: Panel Mapping (Left) and LIDAR Mapping (Right)

4.2 Design Choices

In the section it will be discussed the reasons for the current design. Each module was designed to work with the previous module and give an output to the next module. YOLO was used because it could process data quickly in real time. After investigating many other calibration options, it was decided that the created LIDAR-camera transformation would be faster and more reliable than other sources. The center of the bounding box was used for object locator because it gives more accurate results than anywhere else in the box, since an object is most likely to exist in the center. To track objects, Kalman filtering is the most reliable method, and transcribed MATLAB code into Python. For collision detection, it was assumed that the car would be in an approximate straight line with respect to the previous frame, and updating the data quickly enough, so it could predict a collision in 3 seconds. The final decisions because of the time constraints was to build an audio alert rather than a haptic alert.

4.2.1 YOLO

YOLO was used for the object detection algorithm primarily because of its speed, but also because of its familiarity and ease of use. Keeping processing time minimum was crucial to the project as the application was inherently a real-time one. It was essential that this system took less than 250 milliseconds to process information, as this is close to the fastest reaction times a human is capable of. YOLO was the only algorithm that met this requirement. In preliminary speed testing, YOLO took around 30-50 milliseconds per image, and consistently ran at more than 25 frames per second when tested on video from a webcam. Other algorithms, such as VoxelNet [9], took upwards of one second to process a single image.

YOLO also had the advantage of being familiar. The previous year's senior design team, which had also worked on some real-time object detection, had used YOLO in their system and the availability of some of their code as examples was helpful to create the module in the shortest possible period upgrading what the previous year´s team had. Furthermore, the project´s faculty mentor's graduate students were already knowledgeable with it and could provide guidance if needed. There was helpful documentation online. All of this made it possible to have an object locator algorithm with great precision and that can track and classify multiple objects of different categories at the needed speed.

4.2.2 LIDAR-Camera Calibration

The next attempt at creating an LCT involved a similar process to the previous one however, this time it was decided to calculate the rotation matrix instead of using one calculated with an open-source algorithm. For this task it was necessary that the LCT was using objects that are easily identifiable by eye in the point cloud. For this reason, the point cloud and image of a truck were aligned. This object is easily identified and can be moved on top of the photo of the truck to calibrate our LCT. The owner´s car was moved to the roof of the parking where it was parked and positioned in front of another truck to get the images and make the calculations using those. Figure 8 shows the LIDAR point cloud and photo that we used to set up our LCT calibration. In the point cloud the truck can be seen towards the top left near the rings of the circles.

Figure 8: Point Cloud (Left) and Photo (Right) of Truck LCT setup

After the data the next task was to write code to overlay the point cloud on the photo. The start off the rotation matrix was an identity matrix. At this point the photo was checked to see if it was possible to find the truck in these points. Once the truck was identified, the next stage was to see how our point cloud should rotate to accurately place the truck on top of the photo. At first, the object was only rotated along the x, y, z values by 90 degree rotations. Once this rotation got the truck to be in the right alignment the tune of our rotations was started in order to make our rotation more exact. Then a translation matrix was added to shift the cloud to make the final adjustments. The reached values can be seen below in Table 4. These rotations gave the result shown in Figure 9. It is possible to check that it overlays the points on the truck rather closely. This places the points on the truck and within the region where it would be expected a bounding box to surround the truck. The rotation matrix was then used on several other photos and point clouds that were took in order to verify that it worked in multiple situations. The results of these LCT uses can be seen in Figure 10. From these results it is possible to see that the LCT worked really well on the pillar, rather well on the two trucks, and poorly on the person. Before taking the photo of the person the camera was accidentally knocked to the side. The attempts to correct this mistake were not completely effective so there exists a shift between the point cloud and the image.

Table 4. Final LCT Calibration Measurements

	Rotation (degrees)	Translation (meters)
x	90.53	-1
y	-81.36	3
z	90.53	0.7

Figure 9: Image of point cloud overlaid on truck using LCT

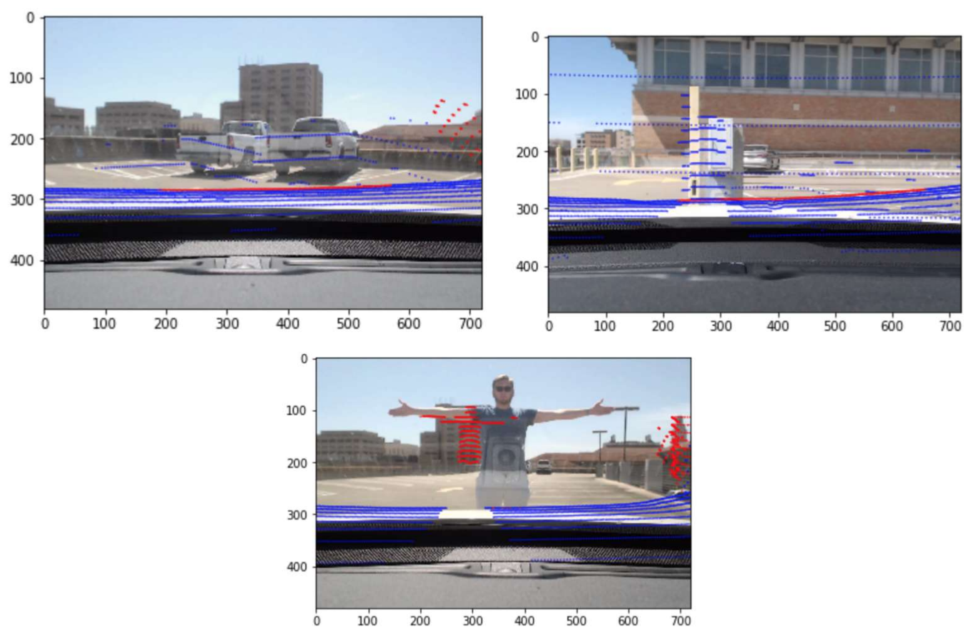

Figure 10: LCT Used on Two Trucks (Top Left), a Pillar (Top Right), and person (Bottom)

4.2.3 Object Locator

Using the data from the object detection and LCT module, it was necessary to get the distance of the object from the car. Using the bounding box coordinates, it is possible to localize the objects within the frame. Since the LIDAR point cloud was transformed into camera coordinates already, it was possible to use the object locator module to find the points from the LIDAR point cloud that are in or near the bounding box and get their corresponding distances. The initial idea consisted of just picking any point within the bounding box. However, after consideration, it was decided this would not work because at any random point inside the bounding box where there is a person, there is a chance that the random point we choose may be a point underneath their arm or beside their head which will give us the wrong distance of the object in the bounding box. Therefore, the solution was to use the center of the bounding box, which is far more likely to actually contain the object. It was also important for this function to work well with our other submodules, so calculations and code were done for the module immediately after the LCT module. We felt this met the time processing and accuracy demands.

4.2.4 Kalman Filter

The initial Kalman filter used code from an Extended Kalman Filter developed by Yunming Shao, a PhD student from Ohio State [10]. This code helped to understand how Kalman Filters worked and what they did. This code used noisy paths generated by MATLAB and tracked them helping to reduce the noise. However, it was soon realized that this system would not suit the needs for real time tracking. The format was based on file input and was difficult to change to fit our needs. As a result, the final idea involved creating the Kalman Filter code in Python. Because writing the code made it easier to understand and modify than the one originally used. After getting the Kalman filter to work on one object the code was changed to work in a class system. This allowed it to create new Kalman filters for each object that was wanted to track.

With enough time to help differentiate between different objects this Kalman filter would already be capable of tracking all of them at the same time. For that an idea that did not materialized was to create a New vs Old subsystem which would allow the system to differentiate objects that were already detected by YOLO from objects that are new and ones that are now out of sight.

Most of the New vs Old subsystems found online were only based on the position of the object and compare it to the predicted position. It was possible during the project no matter time constraints to try this method but it was not accurate enough so it was decided not to implement it. For this system three characteristics would be considered for each object to make sure that it worked and reduce the number of errors. An object would be a previously detected one if it was in a position relatively close to the one predicted by this object´s Kalman filter, if its bounding box height vs width relation was equal or similar and if the object class detected by YOLO algorithm was the same.

4.2.5 Collision Detection

The Collision Detection Module is where the outputs of all of the other modules, as well as from the GPS comes together to output the final alert to the user. The most important assumption that guides the implementation of this module is that within a short period of time, the path of the user's car will be approximately straight. Initially, this short period of time was three seconds, however, later the decision to reduce it to one second in order to maximize accuracy was made. The module uses velocity taken from the GPS to determine the car's path as a narrow rectangle directly in front of it, where its width is the width of the car, and its length is the distance it will go within the time interval (its velocity multiplied by one second). The module then uses information about potential objects (size, location, angle of approach, and velocity) from the object locator and object tracking modules to generates paths for these objects. A collision is defined as an intersecting path. If any are found, an auditory alert warns the driver.

The Module uses object-oriented programming with Car classes representing vehicles. Various functions such as update_speed() to get velocity from the GPS, and update_path() to generate paths work in tandem to detect collisions. Object-oriented structure was used so that the system could be easily extended to multiple objects and so that in the future, different objects could have different attributes - such as a person having a different bounding box size than a car. The system was ready to work for multiple collision detection and tested successfully in lab with fake data of both cars and people.

4.2.6 Alert

The alert module outputs a beeping sound when a collision is detected. The original plan was to use a haptic alert embedded in the driver's seat, where different regions of the seat would vibrate based on the direction of the collision. The haptic alert was considered at first because it would be able to provide the driver information about expected collision direction, and it also seemed more reliable than an auditory alert because all drivers make contact with the car seat, whereas not all people would be able to hear an auditory alert. However, the haptic alert would have been costlier because it would have required to buy many vibration motors, and it would have taken longer to implement because to write code for a microcontroller to interface with each motor and then only vibrate certain ones based on direction is a time demanding task. Due to the increased cost and the limited amount of time, the final decision was an auditory alert because it could be easily generated using a laptop, it still provides the driver collision information without being too distracting, and its ease of implementation allowing to focus more on the functionality and accuracy of the system.

The first and rejected idea was to use a visual alert because it will be too distracting to the driver as during a possible accident, to have your eyes focused on the road is a priority.

4.3 Cost/Benefit Analysis

This section describes the cost that the project involved and compares them with the benefits which are expected to result from the developed system. As the project was intended primarily for research purposes and not to obtain an actual quantifiable benefit, there is no way to give actual financial benefits and it is not possible to calculate a payback period.

The sensors used for the project, the Velodyne PUCK VLP 16 LIDAR, the Logitech C920 1080p HD webcam, the Honda Accord and the laptop, were already owned by the CARSTOP research team so the only actual economic expense of the project was the Modified Garmin GPS. During the project, an important decision was whether to buy a more precise Velodyne LIDAR with 64 layers instead of 16 because it could worth it due to compatibility problems. However, due to the high cost of the 64 layer LIDAR the decisions was not to pursue this path.

Although there only was one actual expense, so in terms of time spent vs benefits, the benefits were well worth the time putted in. In less than a semester a functional collision prediction system accurate in predicting collisions with one object was created. With more time, this system could be developed further into a cheaper alternative to expensive autonomous driving systems. This could potentially provide the benefits of being compatible with many different types of cars and could function as an affordable predictive safety solution which would increase safety while driving. This system is the first step to a complete usable collision prediction and avoidance system which is an essential part of autonomous driving.

5.0 TEST AND EVALUATION

After designing the system it was time to start to work on creating each module. Most of the testing occurred while building the system. Each module was tested individually as it was being built, then began testing how each system worked together, and then finally the total system. The first tests performed were to make sure that the system was receiving the accurate data. Next, the object detection system was tested to make sure that it could output the bounding boxes of detected objects. Then the syncing of data between the object detection and LIDAR was the one tested. After the calibration and location sections were finished it was time to see how they worked in conjunction with the LIDAR and object detection systems to determine the locations of objects in real time. The Kalman filter and collision detection systems where the final ones and used paths that were created in lab. Next step was to make sure that the system operated quickly enough to meet the goal of real time prediction. Once this was verified and all of these systems worked together we tested the entire system in different scenarios with and without collisions.

Each member of the team tested his own module while the testing for sensors, the interaction between modules and the overall system was done by all the members of the team together. This was necessary as each test would need deep knowledge of distinct submodules and the ability to change it during the experiments.

5.1 Sensors

The tests of LIDAR and GPS sensors were done individually. For LIDAR, the idea was to check the validity of the data collected and the timing of the methods with the design constraints in mind. The accuracy of the GPS was also tested with the car running at 5, 10 and 15 mph and with a simple Python code that was able to output to the computer the speed of the car at each moment.

5.1.1 LIDAR

The task was to test the software wrote to collect LIDAR data by measuring how long it takes to receive one packet of data and by writing code to visually verify if the system would be getting the correct LIDAR data.

To test how long it takes to collect one packet of data from the LIDAR a Python module was used to measure the current time in microseconds at the start of the packet and at the end of the packet. The time to receive one packet was measured with the difference between the start and end time. A similar method was used to measure how long it takes to receive one point cloud. A point cloud consists of about 90 packets. Initially, it was found that it took 13 milliseconds to receive one packet of data and 5 seconds to receive an entire point cloud of data, which is too slow for this design requirement of processing each frame in less than 250 milliseconds. This is because the code was using a data structure called a DataFrame to store the point cloud data. So the solution was to switch it to simply using arrays in Python and had much better results. With arrays, it took 1.312 ms (averaged over 90 trials) to receive one packet of data from the LIDAR and the time to receive one point cloud was about 118.141 ms. Because this system is the most time intensive a time of 118 ms is sufficient for the system to reach the goal of 250 ms per frame. A summary of the testing for the LIDAR packet data tests can be found in Table 5.

Table 5. Time to Receive LIDAR Data

Data Structure	Time to receive packet (ms)	Time to receive point cloud (ms)
DataFrame	13	5000
Array	1.312	118.141

More code was used to verify that the system would be collecting data from the LIDAR correctly. The code used receives data from the LIDAR directly from the computer ethernet port. It decodes the packet data and convertes the given spherical coordinates to cartesian coordinates using these equations:

$$x = r \sin\theta \cos\varphi \qquad (10)$$

$$y = r \sin\theta \sin\varphi \qquad (11)$$

$$z = r \cos\theta \qquad (12)$$

Where r is the distance from the sensor, θ is the elevation angle, and φ is the azimuthal angle.

To test the validity of the data point conversions, the data collected from the packets was plotted in 3D coordinates using the MATLAB plotting library in Python and compared it visually against the data collected from VeloView. VeloView is the software that the LIDAR is shipped with, so users know data is correct. To compare the data, it was necessary to caputre one point cloud of the same environment in VeloView and one point cloud with the tool we created within a few seconds of each other and plotted the data for each in 3D. The plots were compared visually then looking at individual points and compared their X, Y, Z coordinates to see if they were similar. Initially, it was found that the data did not match with the VeloView software, as the two plots shown in Figure 11 were drastically different. After some debugging, it was discovered that the difference was because of the use of degrees instead of radians in the calculations. Once changed to radians, the results became more visually similar, showing that the data collected from the packets is correct. It was also verified that the data collected is correct by confirming that their point's X, Y, and Z coordinates matched the VeloView X, Y, Z points.

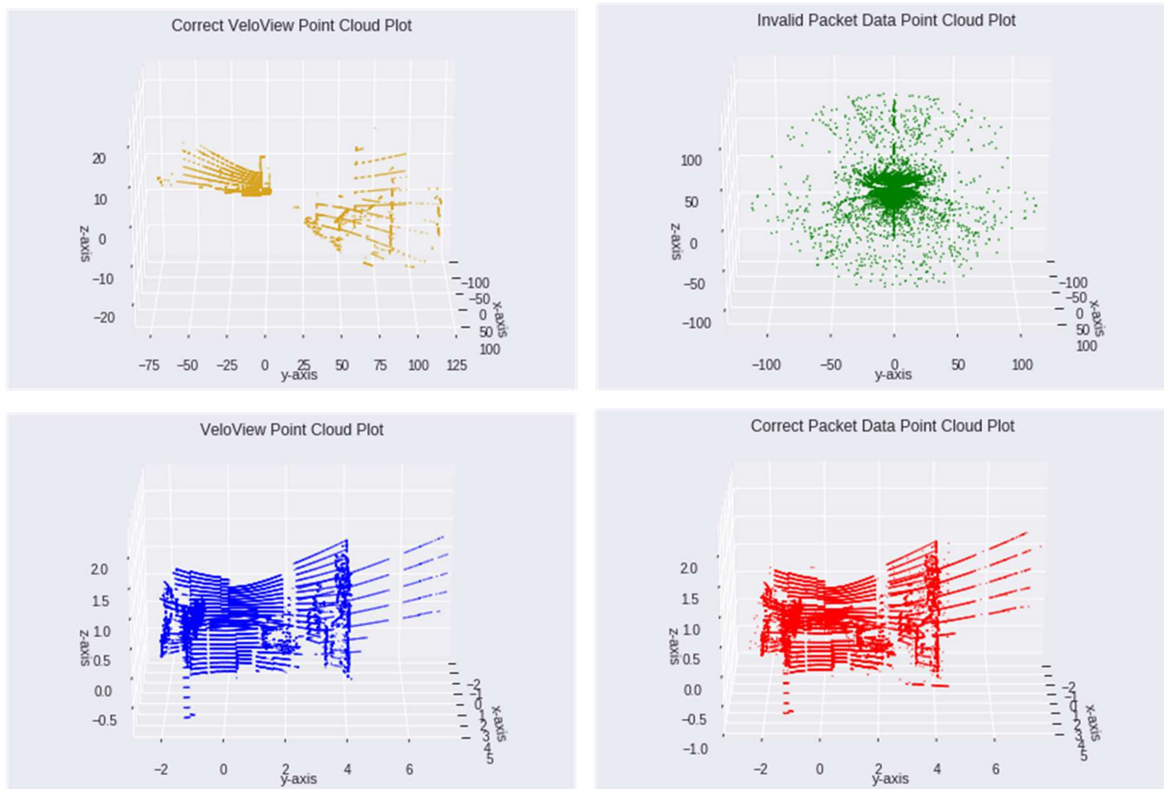

Figure 11. LIDAR point cloud 3D plots

5.1.2 GPS

The test of the GPS's ability to track the speed of a car at 5, 10, and 15 mph found that it is generally capable of tracking the speed of the car but is more accurate at higher speeds. The GPS measures speed in nautical miles per hour (NMPH) so it is necessary to convert this data into miles per hour (MPH) to test it. In order to convert to miles, the code simply multiply nautical miles by 1.15. The test conducted evaluated the accuracy of the GPS speed acquired vs actual speed. A short program that would fetch the GPS data whenever the function is called was wrote, and it would output the current speed which the system use along with longitude and latitude which we do not need for this purposes. To ensure the GPS functions correctly, a simple test was done with the car driving steadily at three different speeds. Then the code gathered the data for five different runs, and the results are seen in Table 6.

Considering that an speedometer cannot have an error of more than a ±2.5% relative to the actual speed according to manufacture tolerances and supposing that the cars conditions were ideal it

got a maximum percent error of 30.841% for 5 mph, 32.996% for 10 mph and 3.746% for 15 mph. Although at very low speeds there are some potential issues with accuracy, the tests at higher speed are more accurate. The conclusion was that the GPS tests were successful and confirmed the functionality that satisfies the requirement of getting the speed of the vehicle as most of the time user´s vehicle is going to be running at speeds higher than 15 mph if it is taken into account that the minimum common United States roadway speed limit is 15 mph for school zones.

Table 6. Results of GPS Speed Test

Run	NMPH at 5	NMPH at 10	NMPH at 15	MPH at 5	MPH at 10	MPH at 15
1	2.0	7.8	11.9	2.3	11.76084	13.694282
2	2.4	8.1	14.1	2.761872	12.21318	16.225998
3	4.1	9.2	12.3	4.718198	13.87176	14.154594
4	4.3	8.9	12.5	4.948354	13.41942	14.38475
5	2.6	9.0	13.5	2.992028	13.6	15.53553
Average	3.08	8.6	12.86	3.5444024	12.96708	14.7990308

5.2 Object Detection

The first test of the object detection module was on images to get some confidence in the YOLO algorithm's functionality, and then on video. To test YOLO's accuracy, the process consisted on running the algorithm on 10 images and calculated its precision and recall on each image. To test the algorithm's speed, it was measured the time it took to process the entire frame. For video testing, the average frames per second (FPS) of the algorithm were calculated.

This test involved calculating the precision, recall, and accuracy for each image in our set of 10 testing images we sourced from the internet. These pictures were taken from positions inside the car from a position similar to the webcam's position in this project´s design and so looked similar to the images the object detection would have to work with. This set of images contained 4 nighttime pictures, and 1 picture which was taken in heavy snow. As this mix of images would provide a better understanding of the algorithm's accuracy and robustness. As seen below, the object detection module was able to detect the vast majority of objects in both daytime and

Figure 12. YOLO Output on a Daytime Image

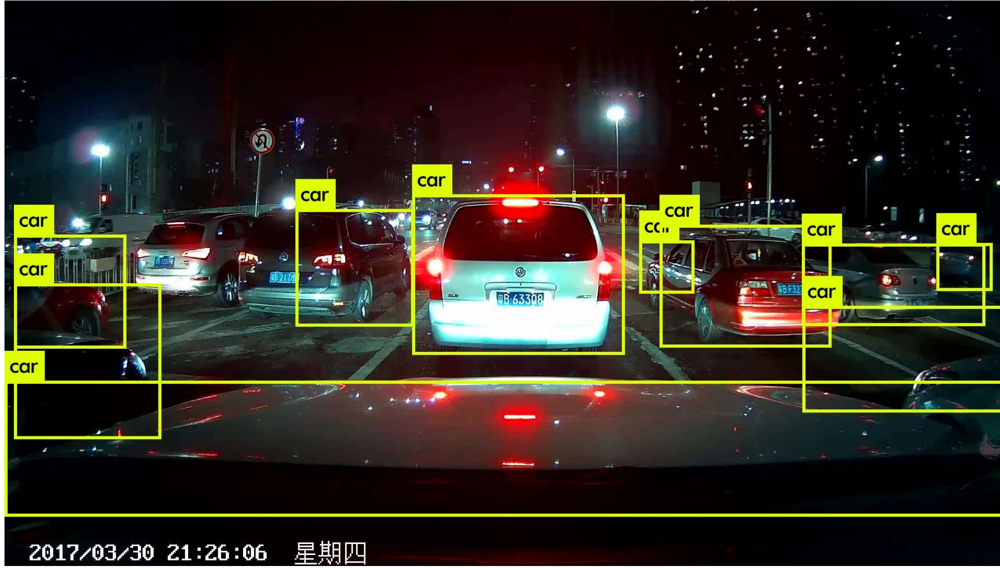

Figure 13. YOLO Output on a Nighttime Image

To get a quantitative value on the accuracy, the metrics of Precision (P), Recall (R), and Accuracy (A) were used. Precision is the proportion of detected objects in the image which were actually objects, recall is the proportion of objects that should have been selected which were actually selected, and accuracy is the proportion of correctly detected objects. These metrics are closely related to the concepts of a true positive, a false positive, and a false negative. A true positive (*TP)* is an object which is actually present in the image, and which is also detected by the algorithm, a false positive (FP)is an object which is not actually present in the image, but which our algorithm erroneously detects, and a false negative (FN)is an object which is actually present in the image, but which is missed by the system. The formulas for P, R and A are given below.

$$P = \frac{TP}{TP + FP} \qquad (13)$$

$$R = \frac{TP}{TP + FN} \qquad (14)$$

$$A = \frac{TP}{TP + FN + FP} \qquad (15)$$

After testing on 10 images, 9 of which had multiple objects, the object detection module had an average precision of 0.812, an average recall of 0.733, and an average accuracy of 0.616. To get the algorithm's processing speed, it was necessary to measure the time it took for YOLO to handle the image. On the same 10 images as before, the module took on average 147.55 milliseconds per image, with a standard deviation of 3.62 milliseconds. The details of this accuracy and speed tests are given in Table 7 below. The accuracy of 0.616 is successful because it it meets our goal of beating last year's accuracy of 0.4639.

Table 7. Accuracy and Speed Test Results

Image Number	**True Positives**	**False Positives**	**False Negatives**	**Precision**	**Recall**	**Accuracy**	**Time (in milliseconds)**
1	1	1	0	0.5	1	0.5	150.7
2	2	0	1	1	0.67	0.67	151.6
3	3	1	2	0.75	0.6	0.5	143.9
4	4	1	3	0.8	0.57	0.5	142.2
5	8	0	2	1	0.8	0.8	148.6
6	4	1	2	0.8	0.67	0.57	149.1
7	2	1	2	0.67	0.5	0.4	151.4
8	5	0	1	1	0.83	0.83	149.9
9	8	2	1	0.8	0.89	0.72	143.2
10	4	1	1	0.8	0.8	0.67	144.9
Average:				**0.812**	**0.733**	**0.616**	**147.55**

After performing the above tests on images, the following step was to test the algorithm's performance on video file inputs. It was not necessary to test separately the accuracy on a video input because the algorithm internally breaks a video down into its constituent frames, so the previous accuracy tests which had been performed on images would still be relevant. Therefore, this test focused on the algorithm's speed when supplied with video. It used 3 videos, each of

which were between 70 and 90 seconds in length and were taken from the dashboard of a car driving in urban traffic situations - one in suburban London, one in downtown Los Angeles, and one on an interstate highway. The average FPS over the 3 videos was 29.4 FPS. The conclusion was that these tests were successful because the average FPS is well within the specification of processing each frame within 250 ms, which is about 4 FPS.

5.3 Data Syncing

The syncing module receives data from both the object detection and LIDAR modules, so in order to test its speed it was necessary to start testing the average rate that each module sends data. The method used to test the LIDAR speed is explained in the LIDAR section. In order to test the rate that the object detection module sent data, the test received 200 packets from it, found the time difference between each packet's timestamp, and then averaged the time differences. After testing each modules' data rate separately, the code then combined them and measured the total time that the sync module took to execute. This was done by measuring the elapsed time to receive both data packets and averaging that over 100 trials.

From the individual module testing, it was found that the LIDAR sends point clouds approximately every 118 ms, while the object detection module sends packets every 82 ms. As a result of this tests, it was discovered that the faster data rate of object detection was causing its buffer to overfill, which produced inaccuracies in the synchronization between point clouds and object detection packets. This was fixed by only sending every 5th object detection packet, which allowed both modules to send data at approximately the same rate and produced synchronization. After fixing that issue, the combined sync test revealed that the average total time the module takes is 118.276 ms. The results of the tests are summarized in Table 8.

Table 8. Results of Data Sync Tests

Module	Time to receive data (ms)
LIDAR	118.141
Object Detection after LIDAR	0.135
Total Sync Time	**118.276**

This testing showed that the syncing time is well under the 250 ms requirement, which allowed the processing time of the entire system to be under our requirement. However, as a result of the testing the necessity of skipping some of the object detection frames in order to synchronize the speeds of the modules was found. This was necessary so that the system could operate in real-time, but it does have the drawback of not being able to process every single frame, which decreases resolution and could possibly decrease the accuracy of the overall system. If the system was tracking a vehicle moving at a high speed, then this skipping could be problematic and possibly cause a false negative.

5.4 Object Locator

In order to test the object locator, it was necessary to set up a testing environment with several different distances located in different parts of the frame. The car was then taken to an open parking lot and parked it facing a direction where the camera did not detect any objects; this helps to ensure that any object detected by the system is the object that we want to measure the distance of. At that moment people were placed in the frame and began to measure the distance from the LIDAR to the person with a measuring tape and comparing that value to the one output by the system. The first two tested scenarios involved a person standing in the center of the frame at 2.5 and 8 meters away. After verifying that these two initial values had an accuracy greater than 90% it was time to proceed to set up the test measuring multiple distances at different angles. This was done by placing markers on the left side, center of, and right side of our image at 5, 10, and 15 meters. Figure 14 shows an example of this test setup. Then someone stand on one of the markers and recorded the distance determined by the object locator. This test was repeated using a car in each location instead of a person.

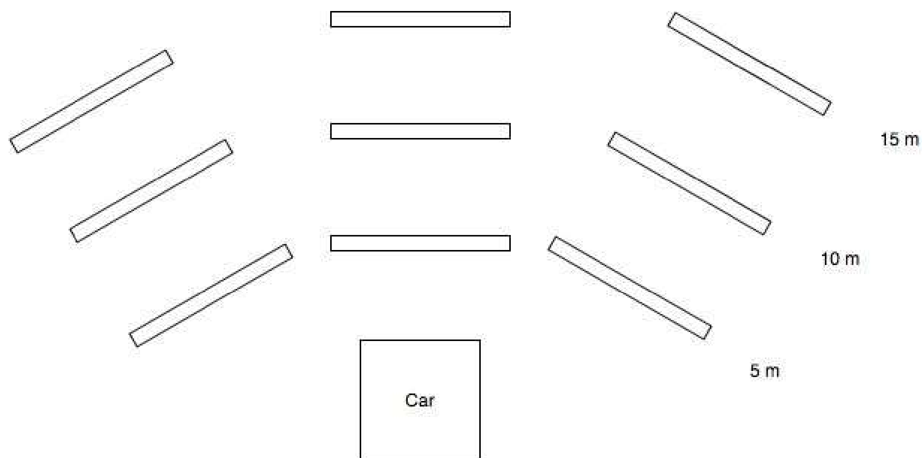

Figure 14. Object Locator Distance Test Setup

Tables of each of these recorded values can be seen in Appendix A. There were two noticeable trends in the data. The system provided more accurate data when the object was closer to the car and when the object was at the center of the frame. Figure 15 shows how the system performs on the different positions. When objects were at the center of the frame the distance accuracy was greater than 90%, on the right around 75%, but it drops dramatically towards 60% for the left side of the frame. While searching for the cause of these drops the solution was to view the calibration projection in real time. The conclusion was that when objects were close to the edge of the screen the calibration projects the points further away from the center of the picture than they actually were. When this mismatch in calibration is small the system can handle it because we pick the closest point to the center of our bounding box, however, when the difference between the object in the picture and the projection become too large the closest point to the center of the box becomes one of the ground points. This causes the system to record a distance which is much closer than the actual distance. This effect was more pronounced on the left side than on the right, but it is still unknown why this affects one side more heavily than the other. With this calibration the goal of higher accuracy than last year's team was achieved for every position. With more time it would have been possible to have further improved the accuracy by focusing on improving the edges of the frame.

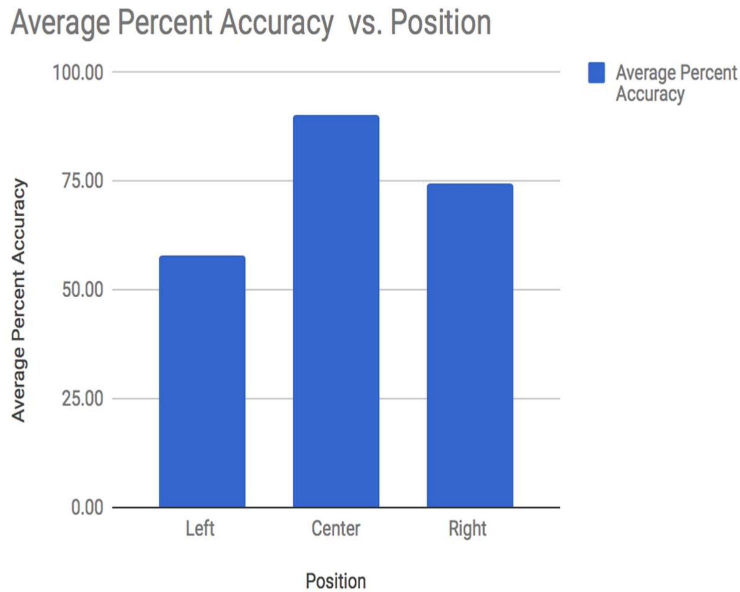

Figure 15. Average Percent Accuracy vs. Position

The next objective was the system performance with regards to the different distances. The results can be seen in Figure 16. As can be seen it has a high level of accuracy higher than 88% for distances less than 10 meters but it starts to decrease sharply after that. At 15 meters, this module only has an accuracy of 59%. The accuracy decreases at further distances because the layers of the LIDAR are further spread apart than they are at closer distances. At 5 meters it can typically have 8 layers overlayed on a person but at 15 meters this number can be reduced to 2 layers of LIDAR on objects the size of a truck. However, once again, the total average accuracy was higher than last year's for all distances. Since this calibration was able to beat the goal for different position and different distances we believe it is sufficient for this particular design.

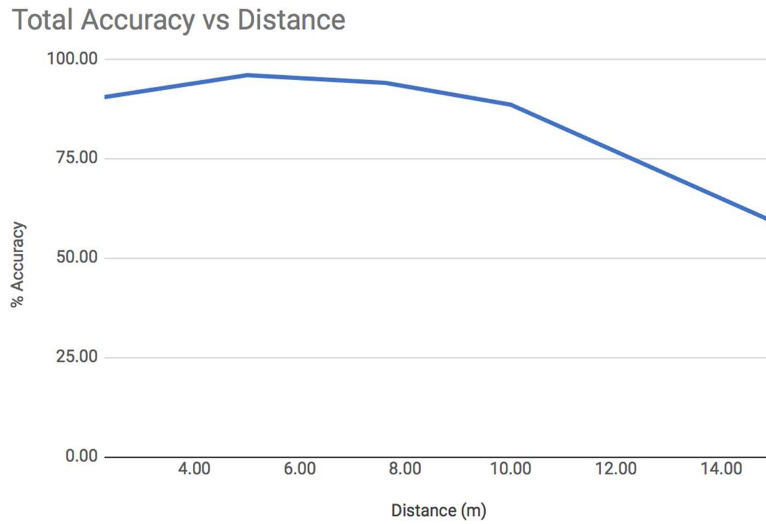

Figure 16. Total Accuracy vs. Distance

5.5 Kalman Filter

The first test performed on the Kalman filter was determining how accurately it was able to track a single object. This test used MATLAB to create a file storing unitless x and y values and a timestamp. Then, it used this file as the input to the Kalman filter. Next, plotted the results of the filter on the same graph as the one that contained the plotted input. As seen in Figure 17, the Kalman filter was able to accurately follow the input path. This shows that the Kalman filter is capable of tracking the system while creating its own prediction model. The Kalman filter is also able to reduce the noise in the system. This can be seen visually at the top and bottom of the graph where there are sharp turns and during the straight edge in between the turns.

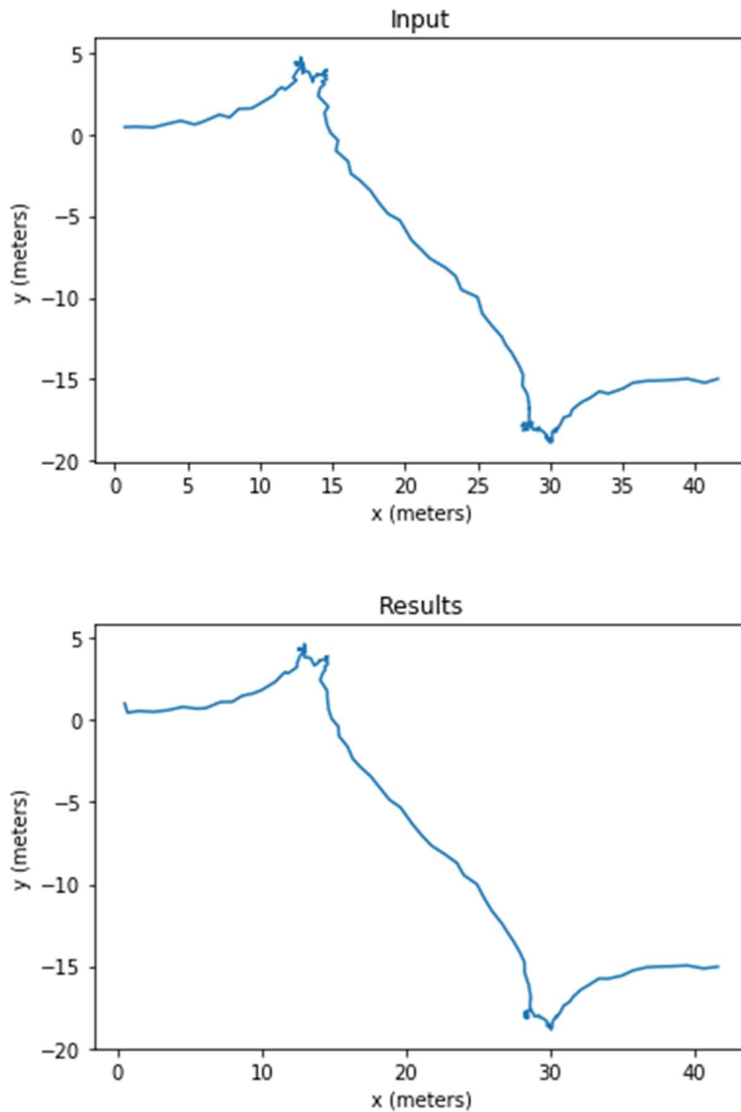

Figure 17. Comparing Inputs and Results for Kalman Filtering with 1 Object

Because Kalman filters use specific coefficients relating to each object it is necessary to create a Kalman filter for each object that the system intend on tracking. A class system was developed where each object it wished to track would have its own gain and matrices. Then ran the tests for the Kalman filter on running two different objects at the same time. The results of this test can be seen below in Figure 18. The results of this test were similar to those of the previous test on one object. For the same reasons the conclusions was that it was successful on one object and that the class-based system was successful on tracking multiple objects.

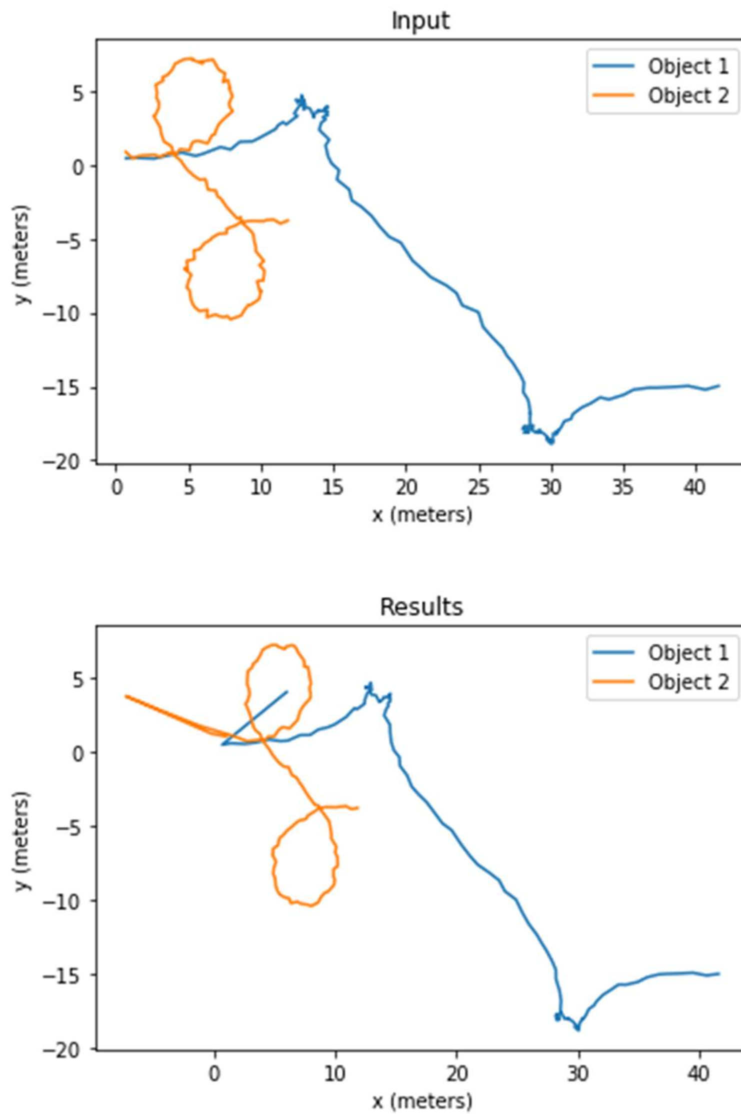

Figure 18. Comparing Inputs and Results for Kalman Filtering with 2 Objects

5.6 Collision Detection

In testing the collision detection module, vehicles and objects were visualized in bird's eye view using the shapely and descartes libraries in Python as shown in Figure 19 below. The dark red box represents the owner´s vehicle, the dark blue box is an object, and the lighter red and blue boxes are their paths, respectively.

```
***COLLISION DETECTED***
```

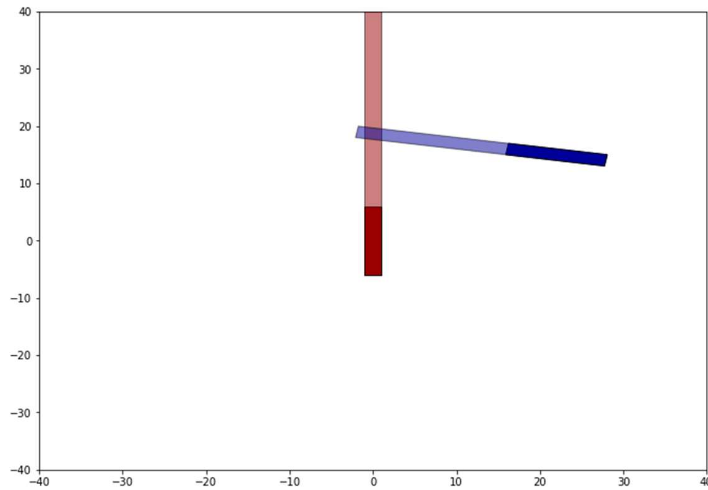

Figure 19. Visualizing Objects and Collisions

The paths in Figure 19 intersect, and the module printed a "***COLLISION DETECTED***" to the console. In a similar manner, testing was conducted with all possible combinations of the following factors: 0 or 1 objects; collision or no collision; and velocities of 0, 1, and an arbitrary number greater than 1.

Another consideration was timing analysis (determining the amount of time this module requires to run, as it scales with the number of objects). After much testing, (the testing methodology is specified in the "Total Time" section below), it was discovered that the module averaged 3.9 milliseconds per object.

5.7 Overall Time

Because processing power is limited, and the system is time sensitive (it is crucial that the user is alerted of a possible collision as soon as possible), timing analysis were performed on all the constituent modules of the system. These time values were calculated as an average of multiple trials while using the laptop that was used in the final system in the car. Since the CPU is typically downclocked when running on battery power, as the laptop would typically be in the car, these tests were conducted with the laptop unplugged.

The vast majority of the processing time is spent in receiving the LIDAR packets. This operation takes up about 118 of the approximately 124 milliseconds required for the entire pipeline. The data syncing and LCT operations together take less than 2 ms, and the collision detection takes slightly less than 4 ms. Timing details for each module are given in the Table 9 below.

Table 9. Time Spent per Module

Module	**Time (milliseconds)**
LIDAR packets	118.14
Sensor Data Syncing	0.14
LIDAR-Camera Transform	1.57
Kalman Filter	0.23
Collision Detection	3.90
Total	**123.98**

Through this time testing, it was confirmed that the system is able to operate in real-time and meet the processing time requirements. The goal was a processing time of less than 250 ms per frame, and the system is well under that, processing frames about twice as fast as required.

5.8 System Testing

To test the overall system, it was necessary a setup similar to the one used to test the object locator module, which consisted of locations 5, 10, and 15 meters from the car in the right, middle, and left of the image. A diagram of this testing setup was shown in Figure 14. An entire suite of tests was performed with a person in the image, signifying pedestrian tests, and another suite of tests with a car, signifying vehicle tests. A successful test was one in which an alert was expected, and the system produced an alert one second before the collision would have taken place. A failed test was one in which the system either did not produce the expected alert or produced one less than one second in advance.

The pedestrian tests were done by using a person walking and running to the car from each of the 9 locations mentioned above. Those also included tests where a person walked sideways through

the frame, diagonally near but not to the car, and where a person ran away from the car. The vehicle tests involved driving a car towards the car with our system. They were analogous to the pedestrian tests but with the exception that these tests were not conducted at two different speeds. Although the second vehicle only drove in low speed out of safety concerns, it has been proved that the system will function as expected within the specified maximum speed of 45 mph.

One of the original goals was to alert the driver 3 seconds before an imminent collision. However, during preliminary testing it was discovered that predicting the movement of the other object 3 seconds in advance led to a substantial worsening of our accuracy and false positive rate. This was due to our assumption that the trajectory of the other vehicle is approximately linear. While this assumption holds for short intervals of time, it is not a reasonable assumption to make for a period of 3 seconds. In that time, a vehicle can speed up, slow down, or change direction substantially. When we tested with the system predicting 3 seconds in advance, on many occasions it predicted an accident even when the driver was slowing down. This deceleration was not captured by our Kalman filter. Faced with this accuracy-time tradeoff, the decision was to focus on accuracy as it is a more important metric for this particular project, especially as the system is currently facing the front of the car. For that reason, the desired warning time was reduced to 1 second.

The results of the tests are summarized in Table 10 below. Our system's 33.3% false positive rate narrowly missed our target of 30%. However, it was found that it was very accurate in predicting actual collisions, with a true positive rate of 90% so the safety concerns of the system were fulfilled.

Table 10. Collision Detection Test Summary

True Positives	**False Positives**
90%	33.3%
False Negatives	**True Negatives**
10%	66.7%

6.0 TIME AND COST CONSIDERATIONS

The result of this project was a working system to track one object relative to a user´s car and predict possible collisions with it within the assigned time and budget. The only expense of it was a Garmin modified GPS which was used to track the speed of the user´s car. However, due to the limited time and budget it was not possible complete some desired features.

First, it was necessary to change its path in late February because of the previously described problems with Apollo due to not having compatible hardware that was above budget constraints. Apollo requires hardware that is very expensive, with the 64-layer LIDAR alone costing up to $85,000. Despite losing time trying to use Apollo with the previously owned lower cost sensors, it was possible to create the described system within 2 months. However, it was not possible to add some previously desired features that Apollo had within this time frame. Multiple object tracking for example, is not available. The intention was to make a New vs Old subsystem to differentiate between objects that were previously detected and the ones that were detected for the first time, but the lack of time to implement it made it necessary to reject that possibility. The LIDAR camera calibration was also not completely finished, as the left and right side of the camera view was less accurate than the center. Another intention that could not be done because of the time constraints was to improve the calibration accuracy.

7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN

With a collision detection system, one of main concerns is reducing injury due to vehicles. The system was designed with a focus on addressing human errors. One way this safety concern was addressed is for the system to process data faster than humans can. This system processes data in about half the time a human can react, so it was a satisfactory improvement. While it does not have perfect accuracy on the overall system, there is a sufficiently high level of accuracy in the system for research purposes. Overall, the that LIDAR is capable of working in many types of weather conditions. Furthermore, the YOLO object detection system works in night conditions, which addresses two main concerns about the usability of this design.

For the alert system itself, the focus was the accuracy of the alert, such as predicting an alert correctly. Accuracy was prioritized rather than giving the driver more than 3 seconds, with the opinion that this is the more ethical choice because giving false positives or false negatives could be more harmful to the driver. This design also detects crashes with both pedestrians and cars. It does not discriminate or prioritize either, and detects both in the same fashion.

A major safety concern when working on this project was how to perform testing. It was necessary to actually be able to test whether the collision prediction system worked, but also to prioritize safety and not put anyone in any potentially dangerous situations. As a result, the system was only tested it in a very controlled environment. This was beneficial for the sake of safety, but it also meant that the system was not tested in any real-life scenarios, like on an actual road. Because of this, if the system were to be used in the future a lot more testing would need to be done in different scenarios. As of now, the system is only usable for research purposes where testing variables can be controlled, and it would need to be improved and further tested before being exposed to real-world driving scenarios.

8.0 RECOMMENDATIONS

Future improvements in the system involve upgrading the hardware, tuning current subsystems, and adding modules for multiple object tracking. For the sensor hardware, the camera resolution was sufficient, but it used only one camera in the forward direction. In the future, more cameras could be added in order to provide a 360-degree field of view. Another potential improvement is to upgrade the LIDAR. The current 16-layer LIDAR is not very accurate at long distances, and with a higher resolution LIDAR the system would be more accurate at longer distances.

Furthermore, the LIDAR-camera calibration could be improved. Although trying a variety of tools, software, and outside resources, it could not achieve a very accurate calibration. This calibration was the main point of inaccuracy in the system, especially when used on objects near the edges of the camera. With more time the calibration could be made more accurate which would improve objectively.

An important addition is to make the track for multiple objects. Currently this system works for one object, so the recommendation is to use the YOLO classification of the object as well as the bounding box to track multiple objects. Finally, future projects could improve the buffer time

between collision detection and alert. It was necessary to decide during the project between improving the detection accuracy or advancing detection time. The decision was to focus on collision detection accuracy, and that meant it could alert the driver only one second in advance. Furue groups can improve on this while not sacrificing accuracy.

9.0 CONCLUSION

Overall, this project was successful in creating a collision prediction system which was able to detect collisions and alert the driver before the collision occurred. The system achieved an object detection accuracy of 61.6%, which met the specification of improving upon last year's accuracy, which was 46.39%. Additionally, it met out processing time requirement of 250 ms, as the system operated at an average of 124 ms per frame. Finally, it was discovered that the system was very accurate in predicting impending collisions, as it achieved a 90% true positive rate. As a result of the project, a conclusion was that it is possible to create a collision warning system with lower-cost sensors than those used in the majority of existing systems.

Although being able to create a functioning collision detection system, it did not meet all of the original requirements. The original goal was to use Apollo, but due to hardware incompatibility and the complexity of the codebase, it was necessary to eventually abandon Apollo and instead create another system using many individual modules. This project also missed the desired false positive rate by 3.3%, achieving a rate of 33.3% instead of the 30% from the specifications. Additionally, this system only alerted the driver one second before the collision occurred instead of three seconds due to a dramatic decrease in accuracy when attempting to provide three seconds of warning. Lastly, it was originally wanted for the system to be able to track multiple objects, but ended up only being able to track one at a time because of the complexity of identifying and tracking old and new objects. Although not meeting all of the original specifications, it is obvious that with more time this system could be improved and exceed the requirements.

There are many improvements that could be made to the system, but one that would have the most immediate impact is LIDAR-camera calibration accuracy, as that was a major point of inaccuracy for our system. With a better calibration, the system would be better at determining object distances and velocities regardless of where in the frame the object is. Other potential

improvements include the ability to track multiple objects and increasing the alert time while still maintaining a high level of accuracy. Lastly, there are hardware improvements that could be made as well - by either adding more sensors or using higher resolution sensors, the system could achieve an even higher accuracy. There is a lot of room for improvement in the system, and this proejct can serve as a strong base for future research in the field of autonomous driving.

REFERENCES

[1] "Critical Reasons for Crashes Investigated in the National Motor Vehicle Crash Causation Survey", Crashstats.nhtsa.dot.gov, 2015. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115. [Accessed: 6- Oct-2017].

[2] T. DePalatis et al, "Object Detection Using Sensor Fusion on Vehicles", University of Texas at Austin. [Accessed: 26- Mar -2018].

[3] "Experiment: How Fast Your Brain Reacts to Stimuli", Backyardbrains.com, 2018. [Online]. Available: https://backyardbrains.com/experiments/reactiontime. [Accessed: 3-May- 2018].

[4] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *arXiv: 1612.08242.* [Online]. Available: https://arxiv.org/pdf/1612.08242.pdf. [Accessed: 22- Feb- 2018].

[5] J.Redmon, "YOLO: Real-Time Object Detection," 2016 [Online]. Available: https://pjreddie.com/darknet/yolo/. [Accessed: 22- Feb- 2018]

[6] OpenCV, "Camera Calibration and 3D Reconstruction," *Camera Calibration and 3D Reconstruction - OpenCV 2.4.13.6 documentation*. [Online]. Available: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html. [Accessed: 22- Feb- 2018].

[7] M. Vernier, "ros-camera-lidar-calibration," *GitHub*. 12-Dec-2016. [Online]. Available: https://github.com/swyphcosmo/ros-camera-lidar-calibration.

[8] Udacity and Mercedes, "Self Driving Car ND - Sensor Fusion - Extended Kalman Filters ," tech.

[9] Zhou, Y. and Tuzel, O. (2018). VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection. [online] Arxiv.org. Available at: https://arxiv.org/abs/1711.06396v1 [Accessed: 4- May- 2018].

[10] Y. Shao, "Object Tracking using EKF by Fusing Lidar and Radar data," *GitHub*. [Online]. Available: https://github.com/ymshao/Object-Tracking-using-EKF-by-Fusing-Lidar-and-Radar.

APPENDIX A – Object Locator Distance Test

Position	Measured distance (m)	Actual Distance (m)	Difference (m)	% Error	% Accuracy
Person Left	4.60	5.00	0.40	8.00	92.00
Person Left	4.88	10.00	5.12	51.20	48.80
Person Left	4.60	15.00	10.40	69.33	30.67
Person Center	5.35	5.00	0.35	7.00	93.00
Person Center	10.40	10.00	0.40	4.00	96.00
Person Center	15.90	15.00	0.90	6.00	94.00
Person Right	5.25	5.00	0.25	5.00	95.00
Person Right	10.40	10.00	0.40	4.00	96.00
Person Right	4.88	15.00	10.12	67.47	32.53
Car Left	5.23	5.00	0.23	4.60	95.40
Car Left	4.82	10.00	5.18	51.80	48.20
Car Left	4.83	15.00	10.17	67.80	32.20
Car Center	5.16	5.00	0.16	3.20	96.80
Car Center	12.50	10.00	2.50	25.00	75.00
Car Center	17.41	15.00	2.41	16.07	83.93
Car Right	5.64	15.00	9.36	62.40	37.60
Car Right	10.24	10.00	0.24	2.40	97.60
Car Right	5.56	5.00	0.56	11.20	88.80
Test Center	2.50	2.29	0.21	9.36	90.64
Test Center	8.06	7.62	0.44	5.77	94.23

Person				
Actual	Average Measured	Difference	% Error:	% Accuracy:
2.29	2.50	0.21	9.36	90.64
5.00	5.07	0.07	1.33	98.67
7.62	8.06	0.44	5.77	94.23
10.00	8.56	1.44	14.40	85.60
15.00	8.46	6.54	43.60	56.40

Car				
Actual distance	Average measured	Difference	% Error	% Accuracy
5.00	5.32	0.32	6.33	93.67
10.00	9.19	0.81	8.13	91.87
15.00	9.29	5.71	38.04	61.96

Both				
Actual:	Average:	Difference	% Error	% Accuracy
2.29	2.50	0.21	9.36	90.64
5.00	5.19	0.19	3.83	96.17
7.62	8.06	0.44	5.77	94.23
10.00	8.87	1.13	11.27	88.73
15.00	8.88	6.12	40.82	59.18

Position	Average Percent Error	Average Percent Accuracy
Left	42.12	57.88
Center	9.55	90.45
Right	25.41	74.59

APPENDIX B – Collision Detection Code

```
import shapely.geometry
import shapely.affinity
import matplotlib.pyplot as plt
from descartes import PolygonPatch
import math, random
# import winsound
import numpy as np
import os

delta_t = 3.0


#Initialize Kalman stuff
delta_t = 1
F_ = np.array([[1, 0, delta_t, 0],[0, 1, 0, delta_t],[0, 0, 1, 0],
               [0, 0, 0, 1]])
R_laser = np.eye(4)*0.0225
H_laser = np.array([[1,0,0,0],[0,1,0,0]])
#P_ = np.array([[1,0,0,0],[0,1,0,0],[0,0,1000,0],[0,0,0,1000]])
Q_ = np.array([[0.25*R_laser[0,0]*delta_t**4, 0,
                0.5*R_laser[0,0]*delta_t**3,0],
               [0,0.25*R_laser[1,1]*delta_t**4,
                0,0.5*R_laser[0,0]*delta_t**3],
               [0.5*R_laser[0,0]*delta_t**3,0,
                R_laser[0,0]*delta_t**2,0],
               [0, 0.5*R_laser[0,0]*delta_t**3,
                0,R_laser[1,1]*delta_t**2 ]])


#convert knots to meters per second
def kn_to_ms(kn):
    return kn*0.514444

#gets data from object tracking module (evan's thing)
def object_tracking(obj):
    obj.calc()
    return obj.x_

def update_F_and_Q(delta_t):
    F_[0,2] = delta_t
    Q_[1,3] = delta_t
```

```
#represents a 2D rotated rectangle, angle is from 0 to 360
class Rectangle:
    def __init__(self, center_x, center_y, width, length, angle):
        self.cx = center_x
        self.cy = center_y
        self.l = length
        self.w = width
        self.angle = angle #0 angle is front to the top
    def get_contour(self):
        w = self.w
        l = self.l
        c = shapely.geometry.box(-w/2.0, -l/2.0, w/2.0, l/2.0)
        rc = shapely.affinity.rotate(c, self.angle)
        return shapely.affinity.translate(rc, self.cx, self.cy)
    #returns intersect area between two rectangles
    def intersection(self, other):
        return self.get_contour().intersection(other.get_contour())
    #returns true if collision
    def collision(self, other):
        overlap = self.intersection(other)
        if (overlap.area > 0):
            return True
        else:
            return False

#extends rectangle
class Car(Rectangle):
    def __init__(self, center_x, center_y, width, length, angle):
        self.cx = center_x
        self.cy = center_y
        self.l = length
        self.w = width
        self.angle = angle
        #new variables
        self.cx_future = self.cx
        self.cy_future = self.cy
        self.speed = 0.0 #meters/second
        #Kalman variables
        self.x_ = np.array([0, 0, 1, 1])
        self.P_ = np.array([[1,0,0,0],[0,1,0,0],[0,0,1000,0],[0,0,0,1000]])
        self.results = np.empty([1,2])
```

```
    self.locarray = np.empty([1,2])
def update_locarray(self, locarray):
    self.locarray = locarray
def calc(self):
    #Predict
    self.x_ = F_.dot(self.x_)
    self.P_ = F_.dot(self.P_).dot(F_.T) + Q_
    # Kalman Update
    self.z = np.array([self.locarray[0],self.locarray[1],
                       self.locarray[2], self.locarray[3]])
    self.y = self.z-self.x_
    self.S = self.P_ + R_laser
    self.K = self.P_.dot(np.linalg.inv(self.S))
    self.x_ = self.x_ + self.K.dot(self.y)
    self.P_ = (np.eye(len(self.P_))-self.K).dot(self.P_)
    self.results = np.append(self.results, [self.x_[0:2]], axis = 0)
#adjusts cx_future and cy_future as where the center of
#car will be after delta_t time
def update_speed(self):
    self.speed = 0

    #self.speed = math.sqrt((self.x_[2]*self.x_[2])+
    #(self.x_[3]*self.x_[3]))
def update_for_box(self):
    hyp = self.speed*delta_t
    changeiny = hyp*math.sin(math.radians(90.0-(-1.0*self.angle)))
    changeinx = hyp*math.cos(math.radians(90.0-(-1.0*self.angle)))
    self.cy_future = self.cy+changeiny
    self.cx_future = self.cx+changeinx
    return hyp
#adjusts cx_future and cy_future as where the center of
#the car's path will be after delta_t time
def update_for_path(self):
    hyp = self.speed*delta_t
    changeiny = (hyp/2)*math.sin(math.radians(90.0-(-1.0*self.angle)))
    changeinx = (hyp/2)*math.cos(math.radians(90.0-(-1.0*self.angle)))
    self.cy_future = self.cy+changeiny
    self.cx_future = self.cx+changeinx
    return hyp
def get_contour_future(self):
    self.update_for_box()
```

```
        w = self.w
        l = self.l
        c = shapely.geometry.box(-w/2.0, -l/2.0, w/2.0, l/2.0)
        rc = shapely.affinity.rotate(c, self.angle)
        return shapely.affinity.translate(rc, self.cx_future, self.cy_future)
    def get_path(self):
        dist = self.update_for_path()
        w = self.w
        l = dist+self.l
        c = shapely.geometry.box(-w/2.0, -l/2.0, w/2.0, l/2.0)
        rc = shapely.affinity.rotate(c, self.angle)
        return shapely.affinity.translate(rc, self.cx_future, self.cy_future)
    def intersection_future(self, other):
        return self.get_path().intersection(other.get_path())
    def collision_future(self, other):
        overlap = self.intersection_future(other)
        if (overlap.area > 0):
            return True
        else:
            return False
    def update_object(self):
        array = object_tracking(self)
        self.cx = array[0]
        self.cy = array[1]
        vx = array[2]
        vy = array[3]

        if vx == 0 and vy == 0:
            angle = 0
        elif vy == 0:
            if (vx > 0):
                angle = -90.0
            elif (vx < 0):
                angle = 90.0
        elif vx == 0:
            if (vy > 0):
                angle = 0
            elif (vy < 0):
                angle = 180
        else:
            angle = math.degrees(math.atan(array[3]/array[2]))
```

```
            if vx>0 and vy>0: #quadrant 1 #pos/pos = pos
                angle = (90.0 - angle)*-1.0
            if vx<0 and vy>0: #quadrant 2 #pos/neg = neg
                angle = (90.0 + angle)
            if vx<0 and vy<0: #quadrant 3 #neg/neg = pos
                angle = angle+90
            if vx>0 and vy<0: #quadrant 4 #neg/pos = neg
                angle = (90 + angle) + 180
        self.angle = angle
        self.speed = abs(math.sqrt((vx*vx)+(vy*vy)))

#returns true if car and obj intersect
def collision_detection(car, obj):
    return car.collision_future(obj)

def alert_linx():
    duration = 1  # second
    freq = 440  # Hz
    os.system('play --no-show-progress --null --channels 1 synth %s sine %f'
              % (duration, freq))

def alert():
    print ("***CRASH DETECTED***")
    alert_linux()
    frequency = 600 # Set Frequency To 2500 Hertz
    duration = 750  # Set Duration To 1000 ms == 1 second
    winsound.Beep(frequency, duration)
```

APPENDIX C – Kalman Fitler Code

```
"""imports"""
import numpy as np
import matplotlib.pyplot as plt


"""Initialize all of the matrices that we will use"""
delta_t = 1
F_ = np.array([[1, 0, delta_t, 0],[0, 1, 0, delta_t],
               [0, 0, 1, 0], [0, 0, 0, 1]])
R_laser = np.eye(4)*0.0225
H_laser = np.array([[1,0,0,0],[0,1,0,0]])
#P_ = np.array([[1,0,0,0],[0,1,0,0],[0,0,1000,0],[0,0,0,1000]])
Q_ = np.array([[0.25*R_laser[0,0]*delta_t**4, 0,
                0.5*R_laser[0,0]*delta_t**3,0],
               [0,0.25*R_laser[1,1]*delta_t**4 ,0,
                0.5*R_laser[0,0]*delta_t**3],
               [0.5*R_laser[0,0]*delta_t**3,0,
                R_laser[0,0]*delta_t**2,0],
               [0, 0.5*R_laser[0,0]*delta_t**3, 0,
                R_laser[1,1]*delta_t**2 ]])

""" Kalman Filter Class """
class Kalman:
    def __init__(self, x, P, locarray):
        self.x_ = x
        self.P_ = P
        self.results = np.empty([1,2])
        self.locarray = locarray
    def update_locarray(self, locarray):
        self.locarray = locarray
    def calc(self):
        for n in range(1,len(self.locarray)):
            #Predict
            self.x_ = F_.dot(self.x_)
            self.P_ = F_.dot(self.P_).dot(F_.T) + Q_
            # Kalman Update
            self.z = np.array([self.locarray[n,0],self.locarray[n,1],
                               self.locarray[n,0]-self.locarray[n-1,0],
                               self.locarray[n,1]-self.locarray[n-1,1]])
            self.y = self.z-self.x_
            self.S = self.P_ + R_laser
```

```
            self.K = self.P_.dot(np.linalg.inv(self.S))
            self.x_ = self.x_ + self.K.dot(self.y)
            self.P_ = (np.eye(len(self.P_))-self.K).dot(self.P_)
            self.results = np.append(self.results, [self.x_[0:2]], axis = 0)
        plt.plot(self.results[:,0], self.results[:,1])
        plt.title("Results")
        plt.xlabel('x (meters)')
        plt.ylabel('y (meters)')
        plt.legend(['Object 1', 'Object 2'])


"""MAIN"""

#sample constructor data
P_sample = np.array([[1,0,0,0],[0,1,0,0],[0,0,1000,0],[0,0,0,1000]])
x_sample = np.array([0, 0, 1, 1])

"""GET LOCARRAY INPUT"""
locarray1 = np.empty([1,3])
with open("obj_pose-laser-radar-synthetic-ukf-input1.txt","r") as f:
    for line in f:
        location = line.lower()
        location = location.replace("\n","")
        location = location.split("\t")
        for n in range(len(location)):
           location[n] = float(location[n])
        locarray1 = np.append(locarray1, [location], axis = 0)
plt.plot(locarray1[1:,0],locarray1[1:,1])
plt.xlabel('x (meters)')
plt.ylabel('y (meters)')
#plt.title("Input")
#plt.show()

locarray2 = np.empty([1,3])
with open("obj_pose-laser-radar-synthetic-ukf-input2.txt","r") as f:
    for line in f:
        location = line.lower()
        location = location.replace("\n","")
        location = location.split("\t")
        for n in range(len(location)):
           location[n] = float(location[n])
```

```
            location[n] = float(location[n])
        locarray2 = np.append(locarray2, [location], axis = 0)
plt.plot(locarray2[1:,0],locarray2[1:,1])
plt.title("Input")
plt.legend(['Object 1', 'Object 2'])
plt.show()

sample_k = Kalman(x_sample, P_sample, locarray1)
#sample_k.update_locarray(locarray1)
sample_k.calc()

sample_k_2 = Kalman(x_sample, P_sample, locarray2)
#sample_k_2.update_locarray(locarray2)
sample_k_2.calc()
```