DATA ACQUISITION AND PROCESSING PIPELINE FOR E-SCOOTER TRACKING USING 3D LIDAR AND MULTI-CAMERA SETUP

A Thesis

Submitted to the Faculty

of

Purdue University

by

Siddhant S Betrabet

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Mechanical Engineering

December 2020

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMITTEE APPROVAL

Dr. Renran Tian, Co-Chair

Department of Computer Information and Graphics Technology

Dr. Likun Zhu, Co-Chair

Department of Mechanical and Energy Engineering

Dr. Sohel Anwar

Department of Mechanical and Energy Engineering

Approved by:

Dr. Jie Chien

Head of the Graduate Program

Dedicated to my family for their support and belief in me.

ACKNOWLEDGMENTS

I would first like to thank my thesis advisor Dr. Renran Tian of the CIGT department and Dr. Likun Zhu of the Mechanical Department. They were extremely helpful in guiding me through this thesis and helping me navigate through problems throughout this experience. I would also like to thank the students and professors involved in the Toyota project out of which this thesis emerged. I would specially like to thank Prof. Tian for being patient with my work and trusting me with the project and the thesis.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

PREFACE

As civilization continues to move towards autonomy, the need for data in order to train AI becomes ever more necessary. This drives furthermore the need for systems that can collect data to help further this technological advance. I have always been interested in projects involving integration of sensors and systems and this thesis is meant to be a representation of such a project involving the coming together of hardware and software. Thank you to my family, my professors and my colleagues in providing me the support to help me complete this year and a half long project.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

LIDAR	Light Detection and Ranging
IMU	Inertial Measurement Unit
MJPEG	Motion Joint Photographic Experts Group
FPS	Frames Per Second
UDP	User Datagram Protocol
ARM	Advanced RISC Machine
USB	Universal Serial Bus
DHCP	Dynamic Host Configuration Protocol
ROS	Robot Operating System
GPU	Graphics Processing Unit
UVC	USB Video Class
UI	User Interface
CSV	Comma Separated Values
L-M	Levenberg Marquadt
SVD	Single Value Decomposition
RANSAC	Random Sample Consensus
SolvePnP	Solve Perspective and Point
EPNP	Efficient Perspective and Point
SLAM	Simultaneous Localization and Mapping

ABSTRACT

Betrabet, Siddhant S. M.S.M.E., Purdue University, December 2020. Data Acquisition and Processing Pipeline for E-Scooter Tracking Using 3d Lidar and Multi-Camera Setup. Major Professors: Dr. Renran Tian and Dr. Likun Zhu.

Analyzing behaviors of objects on the road is a complex task that requires data from various sensors and their fusion to recreate movement of objects with a high degree of accuracy. A data collection and processing system are thus needed to track the objects accurately in order to make an accurate and clear map of the trajectories of objects relative to various coordinate frame(s) of interest in the map. Detection and tracking of moving objects (DATMO) and Simultaneous localization and mapping (SLAM) are the tasks that needs to be achieved in conjunction to create a clear map of the road comprising of the moving and static objects. These computational problems are commonly solved and used to aid scenario reconstruction for the objects of interest. The tracking of objects can be done in various ways, utilizing sensors such as monocular or stereo cameras, Light Detection and Ranging (LIDAR) sensors as well as Inertial Navigation systems (INS) systems. One relatively common method for solving DATMO and SLAM involves utilizing a 3D LIDAR with multiple monocular cameras in conjunction with an inertial measurement unit (IMU) allows for redundancies to maintain object classification and tracking with the help of sensor fusion in cases when sensor specific traditional algorithms prove to be ineffectual when individual sensor falls short due to their limitations. The usage of an IMU and sensor fusion methods relatively eliminates the need for having an expensive INS rig. Fusion of these sensors allows for more effectual tracking to utilize the maximum potential of each sensor while allowing for methods to increase perceptional accuracy. The focus of this thesis will be the dock-less e-scooter and the

primary goal will be to track its movements effectively and accurately with respect to cars on the road and the world. Since it is relatively more common to observe a car on the road than e-scooters, we propose a data collection system that can be built on top of an e-scooter and an offline processing pipeline that can be used to collect data in order to understand the behaviors of the e-scooters themselves. In this thesis, we plan to explore a data collection system involving a 3D LIDAR sensor in conjunction with an IMU and multiple monocular cameras on an e-scooter as well as an offline method for processing the data to generate data to aid tracking and scenario reconstruction.

1. INTRODUCTION

1.1 Motivation

Understanding the behavior of objects on the road to avoid fatalities is crucial to driverless systems. Injuries resulting on the road due to e-scooter accidents continue to mount year after year. Any form of Automated Driver Assistance System (ADAS) will need to build some form of internal model of the vehicles and the world to predict the possible movements of objects on the road once it has classified these objects. Each vehicle behaves differently as the rider tries and tends to use the vehicle depending on all the multitude of factors that go into dealing with driving scenarios. Many users in big cities who primarily use the rented version of the e-scooter do not tend to wear helmets. This fact coupled with their growing popularity in big cities as a cheap and easy to use means of transportation results in more and more injuries every year.

A breakdown of the various accidents recorded in a study [1] by the Center for Disease Control (CDC) shows the reasons for the accidents ranging from interactions with the curb to travelling downhill and a large number also being attributed to interacting with motor vehicles on the roads. E-scooter companies such as Bird and Lime operate in over 100 cities around the world. There has been an estimated $\tilde{4}500$ injuries in 2014 which skyrocketed to $\tilde{1}4000$ injuries in 2018 according to a study by the CDC, an increase of 222% in just 4 short years. In 2018 itself, Bird celebrated its 10 Millionth ride which underlines the need for any sort of safety system that deals with objects on the road to account for e-scooters as a potential cause for accidents.

Behavior of the riders as they make rough turns in order to navigate the curb and the pavement can often influence driver decisions as the drivers attempt to estimate possible interaction scenarios with the constantly unpredictable movement of these e-scooters and their riders. Therefore, to avoid accidents resulting in crashes from the

same, having a thorough understanding of the movement of the e-scooter is required. This requires accurately being able to track an e-scooter in multiple scenarios on the road as well as vehicles of interest to understand the different temporal and spatial relationships between these objects.

To achieve complete scenario reconstruction, the data collection and processing system needs to classify and track the moving objects of interest as well as static objects for a complete picture. As mentioned earlier, since it is relatively common to find cars in urban areas than to find e-scooters, a data collection system mounted on an e-scooter can be driven around urban areas to capture the necessary data. This involves the utilization of multiple perception sensors such as color monocular cameras and 3D LIDAR in conjunction with an IMU, a commonly used sensor combination [2]. The data collection system on such an e-scooter therefore needs an untethered computing platform and a mechanism for storing data. This involves being able to efficiently store high volume and high bandwidth data coming in from the cameras and the LIDAR sensor. The data collection system also needs to be able to compress the data to store it efficiently in its storage space as well as deal with data coming in at different frequencies and synchronize them accurately to aid in the processing.

The system also needs a lightweight frame to house all these parts. Once the data is stored, a data processing system is then required to create a custom pipeline that can then accurately process the data from these sensors. Since scenario reconstruction methods are not standardized, an array of diverse programs tuned for these specific sensors ad this setup is required. This involves being able to format the data and filter data from the IMU to maintain accuracy. The LIDAR data needs to be filtered and corrected for motion compensation to avoid mapping errors by fusing it with data from the IMU. The filtered data from these two sources then needs to be fused accurately to create a 2D map of the environment using an efficient SLAM procedure.

This will allow us to understand the motion of the e-scooter with respect to a fixed world coordinate frame. The data from the camera needs to be pre-processed and run classification algorithms and track the object of interest (the car). The camera data

then needs to be separately fused frame by frame and combined with the classification and tracking algorithms to track the ego motion of the car and its trajectory with respect to the e-scooter. The trajectories can then be simply inverted to understand the relative motion of the e-scooter with respect to the car. This data can then be incorporated into the previously created map of the static environment to achieve a complete scenario reconstruction that can be used to analyze driving behaviors.

The thesis thus primarily addresses these problems of such a system. We propose a data collection system implemented like the one used to create the KITTI benchmark dataset albeit tweaked for the constraints and requirements of being able to fit on an e-scooter as well as designs and scripts for a companion data processing pipeline. We utilized three high resolution CMOS Logitech c920 cameras, a high density 64 Beam 3D Ouster LIDAR with an inbuilt IMU. The data collection system consisted of an NVIDIA Jetson TX2 with the Orbitty carrier board, an external Solid-State Drive to store the data collected from the sensors. The system was battery powered and all the necessary hardware was mounted on the e-scooter itself.

We ran ROS and Gstreamer software on an Ubuntu OS running on the Jetson TX2 in order to record data coming from all the sensors. As we dealt with data from the sensors coming in at different frequencies with different timestamps, synchronization was key. Hence, the data was stored in .bag file format with image data being stored as PNG files which was a simple and effective means of storing and had a simple screen interface to initiate and track the process of the data collection.

The data processing pipeline can mainly be divided into three main areas. The first area dealt with pre-processing the bag files into suitable formats for further processing and sensor calibration. This involved finding the intrinsic and extrinsic calibration matrices to fuse the images and the LIDAR point cloud accurately. This area involved extracting the images from the bag file while maintaining logs concerning their timestamps in order to aid for synchronization process. The second area involved creating fusion maps to detection and tracking of moving objects. The data from the Camera-LIDAR fusion to aid with the DATMO problem. The third area dealt with

implementation SLAM algorithms are used to create static maps of the environment. We used the Google Cartographer program built on top of ROS to aid us with SLAM to generate 2D Map data as an additional output of the offline processing system.

1.1.1 Contribution

This thesis will concentrate on the design and development of a human wearable data collection system and an offline data processing pipeline designed for e-scooter based data collection. The system is designed to be of comparably low cost than other car-based designed systems currently developed primarily using integration of low-cost sensors. The thesis also attempts to fill in the gap existing due to lack of such data collection systems by designing a relatively cheap system geared towards development for e-scooter based data collection.

The thesis aims to provide a system that is human wearable and can be replaced relatively easily if the system breaks and can be up-scaled with less effort and risk due to the relative low cost. The thesis aims at limiting the use of proprietary software and aims at the use of number of open source software's never used in this specific combination with the aim to allow the quick use and easy replication of the system. It will focus specifically on data collection to aid scenario reconstruction of the events and accurate tracking of e-scooters with respect to moving coordinate frames such as on the cars on the road. The data collection system will aim at a design to fit the requirements and constraints of an e-scooter.

The design will be geared towards having features such as online synchronization(UNIX time-stamping) and compression of data coming from the monocular cameras, the LIDAR and the IMU to aid the processing stages. The thesis will detail the software and hardware design of the data collection system. The processing pipeline divided into four parts. The first part of the offline processing pipeline deals with the unpacking and formatting of the sensory LIDAR data and the synchronization data for further processing.

The second part deals with the calibration methods and programs employed for the same. This will primarily deal calibration of the intrinsic and extrinsic matrices will allow for accurate fusion mapping of camera data with the LIDAR point cloud. The third part will deal with the algorithms and implementation of the fusion process involving the camera and the LIDAR to generate fusion tables which can be used for tracking objects. Lastly, we evaluate the implementation of the Google Cartographer SLAM software with the generated LIDAR data to generate 2D and 3D maps of the static environment.

1.1.2 Outline

The thesis will be divided into two main subgroups. The first part will deal with the data collection system, the second area will deal with the data processing pipeline. In first chapter, we plan to discuss the design methods and considerations and the construction of the data collection system on the e-scooter. We then discuss the various computational hardware and software methods considered and implemented to efficiently collect data from the cameras, the 3D LIDAR and the IMU. It also deals with the techniques used to synchronize the data coming from each of the sensors as well as the UI developed for easy data collection.

In the second chapter we explore the methods utilized to pre-process the data to make it ready for calibration, processing, and analysis. This is followed by the explanation on the implementation of existing algorithms to perform calibration of the sensors for fusion. The third chapter will involve the implementation details of the processing(fusion) algorithms of the camera data with the LIDAR data on the collected data including details of code optimization. The final chapter will outline the evaluation of the system and implementation of the Cartographer software system with the collected ROS based LIDAR data and details regarding the generation of 2D maps and concatenated 3D point clouds.

2. LITERATURE REVIEW

2.1 Car Based Data Collection Systems

On road vehicle system equipped with road sensing capabilities have existed for decades. One of the earliest data collection systems that bears resemblance to modern benchmark collection systems is the NAVLAB1 system developed at Carnegie Mellon University [1] consisting of a camera and a laser range finder attached to a Chevrolet panel van. Fast forward to today, the two main recognizable systems, the Google StreetCar [2] developed in 2007 and the Bing car [3]. Both systems were developed to map the streets and structures around the street. These systems usually involve fish-eye cameras and wide-angle cameras enough to roughly cover a 360-degree view and usually two LIDARs arranged at right angles, horizontally and vertically to the surface of the street.

The Bing car similarly involved the use of a GPS integrated IMU along with a panoramic camera and a 3D LIDAR. These systems however are not intended as open source hardware and therefore not much data could be gleaned about their designs. The benchmark dataset collection systems for street and vehicle are well defined sensor arrays on top of the vehicle. The two prime examples of these are the KITTI and Waymo vehicle data collection systems. The KITTI dataset[4] also known as the KITTI vision benchmark suite provides raw benchmark datasets from their array of sensors arranged on a Volkswagen Passat includes an INS system a GPS/IMU system, 64 beam 3D LIDAR, multiple grayscale, color and varifocal cameras aimed at covering a full 360-degree horizontal view of the environment.

The Waymo system [5][6] differs in the aspects of the sensors There is a primary focus on the LIDAR data. The system consists of five different LIDARs, one mid-range LIDAR on the top and four short-range LIDAR's arranged on the sides. The

system also contains five cameras all primarily focused on overlapping 180-degree front view. This system hence utilized the space and size offered by a 4 wheeler and allowed for addition of multiple sensors to the system.

The downside to using a car-based system is to try to find vehicle e-scooter interactions is particularly difficult due the relative lack of e-scooters on the road. The e-scooter system presented in this document will have a much higher number of interactions with cars than a car-based system would have with an e-scooter. The hardware and consequently the software of these car-based systems also directly cannot be used directly on the e-scooter due to the constrains of size and power that need to be considered for an e-scooter based setup.

2.2 Systems Similar to the One Proposed

Although these two systems share their collected datasets for other developers and designers to download, relatively little is known of the actual software of these systems. Papers have also been published on producing datasets distinctly for autonomous automotive perception such as the Precise Synthetic Image and LiDAR (PreSIL) dataset by B. Hurl, K. Czarnecki and S. Waslander[7]. A few systems were found that were closer to the setup that we were trying to create specifically for the e-scooter system. One system by David Blankeau et al [8] aimed at development of a low-cost LIDAR system aimed to be used to collect data from bicycles. The aim was to use the 2D Garmin LIDAR lite in conjunction with a rotating stepper to generate 360-degree data along with a Raspicam camera for visual data.

The system mentioned by David Blankeau et al [8] was designed with simply cameras in mind and a high amount of processing with multiple cameras is required to glean motion information from the same and thus cannot be directly used as an answer to the problem proposed int his document. It must be noted that system proposed in this thesis is designed specifically to cater to the needs of data requirements from both LIDAR and camera sensors has not currently been designed for a system such as the

e-scooter that can collect and store data and also fit the constraints and requirements of an e-scooter based system. When it comes to the hardware software orientation of the system, the system by Bian He et al[9] uses a combination of FPGA's and Gigabit Ethernet hardware to allow for multi channel image collection. The usage of a image data loop acquisition algorithms to achieve this multi channel image acquisition. The data from the system is then sent over gigabit Ethernet to the host computer for storage and is built as a robust system with 0% packet loss for the image data.

Other systems involving FPGA's as a means of image acquisition exist such as the system outlined in the paper by Song Gu and Zhou Hulin[10]. The paper outlines a CCD data acquisition system and a VGA display module with a DMA control module to act as a low power consumption system. The system uses the Nios II interface board as the primary embedded system designed by the Intel Corporation.

Since time-stamping is an important procedure for any data collection system, systems dedicated to the accuracy of the incoming image frame have also been researched and designed. One such paper by S. Esquembri et. al[11] involves combining FPGA's wihth a synchronized time card and is based on the PXIE technology and essentially uses timing devices and hardware to allow for accurate hardware based time-stamping for the incoming image frames with the Precision Time Protocol developed by IEEE thus providing a hardware based solution.

Systems similar to the one employed by this thesis to allow for image acquisition that involve the use of the Linux Platforms have been implemented such as the system by Cheng Yao-Yu et. al[12]. The system exploits a S3C2410(ARMv9) embedded system running Linux to allow for image acquisition from USB cameras. The system touts the application of self composed drivers for allowing communication with the camera and the transplanting of a Linux system onto an embedded system.

Another similar system involves the application of the STM32F407 embedded system to connect directly to a OV2640 CMOS sensor along with the transfer of data over the Ethernet to a host computer in order to allow for data storage on an SD card has been explained in the paper by Yu Chai and Jike Xu[13]. The paper details the

use of DSP and development of the network to facilitate data acquisition and storage. This allows to store data over the network lowering memory requirements.

The specific Windows/ Linux run libraries and software are an integral tool in any data acquisition system. One such software the Gstreamer software has been widely used as a non-proprietary library/collection of features to enable image transport, collection and processing. The paper by G. Sundari et. al[14] outlines the application of the Gstreamer software to achieve h.264 encoding to allow for high quality image streams and achieve high compression ratios.

Apart from USB interfaces, the application if the MIPI camera setup is also widely utilized in data collection systems. A paper by Kyusam Lim et. al[15] details implementation of a MIPI-CSI multi lane configuration system that is meant to collect data at an astonishing 4Gb/s as an implementation for High Definition video applications.

Protocols that attempting to combine the applications of ARM system with the flexibility of the MIPI cameras have been designed such as in the paper by Utsav Kumar[16]. The paper details the usage of a novel i2c protocol to facilitate communication between the embedded system and the MIPI camera.

For battery powered systems, low power is often a given requirement, this developing camera systems that focus on low power consumption are also extremely useful. One such system is an architecture outlined in the paper by Yueh-Chuan et. al[17]. It is a multi-lane MIPI CSI-2 based camera system meant for low power consumption while being able to channel high bandwidth data of up to 4Gb/s while allowing for reductions in clock rates in the system.

A slightly more complex system meant for autonomous driving for Mobility scooters was designed by Liu et al [18]. The perception portion of the three wheeled mobility scooters involved a short range LIDAR couple with a mid-range Laser range finder connected to a 2-DOF servo system in conjunction with a ZED Stereo camera. The processing was done using a Jetson TX2 with the help of a custom built IOT communications module for relaying sensor and processing data. These systems have given the research an ample understanding of the techniques used to design a similar

wearable system for the e-scooter. The research mentioned in this thesis thus draws a lot from such similar projects in areas such as sensor selection and device integration.

A system that comes close to the kind of system that this thesis proposes is the Google Street View Trekker[19]. This system is designed to capture 3D Velodyne Lidar data and 360-degree camera data. The system is meant to be a wearable that records sensory data as the person navigates about collecting street view type data for Google Maps. This system however is also not meant for collecting specifically vehicle interactions and is rather designed for collecting static data such as land topography especially given the height at which the LIDAR and camera records the data with respect to the person wearing the system.

2.3 LIDAR Camera Sensory Data Processing

The use of LIDAR camera systems as means of spatial data collection is well documented. X. Liang et al. shows the application of a LIDAR camera system for Forest Data Collection [20] as well as for coastal mapping by B. Madore[21]. Quite a bit of research has gone into the processing of LIDAR and camera data with papers such as the use of Gstreamer for recording and processing camera data by L.Wang et al. [22] which primarily discusses the use of the PyGI and the Python language and G. Sundari et al. [14] which discusses the use of Gstreamer to compress large amounts of video data using H.264 compression while papers by L. Zheng and Y. Fan[23] focus on data packet decoding for LIDAR data. Processing of LIDAR data has also been a subject of interest with a lot of focus at often times being on feature extraction on LIDAR data. S. Gargoum and K. El-Basyouny's[24] paper, Y. Zhang et al. [25], X. Wang et al.[26] and R. Sahba et al.[27] on feature extraction from LIDAR data using Euclidian clustering, intensity clustering and other methods.

2.4 Sensor Fusion and System Calibration Techniques

Certain methods applied for calibration of stereo camera are also useful for finding the intrinsic matrices of the individual cameras, a paper by Chun Xie et. al[28] which utilizes a spatially coded structured light and a mobile camera in a handheld calibration setup in order to calibrate the cameras. These methods utilize a projector to project this spatially coded light in order to aid calibration. The paper claims results similar to the checkerboard based methods. Designed as a means for calibration to allow scenario reconstruction, it claims a deviation from the temporal coded structured light method more conventionally used. This applies similar to our method for using re-projection methods to fuse the LIDAR and the camera data, as the intrinsic and extrinsic matrices are involved in both methods. Once the initial projection matrix is calculated, it is then refined by decomposing the matrix into the individual intrinsic matrix and transformation parameters and then using non linear optimization.

Other methods for calculating the Projection matrix also involve deviations from the usual re-projection error minimization to instead steer towards a method using Linear Matrix inequality and converting the problem to that of a trace minimization problem such as in the paper by Yoshimichi Ito et. al[29].

When it comes to the task of finding the orientation of the camera with respect to a given co-ordinate frame, the paper by Yankun Lang et. al[30] describes an iterative method utilizing Kernel Density estimation and probability theory to find the alignment of the camera with respect to the ground. This paper can also be then used to find the complete extrinsic matrix between camera and LIDAR if the LIDAR's orientation to the ground is known before hand. Other calibration methods for cameras with wide angle(or close to wide angle lenses) and fish-eye lenses such as by J. Kannala and S.S Brandt[31] that use the similar method of finding the patterns of planar calibration and focus heavily on the geometry of the lenses for achieving results claimed to be comparable to the state of the art methods utilized.

An effort to turn towards more automated methods of calibration owing to the time and effort required for a setup where the calibration shifts due to for example, a changing of the lens or movement of the camera LIDAR or camera-camera setup. One such methods to automate the camera attempts to change the calibration method by utilizing a different template than the standard checkerboard and uses calibration points similar to the use of ARuCo markers now frequently used for calibration such as the paper by Weidong Song et. al[32] changing the traditional calibration with its automated aspect.

The utilization of the camera calibration toolbox in MATLAB has also been researched in detail as a means of calculating the extrinsic and intrinsic matrix has also been detailed such as in the paper by Azra Fetic et al. [33]. Research into the multi-modal camera calibration in order to transfer the projection matrix from one camera to a second camera in a multi camera setup has been published in a paper by Andrey Bushnevskiy et al. [34]

Since the widely used methods for camera calibration involve error L2 minimization problems, other methods involving the use of genetic algorithms as an add on to the widely used methods has been detailed in a paper by Yuanyuan Dong [35] and by Peng Liu [36] where the latter claims better calibration results than Tsai's Calibration algorithm/method with the genetic algorithm consisting of the fitness function and also allowing for crossover in order to achieve better precision. Methods for applying measurement adjustment theory as an add on to camera calibrations have been shown in a paper by Liyan Liu [37]

There have also been papers outlining methods for sensor fusion of camera and LIDAR data as well as calibration of the camera. The camera position estimation has also been outlined in the paper by D. H. Lee [38] utilizes SolvePnP and Kalman filter. The methods involving the camera calibration and the use of Levenberg-Marquadt algorithm (solvepnp) have been significantly used as a basis for calibration and sensor fusion between the cameras and the LIDAR in the methods described in this thesis.

Papers such as those by Y. Yenıaydin and K. W. Schmidt [39] and J. Li et. al [40] outline Camera and 2D LIDAR for lane detection purposes. Similar papers expanding the details and methods of fusion to 3D LIDAR and cameras exist such as those by Ankit Dhall et al. [41] and Velas, Martin et al. [42] utilizes the cost function method as well as the Levenberg Marquadt algorithm for finding the transformation between the camera and 3D LIDAR as well as between two 3D Lidars using ArUco markers. Papers on further processing of camera and LIDAR data for further processing of LIDAR and camera data using Markov chains is then outlined in the paper by A. Rangesh and M. M. Trivedi [43].

Methods involving calibration of camera and a 2D Laser Range Finder using a checkerboard target have been achieved and documented in the paper by PDF Fumio Itami[44]. This paper outlines attempting to reduce the error in the rotation matrix often observed in calibration by rotating the LIDAR slightly to improve the calibration along with a calibration method to measure the rotation of the LIDAR to improve the overall calibration. A paper showing how the camera and 3D LIDAR calibration can be achieved with the application of multiple V-shaped planes and using point correspondences instead of plane mapping has also been shown in a paper by Zhaozheng Hu [45] and involves calculating the rotation and translation matrices separately and involves calibration and testing in both simulation and real world.

Improvement of the method of calibration of 3D LIDAR and a monocular camera by means of pose estimation has been shown in the paper by Jiunn-Kai Huang and J. W. Grizzle[46]. His method involves the use of targets of known geometry to minimize errors in the LIDAR data as well as a cross-validation study to compare the edges of the target in the point cloud space with the camera images and estimates a 70% reduction in error from a baseline has also been documented in his paper.

Packaging the task of calibration of a Velodyne LIDAR and a Point grey camera into a MATLAB toolbox has been done with the documentation written in a paper by Yecheng Lyu[47]. The calibration is designed as an interactive toolbox which utilizes scanning the LIDAR data to find the vertices of a board to calculate the extrinsic

matrix and supports a pin-hole as well as a fish-eye camera model. This multiple model method helps for cameras with fish-eye lenses as the normal model does not work for these ultra-wide lenses due to the high distortions in these lenses.

More complex methods for arbitrary camera LIDAR optimization not involving specific targets and attempting to calibrate based on whatever the camera LIDAR setups views. This method, outlined in the paper by Bo Fu et. al[48]. This method involves moving the camera LIDAR setup and simply capturing camera images in order to attempt to reconstruct a 3D environment alone, and then using graph optimization to fit said reconstructed 3D environment with the LIDAR's 3D point cloud. The method is also outlined as a means to improve upon general methods of optimization such as methods involving checkerboards.

Other simpler methods attempt to calibrate the LIDAR and camera data online, by means of using image processing methods. The paper by Chih-Ming Hsu [49] outlines such a a method involving the application of edge filters to the image and utilizing inverse distance transforms to understand edge alignments and then compare the data with the edges in the LIDAR point cloud in order to calibrate the system.

Although a checkerboard is a common seen object when it comes to calibration, other methods involving using a colored sphere for calibration of multiple RGB cameras and a VLP-16 LIDAR have been outlined in the paper by Geun-Mo Lee et. al[50]. The implication involve being able to locate the center of the sphere only if a few data points are available as well and claims to have a re-projection error of 3 pixels in addition to the benefit given by quick and easy calibration with only a minute of processing required to perform the calibration and obtain the results.

Methods involving the use of PSO or Particle Swarm optimization to aid calibration of a Stereo camera and a LIDAR has been shown in the paper by Vijay John et. al[51]. This paper also aims at usage of arbitrary objects similar to the paper by Bo Fu[48]. However the applications are limited to the usage of a stereo camera or a multi-camera setup where the calibration between the two cameras is known.

The paper utilizes a novel cost function and the Viterbi based stereo disparity estimation as a means to eventually calculate the extrinsic matrix. The paper estimates an improved calibration due to the integration of the LIDAR range image with the Viterbi based disparity estimation to yield better calibration.

An interesting paper written by M.A. Zaiter [52] involves focus on finding the extrinsic matrix between the LIDAR and the ground. It attempts to do the same by combining all LIDAR frames into a single frame in order to find the position of the sensor and therefore the extrinsic matrix of the sensor w.r.t ground. Although the paper is not designed as a means to perform camera LIDAR calibration, it is nonetheless an important method that can be couple with other camera-ground based methods to achieve overall calibration of the sensors.

There have also been research involving calibrations for setups involving multiple LIDARs and camera mounts. This paper written by Radhika Ravi et. al[53] involves using feature matching between multiple cameras and 3D point clouds in order to calibrate all the sensors simultaneously. The proposed calibration feature extraction heavily and is meant to be a solution for terrestrial or aerial systems involving multiple sensors in an outdoor environment. The paper also aims at systems containing an INS component and aims to find the transformations between all the sensors and the INS system as well.

Fusion methods involving calibration similar to the one outlined in this paper are of interesting note such as the paper by Zoltan Pusztai and Levente Hajder[54] attempts to use ordinary boxes to calibrate a LIDAR and camera setup. The system primarily uses a combination of the RANSAC algorithm in conjunction with the Single Value Decomposition(SVD) algorithm and the Levenberg Marquadt algorithm to generate the extrinsic matrix for fusion. The system is the calibrated using the simulation of the entire calibration setup including a model of the VLP-16 LIDAR using the Blender simulation software.

3. DATA COLLECTION SYSTEM

In order to design a data collection system, we looked at some existing systems designed for benchmark data collection of this nature. We specifically looked at vehicles such as the vehicle used to create the KITTI dataset. These cars were equipped with 3D-LIDAR, grayscale and colored cameras, GPS systems and Inertial measurement units. We also looked at certain aerial (UAV) based data collection systems as well. Notably we tried to look at the sensors used in such systems. Since the requirements of the data collection was related to geo-spatial awareness like the ones used in autonomous cars, we also looked at certain self-driving car configurations and the sensors and on board data processing systems employed by the same.

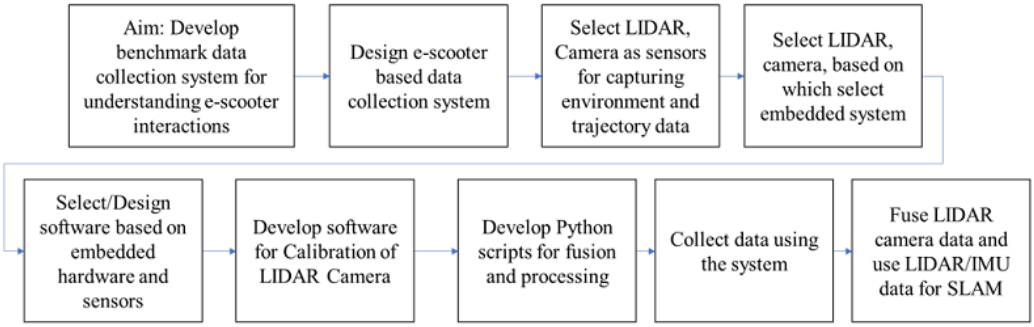


Fig. 3.1. Project Execution Stages

The system requirements stated the need for a system that could collect data for periods of times usually associated with e-scooter rides (2 hours maximum). Since the requirement was that of a wearable system, low weight and relatively ease of handling were important factors. A large unwieldy system would affect the rider and thus affect a naturalistic data collection compared to an easy to use, light system. The system

needed to be battery powered with the battery or any part of the system not being wired to the e-scooter for purposes of safety for the rider in the event of crashes. Being battery powered demanded a system based around relatively low power usage. A simple user interface although not absolutely essential was deemed as a requirement to make it easier to control the data recording process and troubleshoot the system if deemed necessary.

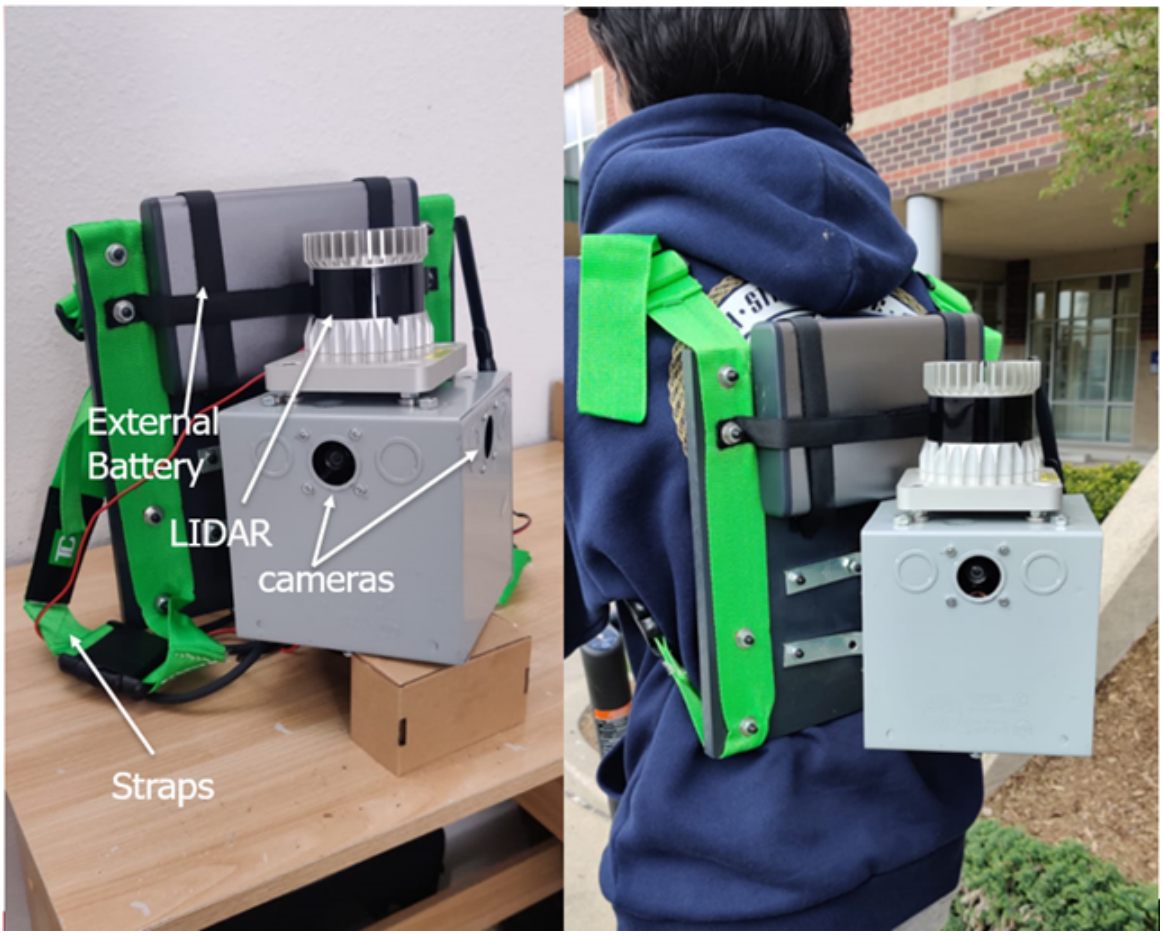


Fig. 3.2. The Data Acquisition System

3.1 Hardware System

There were a considerable number of constraints as well as requirements in developing a data collection system on an e-scooter. Primary considerations were weight and duration of operation of the system. Secondary considerations were ease of use of the system and a modular approach to allow for design changes. Since the e-scooter is meant to collect data about surrounding objects. The final system consisted of 2 USB cameras, a 3D LIDAR with integrated IMU as sensors with a Jetson TX2.

3.1.1 Camera Selection

There were several primary considerations for camera selection to be mounted on top of the e-scooter. Some of these were resolution, frames per second, shutter type, connector type, bandwidth requirements and color or grayscale. Since it was important that the presence of the data collection system should not affect the driver, the weight and volume of each individual part including the camera be taken into consideration and attempted to be minimized.

The three primary cameras selected were the FLIR Grasshoper 3, the Generic USB webcam with the Sony IMX291 sensor and the Logitech c920. It was also vital that the camera selection be compatible with the hardware and software of the data collection system onboard. Since we were planning on using an embedded system running Ubuntu ARM, we chose a camera that was compatible with this type of embedded device. With all these considerations in mind we decided to select the Logitech c920 webcam capable of delivering 1080p @30fps with h.264 or MJPEG format over USB3.0. The camera consumes about 150mA of current at 5V and is equipped with a 1/3" sensor. The camera is quite small and measure approximately 90 grams including the lens. The volumetric dimensions of the camera measure 38cmx38cmx30cm making it a small and portable camera fit for our application.

Resolution was an important factor in camera selection as resolution is one of the factors which determines the number of pixels that will record a given object at a

Fig. 3.3. Logitech c920 camera Source: Logitech website

given distance. We ultimately decided to go with a camera capable of delivering 1080p with a resolution of 1920x1080 pixels which was the active working area of the camera sensor as well. Here the FLIR grasshopper was better and could provide 2048x2048 resolution, and the Generic USB webcam camera could provide 1080p resolution.

The frame rate of the camera was also an important factor in the camera selection. The Ouster LIDAR had a low frame rate of 10 Hz and captures data by constantly rotating a sensor base that captures a radial section of the surroundings and does not capture the entire surroundings data at once. This is similar to the rolling shutter feature in the camera in which the camera does not record the entire image at once and instead takes out pixel rows one by one causing distortion effects.

It is important to note that although the other two cameras could also provide same or better frame rate, only the Grasshopper 3 and the Logitech c920 could provide and adjustable frame rate, where frame rate could be changed depending on the number of cameras. This allowed us to add more cameras on the Tx2 which was a significant hindrance in utilizing multiple cameras. Another reasoning for non-

selection of the Sony sensor based generic webcam was the bandwidth allocation. Generic UVC drivers for the camera allocated a bandwidth that was much higher than the actual observed bandwidth of the camera which was monitored using a Linux utility called USBtop. This meant the Tx2 would throttle the bandwidth for other devices notably the SDD and a second camera which prevented us from using multiple cameras i.e. the Generic USB Webcam's.

Since the LIDAR records an entire 360-degree surroundings in 64 chunks instead of a single image. Thus, if an image could be taken for every single chunk of LIDAR data, perfect synchronization of the two data streams could be achieved. The required frame rate of the camera would thus shoot up to 640 fps. Therefore, a camera with a higher frame rate would allow us to choose an image which was taken closest to the timestamps of the frame.

This would allow one to choose a LIDAR frame and camera frame closest to each other chronologically. Since an ideal frame rate of 640 fps is quite difficult to achieve with cameras that are both relatively inexpensive and not bulky, we decided to go with the highest possible frame rate which we could achieve with USB webcams i.e. 30 fps at 1080p, yet allow for multiple cameras to be attached. The selection of this frame rate would give us roughly 3 frames to select from for every individual LIDAR point cloud frame. The camera also provides 8-bit color data and although the processes of object detection and tracking don't need color and can work with gray-scale images, the presence of color on the camera does allow for a system equipped with the feature which can be exploited later for other applications such as color based tracking methods. One of the drawbacks of the camera is the rolling shutter.

As explained previously, the presence of a rolling shutter may result in the distortion of objects moving at high speeds depending on the direction of their travel. However, since the maximum relative speeds of the e-scooter w.r.t the surrounding and the relative distortion as a result of the same is quite low and does not seem to affect either the segmentation process or the LIDAR camera fusion to a degree that affects the scenario reconstruction. The lens selection discussion was primarily

centered around the field of view of the lens. The Generic USB webcam also included a rolling shutter but the FLIR Grasshopper 3 did provide a global shutter.

The camera came inbuilt with an cs-mount lens and a horizontal field of view of 78 degrees and a diagonal f.o.v of 90 degree. Since the aim was to cover at least 90 degrees at the front of the camera, with the angle centered around the longitudinal axis of the e-scooter and facing in the same direction as the rider. The compatibility on the software side was also a consideration in camera selection. The two primary methods to collect data on a Linux platform were the IEEE standard 1394 driver and the multimedia platform for Linux known as Gstreamer that utilized the v4l2 library. Since this camera used the H.264 as well as MJPG format, it was therefore Gstreamer compatible and we decided to implement software around the same.

It must be noted that while the Generic USB webcam did work with accelerated Gstreamer, but applying the Gstreamer functionality for the FLIR camera was difficult to achieve due to the drivers for the FLIR camera being not part of the Gstreamer library at the time of writing. In addition, workarounds using FLIR's Spinnaker Python API would have to be utilized to enable Gstreamer which would cause more CPU overhead defeating the purpose of using the hardware acceleration aspect of the Gstreamer API.

3.1.2 LIDAR Selection

The selection of the LIDAR had some considerations that were similar to the considerations we made for the camera. The main considerations were size and weight and hardware compatibility that were similar to the camera. The other parameters that were kept in mind for LIDAR selection were 2D vs 3D lidar, resolution, frame rate, sensor speed, number of generated points, hardware connectivity, range, accuracy, and cost. However, the lack of options in the market for a LIDAR relative to that compared to that of the camera meant severely limited options to select for the same. Certain other methods that incorporate the use of a 2D LIDAR that spins

in two planes to create a 3D point cloud was also considered but was rejected due to the added complexity of such a device both in terms of design, installation and processing. For this reason, a 3D LIDAR was chosen.

The other LIDAR's primarily considered were the Velodyne 64E, the Velodyne HDL 32E and the SICK MRS1000. The Velodyne HDL 64E was discarded as a selection due to its extremely bulky weight standing at 12.7Kg which would make an extremely heavy system not suitable for wearing. The Velodyne 32E and the SICK MRS1000 would however compare better weighing at approximately 1 Kg and 1.3 Kg still more than two-folds heavier than the OS1- 64 which measured at just about 455 grams favoring the Ouster LIDAR.

The consideration was selection between a 2D and a 3D LIDAR. Since 2D LIDAR's only capture data in the plane of the LIDAR sensor, the objects above and below the LIDAR do not come into view. In addition, on a constantly tilting system such as an e-scooter, the plane of recording data itself would constantly tilt and rotate. Thus, without some sort of stabilization mechanism, would have the effect of missing out on potential objects of interest. This was a primary reason for not selecting the SICK MRS1000 for our system.

Of the currently existing 3D LIDAR's the one we selected was the best in terms of functionality to cost. We therefore went with an Ouster OS-1 64 beam medium range LIDAR. The LIDAR is capable to generating 1,29,000 points at the rate of 10Hz. It has a horizontal Field of view of 360 degrees and a vertical field of view of 33.2 degrees divided evenly around the mid-plane of the sensor housing.

The LIDAR has a maximum range of 120 meters with an estimated accuracy of +/- 5cm(advertised) for Lambertian targets.In terms of accuracy however although the difference between the LIDAR's is very comparable being between +/- 2 cm for the Velodyne HDL 32 Beam LIDAR.

Although these two Velodyne LIDAR's have greater accuracy, there were other parameters in addition to the cost from the LIDAR's which are in favour of the selection of the Ouster OS1-64 3D LIDAR.

Vertical Accuracy of the Velodyne 32-E also fared worse than the Ouster solution, due to its significantly less horizontal rings(32 compared to 64) in a 3D volume at 1.25° compared to the 0.7° given by Ouster. This value significantly affected the LIDAR beams falling on the cars. Although the Velodyne 32 E LIDAR would get around this issue by distributing the beams closer to the center of the field, this would create issues for a system such as ours that tends to constantly pitch and roll depending on the orientation of the driver.

Although comparable with other LIDAR setups, the reason for selection of the Ouster LIDAR for this particular project is the vertical field of view. The Ouster OS1-64 LIDAR had a 44.5° which edged out over the Velodyne HDL 32E which had a vertical field of view of 40°. The LIDAR comes with a sensor interface that allows to transmit the LIDAR data over the Ethernet using the UDP protocol. The Sensor housing and the gigabit sensor interface operate at 24Volts and consume about 1.8 amps of current.

In terms of hardware functionality and software compatibility, the ability of the sensor interface to stream the data over the Ethernet over UDP allowed a host of different recording methods to be implemented. The implementation of the UDP protocol as a means of sending over the data also made the memory size requirements on the collection systems on-board computer slightly easier. Most notably the LIDAR came with a freely available ROS driver to capture, record and convert data and store with timestamps.

3.1.3 Embedded System Selection

Since volume weight and power consumption of the data processing on-board computer were primary considerations. We decided to go with an embedded system.

Fig. 3.4. Ouster OS-1 64 Beam LIDAR Source: Ouster Website

Fig. 3.5. Sensor Gigabit Interface Source: Ouster Hardware user guide

A single board computer, with appropriate storage capabilities to record data from the cameras and the LIDAR and store them efficiently. With these considerations in

mind, we decide to go with the NVIDIA Jetson Tx2 running an ARM based Ubuntu 16.04 as the primary on-board computer. The other development platforms primarily considered were the Raspberry Pi 4, the Radxa Rock 4 platform and the NVIDIA Jetson Nano/Tx1 platform.

Although the Jetson Tx2 development kit comes with a relatively big motherboard of size 170mm x 170mm x 2.5cm, the actual Jetson SOM(System on Module) measures just 50x 87 mm and can be connected to a Orbitty carrier board of the same size for a total volume of 87mm x 50mmx 50mm sames as the Jetson Tx1. This volume is comparable to the that of the Raspberry Pi 4 which along with the casing necessary for heat removal were about 97mm x 66mm x 41mm with the Rock 4 having similar dimensions as the Pi.

The CPU contains a Dual core NVIDIA Denver 2 in conjunction with an ARM cortex A57 and 8GB of LPDDR4 memory. This was a primary reason for selection of the Jetson Tx2 as the Pi 4 at the time of selection was only available in 1,2 and 4GB variants with the Rock having 2. Although it must be noted that 8GB variants are available at the time of writing. The Rock,Jetson Tx1 and the Nano also fell short in this area with 4GB of memory available. The amount of memory allowed for the ROS based LIDAR processes to use the RAM to act as a buffer space for incoming LIDAR data as the UDP packets were processed.

The carrier board among other things, comes equipped with a USB 3.0 port a Gigabit Ethernet port. Here the Pi 4 came close as it also has Gigabit Ethernet as the Jetson Tx2, with the Tx1 and Nano not providing this feature which would cause considerable packet loss as the LIDAR connection expects Gigabit Ethernet for smooth functioning. The Tx2 and Tx1 platforms also allow for a SATA port that allows us to access an extra PCI-e lane that allows for smoother storage to the hard drive. The Rock 4 and Pi 4 does not have this feature but does have the USB 3.0 same as all the other platforms.

The Jetson Tx2 also allowed for the installation of Ubuntu 16.04 which is widely used as the Operating system to install ROS system. Here the Tx1 and Nano compare

as they can also have Ubuntu installations. The ROS installation on the Raspberry Pi 4 is trickier than other due to the usual operating system on the Raspberry Pi4 being the Debian OS. Another reason for the not selecting the Pi 4 and Rock 4 here is also the relative lack of online community support for the Ubuntu Mate system on the Pi4 and Rock 4 compared to its Debian Linux variant.

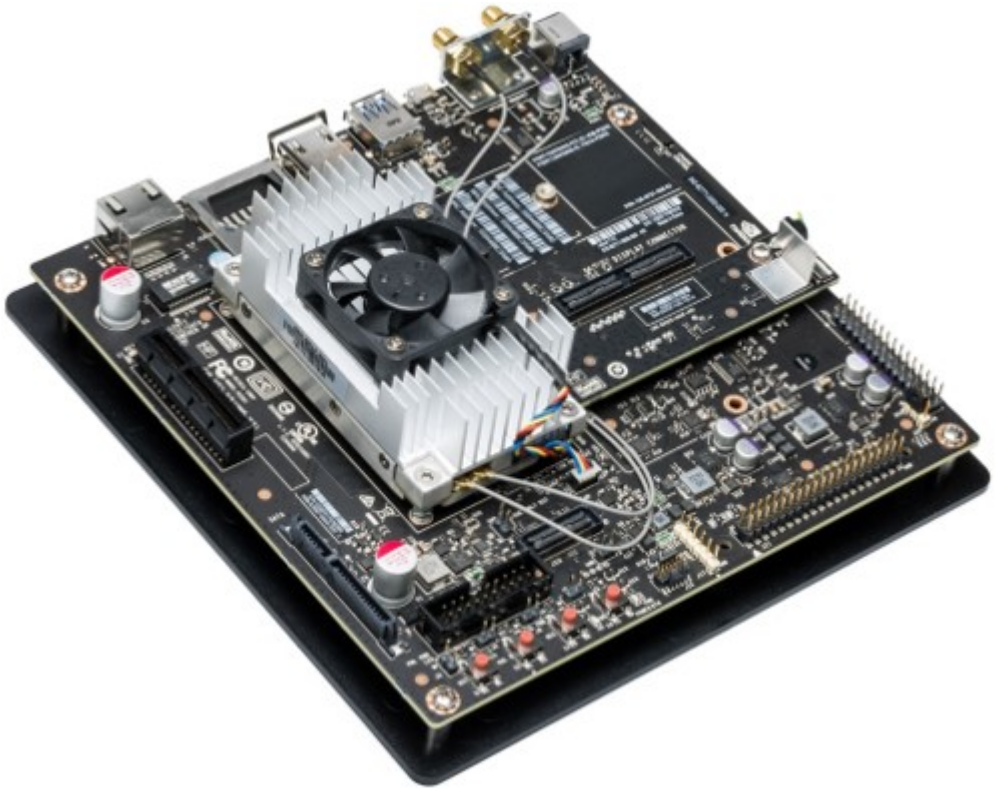


Fig. 3.6. Nvidia Jetson Tx2 Development Board with SOC

Most notably the NVIDIA board comes equipped with methods for hardware accelerated JPEG conversion and video encoding methods. Both of which can be utilized to store the data coming in from the cameras. The presence of a USB3.0 also allows for higher bandwidth and allows for the addition of more cameras. The presence of an ARM based system also allows for less power consumption compared to that of a traditional 32 or 64 bit system. The system operates at an advertised 19 volts at maximum consumption of 4.74 amps. Although the system only comes with

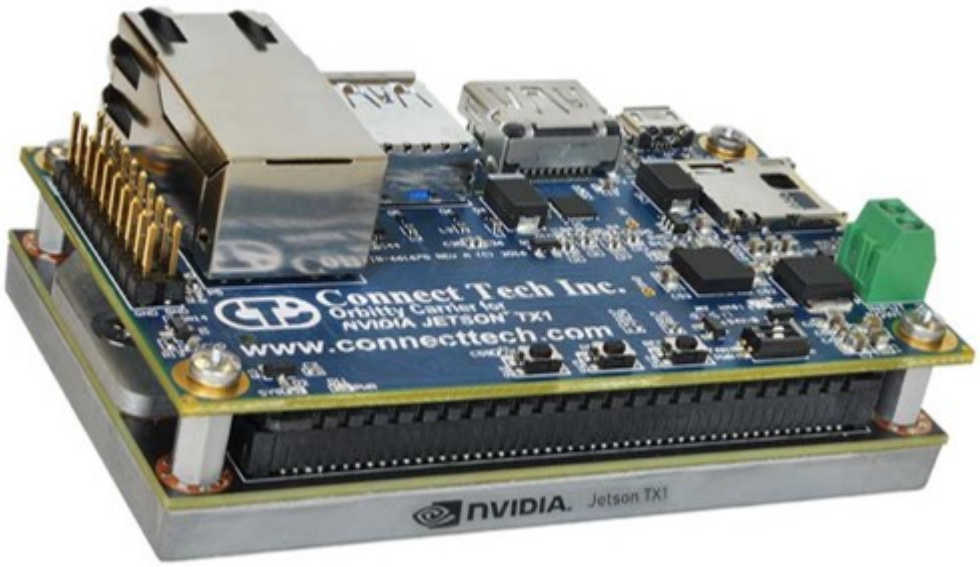


Fig. 3.7. Nvidia Jetson Tx2 System on Chip Source with the Carrier Board

one USB 3.0 port, this was expanded with the use of USB hubs. This advantage of the NVIDIA platform was also considered compared to the Raspberry Pi 4 platform.

3.1.4 Battery and Supporting Electronics Selection

In order to run the data collection system wirelessly on-board, we decided to use batteries to power the whole system. Since the major components that required power were the Jetson Tx2 board and the Ouster LIDAR.

The total consumption is about 34.6 watts. The value for the Jetson Tx2 is the maximum consumption as provided by the manufacturer's datasheet. The current consumption for the OS-1-64 sensor is relatively fixed and does not waiver depend on conditions. The current consumption for the cameras is about 200mA for each camera and about 400mA for two cameras at 5V. The current consumption for the SSD is about 1.6 A at maximum. That coupled with the requirements for the USB hubs internal circuitry as well is thus adds up to about 2A of consumption. For a standard external laptop battery, which is 185 W-h, the time of discharge is calculated

Table 3.1.
Average Measured Power Consumption for the Data Acquisition System (using carrier board)

Sr. No	Component	Voltage (V)	Current (A)	Power (W)
1.	Jetson TX2(carrier Board)	11.92	0.148	1.764
2.	OS1- 64 w sensor interface	23.62	0.749	17.691
3.	USB hub	4.986	1.3	6.482
3.	Logitech c920	4.986	0.35	1.745
4.	Logitech c920	4.986	0.35	1.745
5.	SSD	4.986	0.942	4.697
6.	Buck Boost Converter(Tx2)	20.1	0.004	0.08
7.	Buck Boost Converter(LIDAR)	20.1	0.04	0.804
	Total consumption			35.008

by dividing the total watt-hours possible divided by the max wattage required by the sum of the individual components giving us a rough idea of the working hours for this battery-based setup.

The battery was chosen with these requirements in mind and a standard Laptop external charger was deemed as a good option due to its pre-existing multiple voltage level selections and the higher than required energy capacity. The battery was capable of delivering 185W-h or 50,000mAh, with the ability to give out a maximum of 130Watts. The battery, consisting of 3.7V Li-Ion cells in parallel, had internal current limiting circuitry allowing for over-current and under-current protection. The Power bank could supply voltage of 20V (or12 V) and 5V with the latter being provided as a USB socket. Weighing at 2.7lb, the system could go from 0 percent charge to full charge in under 6 hours.

$$\textit{Time of discharge} = 185\ \textit{W-h} / 35\ W = \ 5.285\ \textit{hours} \tag{3.1}$$

In order to provide the required currents and voltages to each of the individual components, a DC-DC converter, was used to provide the appropriate amount of voltage to each component. Since the Jetson Tx2 needs about 1.5 amps of current(carrier board version), a buck-boost converter with a maximum current draw of 3 Amps was chosen. Two similar buck-boost converters were used to power 20V and 5V to the Ouster Sensor and the USB hub respectively.

Both versions of the board could be used for the system, if the carrier board was utilized the only difference would be that the SSD would be connected to the USB port via a SATA to USB cable. If the Development board was utilized then a SATA SSD could be connected directly to the SATA port on the Development Board as shown in the circuit diagram. Although the TX2 required a higher voltage of 19V compared to the carrier board, the power requirement actually drawn was similar in terms of wattage with the total power draw being roughly 20.5 watts with 1.07 Amps being measure on a multi-meter. The only change to the system required the Jetson Tx2 development board being placed inside a shoulder carry bag due to its greater volume than the Carrier Board. The antenna that sits outside the housing on the outside of the housing of the Tx2 helps the signal connection between the smartphone and the TX2 to be a strong connection.

The usage of a DHCP server allowed for the elimination of a separate router or Ethernet switch and allowed the Sensor Interface box to be directly connected to the Ethernet Port of the Tx2. The process to set up the DHCP server was usually done at the start of each reboot cycle with the help of bash script and by including the command to run the script in the bashrc file in Ubuntu Linux which ran a few seconds at the start after every reboot.

The DHCP server setup primarily involved flushing the IP addresses on the Ethernet interface, followed by the assigning of a static IP address to the interface. This would be followed by the switching on said interface, running the dnsmasq program(which actually started a DHCP server) and allowing devices physically connected to the Ethernet jack to get assigned IP addresses and establish connection in

order to allow the transfer of the LIDAR data. This procedure however required the sensor interface box to remain switched off till the Ethernet interface is switched on.

In order to achieve this, the GPIO on the Tx2 was allowed to toggle the connection to the Sensor Interface/ Ethernet Adapter via an SPDT relay and a NPN transistor. The GPIO pin 38 on the J21 header of the Tx2 was connected to the base of the 2N3904 transistor with a 10K current limiting resistor. The emitter of the NPN transistor was connected to the ground of the USB hub's power supply while the collector was connected to one of the coil pins of the SPDT relay.

The other coil pin of the SPDT relay was connected to the positive rail of the Hub's power supply. A freewheeling diode 1N4001 was connected in reverse bias parallel to the SPDT relay. The positive rail of the incoming power supply was spliced and connected to the NO and COM pins of the SPDT relay. Using the sysfs interface on the Tx2, the GPIO could be toggled, allowing to switch the LIDAR on and off. The command line command for the switch on of the LIDAR was baked into the DHCP server startup script and could be re-initiated via the UI if necessary.

3.2 Software System

3.2.1 Software Recording Process

The software recording process is initiated by pressing buttons on the UI. The recording process involves the UI server initiating a simple DHCP server to start the connection of the LIDAR to the Tx2. This allows for bypassing the router and enabling direct connection between these two devices. The UI also initiates the ROS Lidar driver in order to make the LIDAR to start sending data to the Tx2. The UI can be used to also initiate the Gstreamer recording as the synchronization software.

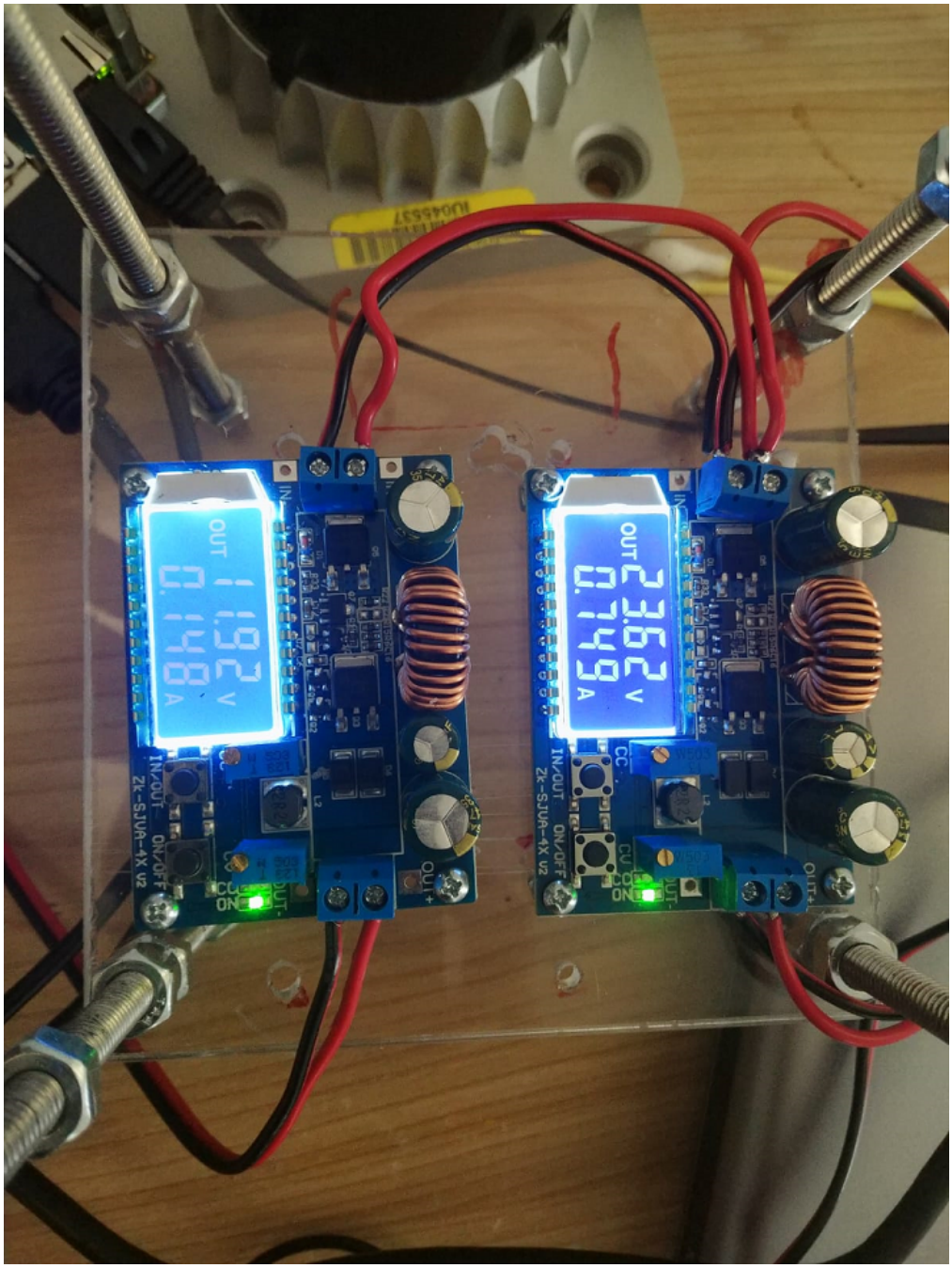


Fig. 3.8. The DC-DC Buck Boost converter displays showing the voltages being supplied to the LIDAR and the Carrier Board. The system is housed inside the metallic box on top of which the LIDAR is fitted.

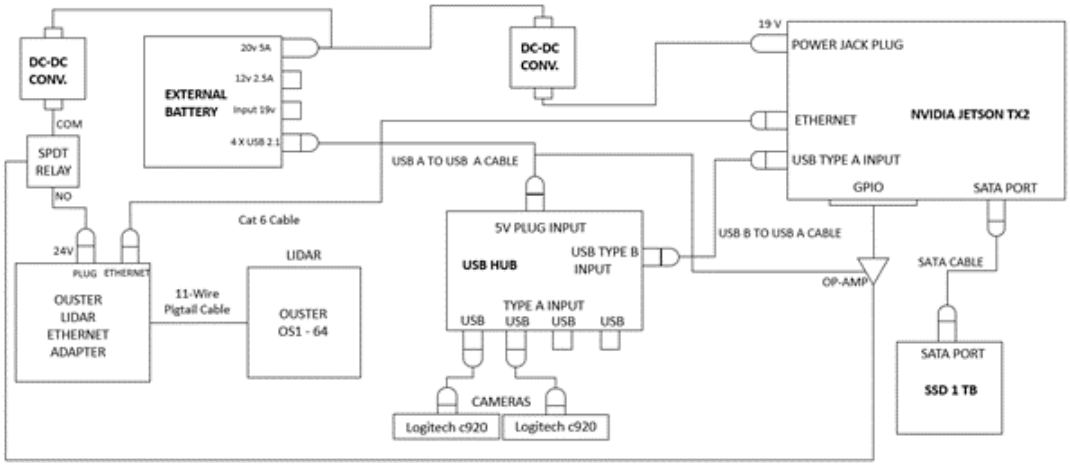


Fig. 3.9. Collection System Architecture

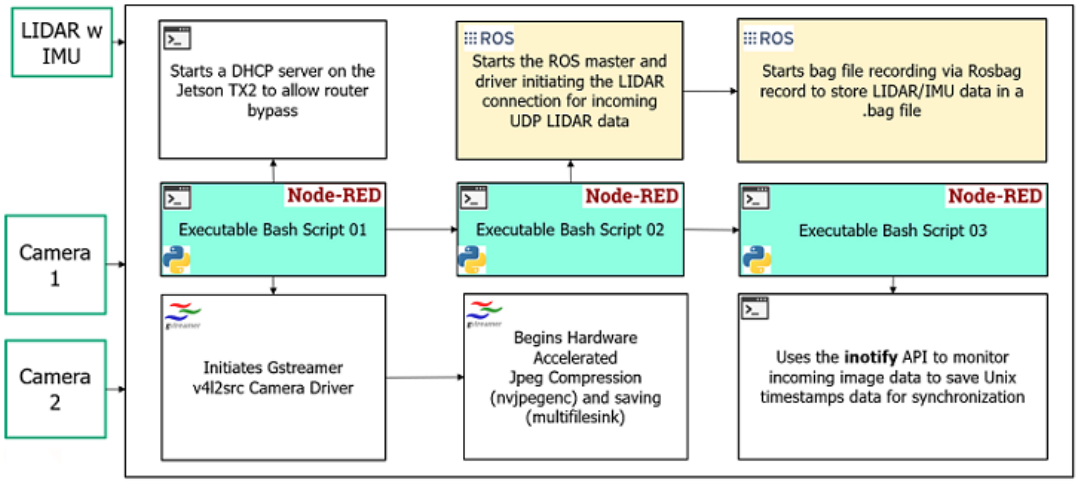


Fig. 3.10. Data Collection Software Execution Flow

3.2.2 LIDAR Data Collection

ROS for Ouster LIDAR

For the Ouster LIDAR there are a couple of ways to record the data. Since the data is available over UDP many different library implementations can be used to initiate data transfer and the record the incoming UDP packets. One efficient way of

recording the data is via the use of ROS. Ouster provides a maintained package for ROS variants, Kinetic and upwards. The ROS driver requires the sensor be connected over the Ethernet with security setting on the host Ubuntu computer and the network router to allow for UDP data transfer over the local network.

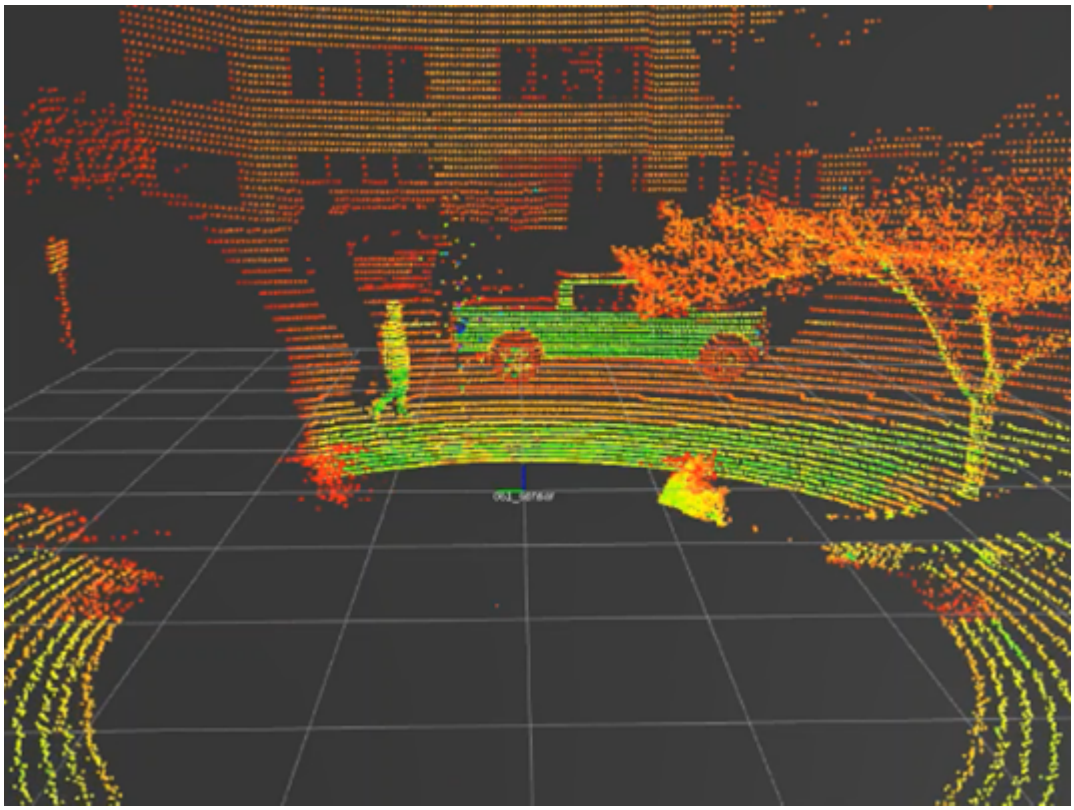


Fig. 3.11. Rosbag LIDAR data visualization using Rviz

The driver was downloaded from its Git repository and cloned into a local folder using the git clone command. The workspace was then built using the 'catkin_make' command. The driver was then simply launched using the "roslaunch ouster_ros os1.launch rviz:=false lidar_mode:=2048x10" command.

The launch file would then start the driver which would publish the available topics. One of the primary reasons for selecting the ROS driver for recording the data is the abstraction ROS allows in terms of writing code to record data. Since all matters of initiating connection, recording the data and synchronization of recording

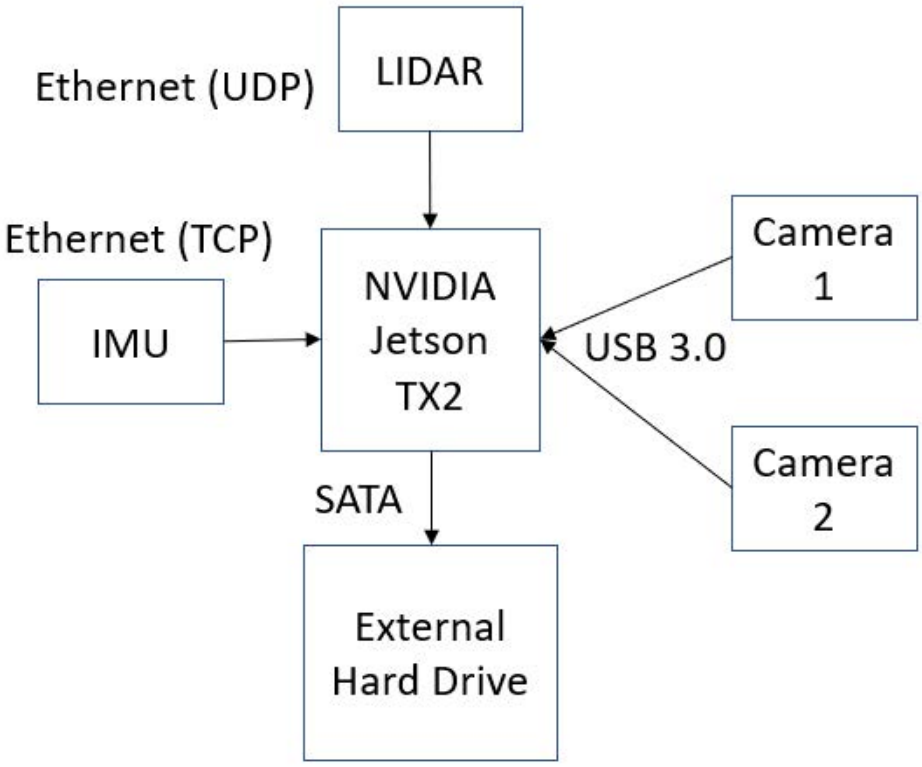


Fig. 3.12. Basic Data flow for the system

is handles by ROS, the complexity is greatly reduced and requires one to only write a launch file to initiate the driver. The driver also efficiently packages the UDP data from the LIDAR as well as the IMU data into the ROS bag to be opened as required. The same driver also allows for conversion between the UDP data to range values. This range values data can then be stored inside another bag file that can then be used for SLAM in post processing. Once the driver is initiated, the rosbag record command line API is used to initiate the actual recording of data from the LIDAR.

This new bag file(s) containing the range values can directly used with the Google Cartographer software with some additional filtering. The recording can be initiated on the Ubuntu command line itself or initiated via a bash script or the Python3 using the OS library.The command looking something like “rosbag record - -udp -b 0 - -split - -size=1024 /os1_node/lidar_packets”. The - -udp flag helps in optimizing

data storage for UDP type data, the -b 0 flag signifies a buffer/RAM memory of how much ever is available from the system for the purposes of recording. The - -split --size=1024 flag divides the recording of the bag file into individual chunks of roughly 1024MB to avoid the formation of large files that would make processing difficult.

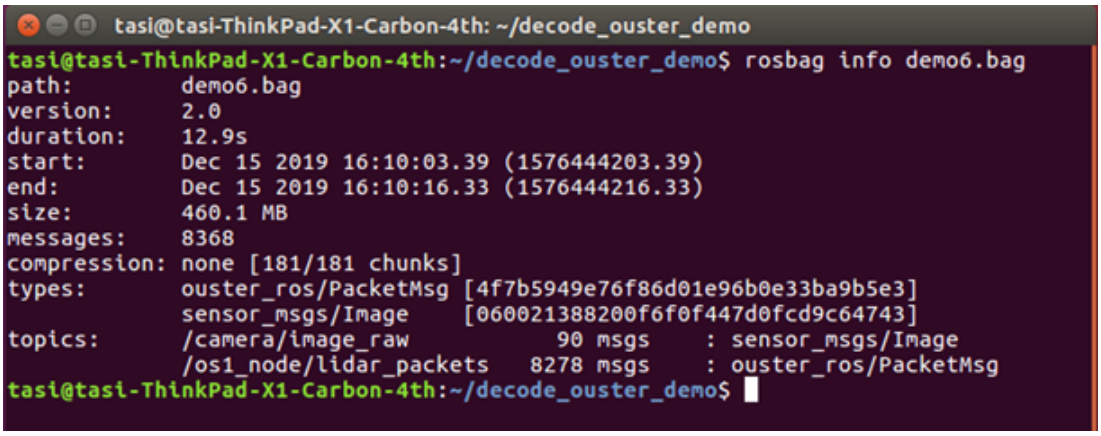


Fig. 3.13. Bag file info displayed using the rosbag command line API

This thus allows for efficient transport and storage of recording and storage due to the data being in the nature of UDP packets and allows for the computational overhead being pushed further into the post processing phase while reducing the memory overhead for both the on-board computer in case of the car and the NVIDIA Jetson TX2 in case of the e-scooter. The ROS based method also uses UNIX timestamps which allows use to synchronize the data with the camera data even if the camera data is recorded using a different method such as the Gstreamer pipeline as it also uses the UNIX timestamp to synchronize the stored images.

3.2.3 Camera Data Collection System

Gstreamer pipeline for e-scooter Web camera

This Gstreamer based method for the web camera was developed for a system consisting of a single web camera that could run USB 3.0 at around 30 FPS. Gstreamer

is a multimedia library designed with drivers and video and audio signal processing features. It also features a command line API as well as plugins for OpenCV and Python. Gstreamer was primarily considered as a means of software for the data collection system due to some of the hardware acceleration-based methods available on the Jetson TX2.

The software allows for the construction of "graphs" of video and audio processing elements. Each of these elements make up a graph that is also referred to as a pipeline. The logic of "sinks" and "sources" is used in Gstreamer where a source is considered as a source of incoming data or signals typically video or audio. A sink is usually referred to as a form of output of the pipeline.

The method was developed for storing either a video of the incoming camera stream or to store frame by frame compressed JPEG images. The method of usage of Gstreamer for our system was to source the data from the UVC web camera using a driver such as the "v4l2src" element in conjunction with a "videoconvert" to convert the data between required formats and then finally to a filesink element. The hardware accelerated features of the TX2, namely the "nvvidconv" and "nvjpegenc" allowed for reducing CPU and memory load.

In all the methods described below the driver always used is the v4l2src Linux driver for Ubuntu 16.04 capabilities for the. The v4l2src element is thus supplied with its main argument i.e the device path. This refers to the path of the webcam in the /dev directory in the root filesystem and for the Jetson TX2 connected on the Orbitty carrier board always refers to the /dev/video0 file.

Unless accessed on the NVIDIA Jetson TX2 development kit, the /dev/video0 file always refers to the default MIPI webcam. A simple Python script could also be used in conjunction with the a Linux library such as evdev to match the details of the camera. A similar Gstreamer pipeline with /dev/video1 argument in the v4l2src element was used for the second camera.

The software-based video recording pipeline utilizes the x264enc to enable conversion of raw video into h.264 compressed data. The major upside of this method as

compared to raw video saving is substantial as the raw video has a bit-rate close to 10 times larger than the compressed image. The downside to the method however is the latency generated is often observed at higher frame rates and higher resolutions.

To mitigate this effect the Jetson TX2 offers a hardware accelerated video encoding block. This block allows for x264 encoding. This element was used together with "nvvidconv", a proprietary NVIDIA plugin for Gstreamer that allows for conversion to other NV12 formats. The method is thus used the nvvidconv element in place of the default videoconvert element in the Gstreamer and similarly the x264enc is replaced by NVIDIA's omxh264enc element.

The software-based image recording pipeline still uses the default v4l2 driver for UVC webcams. In this method the element primarily is composed of the v4l2src source. This is followed by "videoconvert" element that allows the conversion of the video data into the required video formats as requested by the next element. This element is the "jpegenc" element that sources JPEG images.

The quality property of the "jpegenc" element can be set from a value of 0 to 100. This property greatly alters the size of the final saved image(25KB - 1.3MB). The difference between the size of a default saved image thus affects the total number of hours for which data can be saved by the data collection system which is stored using the multifilesink element which stores the formatted JPEG's in the required directory in the External SSD or in the memory.

The hardware accelerated image recording pipeline simply replaces the videoconvert and jpegenc elements with the proprietary Gstreamer plugins on nvvidconv and nvjpegenc. The important hardware acceleration being provided by the nvjpegenc block. The quality property can also be set for the nvjpegenc block as well and generates the same sizes of images as shown in the figure 3.15.

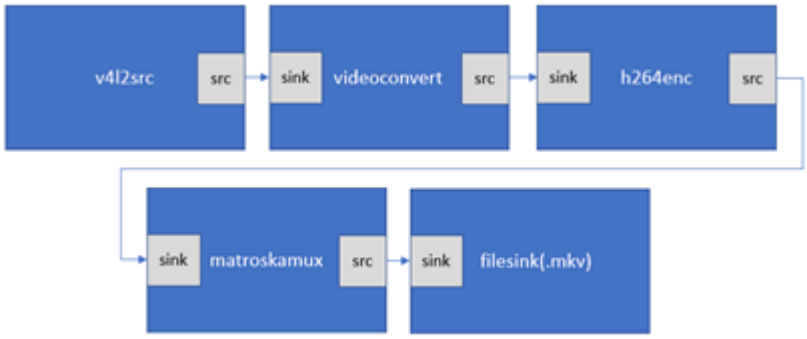


Fig. 3.14. G-streamer Non Hardware Accelerated video recording pipeline

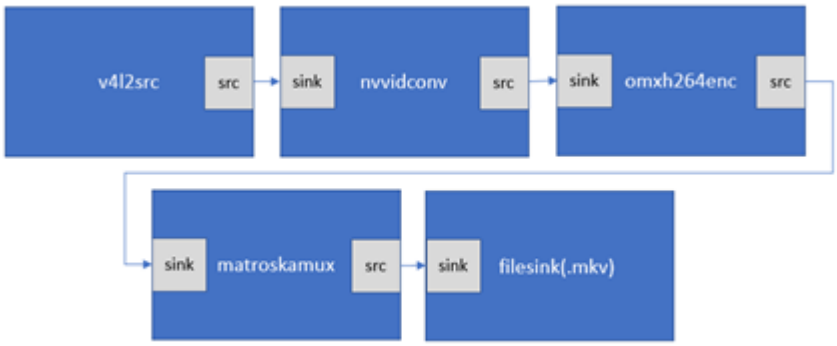


Fig. 3.15. G-Streamer Hardware accelerated video recording pipeline

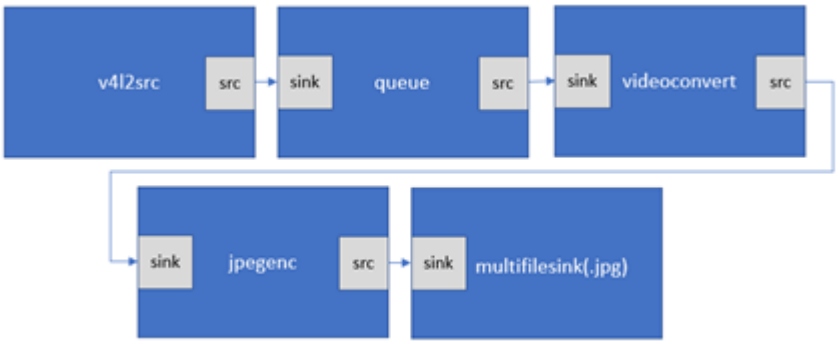


Fig. 3.16. Gstreamer Non Hardware accelerated frame storing pipeline

ROS for Webcam camera

At the time of writing, there exists an available ROS driver for cameras compatible with the Gstreamer interface. The driver basically attaches itself to the end of the

Gstreamer pipeline and takes in raw RGB-x images. The driver then publishes out topics such as camera/image_raw and camera/image_raw/compressed consisting of the standard ROS standard messages for compressed and uncompressed messages. The messages then can be simply recorded using the "rosbag record" command. This method although easy to implement with a working driver was initially investigated as but dropped for several reasons.

The chief reasons among this being that the frame rate on the rosbag based recording depends upon available CPU power. Since the NVIDIA Jetson TX2 still uses a smaller CPU than most standard desktops, the frame rate dropped considerably as a result compared to the Gstreamer method. For a single camera connected on the saving raw images the frame rate dropped to about 11.3 fps from a possible 30fps. This is primarily due to the ROS driver having to handle, transport and compress images on the CPU. This method thus eliminated the benefits hardware acceleration provided by the NVIDIA Gstreamer proprietary compression techniques and the pure Gstreamer method was implemented.

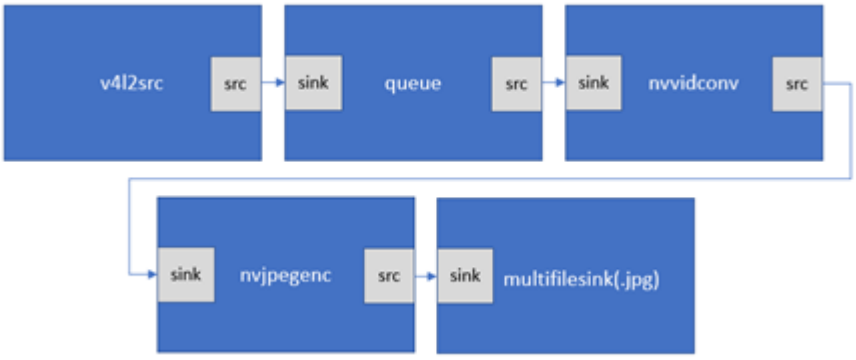


Fig. 3.17. Gstreamer Hardware accelerated frame storing pipeline

3.2.4 Synchronization

In order to achieve synchronization, i.e a timestamping of the camera data in a manner similar to that of the LIDAR data, a Linux program by the name of inotify-

wait was used. The inotifywait can accurately create a flag when a certain file enters a folder and retrieve the name of the said file as output. The logic of the program was to use the inotifywait to monitor the folder which would be continuously filled with images each with a distinct name as the gstreamer program generated the compressed images. This method was implemented via a bash file that would trigger the inotifywait to monitor the folder for changes and then as inotifywait returned the filename, the bash script would then record the current Unix timestamp and then append the name of the inotifywait file with the Unix timestamp value into an empty text file. Once the collection process was done this file could then be used as reference to find out the timestamp of a given image name.

3.2.5 User Interface

A simple but robust User Interface was also developed to aid the process of data collection. The primary aim of the interface was to allow the e-scooter rider control over the data collection process. The UI system was developed to be accessible via a mobile device in the vicinity of the Tx2 system. This allowed the user to open a browser-based interface on his smartphone to allow him to start and stop the data collection process. The implementation of the data collection system was done via the use of two softwares running on the Tx2.

The first one is the Node-Red programming tool that provides the UI, the second is a Python script that uses certain ROS and os libraries to begin the required ROS/Linux processes. The two programs are connected via a simple local websocket implementation. The Node Red server runs uses the Node-JS, a Javascript runtime system. Node-Red was primarily selected over other languages due to it primarily being a browser-based UI that allowed for the creation of a quick and easy dashboard that can be quickly viewed on a browser window on the local network.

This eliminates the need to write server-side programs on the TX2 which would be required to serve the necessary UI webpages and allowed for faster testing and

implementation. The additional benefit of using Node-Red is the use of websockets allows for the connection of node-red over a virtualized TCP to the Python programs that can be used to launch and handle the ROS elements. As mentioned earlier, any changes to the Python code or the adding of any features to the system can be quickly reflected in the UI due to its Simulink like approach to designing code and simplicity in designing UI interfaces.

Node-Red was also selected over Python due to its simplicity when it comes to designing quick UI's and also to provide a Simulink like structure to the code of the UI as Node-Red uses Flows similar to the of Simulink wiring and block diagram that allows for quick changes to be made to the system.

Node-Red Server

The software that primarily served as the back-end for the UI was the Node-Red server / programming tool. This programming tool enables IOT type applications and was able to run JavaScript functions(built on NodeJS) along with many other features. One such feature was the Node-Red dashboard library. This library allowed for the creation of UI that could be accessed within the local network. The system would act as a server that allowed one to open the UI page through a browser on a device on the local network.

The pressing of a button on the UI page creates an event that is sent back to the node-red server. The system simultaneously also runs a web-socket server. This server allows the Python client on the other end of the connection to wait for data signals to be sent over to initiate the ROS/Windows processes. Here the flows are wired to either trigger the processes directly, via the exec node provided natively by Node-Red. The software is designed to send over a distinct character over the websocket to the Python client waiting on the other end.

The exec node upon triggering, allows to start certain Linux processes, such as killing the recording processes of ROS and Gstreamer as well as inotifywait using

the “killall” Linux command. The reason for using the exec node to kill the linux process instead of doing it via the web-socket allows for redundancy in the event the Python program stops responding. Node-Red Dashboard. The UI can be accessed via a smartphone browser on the same network. For ease of use, the Tx2 connects to a hotspot running on the users phone.

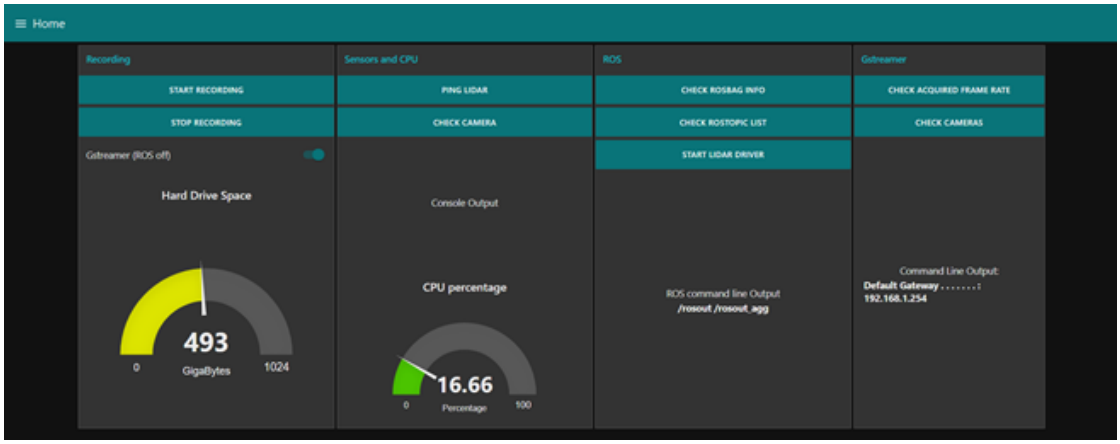


Fig. 3.18. Node-Red UI Dashboard

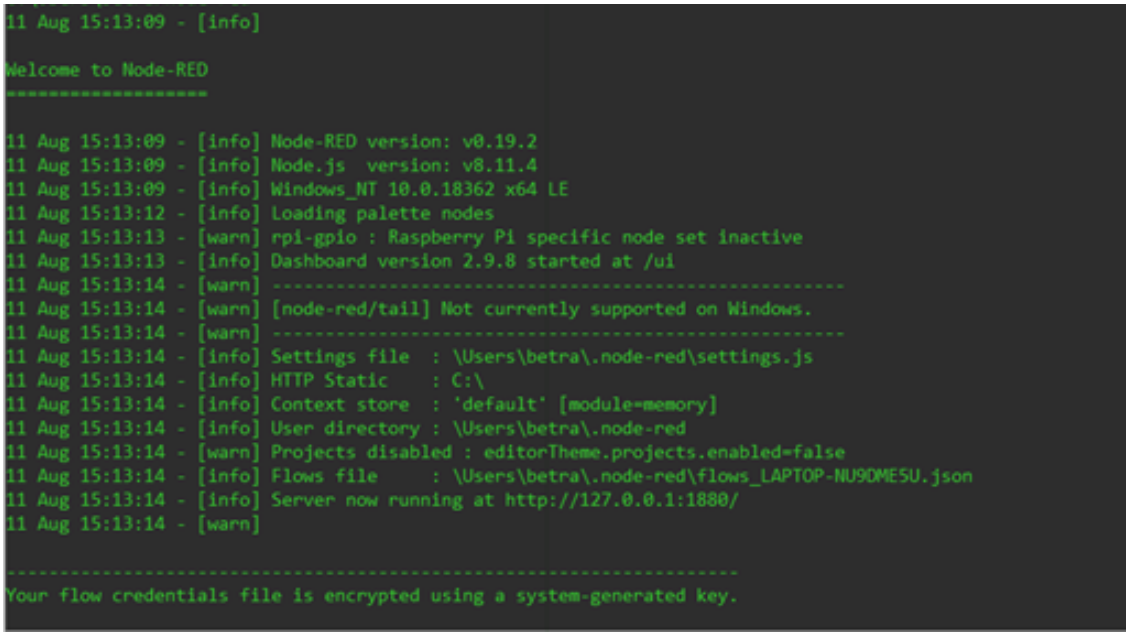


Fig. 3.19. Node-Red Command Prompt

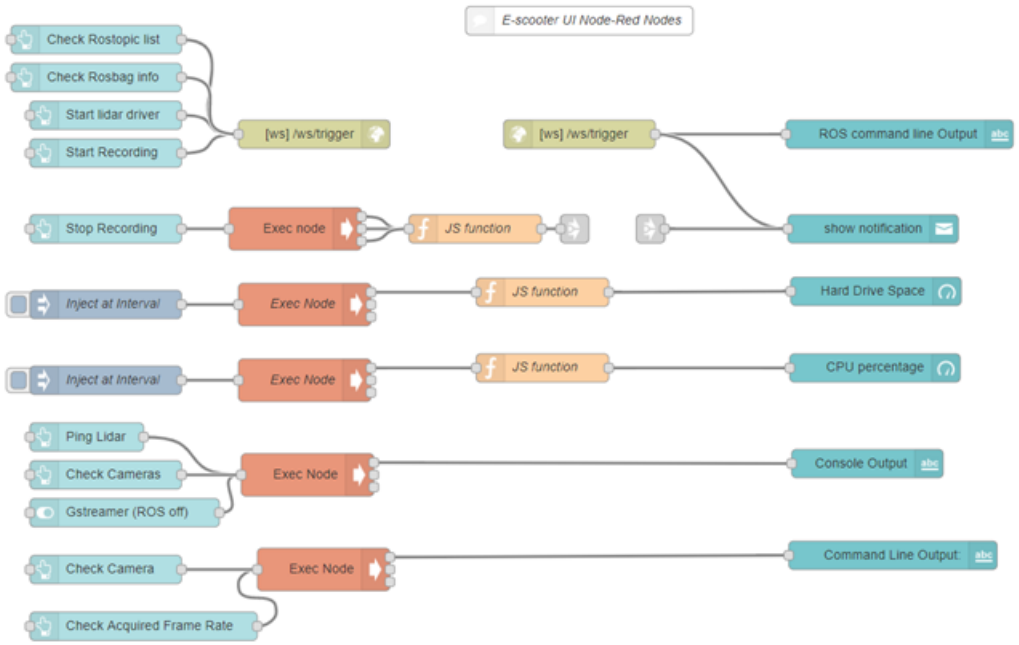


Fig. 3.20. Node-Red Programming Flows for UI

Python Websocket Client

On startup the script connects to the websocket server hosted by Node-Red and starts listening for data. The Python script web-socket client, upon receiving strings from the Node-Red web-socket server, parses the required data and then runs a given function that either starts the ROS lidar driver, the ROS based lidar data collection, the Gstreamer based camera data recording or the synchronization code i.e inotifywait. The script then uses the roslaunch library and a threading system to accordingly initiate and launch the ROS drivers and start the bash file which contains the recording processes.

4. DATA PRE-PROCESSING AND SYSTEM CALIBRATION

4.1 LIDAR and IMU Data Pre-Processing

The data collected by the e-scooter collection system primarily consisted of LIDAR, IMU and camera data. The LIDAR data was usually generated into split bag files each containing a section of the overall duration of LIDAR data. Bag files are the general methods of recording ROS data. A Python API exists to deal with ROS bag data. The Python library "rosbag" was primarily used to extract data from the bag files. The LIDAR sensor interface would send UDP packet data over the Ethernet, the ROS driver for the LIDAR would receive the data on the application layer and then compress the data if instructed and transport each UDP packet as a separate ROS message.

The bag files, if compressed during recording, were first decompressed using the rosbag command line tools. A Python script would then use the OS library to iterate over the bag files in the bag file folder. The decompressed bag files would then be read one by one and then each "message" containing the UDP binary data be saved into a string and saved as a text file. Since the LIDAR generated 1280 UDP messages per second, the output folder for the Python script would very quickly fill up a folder with large number of files.

Since the OS such as Ubuntu would find it difficult to add more files into the folder once the number of files exceeded upwards of 5,00,000. The process would get incrementally slower. To save time, the Python script would dynamically create new folder and start dumping the data into a new folder once a certain threshold was reached in a given folder. The LIDAR and the IMU data could be unloaded into the text files by simply passing the LIDAR topic or the IMU topic into the

bag.read_messages(), a function which is part of the rosbag Python Library. The naming convention for each LIDAR or IMU text file would also be to store the count of the packet and the timestamp of the same packet. This enabled quick sorting of the data during processing/fusion phase and reduced execution time.

4.2 Camera Data Pre-Processing

When Gstreamer was employed as a means of camera data collection, the only pre-processing of the data required was to rename the images in a similar convention as the LIDAR data as mentioned before to enable quick sorting and aid synchronization. The text file generated by the inotifywait method-based bash script would contain the names of the files and there corresponding UNIX timestamps. A simple Python script was developed to rename the filenames of the camera data with the same naming convention as that of the LIDAR data. The Python os library was utilized for this purpose and could efficiently rename files at the rate of 50,000 files per second on an Ubuntu system.

Distortion Correction Equations:

$$x\_corrected = x(1 + k1 * r^2 + k2 * r^4 + k3 * r^6)$$

$$y\_corrected = y(1 + k1 * r^2 + k2 * r^4 + k3 * r^6)$$

4.3 Calibration Process

The process of calibration involves primarily finding three matrices of importance to achieve fusion. These three matrices are namely the intrinsic matrix or the camera matrix, the distortion parameters, and the extrinsic transformation matrix. The

camera matrix contains data that represent camera parameters such as focus, shear and image width and height. The distortion parameters primarily represent the corrections due to lens distortion such as for the fish-eye lens. These parameters are assumed zero for low f.o.v lenses. These are added on to the final attained pixel values as shown in the distortion correction equations where k1, k2 and k3 are the distortion correction equations.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} fx & s & ox & 0 \\ 0 & fy & oy & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} r11 & r12 & r13 & tx \\ r21 & r22 & r23 & ty \\ r31 & r32 & r33 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Fig. 4.1. Intrinsic-Extrinsic Matrix Equation governing LIDAR to camera coordinate transforms

4.3.1 Calculating the Intrinsic Camera Matrix

The Intrinsic Camera Matrix and the distortion parameters were calculated using either of two methods. In order to calibrate the intrinsic matrix, for either method involve the need for pictures of a checkerboard. We used a 9x6 checkerboard with each square having a 25mm side. For collecting the images, the checkerboard was simply placed in front of the camera at various angles. For the webcam, Gstreamer was used to capture the images using the Gstreamer command line shown below.

The file name at the end of the line was changed for every image taken to make sure that the new image would not be overwritten on top of the old image. Once about 15-20 images of the checkerboard at different angles were collected, the first method to find the intrinsic matrix was simply to use the MATLAB Camera Calibrator App. As shown in the image below, images were loaded into the App. The app then cycles through the different images and then outputs the intrinsic matrix as well

as the distortion parameters. These parameters would then be utilized to calculate the extrinsic matrix using OpenCV. The distortion parameters would have 3-6 values depending on the lens distortion, however, the camera matrix would have a 3x3 shape.

Alternatively, the OpenCV function calibrateCamera() was be used to find the intrinsic matrix and the distortion parameters. This is done by first finding the findchessboardCorners() function, this function returns the edges of the checkerboard square. These corner pixel values and square intersections' values are then given to the calibrateCamera() function to find the camera matrix and the distortion parameters.

Gstreamer command for checkerboard image capture:

gst-launch-1.0 v4l2src device=/dev/video0 ! nvvidconv ! nvjpenenc ! filesink location= /home/tasi/intrinsic/image.jpg

4.3.2 Calculating the Extrinsic Matrix

The process of finding the extrinsic matrix involved finding an initial extrinsic matrix using a well established algorithm known as the Levenberg-Marquadt algorithm also known as the damped least squares method for solving A(X) =B matrix equations solving for the extrinsic matrix X. The algorithm was implemented using the OpenCV solvepnp function. This initial matrix was then manually fine-tuned with the aid of a Python script.

Initial Calibration

The inputs for the OpenCV solvePnP function involved the intrinsic matrix, the distortion parameters, and at least six correspondences of pixel values and their corresponding coordinate values which in our case were, LIDAR coordinates. The process of finding the initial calibration extrinsic matrix involved a setup in which three poster boards were kept at relatively same distance from the camera LIDAR setup as shown below in fig 4.4. The data collection system was then used to record calibration data

of this poster boards. This process was repeated as the setup was kept roughly 3,5 and 7 meters away from the LIDAR for each iteration. The aim here was to obtain the point and pixel pairs of the LIDAR and camera data respectively. The poster board was chosen primarily for their sharp vertices which could be detected both in the LIDAR point cloud as well as the camera image.

The poster boards were intentionally evenly spread out over the width of the image as it was observed that selecting point pixels only from a specific region of the image resulted in bad fusion on the other areas of the image notably in terms of width. These point and pixel values of these vertices were used as inputs to the solvePnP function. The pixel values were simply attained by opening the image file collected using an image app such as MS-Paint or Linux ImageMagick and finding out the pixel values of the corners.

To find out the equivalent point in the LIDAR data, there were either of three methods used. The first one involved the use of Ousterstudio. The Ousterstudio software could easily be downloaded and run on Windows/Linux allowed the user to collect and playback LIDAR data. As shown below in fig 4.3, the poster edge could simply be manually selected via the UI and the XYZ coordinates of the poster. This method however required that the LIDAR be captured via Ousterstudio and thus required the user to separately record first the LIDAR data using Ousterstudio, and then the image data using Gstreamer or ROS. This meant that a display needed to be connected to the NVIDIA Tx2 along with input devices.

The second method involved using the ROS LIDAR driver which came with the visualization playback functionality using the Rviz ROS software. The LIDAR driver could be set in the replay mode in a terminal. The Rviz software could then be also started via a second terminal. As shown in the image below, by playing and then pausing the recorded bag file in a different terminal with the LIDAR data, the Rviz software also had the ability to allow the user to select a point on the point cloud.

The third method to select the points from the point cloud involved first recording LIDAR data through the UI via ROS. The bag file was then extracted using the

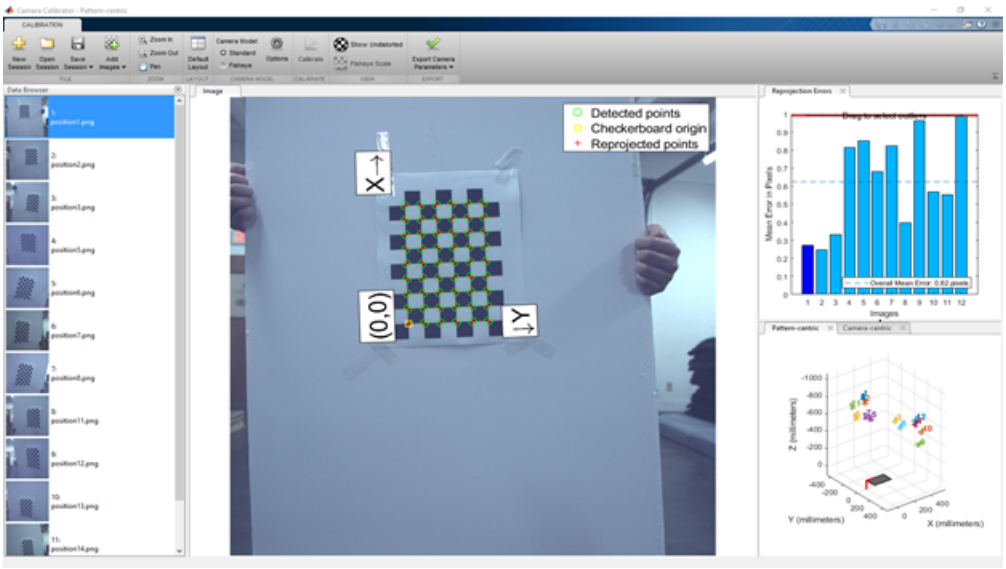


Fig. 4.2. Intrinsic Matrix calibration using MATLAB Camera Calibrator

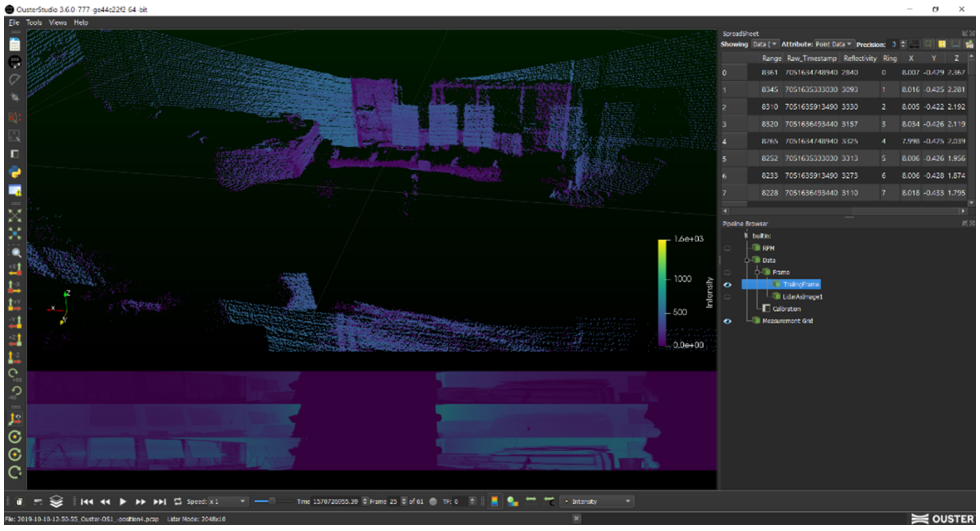


Fig. 4.3. Ouster Studio Interface replaying data from the OS-1 64 LIDAR

preprocessing steps outlined previously. Once this was achieved a simple Python script was used to convert the text files generated by the bag extraction script into a dataframe containing X,Y and Z values of the LIDAR point cloud data. This dataframe was then plotted as a scatter plot using the Python Plotly library. The scatter plot then allowed for selection of a point (the poster edges) by simply hovering

the cursor on top of the point cloud as shown in the image below and a label would display the XYZ value of that point. The details regarding the conversion of the binary data in the text file into XYZ values in described in detail in chapter C.

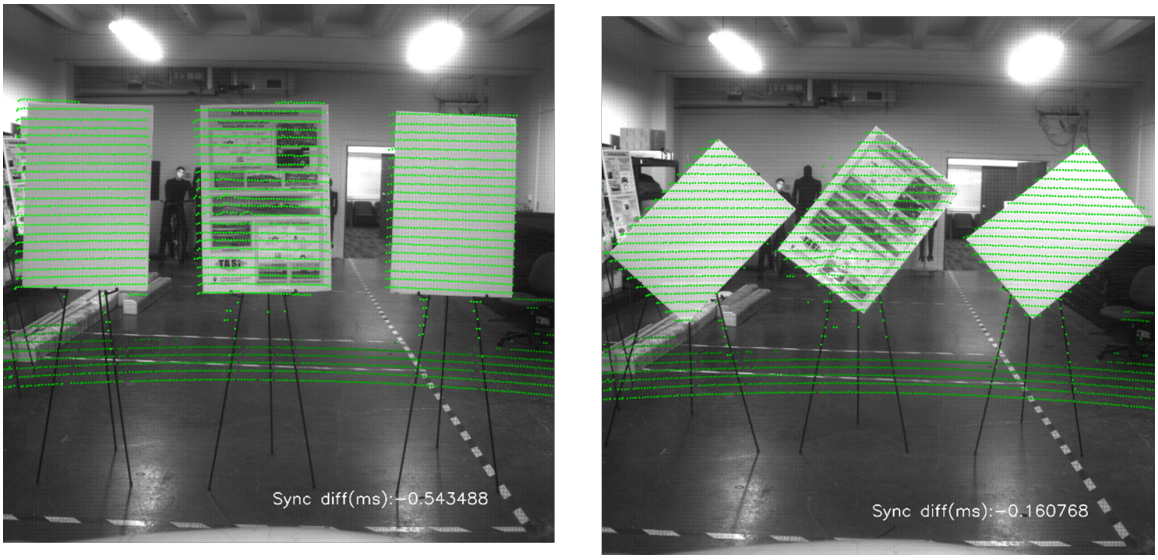


Fig. 4.4. Fusion Overlay on 3 poster boards using collected LIDAR and camera data

Three poster boards kept at different distances generated a total of 36 pixel-point pairs that were then supplied to the SolvePnP function along with the camera matrix and the distortion coefficients. The function would then return two vectors each of them of a size 3x1. These vectors called the rotation vector and the translation vector that would represent the extrinsic matrix. This rotation vector and translation vector represented the initial extrinsic matrix which would then be later refined.

```
76 sucess, rotation_vector, translation_vector = \
77   cv2.solvePnP(lidar_points, image_points, camera_matrix, dist_coeffs)
78
```

Fig. 4.5. Python code snippet to find the Extrinsic Matrix's Rotation and Translation Vectors

```
(camera_pixels2D, jacobian) = \
        cv2.projectPoints(Lidar_Points_3D, rotation_vector,\
                          translation_vector, camera_matrix, dist_coeffs)

u,v = ( int(camera_pixels2D[0][0][0]), int(camera_pixels2D[0][0][1]))

```

Fig. 4.6. Python code snippet to find corresponding pixels for LIDAR points

Fig. 4.7. Overlay showing fusion on data at longer distances

The accuracy of the initial extrinsic matrix was checked by using the OpenCV projectPoints() function. This function allowed to give a visual understanding of the accuracy of both matrices. The inputs to the function were the LIDAR point, the Camera matrix, the distortion parameters as well as the transformation matrix or vectors. The important output of the function was the resulting 2d point which contained the u,v value. The fusion was generally tested on the data collected for calibration and on some collected data. Each generated 2D pixel value that fell inside the bounds of the image was displayed using a green dot on the image to represent

Fig. 4.8. Overlay showing fusion on data with LIDAR points filtered to allow points only within a distance (LIDAR Horizontal Resolution = 1024)

the fusion overlay. Sometimes the colored dot was given a hue mapping from green to black proportional on the distance corresponding point from the LIDAR frame.

Fine tuning the extrinsic matrix

Once an extrinsic matrix was obtained, it was then manually refined with the aid of a Python script. The setup for the fine tuning involved placing four poster boards at various distances as shown in the image below, the LIDAR and camera data collected for the same. The data was prepocessed as outlined earlier. The script then loaded a given dataframe and the corresponding image with poster board data. The camera matrix, the distortion parameters, the rotation and the translation vectors were supplied to the script. The script then performed a simple fusion using the project points function as outlined earlier and then display the fused image. The script would make small increments or decrements to the rotation vector or

the translation vector depending on which key was pressed. The script would then recalculate the point projection and update the fusion image the keyboard library was used to monitor key changes while the script was running.

Pressing of the keys ‘q’,’w’,’e’ would result in a small addition of 0.01 radians was made to the first, second and third value of the rotation vector. Pressing of the keys ‘a’,’s’,’d’ would result in a small subtraction of 0.01radians was made to the first, second and third value of the rotation vector. Similarly, ‘r’,’t’,’y’ and ‘f’,’g’,’h’ keys were used to make adjustments of 0.01 meters to the translation vector. As the keys were changed to fit the fusion better, the recent most values of the rotation and translation vectors were updated and saved. Once the user was content on the fusion, the program could be ended by hitting the escape key and the program would print out the last updated rotation and translation vectors before exiting. These vectors would then represent the fine-tuned extrinsic matrix. The evaluation of this fine tuning is shown in the evaluations section of this document.

5. DATA PROCESSING SYSTEM

5.1 LIDAR Data Processing

ROS was used as a means to collect road data instead of the Ousterstudio software due to its 10 minute limit between records. The Ouster OS1-64 LIDAR's data sheet along with the os1 Python library were parallelly used for processing the LIDAR data. Primary processing of the LIDAR data involved converting the text files generated by the pre-processing step of the pipeline. According the datasheet, the LIDAR would forward the UDP data to the port 7502 of the designated destination computer.If the OusterStudio software were used for recording the data for calibration purposes. The LIDAR data can be easily extracted through the software via export data feature.

Each data packet consisted of 16 azimuth blocks; the packet always has a length of 12609 bytes. However, when the packet was stored as a text file, the packet would show a total length of the packet as 12609 bytes. The extra byte was attributed to a newline "\n" character that the preprocessing script would append to the end of the text file. The LIDAR could be set in either a 1024x10 or a 2048x10 resolution mode. The LIDAR would output either 64 packets of data or 128 packets of data depending on the mode in which the LIDAR was set.

A higher resolution of 2048x10 would generate twice as many points than the 1024x10, however the increased resolution did have the effect on the bandwidth of the Ethernet and packet dropping would be relatively more frequent as the data collection went on. Either way, each packet would always consist of 16 azimuth blocks arranged as shown below. Each azimuth block would contain the values of timestamp, Frame ID, measurement ID, Encoder count, Data block and the Packet status. The encoder referred to the hall encoder inside the LIDAR which would indicate the yaw rotation

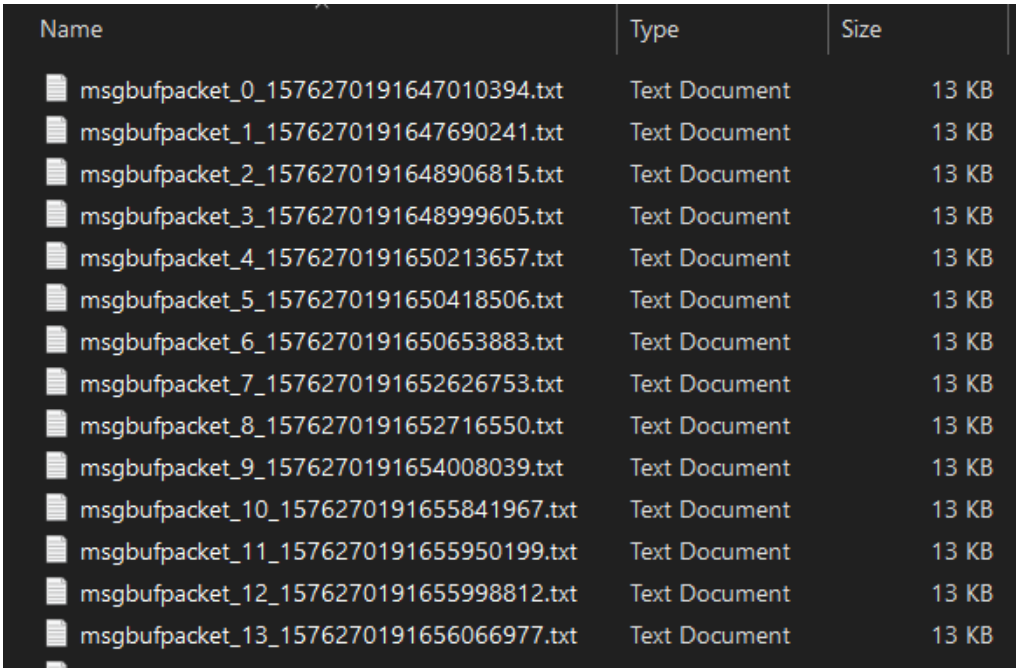


Fig. 5.1. LIDAR files with filename containing packet count and timestamp for synchronization collected from the ROS method containing buffer data after pre-processing the bag data

angle of the LIDAR scan for any given packet with the direction of the LIDAR connector wire representing the positive x-axis of the 3D Ouster LIDAR.

Each data block contained 64 values of range, reflectivity, ambient noise, and the number of signal photons received. Each range inside the data block represented the 64 beams vertically in the 3D LIDAR. Each of the ranges inside the LIDAR packet had a corresponding azimuth and altitude angle that could be used in conjunction with the encoder value of the block to covert the range values into XYZ coordinates.

The first task was to convert the binary UDP data into words this was performed by using the formatting key of the binary data which explained the repeating data format of each word. This was implemented simply using the struct library. Once the main values of range and encoder value were discovered, the XYZ coordinate could be converted into 2 ways to implement the equations mentioned below for conversion.

5.1.1 Decoding UDP Packets

The first way to implement the equations were using the Python math library to compute the coordinate values. This was required due to the presence of sines and cosines in the equations. The second method involved using the logic of lookup trigonometric tables to calculate the pre-calculated sign and cosine values while implementing the equations to save computation time. Although the data processing step is an offline process, nevertheless, the scale of converting millions of LIDAR data blocks into coordinate values required a certain level of optimization to meet project requirements. The second downside of using the math library was an induced 2-degree yaw difference compared to using trigonometric tables, which yielded in faster data processing.

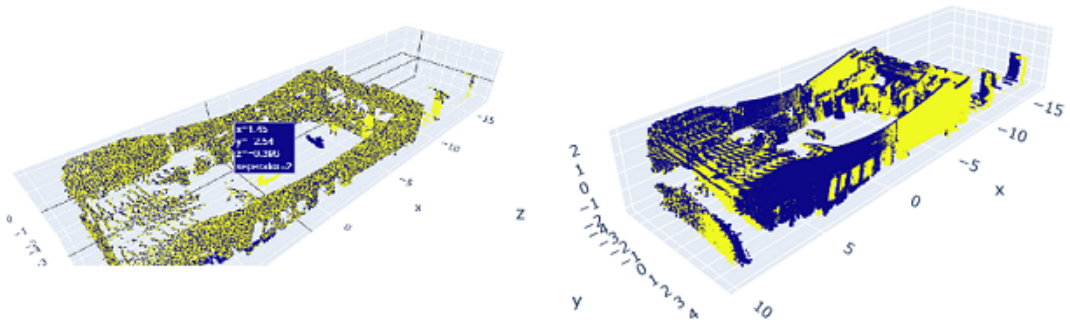


Fig. 5.2. 2 Degree yaw difference induced in the point cloud due to the difference in UDP packet decoding methods. Left Image shows the same point cloud processing method (yellow and blue mixed) and Right image shows difference due to difference in math library and Trigonometric Table based processing

The Cartesian conversion depended upon the encoder count which pointed to the rotation angle in which the LIDAR beam was angled at that moment. The beam altitude angle is a function of the specific ring number of the LIDAR, for a 64 beam LIDAR, there are 64 "rings" in total. The values for which can be taken from the configuration file provided by Ouster. As shown in the picture below, the error was induced due to the difference in computation of the libraries and the effects of the

error being scaled due to the larger range distances by which the sines and cosines of the azimuth angles that were multiplied. This error in the XYZ coordinate creates problems with fusion when used alongside calibration data from ROS or OusterStudio as both programs used the trigonometric tables for faster computation. Thus, the code required a 2-degree offset be added to the theta or yaw angle if the math library was utilized. In course of the optimization, however, the trigonometric tables were finally utilized for both its speed as well as its compatibility with the ROS and OusterStudio programs. The Pandas library was used primarily to store XYZ values for a given rotation or "frame" of data. A separate Python script was also written that was used purely for the generation of LIDAR data points as CSV files.

It must be noted that it became important that the means used to convert the UDP packet to Cartesian needed to be the same for the extrinsic calibration and usage on the processed data as the difference in the two methods of calculation induced an error that could be seen on the final fusion. This difference is due to the relative lack of resolution in the sin and cosine values in the trigonometric tables that allow for speed over accuracy which increase linearly with distance from the LIDAR thus appearing as a yaw shift of 2 degrees.

Word	Azimuth Block 0	Azimuth Block 1	...	Azimuth Block 15
(Word 0,1)	Timestamp	Timestamp	...	Timestamp
(Word 2[0:15])	Measurement ID	Measurement ID	...	Measurement ID
(Word 2[16:31])	Frame ID	Frame ID	...	Frame ID
(Word 3)	Encoder Count	Encoder Count	...	Encoder Count
(Word 4,5,6)	Channel 0 Data Block	Channel 0 Data Block	...	Channel 0 Data Block
(Word 7,8,9)	Channel 1 Data Block	Channel 1 Data Block	...	Channel 1 Data Block
	.	.		.
(Word 193, 194, 195)	Channel 63 Data Block	Channel 63 Data Block	...	Channel 63 Data Block
(Word 196)	Packet Status	Packet Status	...	Packet Status

Fig. 5.3. LIDAR Data Packet format Source: Ouster Software Guide

$$r = range(mm)$$

$$\theta = 2\pi\left(\frac{encoder\ count}{90112} + \frac{beam\ azimuth\ angles[i]}{360}\right)$$

$$\emptyset = 2\pi\ \frac{beam\ altitude\ angles[i]}{360}$$

$$x = r\ \cos\theta\ \cos\emptyset$$

$$y = -r\ \sin\theta\ \cos\emptyset$$

$$z = r\ \sin\emptyset$$

Fig. 5.4. LIDAR spherical to cartesian conversion Source: Ouster Software Guide

5.2 Camera Data Processing

As data is mapped from the LIDAR data to the camera, the processing pipeline does not need the camera picture data to the fusion data. This section in the pipeline ideally only needs the LIDAR data, the intrinsic matrix, the extrinsic matrix, the distortion parameters, and the synchronization data to generate the mapping table containing the LIDAR-camera coordinate mapping. The one reason however for requiring the camera data is to generate the fusion overlays to check and confirm the working of the fusion for the collected data.

The script for processing was written in Python and the OpenCV library was used over other libraries for a couple of reasons. The library was chosen over an imaging library like PIL primarily due to its lack of direct complexity compare to that of PIL, the previous use of OpenCV library for the use of solvepnp() and projectPoints() kept the implementation relatively simpler by using as many functions possible from the same library. The other advantage of the library is the application OpenCV GPU to

use hardware acceleration the generation of fusion images when dealing with image generation. A similar logic was use for the selection of cupy, a CUDA accelerated library with a compatible syntax as the widely used numpy.

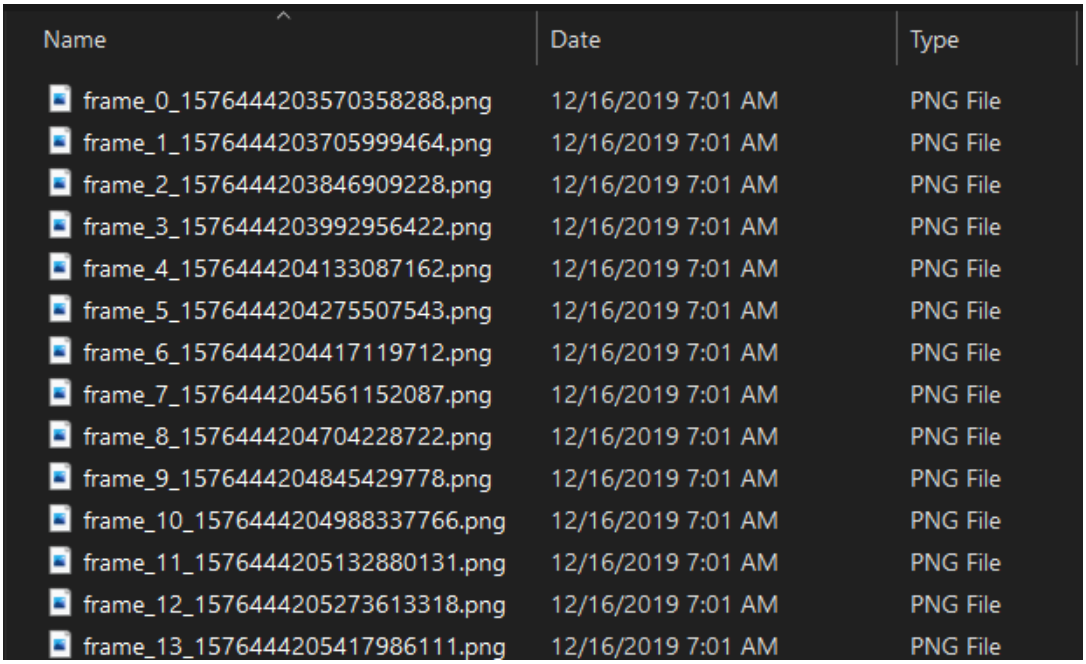


Fig. 5.5. Image file names with framecount and timestamp for synchronization collected from either Gstreamer or ROS method

5.3 Synchronization and Fusion

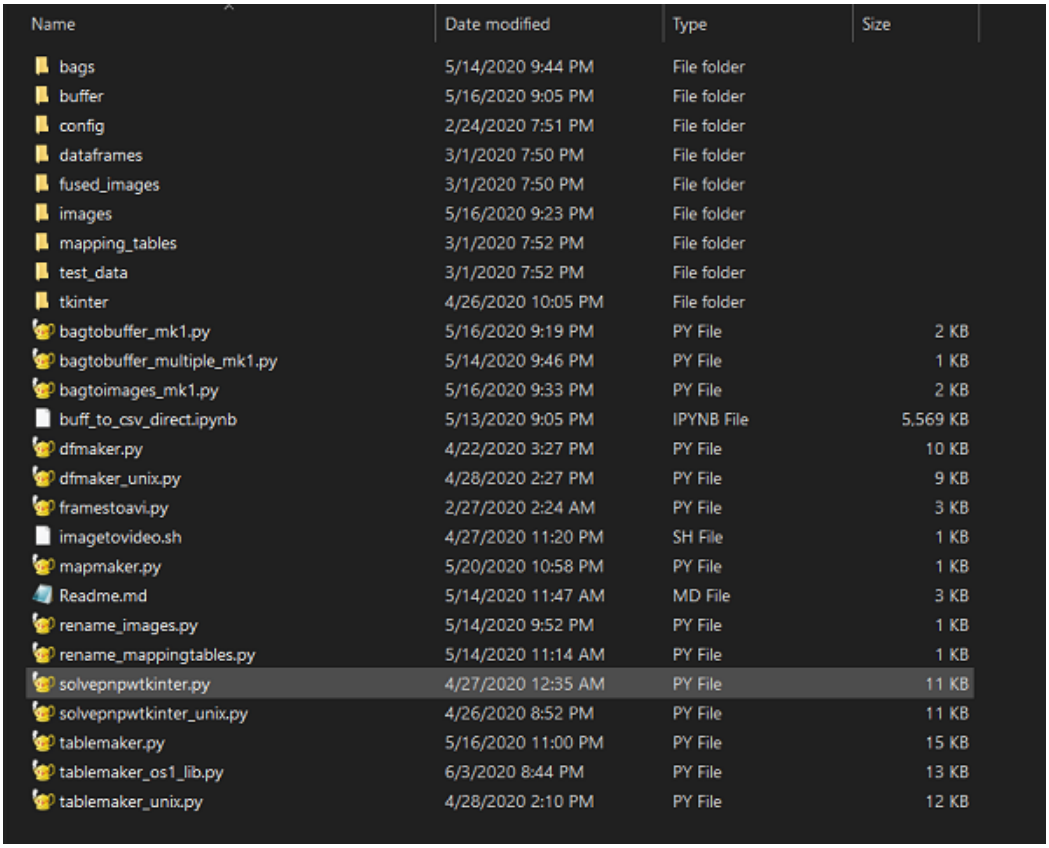
The need for synchronization arises from the need to process collected data that is continuous and also being collected at different rates along with the very nature of the sensors also meaning that while the camera data is considered to be for practical purposes, taken instantly (this is not true for the Logitech cameras as they use a rolling shutter), the LIDAR data is available continuously. To match the camera data to the LIDAR data, each frame of the camera is timestamped, and the timestamp is stored inside the filename of the image. Similarly, each packet of the LIDAR data is timestamped as well.

The fusion script first iterates over the camera and LIDAR data directories, storing the names of the files as a list along with the timestamp values of each frame/packet alongside in two separate lists, one for each sensor. The script then loops over every image (or every other image) in the file directory. For every image, the script then finds the corresponding timestamp and then subtracts the value of the timestamp with the list containing the LIDAR timestamps. This new list thus contains timestamp difference values, the lowest value in the array would represent the prime LIDAR packet closest time difference with the camera frame.

The script then chooses the LIDAR packet as the center point of the LIDAR data. Since 128 LIDAR packets make up one rotation of LIDAR data(for 2048x10 mode), the script selects 64 sequential packets before the prime LIDAR packets and 63 sequential packets after the prime LIDAR packet. To make a total of 128 LIDAR packets. The data by this means is synchronized. The process is quick as using Python Lists for purposes of sorting and scanning in Python is faster than using Pandas dataframes.

The script then converts these 128 LIDAR text files containing UDP binary data and converts them into a dataframe of XYZ values using the trigonometric tables method mentioned earlier in the chapter. The script then performs fusion on the image frame and the LIDAR. It does this by iterating over the pandas dataframe and fusing each LIDAR XYZ value, the intrinsic and extrinsic matrix, and the distortion coefficients through the projectPoints() function, The output of the function are the corresponding pixel values (u,v) in the image.

Since the LIDAR data contains 360-degree data, many LIDAR point that do not belong in the image are projected at values that lie outside the bounds of the image. These data points are eliminated using a simple conditional statement that checks if the resultant pixel values exceed the bounds of the image. For an image width and height of 1920 and 1080 respectively, a pixel is permitted only if 0 >u >1920 and 0 >v >1080 is satisfied for the resultant pixel. A mapping table using lists at first is generated and at the end of the loop is converted and stored into a CSV file.



Name	Date modified	Type	Size
bags	5/14/2020 9:44 PM	File folder	
buffer	5/16/2020 9:05 PM	File folder	
config	2/24/2020 7:51 PM	File folder	
dataframes	3/1/2020 7:50 PM	File folder	
fused_images	3/1/2020 7:50 PM	File folder	
images	5/16/2020 9:23 PM	File folder	
mapping_tables	3/1/2020 7:52 PM	File folder	
test_data	3/1/2020 7:52 PM	File folder	
tkinter	4/26/2020 10:05 PM	File folder	
bagtobuffer_mk1.py	5/16/2020 9:19 PM	PY File	2 KB
bagtobuffer_multiple_mk1.py	5/14/2020 9:46 PM	PY File	1 KB
bagtoimages_mk1.py	5/16/2020 9:33 PM	PY File	2 KB
buff_to_csv_direct.ipynb	5/13/2020 9:05 PM	IPYNB File	5,569 KB
dfmaker.py	4/22/2020 3:27 PM	PY File	10 KB
dfmaker_unix.py	4/28/2020 2:27 PM	PY File	9 KB
framestoavi.py	2/27/2020 2:24 AM	PY File	3 KB
imagetovideo.sh	4/27/2020 11:20 PM	SH File	1 KB
mapmaker.py	5/20/2020 10:58 PM	PY File	1 KB
Readme.md	5/14/2020 11:47 AM	MD File	3 KB
rename_images.py	5/14/2020 9:52 PM	PY File	1 KB
rename_mappingtables.py	5/14/2020 11:14 AM	PY File	1 KB
solvepnpwtkinter.py	4/27/2020 12:35 AM	PY File	11 KB
solvepnpwtkinter_unix.py	4/26/2020 8:52 PM	PY File	11 KB
tablemaker.py	5/16/2020 11:00 PM	PY File	15 KB
tablemaker_os1_lib.py	6/3/2020 8:44 PM	PY File	13 KB
tablemaker_unix.py	4/28/2020 2:10 PM	PY File	12 KB

Fig. 5.6. Template folder containing the required scripts for calibration and processing, to be used for every new data set

This file is referred to as the mapping table that contains the camera and corresponding LIDAR coordinates. The mapping table were used for cross-referencing with the object detection algorithms to find the relative coordinates of the object such as the car in question. To generate the fusion image, the script loops through the pixel values in the mapping table and draws green colored points on the image frame loaded via OpenCV. Since the mapping table contains the LIDAR values as well, the intensity of the hue is mapped to a given distance.

This causes the points closer to appear a light green, and far away objects to have a darker green color. This helps in understanding the fusion as it is useful to understand if the fusion on a car with a distant background is happening correctly by comparing the color of the pixels in the background to the pixel on the car. Once

fusion for a frame is done, the process is then simply iterated over for the remaining images. In terms of code optimization, here it is noticed that again using Python Lists for appending related data sped up the loop execution.

	A	B	C	D	E	F	G
1	Sr. No	CamU	CamV	LidarX	LidarY	LidarZ	Range
2	1	11	366	-31.7129	-12.0094	10.27441	35.433
3	2	30	368	-31.3877	-11.6665	10.14567	34.989
4	3	18	598	-31.7243	-11.8232	7.519933	34.681
5	4	13	710	-31.7938	-11.8567	6.20883	34.496
6	5	4	927	-31.7585	-11.8391	3.673632	34.092
7	6	49	370	-30.9947	-11.3045	9.996044	34.473
8	7	37	599	-31.3562	-11.4674	7.415854	34.201
9	8	32	712	-31.3671	-11.4788	6.111637	33.956
10	9	23	928	-31.4048	-11.4883	3.624495	33.636
11	10	19	1037	-31.4274	-11.4959	2.386403	33.549
12	11	12	1254	-31.6625	-11.5606	-0.06412	33.707
13	12	6	1364	-25.1277	-9.17413	-1.03184	26.77
14	13	68	372	-30.631	-10.9594	9.856859	33.993
15	14	56	601	-30.9739	-11.1126	7.309173	33.709

Fig. 5.7. Mapping table generated at the end of the fusion processing code

Writing a separate loop to draw the points on the image rather than joining it with the initial loop also sped up the speed of iteration. The images are also saved in JPEG as this format is faster compared to other file formats such as RAW and PNG file formats. The size of the files fused images can also be reduced by reducing the dimensions of the fused image reducing the size of the overall data generated during the processing stage. The fusion script also allowed to specifically allow for the selection of particular subset of images from the entire collected data.

This proved important for analysis as the data could be selectively chosen and fused to not needing to process all data. Since huge chunks of the collected data would not involve interactions with cars on the road, the camera data could quickly scanned manually to find the interested region involving interactions and only a small portion of the collected data could then be fused as required.

As the actual interactions with the cars would be relatively quick portions of collected data, with the maximum interaction with a car not exceeding upwards of 30 seconds, the actual data that required to be fused could be as short as 30 seconds.

5.4 Image Processing

The later portion of the processing stage was developed by Apurva Kumar. The process involved using the raw unfused images collected and the performing segmentation as well as tracking of cars in the frames. A Kernelized correlation filter was used to track the cars between each individual frames. This was done in two stages. Firstly, a simple YOLO base object detection was utilized that would track the cars on the road for a given frame. The ouput of this algorithm would be in the form of simple bounding box outputs. These bounding box values could then be fed into a KCF type filter which could then be used to subsequently track down the cars in the subsequent frames. This process was then compounded by performing semantic segmentation on the same frames. However, the segmentation labels for every given pixel would be the same irrespective of them belonging to the different car.

Thus, the segmentation data would be first cross reference with the image tracking bounding box data. This would then produce pixel data specific to each car spanning over multiple frames. A Python script then cross references the pixels of the detected car with the corresponding values in the mapping tables. Each car in each frame therefore produce a cluster of 3D points. The median of the points was then taken and converted to 2D values by removing the z-axis values from the 3D point. Thus, combined with the tracking data produced a set of 2D points. These 2D points were then used to calculate average velocity values for the cars between two frames by using the change in distance of the median values and the change in time values by using the differences in the recorded timestamps for the frames in question.

5.5 Image to Video Processing

For purposes of demonstration, it was often required to process the collected images of the data as well as the fused images as a cohesive video. This was primarily useful as a file input to certain tracking-based script that were video images. To be noted that a considerable amount of storage space could be saved by converting the images into compressed video formats such as MJPEG and H.264. The advantages of having fused image data in a video format also would allow to quickly scan through the video to check the validity and accuracy of the fusion in the sections of the video. The implementation for converting image to video was done in two primary ways using the OpenCV library and the Gstreamer library. For both methods, the images were stored in a single folder.

5.5.1 OpenCV Method

The OpenCV method involved a Python script that would iterate over the folder which contained the files via the OS Python library. The script would then sort the images based on the count values in the filename and then convert them into OpenCV numpy arrays. The script would then append the numpy arrays into a long list and then use an OpenCV function to convert the array list into an entire video of a specified file type and as specified compression. The system would then save the output in a compressed video in a pre-specified directory. This method has its limitations the primarily limitation being the number of files that the system could load in one continuous execution.

For an image resolution of 2048x2048, for a raw file format, the software would fail at about 500 frames and then would require the script to load 500 frames at a time and then create a video and continue to create short videos of 500 frames each. Once all the images were converted into short videos, a separate script would then string the individual short videos into one long uncut video via another OpenCV function to allow to view the collected data as an MPEG video. The length of the

video was determined by the available allocated space given by the CPU to the Spyder application and thus the length is limited based on the system's RAM allocation.

5.5.2 Gstreamer

The Gstreamer method used a simple command line tool interface that could be deployed by a single command over the command line and thus made easily executable via a bash script. The Gstreamer pipeline/command line command is shown below. The Gstreamer implementation involved a pipeline that made the use of a multifilesrc element. This element then had the folder containing the images passed through it and functioned as the primary source element for the pipeline. This element would then be followed up by a jpegdec element that would decode the jpeg compression from the images.

This was followed by the videoconvert element that aids in converting data from an image format into video format. A simple filesink sink element is then used to save the converted images into one video. This method would however limit the use of the system to Linux systems with Gstreamer being a Linux based software. This method does not suffer the memory limitations as that of the OpenCV method and can be used to convert images into a video. The system was tested for upwards of 30,000 frames each frame stored into a 1920x1080 resolution.

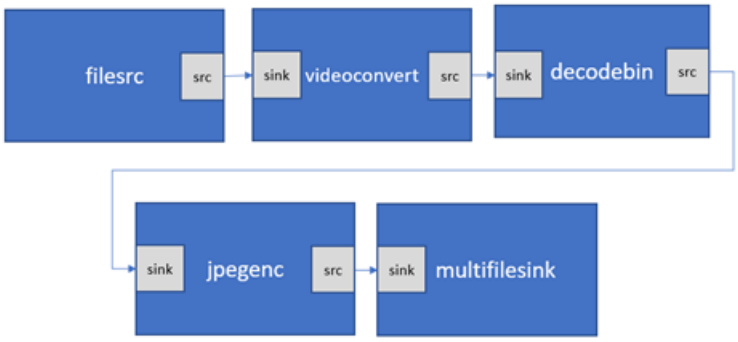


Fig. 5.8. Gstreamer Pipeline for converting captured image frames into a single video.

6. RESULTS/EVALUATION

6.1 LIDAR UDP Packet Drop Evaluation

The LIDAR on the system could operate in multiple horizontal resolutions; however these different resolutions require different levels of Ethernet bandwidth. Although a sensor Gigabit interface is required and was used in the system, occasionally packet loss was observed. This packet loss could have been due to load on the system memory of the TX2. ROS recording software rosbag record would be processing the incoming UDP packets and store them into the bag file and the buffer -b flag was set to 0(which allowed it use as much system RAM that was available). However, occasional packet loss was still observed. The figure 6.1 shows the visual depiction of two adjacent frames of visualized LIDAR data. The top photo shows a normal frame with no packet loss whereas the image below shows another frame with packet loss with the dark area showing the absent packet.

It was also observed that depending on the horizontal resolution and scan rate, that the packet loss was exacerbated by the amount of incoming data. For a resolution of 2048, the incoming data was roughly 129Mbps, whereas for a resolution of 1024, the incoming data rate was roughly 64 Mbps. The figure demonstrates the packet loss by comparing the no of received packets for a period of 20 minutes and then compared it to the expected number of packets based on the rotation speed of the LIDAR and the LIDAR resolution.

Percent Packet Loss = (Exp. Packets - Received Packets / ExpectedPackets)*100 (6.1)

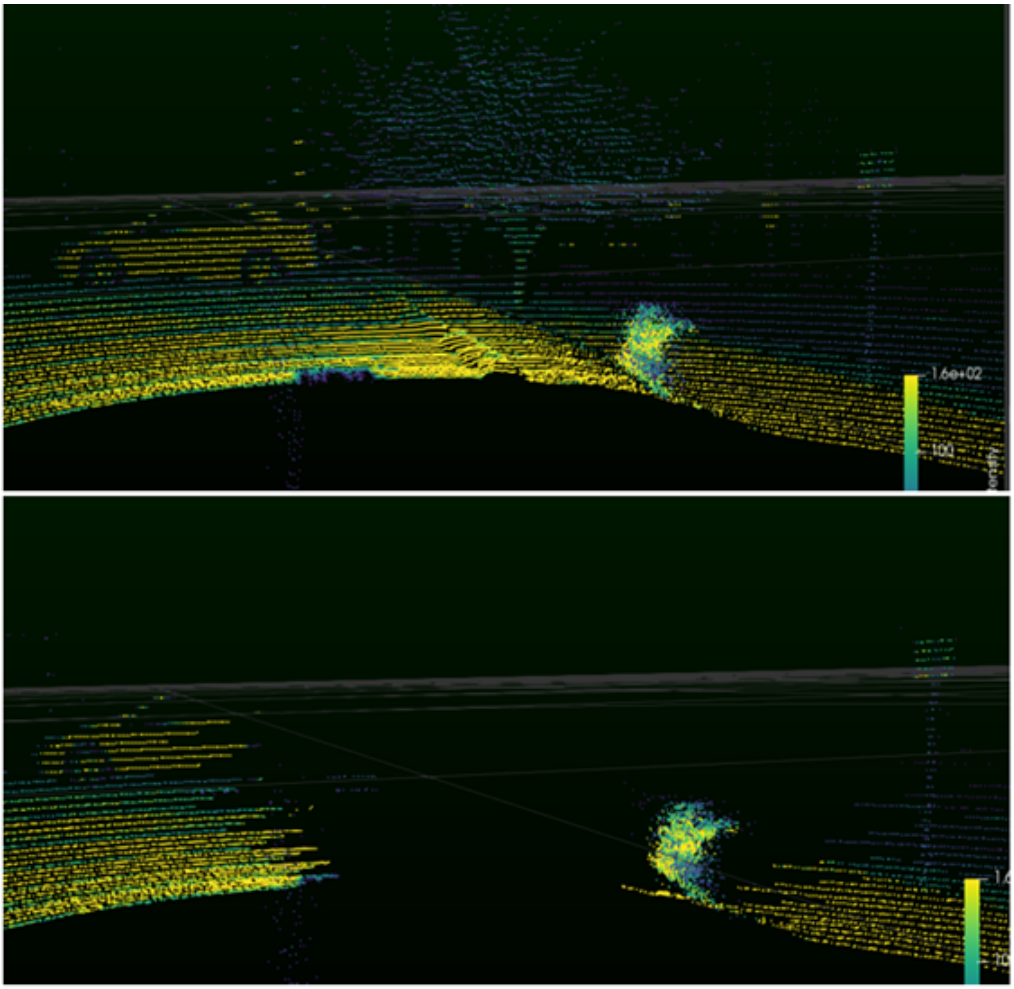


Fig. 6.1. Visualization of Packet Loss in a frame in the dark area (below) compared to no packet loss in a frame (above)

6.2 Camera Frame Rate Evaluation

The selected camera was evaluated with some of the other cameras that were shortlisted and selected. The acquired frame rates were compared to the max possible frame rates achieved with from any method. On the Logitech camera and the Generic USB webcam, the Gstreamer software was used and for the FLIR Grasshopper 3, the Spinview software was utilized. Although the FLIR grasshopper 3 could achieve higher frame rates, the cost and weight of the cameras was a detriment to selection.

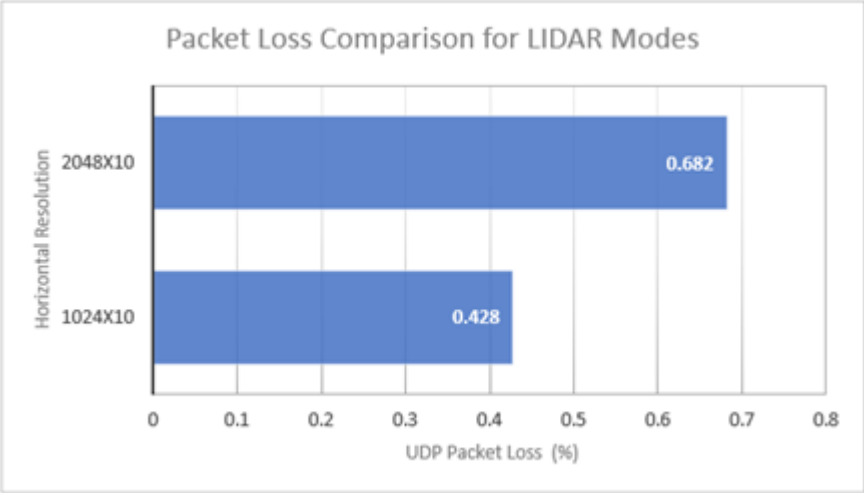


Fig. 6.2. Percentage Packet Loss Comparison for two LIDAR modes

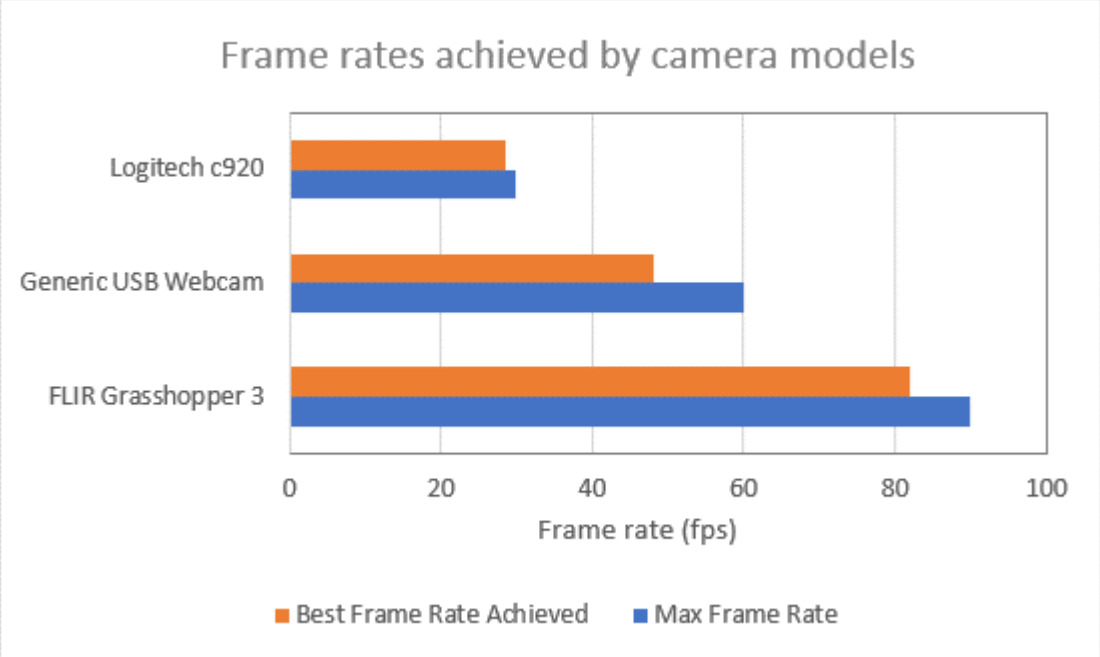


Fig. 6.3. Best frame rates achieved with the experimented camera solutions

The Generic USB webcam fares better than the Logitech camera in terms of both frame rate and cost, but lacks driver support. This tends to create an issue for low cost USB devices that are not supported well on a software level and causes issues

when attempting to connect multiple cameras. On a Jetson Tx2, we used the generic v4l2 (video4linux) driver which allows for connection to the camera. The camera on connection, would communicate the required bandwidth to be allocated for the USB connection. This was a problem with the generic USB webcam as it would set a value that would stop the Tx2 from receiving data from other cameras. This issue limited the use of more than one camera per USB 3.0 (assuming the USB 3.0 port's are connected to single PCI-E slot) as was the case with the Jetson Tx2. The Logitech camera allowed for connecting the same number of cameras as the FLIR Grasshopper 3 but at a much lower price, and weight than the FLIR camera.

Camera Name	Number of Cameras Possible	Driver
FLIR Grasshopper	3	libflycapture2
Generic USB Webcam	1	v4l2
Logitech c920	3	v4l2

Table 6.1.
Shows the number of possible cameras over a USB 3.0 connection for specific cameras and their drivers

Different software's were used to find one that allowed us to obtain the maximum frame rate from the Logitech c920 camera. The primary option attempted were ROS, Gstreamer, and OpenCV. Among these four, ROS was given priority as the software was already being utilized for collecting the LIDAR data. This would be helpful to use a single bag file or sets of bag files in chunks to collect both the LIDAR and camera data. ROS was also useful as the underlying driver for the same could be either generic v4l library or Gstreamer library which used v4l2 library but allowed to exploit the acceleration offered by the TX2. However due to the nature of the ROS bag-based recording process which involved image transport to ROS images, the observed frame rate was lower on the ROS system compared to that of the accelerated Gstreamer method and was attributed to the ARM CPU on the Tx2 and the lack of optimization.

It must be noted that the basic cvvideocapture function from the OpenCV library was used for the OpenCV method, which also utilized the v4l2 library but in our testing fared worse than the other methods. It was observed that the accelerated Gstreamer won out with getting the highest frame rate from the two cameras. The Gstreamer also took out the potential load of image transport and software-based compression on the CPU that limited the ROS method to a lower frame rate.

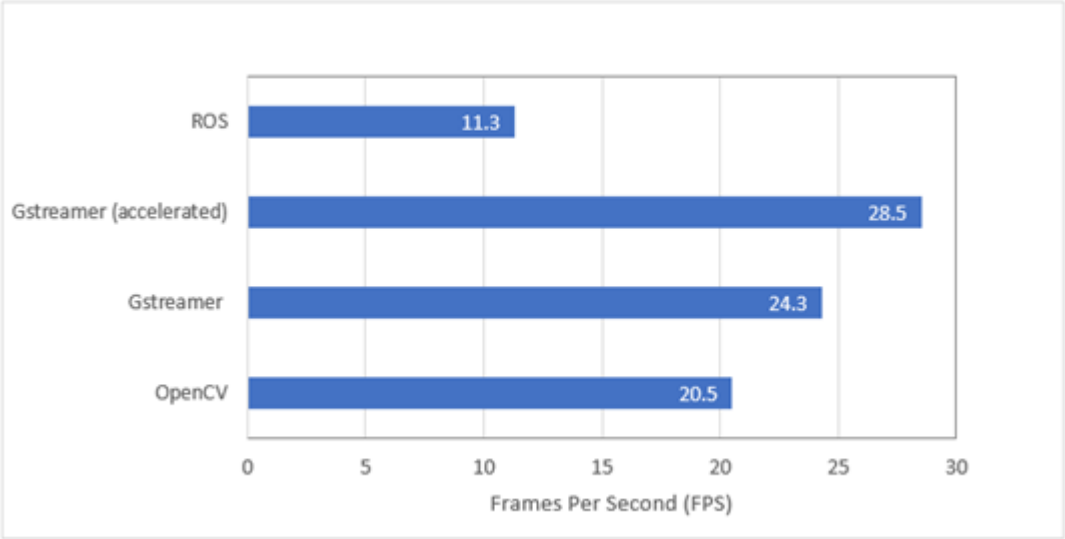


Fig. 6.4. Frame rate comparison for 2x Logitech c920 cameras over USB 3.0 utilizing different recording methods

The accelerated Gstreamer method allowed for hardware-based JPEG compression for the incoming camera data. The API also allowed for choosing the level of quality for compression, set by a number between 0 to 100. As it was observed that the image size could be kept as high as 1.3 MB per image or reduced to as low as 25KB without any kind of observable surface level impact on image quality.

6.3 Extrinsic Matrix Calibration Accuracy

Since solving the extrinsic matrix was paramount in our fusion process, accuracy of calibration for the same was also very important. The extrinsic matrix was tuned

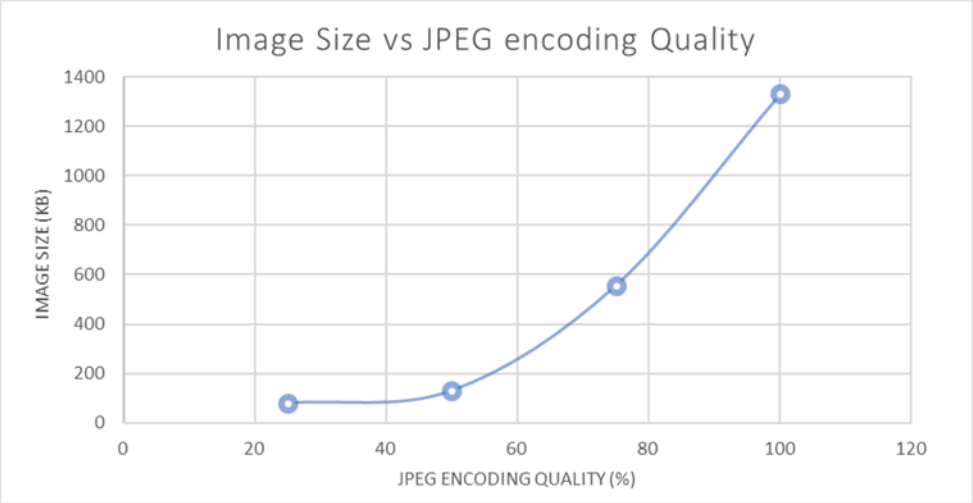


Fig. 6.5. Image size vs encoding quality

manual once the output was obtained via SolvePnP, i.e the Levenberg Marquadt algorithm. The extrinsic matrix, as mentioned earlier, was initially calculated using pixel point pairs along with the pre-calculated intrinsic matrix. It was however noticed that the fusion was not particularly good with the value obtained via SolvePnP and varied greatly depending on the number and accuracy of the pixel point pairs themselves. Thus we further improve the accuracy of the extrinsic matrix by adding and subtracting values to its rotation and translation vectors. The images below show the effects of the tuning can be seen visually in the figure below as the intrinsic matrix with relatively worse fusion can be corrected by using the tuning script.

We performed some ground truth tests with the system to determine the accuracy of the fusion before and after the Manual tuning process. We used a laser scanner to determine ground truth values, then we utilized data from the pixel closest to the point where the ground truth was determined for which we had mapped LIDAR values from the mapping tables that were generated. These LIDAR values were then compared to the ground truth values to generate the table below. It showed that the adding the Manual Tuning Process bettered accuracy of fusion from 8 cm to 3 cm on average assuming equivalent 3D Gaussian noise for both the laser range finder and the

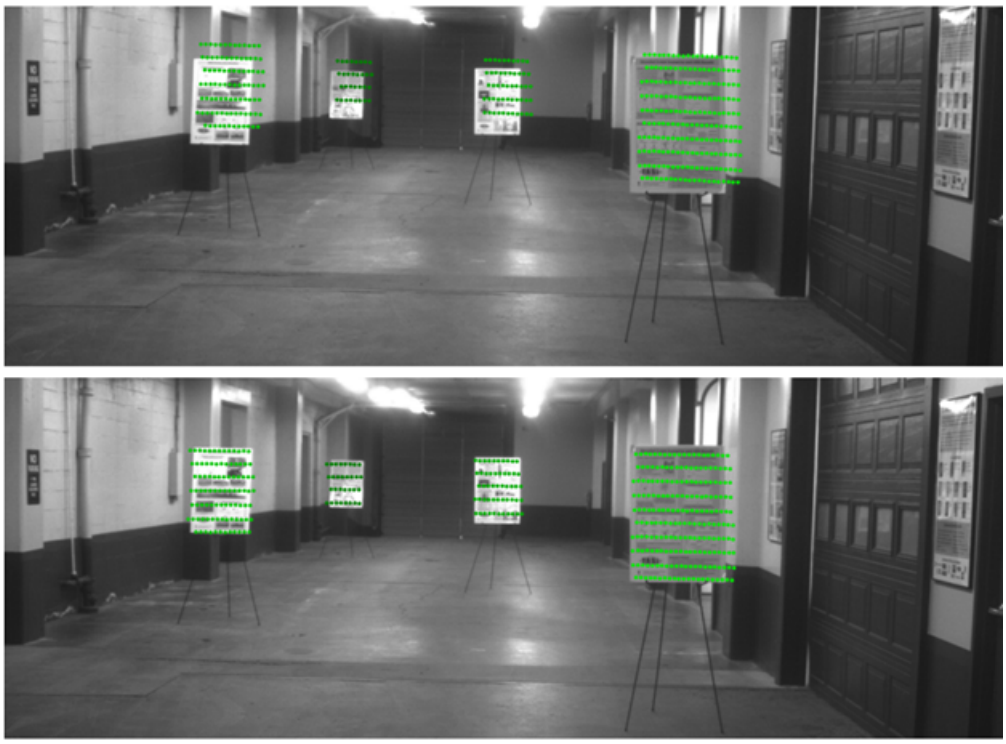


Fig. 6.6. Overlay image showing fusion before and after fine tuning manually

LIDAR. It must be noted that the same white poster as shown in the images above was utilized for the experiment. The evaluation was primarily conducted for fusion accuracy and the error of the distance given by the LIDAR was itself discounted from the equation on account of the error being less than 1 cm which itself was within the bounds of our experimentation error. It must be noted that this 1cm value is in the range of the advertised +/- 5cm error as it depends greatly on the material and color of the object. A ground truth error closer to the +/- 5cm was observed for darker objects, hence the white matted checkerboards were used for the calibration of the sensors and the evaluation.

The values gained by the fusion method used in this thesis was compared to some earlier work done in camera LIDAR calibration in attempt to compare the fusion results. The values obtained in the experiment mentioned earlier were compared to the method by Ankit Dhall et al [22] involving RANSAC along with the EPNP

Table 6.2.
Shows comparison of fusion accuracy differences due to the added manual tuning

Absolute Error w.r.t Ground Truth	SolvePnP	SolvePnP + Manual tuning
Minimum Absolute Error (m)	0.05	0.003
Maximum Absolute Error (m)	0.12	0.06
Average (m)	0.085	0.0315

Table 6.3.
Shows comparison of fusion accuracy based on calculation methods for extrinsic matrix

Absolute Error w.r.t Ground Truth	RANSAC+EPNP	RANSAC+SVD+L-M	SolvePnP+Manual
Minimum Absolute Error (m)	0.009	0.05	0.003
Maximum Absolute Error (m)	0.05	0.5	0.06
Average Absolute Error (m)	0.0295	0.275	0.0315

software an also with the paper written by Z. Pusztai and L. Hajder[24] which involved RANSAC along with single value decomposition (SVD) finally followed by Levenberg-Marquadt algorithm to calculate the extrinsic matrix. It must be noted that 3D - LIDAR utilized by [22] was a Velodyne 32 beam sensor whereas [25] used a Blender simulation of the Velodyne 32 sensor for their experiments and it must be noted that the LIDAR sensor used in this thesis was the Ouster OS1-64 has a much lower accuracy of upto +/-5cm compared to the Velodyne sensor which has a stated accuracy of +/- 2cm. In addition, the effects of the Pustzai paper using a simulation might also not account for certain real-world effects such as the effect of texture and color on reflectivity of the object viewed and hence its accuracy.

The fusion accuracy using the SolvePnP+Manual Correction was comparable with the result obtained from the other methods. Although it must be noted that the other methods were attempting for a more automated approach in finding the extrinsic matrix, whereas the prime purpose for our project was to get higher accuracy at the

expense of some manual labor although it must be noted that elements of our method could be automated with a few changes to the method and the system. The pixel reprojection errors for the selected pixel point pairs were also compared to that of results by [22]. Here the results were better with the average pixel reprojection also being very close to that of the RANSAC + EPNP method. It must be noted that more point pairs were used than the Method used by RANSAC+EPNP which could have helped reduce the error by providing more points to the Solvepnp algorithm but also could have increased the error due to the extra noise introduced by adding extra point-pixel pairs to the equation.

Table 6.4.
Pixel Reprojection Errors between the Extrinsic Matrix Calculation methods

Absolute Error w.r.t Ground Truth	RANSAC + EPNP	SolvePnP + Manual Correction
Number of pixel-point pairs used for calibration	20	40
Pixel Reprojection Errors (in pixels)	0.5759	0.43

6.4 Cartographer SLAM

The Cartographer software developed by Google was used as direct tool in order to generate the 2D maps from the collected LIDAR 3D bag data. The system was designed to work with ROS and developed by Ouster using various launch files to step by step use the system to generate the 2D and 3D probability grids of the environment. The Cartographer system uses a complex set of programs that use various algorithms as shown in the figure below. We desist going into too much detail into the system as it is a massive and complex program that is not the primary subject of the thesis.

The Cartographer system essentially down-samples the LIDAR data using Voxel filter that convert the high amount of LIDAR data into relatively easily processed chunks. The system then runs a scan matching algorithm that tries to match each scan to the next one creating local sub-maps. In order to reduce complexity involved

in scan matching, the system uses the data from the IMU to aid in the scan matching and guess where the scan should be inside the sub-map using orientation data. The system is designed to be configurable for any type of 3D LIDAR/IMU data. The system requires a bag file containing data from the LIDAR and the IMU. The final output being a .pbstream file containing the mapping and trajectory data.

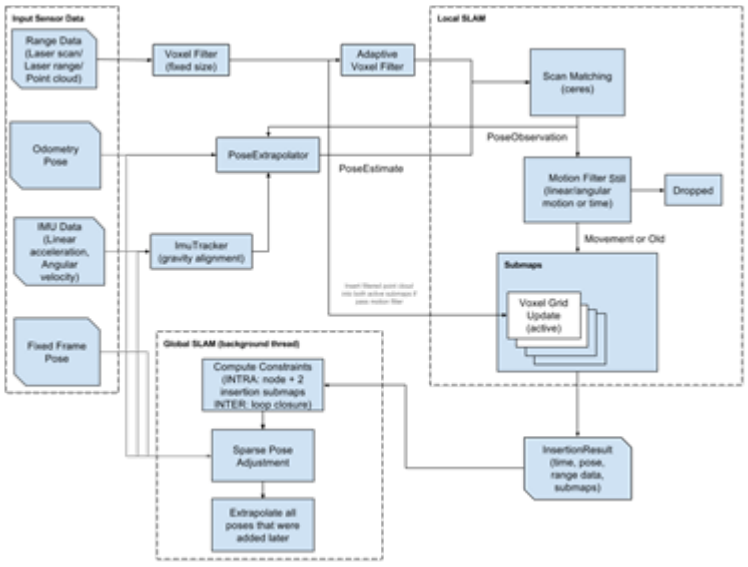


Fig. 6.7. Cartographer Architecture. Source: Google

Every bag file created by ROS contains messages information containing the topic names associated with each message along with other important info including the message data itself. The Cartographer process begins with a Python script that is used to filter the bag files. This pre-processing involves parsing the bag file and removing the leading slashes from every message topic inside the bag file. A Cartographer ROS launch file is then used to validate the bag file to check for any gaps in the data that may exist in the bag file that might make it difficult for cartographer to process the bag file. The system does not make accurate predictions as to the potential problems faced by the Google Cartographer setup. The system only alerts the users as to these gaps in the bag files. Once this validation is complete, the bag

file is then fed to a ROS launch file cart_2d.launch that performs the majority of the Cartographer programs and processes including scan matching and map generation.

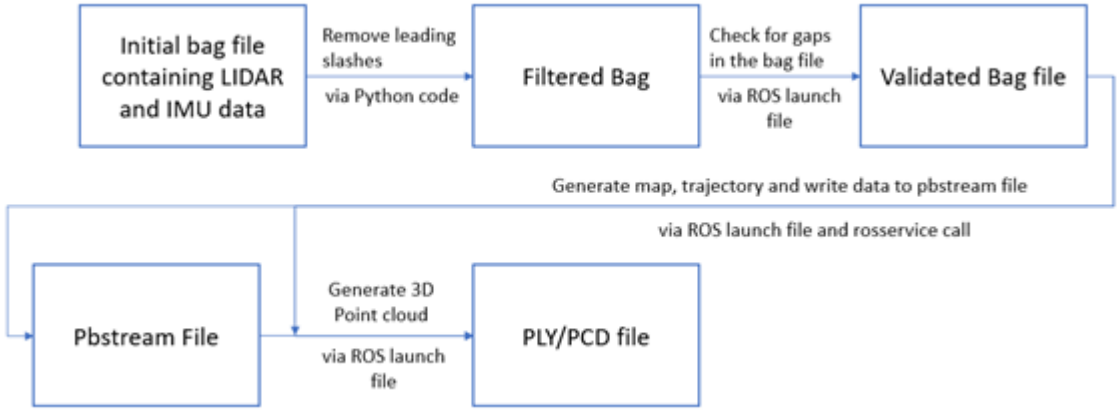


Fig. 6.8. SLAM Cartographer Steps/Execution order

As the launch file processes the bag file, Cartographer also has a rosservice that can be run simultaneously with the cart_2d.launch to generate the pbstream file which contains the map and trajectory information. In addition, Cartographer also provides with certain other launch files that run a program known as assets writer that can be used to use the previously generated pbstream files and the filtered bag file to generate full 3D point cloud maps. These point clouds can be stored in pcd, ply file types and can be viewed using software's such as CloudCompare and Ccviewer.

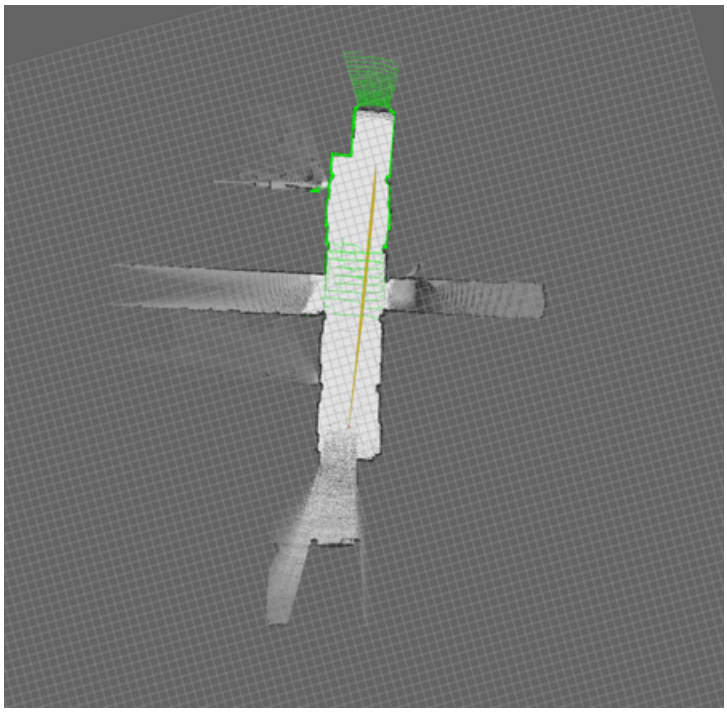


Fig. 6.9. 2D Probability grid showing the map of stutz lab(white and grey) and vehicle trajectory(Yellow line) in the Stutz Lab from collected data

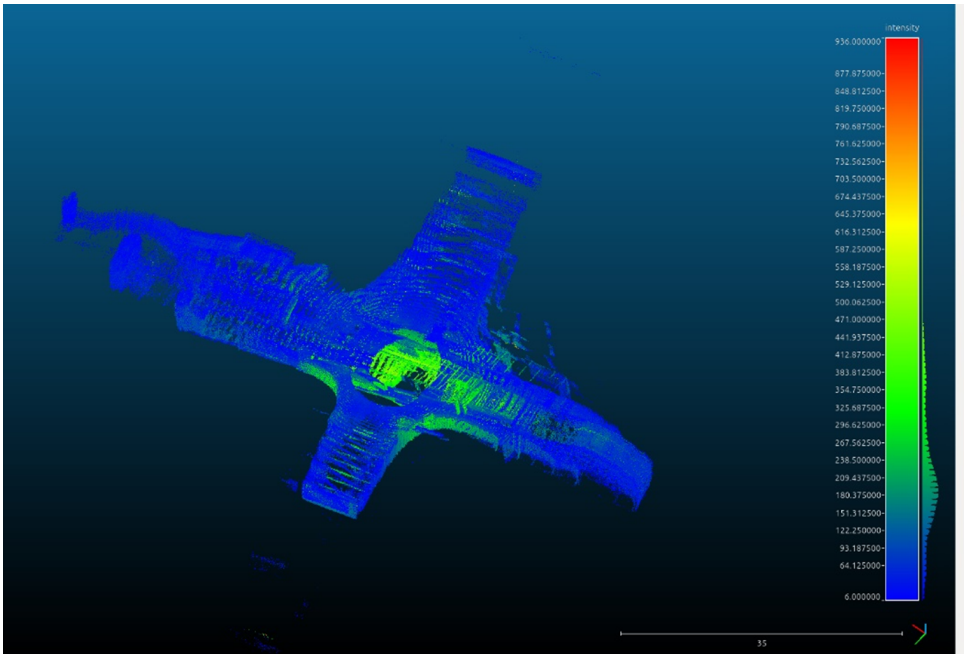


Fig. 6.10. 3D mapping (.ply file) of the Stutz lab viewed using ccviewer from collected data

7. CONCLUSION AND FUTURE WORK

This thesis demonstrates the design and implementation of a unique wearable data acquisition system for e-scooter consisting of LIDAR and cameras collection that can be used as a data collection platform to provide data for autonomous systems of the future. The data collected consists of high-resolution 3D LIDAR data and FHD Camera data from multiple cameras. The thesis delves in detail in the properties of the sensors and why they were selected with both non tangible parameters such as resolution, feature-set, software support, driver availability as well as tangible parameters such as size weight and cost taken into account.

The thesis provides details of the sensor selection for the LIDAR, camera as well as the accompanying embedded system and details of how the system works on a hardware level. We explain the various changes that required to be made to the system in order to achieve the continuous data collection that was possible within the limitations and the constraints placed upon the project due to a myriad of factors. On a software level we explained the various methods used to collect data from the LIDAR sensor and the camera to provide methods for keep track of the data for synchronization purposes utilizing Linux based inotify API. The selection criteria and reasoning of selection of the Gstreamer library over the standard ROS Library for camera data collection to exploit the advantages offered by hardware accelerated Gstreamer NVIDIA's Tx2 platform were also detailed and evaluated.

We used the ROS ouster driver for the LIDAR instead of the standard out of the box Ousterstudio interface applications due to its ability to initiate the recording without the need of a manual user interface as well as allowing us to timestamp the UDP data and store it in a ROS bag file. We also selected ROS for its ability to break down the files and store it in compressed chunks using the ROSBAG API. We used a combination of two programming interfaces, Node-Red (running on Node JS) and

Python to implement a wireless User Interface accessible on a smartphone browser on the same local network. The interface allowed the user to start and stop the data acquisition process and also to check if the system was properly functioning as well as details on the current state of the LIDAR and relayed output to the screen from the ROS terminal.

In the data pre-processing and calibration stage, we used a Python program to convert the LIDAR bag data into individual binary UDP packets and format the names with their UNIX timestamps. We calibrated the camera intrinsic matrix and the distortion parameters using both OpenCV as well as the MATLAB camera calibrator toolkit. This was achieved by feeding in numerous images of checkerboards.

The extrinsic matrix was calculated using the OpenCV SolvePnP functions based on the Levenberg Marqaudt Algorithm. The values of the camera matrix, distortion parameters and tens of LIDAR camera point-pixel pair values were fed to the SolvePnP function which calculated the extrinsic matrix using the Levenberg-Marquadt algorithm. The matrix was then further refined using a script which used the LIDAR camera fusion overlay image and allowed to manually correct the extrinsic matrix to fit the fusion better. The fusion overlay was created by using the fusion equation to find the corresponding pixel values for every LIDAR point in the direction of the camera lens and then iteratively drawing the pixels on the calibration images.

We also went into detail as to the data processing system and the manner in which the program went frame by frame and created the point pixel pair tables as well as the overlay fusion image fusing the collected LIDAR and camera data in the process. The process utilized the pre-calculated intrinsic parameters and the extrinsic matrix to then perform fusion. The thesis also delved into the processing of the UDP binary text data into the Cartesian 3D coordinates as well as methods such as using the using the Cupy library to exploit NVIDIA's GPU hardware acceleration as well as the use of trigonometric tables to speed up the calculation of the millions of sines and cosines encountered per rotation to calculate the 3D coordinate of the LIDAR point values. We then discussed the use of the Cartographer system developed by google to help

convert the LIDAR bag data into a 3D map using the ROS/RVIZ integration. Lastly, we evaluated the system and discussed the problems faced and the improvements made to the system. This involved evaluating the different methods of camera and LIDAR data collection as well as the calibration improvements for the calculation of the intrinsic and extrinsic matrix in allowing us to obtain better fusion.

7.0.1 Future Work

For future work, the system and the scope could be improved in several ways. Utilization of a lighter LIDAR setup could allow for easier use of access and riding while carrying the setup. An even lighter LIDAR and carrying setup would result in a more naturalistic data collection. As a system where the rider is unaware of the setup while riding would lead to more natural riding. The use of cameras that allow for frame triggering via hardware would allow for better synchronization with the rotation of the LIDAR.

This could be primarily be achieved by either some form of software or hardware triggering. In addition, due to the sweeping nature of the LIDAR, frame triggering for the camera would result in an image where the delays between the points in the LIDAR point cloud and the image itself would be relatively well known. This could lead to better fusion on fast moving objects.

A balance from some form of control system to stabilize the camera and LIDAR would also allow for the image data to constantly capture vehicle in view allowing for better capture during moments when the e-scooter rider tilts or shifts his weight in order to maneuver the curb. This addition could greatly enhance the quality of the collected data and might provide for easier analysis of the fusion over a period of time as it would be easier to analyze the fusion video for a camera video that remains smoothly attached. Connection to the stabilizing system would also reduce the torques experienced by the camera reducing chances of mechanical failure.

It would also help reduce weight if the battery setup were moved by some means to the e-scooter itself. This would remove the extra battery weight that the rider would have to other wise carry. Although this system would have to be designed to make sure to allow for easy disconnection between the rider and the device with the battery system in event of an accident. A wire dangling from the e-scooter connecting to the rider could be detrimental or dangerous to the rider.

A greater coverage of the surroundings would also be a useful addition for the future. The LIDAR presently cannot cover all 360°, however, if the LIDAR is somehow mounted on the head so as to not be blocked by the body of the person, it would get a greater coverage. The application of an additional LIDAR on the back might also allow for greater LIDAR coverage. Addition of extra cameras to cover a greater field of view might also help data analysis and fusion.

The use of MIPI cameras might also allow for adding extra cameras to cover a greater view of the cameras allowing to capture data to the sides of the riders direction of travel as well as in the opposite direction. The use of MIPI cameras also might help cut down slightly on the weight of the USB cables from the cameras. Additional camera and LIDAR data would help in covering a greater number of scenarios occurring on either side of the e-scooter as opposed to a single direction.

Incorporation of additional sensors into the setup would also help provide greater functionality. Additions of sensors such as RADAR and GPS could allow for combining LIDAR trajectories with trajectories from the INS system to allow for better understanding of where the interactions of interest take place withing the given geography of an urban area. Although outside the scope of our current project, such data would be very useful.

Although the LIDAR sensor mounted on the system has an IP rating, the rest of the system does not. If the system can be sealed shut to disallow the entry of water, the system could be useful during rainy weather as well. Since e-scooter injuries can happen on slippery roads as well, this addition would greatly increase the diversity of scenarios for data collection.

Automation of the manual fusion process would also greatly reduce calibration time. Employment of methods like RANSAC and edge detection if they could be applied without a decrease in calibration accuracy would be helpful as well. Application of other fusion methods such as Singular visual odometry (SVO) might allow for additional reference trajectories that could be compared/fused with the LIDAR trajectory for greater accuracy.

REFERENCES

REFERENCES

[1] Pomerleau, Dean. “ALVINN: An Autonomous Land Vehicle in a Neural Network.” NIPS (1988).

[2] A. Frome et al., "Large-scale privacy protection in Google Street View," 2009 IEEE 12th International Conference on Computer Vision, Kyoto, 2009, pp. 2373-2380, doi: 10.1109/ICCV.2009.5459413.

[3] “StreetSide: Dynamic Street-Level Imagery - Bing Maps.” StreetSide: Dynamic Street-Level Imagery - Bing Maps, 12 Feb. 2020, www.microsoft.com/en-us/maps/streetside. doi: 10.1109/ICTIS.2017.8047822

[4] A. Geiger, P. Lenz and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," 2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, 2012, pp. 3354-3361. doi: 10.1109/CVPR.2012.6248074

[5] Team, Waymo. “Waymo Open Dataset: Sharing Our Self-Driving Data for Research.” Medium, Waymo, 21 Aug. 2019, medium.com/waymo/waymo-open-dataset-6c6ac227ab1a.

[6] D. L. Rosenband, "Inside Waymo's self-driving car: My favorite transistors," 2017 Symposium on VLSI Circuits, Kyoto, 2017, pp. C20-C22.

[7] B. Hurl, K. Czarnecki and S. Waslander, "Precise Synthetic Image and LIDAR (PreSIL) Dataset for Autonomous Vehicle Perception," 2019 IEEE Intelligent Vehicles Symposium (IV), Paris, France, 2019, pp. 2522-2529. doi: 10.1109/IVS.2019.8813809

[8] Blankenau, Isaac Zolotor, Daniel Choate, Matthew Jorns, Alec and Homann, Quailan and Depcik, Christopher. (2018). Development of a Low-Cost LIDAR System for Bicycles. 10.4271/2018-01-1051.

[9] B. He, L. Guangsen, W. Huawei, F. Jia, G. Bo and Y. Hongtao, "Design of multichannel data acquisition system based on Ethernet," 2017 IEEE 17th International Conference on Communication Technology (ICCT), Chengdu, 2017, pp. 692-695, doi: 10.1109/ICCT.2017.8359725.

[10] Song Gu and Zhou Hulin, "The design of image acquisition and display system" 2010, 2nd International Conference on Education Technology and Computer,
Shanghai, 2010, pp. V5-23-V5-26, doi: 10.1109/ICETC.2010.5529954.

[11] S. Esquembri et al., "Hardware timestamping for image acquisition system based on FlexRIO and IEEE 1588 v2 standard," 2014 19th IEEE-NPSS Real Time Conference, Nara, 2014, pp. 1-1, doi: 10.1109/RTC.2014.7097499.

[12] C. Yao-yu, L. Yong-lin, L. Ying and W. Shi-qin, "Design of image acquisition and storage system based on ARM and embedded," 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), Yichang, 2012, pp. 981-984, doi: 10.1109/CECNet.2012.6202208.

[13] Y. Chai and J. Xu, "Design of Image Acquisition and Transmission System Based on STM32F407," 2018 2nd IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Xi'an, 2018, pp. 1085-1089, doi: 10.1109/IMCEC.2018.8469227.

[14] G. Sundari, T. Bernatin and P. Somani, "H. 264 encoder using Gstreamer," 2015 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015], Nagercoil, 2015, pp. 1-4, doi: 10.1109/ICCPCT.2015.7159511.

[15] K. Lim, G. S. Kim, S. Kim and K. Baek, "A multi-lane MIPI CSI receiver for mobile camera applications," in IEEE Transactions on Consumer Electronics, vol. 56, no. 3, pp. 1185-1190, Aug. 2010, doi: 10.1109/TCE.2010.5606244.

[16] U. K. Malviya, A. swain and G. Kumar, "Tiny I2C Protocol for Camera Command Exchange in CSI-2: A Review," 2020 International Conference on Inventive Computation Technologies (ICICT), Coimbatore, India, 2020, pp. 149-154, doi: 10.1109/ICICT48043.2020.9112536.

[17] Y. Lu, Z. Chen and P. Chang, "Low power multi-lane MIPI CSI-2 receiver design and hardware implementations," 2013 IEEE International Symposium on Consumer Electronics (ISCE), Hsinchu, 2013, pp. 199-200, doi: 10.1109/ISCE.2013.6570183.

[18] Liu, Kaikai, and Rajathswaroop Mulky. "Enabling Autonomous Navigation for Affordable Scooters." Sensors (Basel, Switzerland), MDPI, 5 June 2018, www.ncbi.nlm.nih.gov/pmc/articles/PMC6022038/.

[19] Cheung, Danny. “Mapping Stories with a New Street View Trekker.” Google, 18 Dec. 2018, www.blog.google/products/maps/mapping-stories-new-street-view-trekker/.

[20] X. Liang et al., ”Forest Data Collection Using Terrestrial Image-Based Point Clouds From a Handheld Camera Compared to Terrestrial and Personal Laser Scanning,” in IEEE Transactions on Geoscience and Remote Sensing, vol. 53, no. 9, pp. 5117-5132, Sept. 2015. doi: 10.1109/TGRS.2015.2417316

[21] B. Madore, G. Imahori, J. Kum, S. White and A. Worthem, ”NOAA’s use of remote sensing technology and the coastal mapping program,” OCEANS 2018 MTSIEEE Charleston, Charleston, SC, 2018, pp. 1-7. doi: 10.1109/OCEANS.2018.8604932

[22] L. Wang, L. Zhang and Y. Ma, ”Gstreamer accomplish video capture and coding with PyGI in Python language,” 2017 First International Conference on Electronics Instrumentation Information Systems (EIIS), Harbin, 2017, pp. 1-4.

[23] L. Zheng and Y. Fan, ”Data packet decoder design for LiDAR system,” 2017 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-TW), Taipei, 2017, pp. 35-36, doi: 10.1109/ICCE-China.2017.7990982.

[24] S. Gargoum and K. El-Basyouny, ”Automated extraction of road features using LiDAR data: A review of LiDAR applications in transportation,” 2017 4th International Conference on Transportation Information and Safety (ICTIS), Banff, AB, 2017, pp. 563-574.

[25] Y. Zhang, J. Wang, X. Wang and J. M. Dolan, ”Road-Segmentation-Based Curb Detection Method for Self-Driving via a 3D-LiDAR Sensor,” in IEEE Transactions on Intelligent Transportation Systems, vol. 19, no. 12, pp. 3981-3991, Dec. 2018.

[26] X. Wang, P. Ma, L. Jiang, L. Li and K. Xu, ”A New Method of 3D Point Cloud Data Processing in Low-speed Self-driving Car,” 2019 IEEE 3rd Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Chongqing, China, 2019, pp. 69-73.

[27] R. Sahba, A. Sahba, M. Jamshidi and P. Rad, ”3D Object Detection Based on LiDAR Data,” 2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON), New York City, NY, USA, 2019, pp. 0511-0514.

[28] C. Xie, H. Shishido, Y. Kameda and I. Kitahara, ”A Projector Calibration Method Using a Mobile Camera for Projection Mapping System,” 2019 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct), Beijing, China, 2019, pp. 261-262, doi: 10.1109/ISMAR-Adjunct.2019.00-33.

[29] Y. Ito and Y. Oda, "Estimation of Camera Projection Matrix Using Linear Matrix Inequalities," 2016 Joint 8th International Conference on Soft Computing and Intelligent Systems (SCIS) and 17th International Symposium on Advanced Intelligent Systems (ISIS), Sapporo, 2016, pp. 72-75, doi: 10.1109/SCIS-ISIS.2016.0028.

[30] Y. Lang, H. Wu, T. Amano and Q. Chen, "An iterative convergence algorithm for single/multi ground plane detection and angle estimation with RGB-D camera," 2015 IEEE International Conference on Image Processing (ICIP), Quebec City, QC, 2015, pp. 2895-2899, doi: 10.1109/ICIP.2015.7351332.

[31] J. Kannala and S. S. Brandt, "A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 28, no. 8, pp. 1335-1340, Aug. 2006, doi: 10.1109/TPAMI.2006.153.

[32] W. Song, Z. Miao and H. Wu, "Automatic calibration method based on improved camera calibration template," 5th IET International Conference on Wireless, Mobile and Multimedia Networks (ICWMMN 2013), Beijing, 2013, pp. 301-305, doi: 10.1049/cp.2013.2429.

[33] A. Fetić, D. Jurić and D. Osmanković, "The procedure of a camera calibration using Camera Calibration Toolbox for MATLAB," 2012 Proceedings of the 35th International Convention MIPRO, Opatija, 2012, pp. 1752-1757.

[34] A. Bushnevskiy, L. Sorgi and B. Rosenhahn, "Multimode camera calibration," 2016 IEEE International Conference on Image Processing (ICIP), Phoenix, AZ, 2016, pp. 1165-1169, doi: 10.1109/ICIP.2016.7532541.

[35] Y. Dong, X. Ye and X. He, "A novel camera calibration method combined with calibration toolbox and genetic algorithm," 2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA), Hefei, 2016, pp. 1416-1420, doi: 10.1109/ICIEA.2016.7603807.

[36] P. Liu, J. Zhang and K. Guo, "A Camera Calibration Method Based on Genetic Algorithm," 2015 7th International Conference on Intelligent Human-Machine Systems and Cybernetics, Hangzhou, 2015, pp. 565-568, doi: 10.1109/IHMSC.2015.246.

[37] L. Liu, S. Cao, X. Liu and T. Li, "Camera Calibration Based on Computer Vision and Measurement Adjustment Theory," 2018 Eighth International Conference on Instrumentation Measurement, Computer, Communication and Control (IMCCC), Harbin, China, 2018, pp. 671-676, doi: 10.1109/IMCCC.2018.00145.

[38] D. H. Lee, S. S. Lee, H. H. Kang and C. K. Ahn, "Camera Position Estimation for UAVs Using SolvePnP with Kalman Filter," 2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN), Shenzhen, 2018, pp. 250-251.

[39] Y. Yenıaydin and K. W. Schmidt, "Sensor Fusion of a Camera and 2D LIDAR for Lane Detection," 2019 27th Signal Processing and Communications Applications Conference (SIU), Sivas, Turkey, 2019, pp. 1-4.

[40] J. Li, X. He and J. Li, "2D LiDAR and camera fusion in 3D modeling of indoor environment," 2015 National Aerospace and Electronics Conference (NAECON), Dayton, OH, 2015, pp. 379-383.

[41] LiDAR-Camera Calibration using 3D-3D Point correspondences Ankit Dhall, Kunal Chelani, Vishnu Radhakrishnan, K.M. Krishna (Submitted on 27 May 2017)

[42] Velas, Martin et al. "Calibration of RGB camera with velodyne LiDAR." (2014).

[43] A. Rangesh and M. M. Trivedi, "No Blind Spots: Full-Surround Multi-Object Tracking for Autonomous Vehicles Using Cameras and LiDARs," in IEEE Transactions on Intelligent Vehicles, vol. 4, no. 4, pp. 588-599, Dec. 2019, doi: 10.1109/TIV.2019.2938110. https://arxiv.org/abs/1802.08755

[44] F. Itami and T. Yamazaki, "An Improved Method for the Calibration of a 2-D LiDAR With Respect to a Camera by Using a Checkerboard Target," in IEEE Sensors Journal, vol. 20, no. 14, pp. 7906-7917, 15 July15, 2020, doi: 10.1109/JSEN.2020.2980871.

[45] Z. Hu, Y. Hu, Y. Li and G. Huang, "Registration of image and 3D LIDAR data from extrinsic calibration," 2015 International Conference on Transportation Information and Safety (ICTIS), Wuhan, 2015, pp. 102-106, doi: 10.1109/ICTIS.2015.7232189.

[46] J. -K. Huang and J. W. Grizzle, "Improvements to Target-Based 3D LiDAR to Camera Calibration," in IEEE Access, vol. 8, pp. 134101-134110, 2020, doi: 10.1109/ACCESS.2020.3010734.

[47] Y. Lyu, L. Bai, M. Elhousni and X. Huang, "An Interactive LiDAR to Camera Calibration," 2019 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2019, pp. 1-6, doi: 10.1109/HPEC.2019.8916441.

[48] B. Fu, Y. Wang, X. Ding, Y. Jiao, L. Tang and R. Xiong, "LiDAR-Camera Calibration Under Arbitrary Configurations: Observability and Methods," in IEEE Transactions on Instrumentation and Measurement, vol. 69, no. 6, pp. 3089-3102, June 2020, doi: 10.1109/TIM.2019.2931526.

[49] C. Hsu, H. Wang, A. Tsai and C. Lee, "Online Recalibration of a Camera and Lidar System," 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Miyazaki, Japan, 2018, pp. 4053-4058, doi: 10.1109/SMC.2018.00687.

[50] G. Lee, J. Lee and S. Park, "Calibration of VLP-16 Lidar and multi-view cameras using a ball for 360 degree 3D color map acquisition," 2017 IEEE International

Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI), Daegu, 2017, pp. 64-69, doi: 10.1109/MFI.2017.8170408.

[51] V. John, Q. Long, Z. Liu and S. Mita, "Automatic calibration and registration of lidar and stereo camera without calibration objects," 2015 IEEE International Conference on Vehicular Electronics and Safety (ICVES), Yokohama, 2015, pp. 231-237, doi: 10.1109/ICVES.2015.7396923.

[52] M. A. Zaiter, R. Lherbier, G. Faour, O. Bazzi and J. C. Noyer, "3D LiDAR Extrinsic Calibration Method using Ground Plane Model Estimation," 2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE), Graz, Austria, 2019, pp. 1-6, doi: 10.1109/ICCVE45908.2019.8964949.

[53] R. Ravi, Y. Lin, M. Elbahnasawy, T. Shamseldin and A. Habib, "Simultaneous System Calibration of a Multi-LiDAR Multicamera Mobile Mapping Platform," in IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, vol. 11, no. 5, pp. 1694-1714, May 2018, doi: 10.1109/JSTARS.2018.2812796.

[54] Z. Pusztai and L. Hajder, "Accurate Calibration of LiDAR-Camera Systems Using Ordinary Boxes," 2017 IEEE International Conference on Computer Vision Workshops (ICCVW), Venice, 2017, pp. 394-402, doi: 10.1109/ICCVW.2017.53.

[55] S. Kružic, J. Musić, I. Stančić and V. Papić, "Influence of Data Collection Parameters on Performance of Neural Network-based Obstacle Avoidance," 2018 3rd International Conference on Smart and Sustainable Technologies (SpliTech), Split, 2018, pp. 1-6.

[56] Selby, Wil. "Building Maps Using Google Cartographer and the OS1 Lidar Sensor." Ouster, 9 Oct. 2019, ouster.com/blog/building-maps-using-google-cartographer-and-the-os1-lidar-sensor

[57] Holger, Ceaser, et al. NuScenes: A Multimodal Dataset for Autonomous Driving. arxiv.org/pdf/1903.11027.pdf

[58] K. Oishi, S. Mori and H. Saito, "An Instant See-Through Vision System Using a Wide Field-of-View Camera and a 3D-Lidar," 2017 IEEE International Symposium on Mixed and Augmented Reality (ISMAR-Adjunct), Nantes, 2017, pp. 344-347.

[59] S. Gatesichapakorn, J. Takamatsu and M. Ruchanurucks, "ROS based Autonomous Mobile Robot Navigation using 2D LiDAR and RGB-D Camera," 2019 First International Symposium on Instrumentation, Control, Artificial Intelligence, and Robotics (ICA-SYMP), Bangkok, Thailand, 2019, pp. 151-154.

[60] X. Zuo, P. Geneva, W. Lee, Y. Liu and G. Huang, "LIC-Fusion: LiDAR-Inertial-Camera Odometry," 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Macau, China, 2019, pp. 5848-5854.

[61] T. Dieterle, F. Particke, L. Patino-Studencki and J. Thielecke, "Sensor data fusion of LIDAR with stereo RGB-D camera for object tracking," 2017 IEEE SENSORS, Glasgow, 2017, pp. 1-3.

[62] M. Y. Lachachi, M. Ouslim, S. Niar and A. Taleb-Ahmed, "LIDAR and Stereo-Camera fusion for reliable Road Extraction," 2018 30th International Conference on Microelectronics (ICM), Sousse, Tunisia, 2018, pp. 64-67.

[63] L. Zheng and Y. Fan, "Data packet decoder design for LiDAR system," 2017 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-TW), Taipei, 2017, pp. 35-36.

[64] W. Xuan, Z. Huang, X. Chen and Z. Lin, "A combined sensor system of digital camera with LiDAR," 2007 IEEE International Geoscience and Remote Sensing Symposium, Barcelona, 2007, pp. 589-592.

[65] Y. Wu and C. Tsai, "Pedestrian, bike, motorcycle, and vehicle classification via deep learning: Deep belief network and small training set," 2016 International Conference on Applied System Innovation (ICASI), Okinawa, 2016, pp. 1-4. doi: 10.1109/ICASI.2016.7539822

[66] B. He, L. Guangsen, W. Huawei, F. Jia, G. Bo and Y. Hongtao, "Design of multichannel data acquisition system based on Ethernet," 2017 IEEE 17th International Conference on Communication Technology (ICCT), Chengdu, 2017, pp. 692-695, doi: 10.1109/ICCT.2017.8359725.

[67] Xu Lijun, Gao Guohong, Li Xueyong and Qu Peixin, "Video collection system based on embedded system and USB communication," 2010 Second International Conference on Communication Systems, Networks and Applications, Hong Kong, 2010, pp. 112-114, doi: 10.1109/ICCSNA.2010.5588759.

[68] C. Yao-yu, L. Yong-lin, L. Ying and W. Shi-qin, "Design of image acquisition and storage system based on ARM and embedded," 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), Yichang, 2012, pp. 981-984, doi: 10.1109/CECNet.2012.6202208.

[69] Jian Yuan and DongSheng Yin, "Wireless camera based on ARM11," 2011 International Conference on Computer Science and Service System (CSSS), Nanjing, 2011, pp. 1887-1890, doi: 10.1109/CSSS.2011.5974122.

[70] Lu and P. Li, "A Data Acquisition and 2-D Flow Measuring Technology in Agricultural Spray Field Based on High Speed Image Processing," 2009 International Conference on Measuring Technology and Mechatronics Automation, Zhangjiajie, Hunan, 2009, pp. 446-449, doi: 10.1109/ICMTMA.2009.122.

[71] Song Gu and Zhou Hulin, "The design of image acquisition and display system," 2010 2nd International Conference on Education Technology and Computer, Shanghai, 2010, pp. V5-23-V5-26, doi: 10.1109/ICETC.2010.5529954.