Universiteit Leiden
The Netherlands

Master Computer Science

A research platform for autonomous navigation of pedestrian spaces.

Name: Mees Delzenne
Student Id: s1531255
Date: 04/06/2023

1st supervisor: Erwin M. Bakker
2st supervisor: Michael S. Lew

MSc Thesis computer science

Leiden University
Niels bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

In this paper we study the problem of autonomous driving in pedestrian environments. We focus on End-to-End models trained using RGB monocular images for ego-position estimation and waypoint driving. A mobile platform was designed and implemented that was able to capture RGB monocular images as well as precise RTK-GNSS location data while driving in a pedestrian environment. This platform was used to construct a dataset with applications for research on autonomous driving in pedestrian spaces. The dataset is used for two experiments on waypoint driving as well as ego-position estimation. For each of these two tasks we trained and validated a baseline neural network. The resulting experiments show interesting preliminary results for both tasks in pedestrian environments.

Contents

1 Intro

Over the last decade interest in autonomously driving cars, by both the general public as well as research, has greatly increased, in part due to various companies promising to deliver fully autonomous driving cars in the coming years. As a result we have seen many papers detailing new methods for the various tasks that are required for autonomous driving; like object detection [1], localization [2] and trajectory planning [3]. Other researches try, instead of training various models for there own tasks, to train a single model capable of autonomous driving in an End-to-End manner [4]. While commercial methods seem to favor creating autonomous driving systems by combining multiple models [5], End-to-End learning seems to be used more often on platforms with limited computational power as End-to-End models only require running a single model instead of multiple models.

Research has already provided various methods for training End-to-End autonomous driving models, however these models are often trained and tested in a simulator. Simulators have their advantages as they are easy to setup and require little effort for collecting data. However because models trained in a simulator can have drastically different performance in the real world [6], it also remains important to train and test models in the real world. However, this is not always as straight forward as applying the methods used in the simulator to data from the real world. For example, methods which use reinforcement learning to train a model rely implicitly on the ability of a simulator to be able to crash the vehicle without incurring any cost or pose danger to other road user, which would be prohibited in the real world [7]. Other methods might use other features of the simulator which are hard to achieve in the real world. For instance, World on Rails [4] uses extremely precise location information to train a forward model of the car. The use of simulator features which do not translate well to the real-world, in combination with unpredictable generalizability of models trained in simulators make these methods less widely applicable than methods which are trained on real-world data. However, collecting data and testing in the real world is often more involving and more time consuming as it requires a physical platform.

Many researches in the field of autonomous driving target a car driving on the road, ignoring other spaces where vehicles may navigate, such as pedestrian spaces: public walkways, sidewalks and cycle paths. These spaces provide there own challenges, as walkways are often less distinct than roads without clear, uniform, markings. Furthermore, vehicles navigating public walkways would also need to be able to handle pedestrians which often behave in a less predictable manner than cars as they do not have to follow designated lanes.

In this paper we focus on supervised End-to-End learning for autonomous driving in pedestrians spaces as well as ego-position estimation for these kind of spaces. Specifically we present the design and implementation of a low-cost platform which is capable of capturing real-world datasets in pedestrians spaces that can be used to train and test autonomous driving methods in the real world. Our goal is to create a platform which has both the sensors and actuators to be able to autonomously drive, as well as sensors to be able to establish a ground truth and properly measure the performance of trained models.

There is much research into another related and important problem: tracking the position of a vehicle in a given environment. The most common setting of this research problem seems to be in simultaneous localization and mapping or SLAM. SLAM methods try to estimate the difference in orientation and location between multiple frames of sensor data. This problem is similar but not equivalent to our problem of ego-position estimation. With ego-position estimation we mean the

task of estimating a global position from a set of sensor data. SLAM methods produce relative positions: All positions generated are relative to some starting position which is often the position of the first frame of data. Ego-position estimation aims to estimate the coordinates of the position of a frame or window of sensor data independent from the order of frames or windows.
In order to estimate the performance of models which where trained for autonomous driving, performance measures are required. There are several ways to measure the performance of an autonomously driving vehicle, but for our platform we found that the best way to measure performance was by tracking the position of the vehicle during autonomous driving and compare this to where the vehicle was positioned in it's training data. For this we choose to add a high precision GPS module to the platform to be able to track the position of the car.
Basic GPS setups, with single-frequency receivers, are able to attain an accuracy of roughly 2 meters 95% of the time[8]. This imprecision would mean that we cannot determine the difference between a model preforming well and a model performing poorly, as roads in pedestrians spaces are often less than two meters wide. A position measurement with an accuracy of more then two meters could not differentiate between whether the car is on or off the road. In order to improve the accuracy of GPS GNSS systems it is possible on certain receivers to augment the measurements with additional data using Real Time Kinematics (RTK).
Our platform is equipped with an RTK system and is able to track its own position to single centimeter accuracy.
Using this platform we created a real world dataset with applications for learning autonomous driving models in pedestrian spaces. This dataset contains roughly 35.000 images accounting for more then 3 hours of driving. The images are accompanied by controlling input (steering and acceleration input), precise position information (GNSS RTK), as well as speed measurements.

2 Related Work

Research into the problem of autonomous driving has been active since at least 1989 when the first paper presenting an autonomous driving method using neural networks was published [9]. ALVINN, the vehicle presented in the paper, was an autonomous driving car using a small neural net which was capable following a road by itself. Since then, as computers have become more capable, we have seen an increase in autonomous driving papers. The field of autonomous driving is varied, with many different methods, sensors, and vehicles used. One common approach is to break the task of autonomous driving into a variety of different tasks solved with different models. A few of these possible tasks are detecting interesting regions in sensor data. This can be predicting a 2D bounding box in RGB image data [1] [10] or regions of pixels [11] [12], or with LiDAR or other point cloud sensors predicting 3D bounding boxes [13] or a 3D representation of the detected region [14]. Next is the task of predicting the position, orientation and velocity of vehicles called localization [2], and finally, for tasks relevant to our thesis, we have trajectory planning, where the task is to find a good trajectory through an environment [3] [15]. Anyone looking to build an autonomous driving system using the methods presented in these papers would need to combine several of these methods to create a system that is capable of driving autonomously. The current state of the art seems to be a method which combines multiple different systems to create a single autonomous driving system [16].
An alternative to this modular approach is the End-to-End approach as popularized by Nvidia

[17]. Instead of combining multiple models, End-to-End driving tries to train a single model into doing the full range of tasks for autonomous driving. Also here, there is a wide variety of sensors and methods used. Some methods are using reinforcement learning [18] [4] and are trained in a simulator [19], allowing a model to freely make mistakes without incurring costs. There are also some researches that propose methods that are trained on real world data, a dataset generated with their own platform [20] [17]. Training an End-to-End model can be done by creating a model from scratch [17], but it might also be useful to instead base the network model on an existing, pretrained, model [4]. One of the classes of networks, often used as the basis for training, are Imagenet models: networks trained to classify images [21] [22]. Imagenet [23] itself is a dataset which is often used to train and test these models. Throughout this paper we often refer to 'Imagenet networks', with this we mean any network trained for the task of image classification often using the Imagenet dataset. While some papers choose to use older, less accurate, but more commonly known networks as their basis [4], it seems to be the case that when choosing a pretrained Imagenet model as a basis for a specialized model picking the model with the highest accuracy during pre training also seems to lead to better accuracy once specialized on the data at hand [24].

There is a wide variety of datasets with applicability in autonomous driving [25] [26]. These datasets are often from the point of view of a car driving on a road and are frequently used in the training of the sensory tasks of autonomous driving, like object detection. Usage in End-to-End driving seems less frequent as most End-to-End methods seem to either implement their own platform [17] or use a simulator [4], of which CARLA [19] is one of the most popular simulator. CARLA is a driving simulator aimed at providing a platform for research in autonomous driving. CARLA is implemented in a recent game engine and thus is capable of producing photo-realistic data. However, as realistic as CARLA is, training in a simulator does not necessarily transfer to the real world due to a lack of adequate fidelity and environmental modeling. This can cause model failure to only be exposed in real-world conditions and scenario's [6]. This difference between simulator and real-world performance makes it important to train and test in real-world environments. In order to facilitate real-world training, we need a platform capable of capturing data and running models in the real-world. As using full size cars for training autonomous driving methods could be either rather dangerous or prohibitively expensive, there are a range of small scale driving platforms available like Deepracer[27], a small pre-built platform from Amazon, or Donkey car [28] and Jetracer [29] which are open-source platforms made from separately bought components.

Most work in the field of autonomous driving is focused on full sized vehicles on the road as apposed to navigating pedestrians spaces like sidewalks or parks. These spaces could present additional challenges as vehicles would have to drive themselves in close proximity to pedestrians. Autonomous driving techniques that are able to drive in these spaces could have a variety of use-cases in for-instance delivery and autonomous mobility vehicles. Some work has already been done in researching vehicles that are able to navigate these spaces. A lot of this research focuses on modeling the behaviour of pedestrians, which is useful for autonomous navigation to avoid collisions with pedestrians [30]. There are also some papers presenting methods for navigation among pedestrians [31] as well as models for detecting sidewalks [32]. Furthermore, there are several datasets that target this space [33] [34]. However we are unaware of any paper presenting End-to-End trained models using a dataset captured in a pedestrian space.

Ego-position estimation has a lot of overlap with the field of SLAM. SLAM methods vary in the types of signals they use such as LiDAR data, IMU measurements, single monocular RGB images, RGB-D images, or stereo RGB images [35]. SLAM which uses RGB images, also called Visual-SLAM

can be implemented using classical methods [36] but also using deep learning methods [37]. The field of research into SLAM is related to the problem of ego-position estimation. However it does have a few significant differences. Research into SLAM tries to create a general method which can estimate differences in position and orientation between two frames of data for all possible locations, while ego-position estimation tries to estimate a single global position for each frame of data, independent of other frames. We found current research into this specific problem to be relatively sparse as we could not find research specifically addressing this problem. To our knowledge this is one of the first studies addressing this problem in pedestrian space. Acknowledgment to Marc Hilbert who pointed us to this problem.

3 Fundamentals

In this section we give a short overview of terms and technologies used for the design and implementation of our mobile robotic platform as well as techniques that are introduced in the methods section.

3.1 End-to-End Autonomous Driving

The first research problem we focus on is End-to-End autonomous driving. With End-to-End we understand a single model, in our case a single neural net, which can take as input images, speed measures, navigation commands, and possible available sensor data and produces controlling output. In our case our model takes only image data and produces acceleration and steering.
End-to-End autonomous-driving is different from the multi model approaches we often see in the current approaches used in experimental autonomous-driving cars. These approaches often use a combination of several trained models which all have different tasks like processing sensor input, predicting trajectories, and planning actions. These multi-model approaches are however more difficult to implement have lots of moving parts and require often more processing power then a single model. They often require more computational power than currently a mobile platform is capable of.
Autonomous driving methods can differ in what they understand as autonomous driving. Most models that are for road going vehicles try to implement a method where a model is fed instructions from a GPS navigation system. These systems tell the car which lane to take and which turn to make. It is up to the model to handle steering the car to follow the commands. Pedestrian spaces might not always have maps as accurate as roads have. Furthermore, there are no lanes in pedestrian spaces. Therefore our preliminary experiments focus on waypoint driving, where a set of waypoints, GPS coordinates are given, that specify a route. The vehicle would then be tasked with autonomously navigating the pedestrian space, driven, from waypoint to waypoint while completing a route.

3.2 Ego-postion estimation

.
The second research problem we focus on is Ego-postion estimation. With Ego-postion estimation we mean the task of a model to predict its own position in a space from a single frame or window

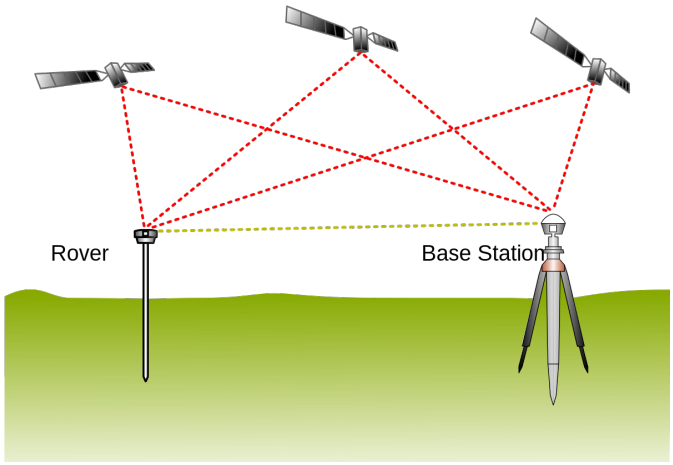

Figure 1: An RTK setup. (image source [38])

of sensor data. In the case of our experiment we focus on training a model to estimate its position from RGB image data. While the coordinates used for training and prediction are not global in relation to a global coordinate system but instead local to a local fixed reference point, together with the global coordinates of the reference point they can be translated to global coordinates. During the task of ego-postion estimation the model is given an image from which it should produce the local position coordinates.

3.3 Real Time Kinematic Global Navigation Satellite System (RTK-GNSS)

An RTK system (see Figure 3.3) consists of two GNSS receivers which are capable of measuring the phase of a GNSS signal. One of the receivers acts as a base station which measures signals received from GNSS satellites and sends its measurements over, an often wireless connection, to the second mobile receiver; the rover. The rover uses the received measurements together with its own measurements of signals received from the satellites to generate a precise measure of its position relative to the base station. Using an RTK system it is possible to generate positions accurate to within a centimeter.

The most commonly used RTK setup uses a base station which is at a static location and is connected to the internet to serve connections with possible rovers. The messages created by these stations are often freely available on the internet for use with a rover. There are several limitations with such a setup: Firstly, the station typically only sends messages every second or so, which is not sufficient for our use case. Secondly, for our receiver, the station must be within 20 kilometers of our rover. Unfortunately for our location no station was available within 20 kilometers. Finally, an RTK base station requires a somewhat involved and time consuming setup where it must record its position for several hours in order to acquire precise, global position information. The base station requires this precise position information to be able to generate the correct messages which a rover with RTK requires for global positioning. These three downsides made this setup not feasible for our use case.

The RTK receiver we use is able to act in a moving RTK base station setup. In this mode the base

station does not need to be stationary but instead has the ability to move while connected to the rover. The downside to this setup is that we lose global coordinates and instead get coordinates relative to base station, but as a trade of we gain the ability to quickly create an RTK setup with 1cm local relative positioning accuracy at 10 measurements per second.

3.4 Hardware Setup: The Rover and the Base station

In order to generate our dataset and to validate our models, we created a rover platform which is capable of running relatively complex neural networks for autonomous navigation, collect images, measure speed and position data, with 1cm local accuracy at 10 measurements per second.
Our rover platform (depicted in Figure 2) is derived from the Jetracer platform [29] and uses an off-the-shelf RC car as its base. An overview of all the components and how they are connected can be seen in Figure 3. The rover is controlled, by an onboard controller board, using Pulse Width Modulated (PWM) signals. These signals are fed into a motor controller which then drives the motors. In order to control these motors from our computer module, the RC cars components are connected to a circuit board, which is able to generated PWM signals and can be controlled using an I^2C bus. This I^2C bus is connected to our computer module which is a Nvidia Jetson Nano B1, equipped with a WiFi module for connectivity. The Jetson Nano is a small form factor computer which is capable of running relatively large neural network thanks to its integrated 128-core Maxwell GPU.
The Jetson Nano is connected to several sensors, which are used to generate data for building datasets as well as running trained models. Firstly, a stereo camera is used: the IMX291-83, these camera's are capable of capturing a pair of images in 2x1640x1320 resolution at 30 fps.
Secondly, in order to be able to measure the speed of the car, which might be useful for methods that use a measure of speed as input for there models, a speed sensor is added. The speed sensor, which reads the rotations per second of the motor, is somewhat similar to speedometers in real cars. As our RC car has no clutch, the RPM sensor is connected to the wires of the brushless motor. This sensor sends a pulse each time the motor completes a rotation. In order to read this pulse we used a small programmable board: the Digispark. This board is capable of reading raw pulse signals that are converted into a number. The Digispark is plugged into a USB port of the Jetson Nano from which it communicates over a serial port. The Digispark sends the number of rotations at regular intervals 1/30 seconds over a serial interface to the Jetson.
Finally, the Jetson is also connected to the ZED-F9P GNSS module which, in turn, is connected to a multi-band GNSS antenna which is mounted on the car. This module is RTK (see section 3.3) capable and is used to track the position of the car. In order to control the ZED-F9P we created a software tool set which is tailored to our use-case. During normal use the ZED-F9P provides 10 position updates every second with a precision of a single centimeter. This position is relative to a base station which is connected to a second PC. This PC, in turn, is connected via WiFi to the Jetson Nano for communicating the RTCM messages (messages of the Radio Technical Commission for Maritime Services SC-104 communication protocol for differential GPS) required for RTK.
The whole vehicle is powered by two sets of batteries, one set for the car motors implementation of and one set for the Jetson Nano.
For manually controlling the car during testing and dataset generation, we used a bluetooth controller connected to the Jetson. Processes on the Jetson then read the input from the controller and, in turn, control the motors and steering servo of the car.

Figure 2: A picture of the vehicle. (A) The IMX291-83 stereo camera. (B) A Digispark used to read data from the RPM-meter. (C) The RPM-meter connected to the brushless motor plugs. (D) The Nvidia Jetson Nano. (E) A Multi-band GNSS antenna. (F) The ZED-F9P GNSS module. (G) The base station reciever. (H) A WiFi router for connectivity between base station and rover. (I) A laptop which sends messages from the base station ZED-F9P GNSS module to the rover over UDP. (J) The Bluetooth controller used to control the car.

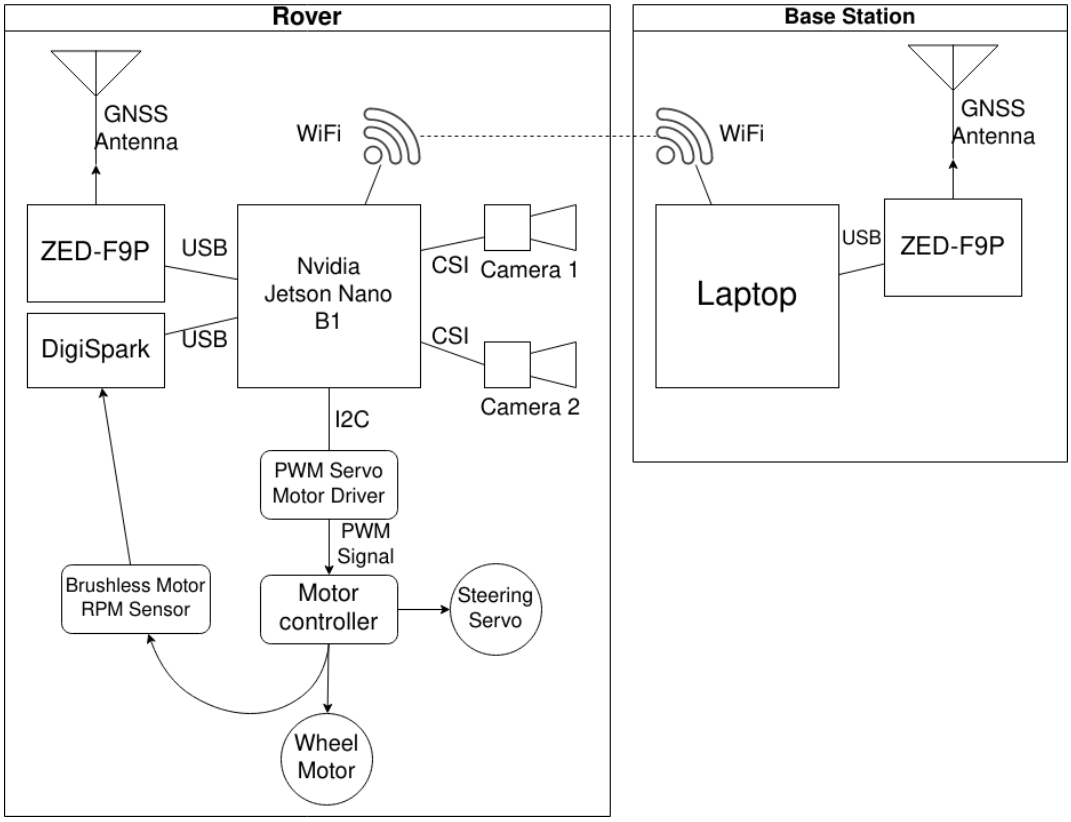

Figure 3: A block diagram of the RC car connected to a laptop for RTK.

4 Methods

In this section we give an overview of our rover platform, the dataset we collected, as well as the setup for the experiments.

4.1 Software tools

For capturing images on the Jetson, as well as controlling the ZED-F9P we designed and implemented two software libraries; Jepture and UBX GPS Server, respectively. Jepture is a lightweight image capturing library for python using the Jetson's builtin hardware accelerated pipeline to be able to quickly capture and save images without significant overhead. UBX GPS Server is a set of tools for working with the ZED-F9P chip with support for high frequency RTK measurements over IP.

4.1.1 Jepture

The RC car is equipped with a stereo camera, which is able to record 30 fps at 2x1640x1320 pixel resolution. As most of the machine learning libraries are written in python we needed a way to be able to control capturing these images from within python, while also being able to access the generated image data. Existing python libraries for capturing images either did not have the required functionality or the required performance, adding so much overhead that the computer is not able to run the neural network models while capturing images.
Therefore, a small python library, called Jepture, was designed which uses the Jetson's builtin hardware acceleration tools to be able to capture and process images in python. The resulting library is able to capture images from the stereo camera at full resolution and frame rate with minimal overhead. This library furthermore also integrates with Nvidia's hardware accelerated JPG codec for fast encoding and storage of the recorded frames.
The library is now publicly available on github (https://github.com/DelSkayn/jepture) and can be easily installed using the python package manager pip.

4.1.2 Ubx GPS Server (UGS)

Our second tool is a library with the somewhat uninspired name UBX GPS Server (UGS). This tool consists of several executables that are able to communicate with each other, and with a GNSS chip that implements the UBX protocol in order to facilitate the tracking of positions using RTK GNSS.
UBlox, the creator of the ZED-F9P, has tools to work with ZED-F9P however these tools only work on windows and access to data from the ZED-F9P is limited. Linux, has a tool called GPSd which is also able to communicate with the ZED-F9P and works similarly to our own implementation. However the tool did not support moving base RTK configuration that was required for our application. Furthermore the daemon only supported a method of communication using TCP connections, which is a deal breaker for our requirements because it introduces high latencies when communicating RTCM messages. Hence, a new tool for our use-case was designed and implemented, called UBX GPS Server (UGS), see Figure 4 for the depiction of the UGS setup.
The primary functionality of the tool is a server that communicates with a chip that implements the UBX protocol over a serial interface. This server acts as source and sink of UBX messages sent over TCP, which enables configuration and communication with the chip across an IP network. When

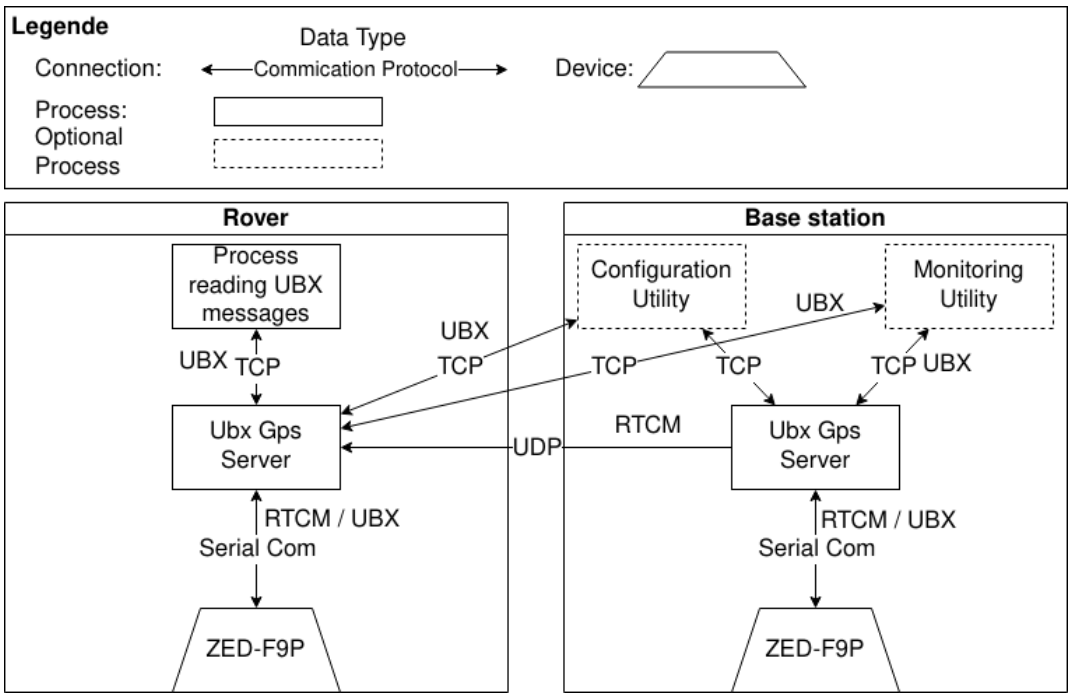

Figure 4: A block diagram of a network for running an RTK setup with UBX GPS Server.

started, the UGS executable starts a TCP server which can be used by other tools to send messages to and read message from a GNSS chip. The server is also able to both receive and forward RTCM messages over UDP for low latency connections. This functionality is used for facilitating RTK GNSS setups. A diagram for the network connections required for a running RTK setup can be seen in Figure 4.
Several auxiliary tools were implemented: a configuration tool which is able to write a configuration to the GNSS chip, a monitoring tool which implements a terminal user interface for monitoring messages and track positions returned by the GNSS chip, and a python library for connecting to the server and reading and processing messages from the GNSS chip.
The library is written in the rust programming language and is able to run on any major platform like linux, windows, or mac-os.
The UGS library is also publicly available on github (`https://github.com/DelSkayn/ubx-gps-server`).

4.2 The dataset: Ped-Data

The rover described in the previous section is used to capture a dataset of driving data in a pedestrian space. We call this dataset: Ped-Data. As our rover is built on an RC car it is very suitable to be driven on pedestrian roads: currently mostly a park. This allows us to capture novel and different driving datasets, where existing datasets are mostly captured on roads. Another difference from other datasets is the perspective of the images: most autonomous-driving datasets are taken from a car often with cameras mounted to the roof of the car given a good perspective of the road. Our RC car has its camera mounted roughly 15 cm from the ground. This makes images somewhat more challenging to interpret as curvature of the road as well as obstacles further on the road are less distinct.

Figure 5: Two samples images of the dataset. Note the differences in appearance due to the weather.

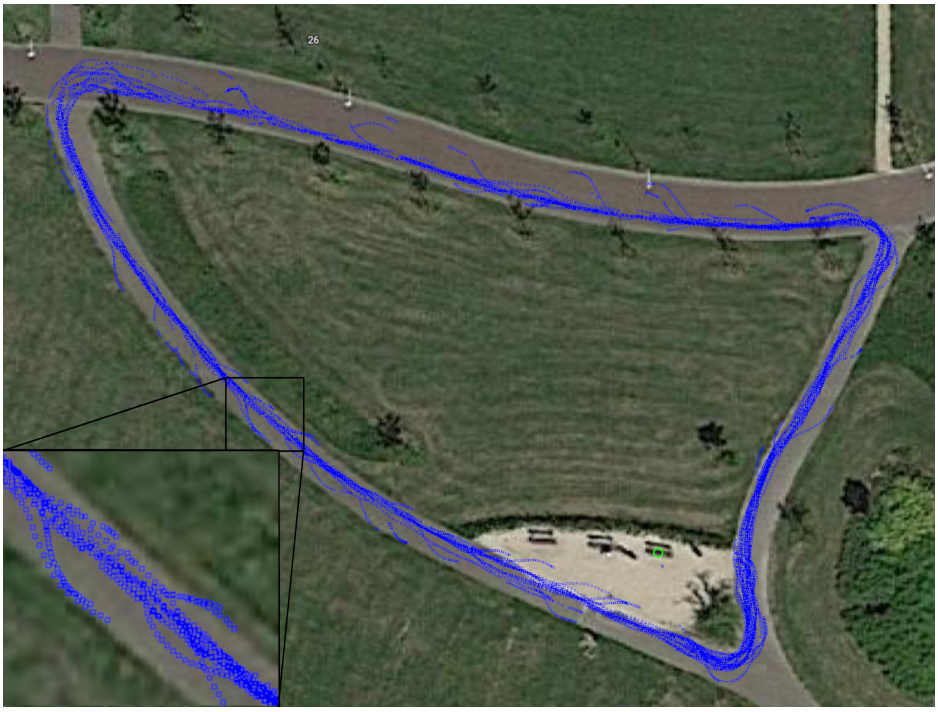

Figure 6: A visualization of some of the positions recorded during dataset creation, overlaid on a satellite image of the location. Each blue square represents a recorded position with the size of the square representing the accuracy of the recorded position. The green circle represents the base station position. All coordinates in this picture where approximately lined up to the satellite image for visualization only. Satellite source: Google maps, Map Data © 2023 Images © 2023 , Aerodata International Surveys, Maxar Technologies

Datatype	detail	number per second
Images	349168 (174584 pairs) at 2x1640x1320	30
Position data	86297 measurements at 1cm accuracy	10
Rpm Measurement	number of rotations recorded in 1/30 sec. intervals	30

Table 1: Table with information about the contents of the dataset.

The dataset consists of the following data:

1. A set of recorded image pairs from the stereo camera. These images are recorded at 30 fps with a resolution of `2x1640x1320`. For each image pair a timestamp is recorded. The timestamps are used to be able to combine the images with the other types of data recorded. Several examples of the recorded images can be seen in Figure 5.

2. Every 0.1 seconds a position is recorded relative to the base station. The position of the base station can vary between recording sessions and as such the coordinates between sequences of images of different recording sessions are not always the same for the same global position of the vehicle in the world. Furthermore, the accuracy of the measurement is recorded, which should mostly be a single centimeter but might vary due to environmental factors, together with a timestamp of the measurement. A visualization of the recorded positions is depicted in Figure 6.

3. RPM measurements are recorded at 30 measurements per second. These speed measurements are recorded together with a timestamp indicating the time the measurement was received.

An overview of the numbers of data can be seen in Table 1. A full description of the structure and format of the dataset can be found in Appendix A.

4.3 Experimental setup

The dataset is used for two experiments: autonomous waypoint driving and ego position estimation. In the following subsections we describe the experimental setups.

4.3.1 Autonomous waypoint driving

Our first experiment is an adaptation of the End-to-End learning method for Autonomous-Driving Cars presented in [17]. The method is adapted to our platform, target space, dataset and with some additions from advances in machine learning made since the paper, such as the usage of a more recent backbone DNN. The task here is to train a model to predict the right steering and acceleration input for the ego-vehicle to be able to drive autonomously. In our case the steering and acceleration are both values ranging from -1 to 1. Steering a positive 1 means that the car should steer as much left as it can while a steering position -1 means that the car should steer as much right as it can with other values in between resulting in a shallower steering angle. For acceleration a value equal to 1 means full acceleration while a value equal to -1 means breaking while in forward motion and then, once fully stopped, accelerating backwards.
In contrast to roads for cars, public walkways often come in a wide variety of forms. Even within the same neighbourhood the limits of what constitutes a walkway are often represented in a variety

of different ways. Some public walkways have a ledge, while others are just a colored part of the road, and other paths might only be delimited by the lack of grass. This in contrast to roads for cars which have, by law required, indications of where a car should drive. This makes the task of following roads in pedestrian spaces somewhat more challenging as any model trained for such a task has to deal with more ambiguity than a model trained for regular car roads would.
Furthermore, unlike the original paper we conducted our experiment on a specific trajectory with several intersections. The original paper [17] used a dataset where the roads for cars often have long stretches without any intersections, Pedestrian areas often have lots of intersections. The task of following a road then becomes somewhat more ambiguous when encountering a crossing as there is no single road to follow but multiple options. This preliminary experiment aims to research the question of how well an autonomously driving model behaves in this more ambiguous environment, where we focus on training a model to replicate a driven path from the dataset. The results of this experiment provide preliminary insights into the possible performance of autonomously driving vehicles, like autonomous mobility scooters or delivery bots, in mobile pedestrian spaces.

Mobile Platform The mobile pedestrian platform from the reference paper [17] had 3 cameras, our mobile platform only has 2. One of the innovations of the reference paper was to record with 3 cameras simultaneously. For each frame of driving data 3 images where recorded. One from the camera in the center of the car, and 2 images from the cameras to each side of the center camera. Apart from training the model to predict the target steering for each image from the center camera the two images from the offset cameras where used as training examples of situations where the car has drifted too far from the center of the road and should correct by steering back to the center. When training on these images, the actual driven steering target is offset with a correcting steering angle to steer the car back to the center of the road.
Our mobile platform does not have a camera in the center of the car but two cameras to each side of the vehicles center. We use these cameras as offset cameras similar to the reference papers applying a correcting steering angle when training on images generated by these cameras. As our mobile platform does not have a third central camera we do not train on images without a correcting steering angle.

Neural Network architecture A pre-trained imagenet network, i.e., a DNN pretrainted on the Imagenet dataset [23], is used as a feature extractor with a customized decision head. The original paper used a CNN that was trained from scratch, but modern approaches (e.g. [4]), often use pre-trained imagenet networks as a backbone network. Pretrained imagenet networks are capable of being powerful feature extractors which can be fine-tuned for new tasks [39]. For our selection of our backbone network we took into account a few factors; Firstly, the size of the neural network. Imagenet networks can scale up to hundreds of million parameters. While the Jetson Nano is a rather capable platform for its size, it is still limited in performance when compared to a desktop with a modern state of the art GPU. As such we need to select a comparatively small imagenet network. Secondly, we looked at the imagenet network performance. It is generally found that imagenet networks which perform better also fine-tune better [24]. Taking this into account we selected a recent network EfficientNet B01 [21] a model with a relatively low parameter count of roughly 5 million parameters. This model consists mostly of convolutional layers with a single final fully connected layer at the end. Pretrained Imagenet networks, including EfficientNet B01, are

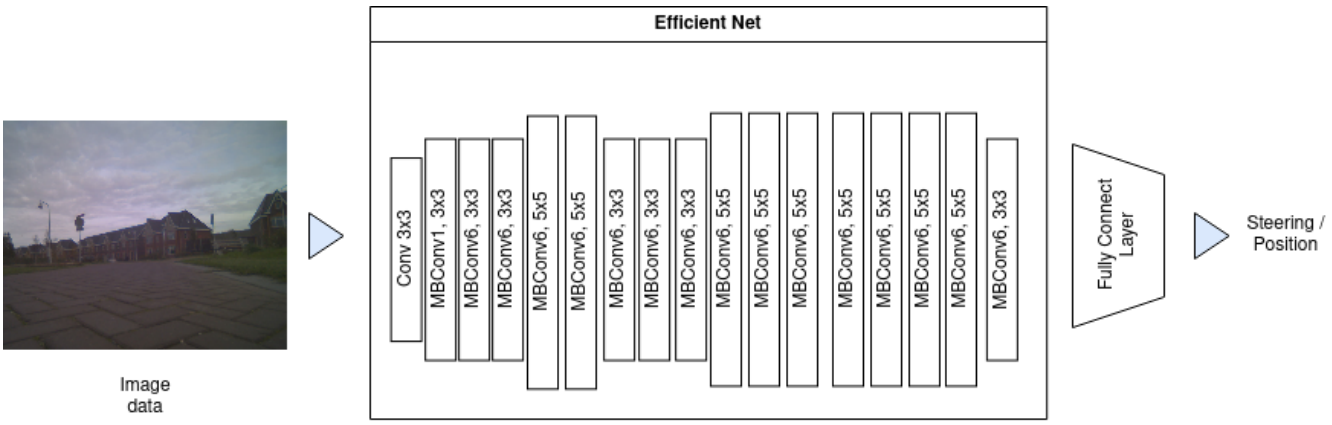

Figure 7: A diagram of the used network architecture with an EfficientNet B0 backbone.

generally trained with a final target layer of around 1000 outputs, a number equal to the number of imagenet classes. As road following has only two targets: steering and acceleration, we replaced the final layers of the network. We removed the included layer from the imagenet network and added a fully connected layer with newly initialized parameters and only two numbers as outputs, representing the predicted steering and acceleration. We also tried different architectures with additional fully connected layers reducing the features generated by the convolutional layers more gradually to two outputs but we found that the resulting architectures performed worse on the validation set. We will refer to this resulting neural net as the *waypoint-model* in further sections. A diagram of the network architecture can be seen in Figure 7.

Data augmentation As several studies showed the benefits of data augmenting several data-augmentation techniques were applied. Apart from the previously mentioned technique using offset in the target together with the offset of the camera, we further augmented the data by randomly applying a horizontal flip, and mirroring the recorded steering input for the flipped images, as left turns in a flipped image are right turns and vice versa. This flipping allows us to create a dataset which is perfectly balanced between left and right turns. Next we apply some common regularization techniques like: random erasing and color jitter, which could prevent overfitting, and random blurring which might allow the model to better respond to difference in focus. The cameras on our rover need to be manually focused by hand. This could have resulted in the focus of images in our dataset being different between different recording sessions.

Performance measures As we have access to other data and have a different target space than the original paper [17], we also modified the performance measure. The paper "End-to-End learning for autonomous-driving Cars" target is to train a car to drive on the road in its lane. The paper furthermore defined its performance as the amount of corrections that where required while the car was driving. The reference paper calls this measure autonomy. Autonomy is defined as the time the car is able to drive without intervention. An intervention happens when the car is one meter away from the center of the line. The car's path is then corrected which is assumed to take 6 seconds. The final autonomy is calculated by taking the total amount of corrections times 6 seconds divided by the amount of time spent driving and then subtracting the resulting number from 1 (see the

$$\text{autonomy} = 1 - \frac{\text{interventions} \times 6s}{\text{driving time in seconds}}$$

Figure 8: Equation for autonomy

equation in Figure 8).
This is a relatively good measure when driving on long straight sections with clear indications of what the road is, but breaks down in our target space where there is a less clear distinction between what is part of the road. Furthermore, as it is using 6 seconds for a correction it is implicitly geared to a certain speed of a specific vehicle. Therefore, we needed to alter the performance measure. As can be seen from the tracks driven and depicted in Figure 6, in our dataset the human driver shows large differences in the path that was driven during turns. Apparently at those places it is challenging or not natural for a human to each time drive the same path. This is something we wanted to take that into account when defining the performance measure. The error should be bigger, if the car deviates a certain distance from its normal route in a place where the human mostly drives the same route, whereas it should be relatively smaller at places where the human has large differences in routes driveni
For our performance measure we started with define where the ego vehicle should be allowed to drive. For this we defined an 'allowed path'. The allowed path is a trajectory together with a distance from that trajectory within the autonomously driving vehicle is valued as driving well. We calculate this target path from the dataset we created using the human driven paths as examples of good driving, i.e., driving similar to human driving.
The target path is calculated as follows: First, we convert the coordinates from paths driven in the dataset to polar coordinates with respect to the paths center, then we fit a curve through the points from the dataset. This curve is determined using a function from the scipy library called `scipy.optimize.curve_fit` [40]. This function takes a model function that accepts a free variable and a number of parameters and produces an output curve. The `curve_fit` function will then try to optimize the parameters such that, the output of the model function will approximate the data given to the `curve_fit` function. For our model function we choose to use a function that is similar to a Fourier series and can be defined with an arbitrary amount of parameters. The definition of this function can be seen in Algorithm 1.

Algorithm 1: The model function which is fitted to the data.

Input: The free variable X, and P a list of 2N parameters
Res $\leftarrow$ 0
for $i \leftarrow 0$ **to** N **do**
  Res $\leftarrow$ Res + P[2i]$cos(iX)$ + P[2i + 1]$sin(iX)$
end
return *Res*

Once we fitted the function to the data the output of this function will represents the 'best' path. Then we split the path into waypoints, for each measured position in the dataset we calculate the point of the curve which is closest to the measured point and calculate it's distance to this waypoint. This distance is a deviation from the 'best' path. For each waypoint on the curve we record the largest deviation in the dataset, e.i., the allowed deviation of the path at this waypoint.

If a waypoint on the curve had no closest recorded position, the allowed deviation is the mean of the allowed deviations of its two closest neighbouring waypoints. The pseudo code for this algorithm is given in Algorithm 2.
As long as the car is driving within the allowed deviation from a waypoint on the curve we define that good driving. If the car is driving beyond the allowed distance of any waypoint on the curve then the car is off-course. A visualization of an allowed path that is determined with this algorithm is depicted in Figure 9.
During the evaluation of our models we use this allowed path to measure the performance of a model. If a model, during autonomous driving, managed to drive the entire time within the allowed path, then it managed to drive with a performance similar to the human driver. Thus performance is measured by the amount of time the model spent outside the allowed path during autonomous driving.
With the previously calculated allowed path we also calculated a best mean path. This is the average path the human drove the mobile platform in the dataset. We can use this average path to compare our model to the reference paper. The reference paper used a deviation from the center of the road for its measure of autonomy. While in a pedestrian space we cannot clearly define the center of the road. But we can use our previously calculated mean path as the center of the road when calculating the autonomy of our model.
Before we calculate the autonomy of our model we do need to make one more adjustment. As we do not have a clearly defined center of the road, it is not possible to report on interventions while validating model. Therefore we need to make some small adjustments such that we can define interventions. An intervention is defined as the moment the ego vehicle is 1 meter from the mean path. However, as we did not intervene during validation the ego vehicle could just skirt the 1 meter where interventions happen; resulting in numerous 'interventions'. Therefore, it is reasonable not to count additional interventions untill the car is within the 1 meter distance from the mean path for at least a full second. If the car exits the 1 meter range again within a second, we do not count it as a new intervention.

4.3.2 Ego Position Estimation

For ego position estimation a network is trained to predict the position of a given recorded image. The recorded position in our dataset is used as the target for the training. As these positions are relative to the base station and not global coordinates, we cannot use all sequences in the dataset but only those where the position of the base station did not change, which could have happened in between separate dataset recording sessions. The dataset contains information about when the position of the base-station could have been changed.
We previously mentioned that the accuracy of the recorded position of the car is generally a single centimeter. This is only actually true for its north-south and east-west coordinates. The elevation coordinates are often less precise. For the current experiment this is irrelevant as we ignore the elevation and assume that the car is driving on flat terrain. A further assumption for this task is that only the north-south and east-west coordinates have to be predicted.

Neural Network architecture For this task the input is an image and the output consists of two targets. The targets are respectively the north-south and east-west predicted coordinates of the input image. We scale the output of the neural net by a scalar factor which is derived from the

Algorithm 2: Pseudo code algorithm for calculating the allowed distance for a given drive of the rover.

Input: C: The set of driven coordinates; R: The set of reference coordinates from the dataset

Output: D: The set of allowed distances.

```
// Calculate best path P resulting from a curvefit on R.
```
$R_{mean} \leftarrow$ mean(R) `// Calculate the center of R.`

$R_{polar} \leftarrow$ toPolar(R - R_{mean}) `// Transform R to polar coordinates.`

F $\leftarrow$ apply `curve_fit` on R_{polar} `// The fitted curve given as a function F.`

```
// Divide path into 1000 waypoints uniformly distributed along the fitted curve. This
   will be stored in P.
```
X $\leftarrow$ A list of 1000 points distributed with equal distance in the range of $0..2\pi$.

P $\leftarrow$ An empty list

for $i \leftarrow 0$ **to** $|X|$ **do**

  P[i] = fromPolar($F(X$[i])) + R_{mean}

end

```
// For each point on C calculate the closest waypoint on P.
```
$D \leftarrow$ list with length equal to $|P|$, with elements initialized to 0

Assigned $\leftarrow$ list with length equal to $|P|$ with elements initialized to False

for $i \leftarrow 0$ **to** $|C|$ **do**

  Dist $\leftarrow$ Infinity

  Point $\leftarrow 0$

  for $j \leftarrow 0$ **to** $|P|$ **do**

    if *Distance(C[i],P[j])* $<$ *Dist* **then**

      Dist $\leftarrow$ Distance(C[i],P[j])

      Point $\leftarrow$ j

    end

  end

  Assigned[*Point*] $\leftarrow$ True

  D[*Point*] $\leftarrow$ Min(*Dist*,D[*Point*])

end

```
// Some waypoints on P may be unassigned and have no allowed distance, for unassigned
   points P, calculate its allowed distance by interpolating allowed distances from
   neighbouring waypoints
```
forall i **in** $0..|P|$ **where** *Assigned*[i] = *False* **do**

  left $\leftarrow$ (i - 1) Modulo $|P|$

  while *not Assigned[left]* **do**

    left $\leftarrow$ (left - 1) Modulo $|P|$

  end

  right $\leftarrow$ (i + 1) Modulo $|P|$

  while *not Assigned[right]* **do**

    right $\leftarrow$ (right + 1) Modulo $|P|$

  end

  D[i] $\leftarrow$ (D[left] + D[right]) / 2

end

return D

Figure 9: A visualization of the allowed path. Each circle represents a point on the curve with the radius being the allowed distance from the point. The allowed distance is determined using human driving trajectories. The allowed distance is the distance the human driven trajectories deviated from the mean path.

maximum range of values of the dataset. The to be predicted coordinates a bigger range then $[0, 1]$, whereas pretrained networks often have output nodes with values in range $[0, 1]$. In order to save training time for networks to predict values from a much larger range, it can be useful to rescale the coordinates so that they fall within the 0 to 1 range.
If we just look at what input and outputs the network produces, we notice that this tasks has similarities, to the previous experiment. The input is an image and the output are the two numbers representing the two coordinates instead of two numbers representing steering and acceleration. This allows use to use the same backbone neural network architecture as for the previous task. In the next sections we will refer to this model as the ego-position neural net.

Performance measure The goal of this experiment is to train a model which is able to precisely determine the position coordinates of an image. This means our performance measure should be a form of position accuracy. In order to interpret the performance of our model, we need some numbers to compare to. If this model where to be used in a real-world application, it would compete with a GNSS device, therefore a good performance reference for comparison would be the performance of GNSS devices. Ideally this model would perform as good as the GNSS RTK measurements we made. It is however highly unlikely that we would be able to achieve such precision. Instead, as a reference we compare it to the less accurate, normal GNSS performance without RTK corrections. The data for comparison is from three different sources, First,a report from the US Government National Transportation Safety Board or the NTSB [8]. This report details the performance of GNSS signals and represent an upper bound on the accuracy of GNSS equipment. GNSS signals are processed by GNSS devices in order to generate a postion. However, poor equipment, signal

access, or a bad antenna can degrade the performance relative the signals. Real GNSS devices rarely achieve the highest possible accuracy with the received signals. The report only cites, what it calls, 95% accuracy, or the error distance below which 95% of the measurements fall, also called the 95% quantile, to be less then 2 meters. The next reference is from a report on the accuracy of mobile phones [41]. This paper only cites the mean error of smartphones which is 4.9 meters. Finally, we have the performance of ZED-F9P as cited by its manufacturer, which is a median error of 1.5 meters in GNSS without RTK mode. The manufacturer does not cite any other measurement of the performance of the ZED-F9P without RTK. These performances are used in the evaluation of our results on ego-position estimation.

5 Experimental Results

In the next section we present the results of our experiments on autonomous waypoint following and ego position estimation.

5.1 Autonomous waypoint following

After training our waypoint neural net on our Ped-Data dataset we ran this model on the track. As the car was autonomously driving on the track we recorded the position of the car, a visualization can be found in Figure 10. The car was mostly able to drive on its own. The waypoint neural net did have problems with crossings as it sometimes would steer too late or not at all. Furthermore, the car seemed to have trouble staying on its path. The car would often sway across the road, only adjusting its trajectory when the car would be relatively far from the path driven in the dataset. This can be seen in Figure 10 as the path for the car sometimes snakes across the road.
On average the car managed to stay within 51 cm of the mean path which is better than the allowed mean deviation of 52 cm. The car was able to stay within the allowed path on 72% of the recorded positions. During this test the car was driving autonomously for 13 minutes, during which we had to intervene 4 times when the car would drive too far from the path without any sign of recovering. Remarkably, only places where we needed to intervene were at crossings. The car would often not turn into the right direction, or turn too late. The car did manage to stay on the road, on its own, while driving the sections without intersections.
Using our version of the autonomy measure, we found that our model required 32 interventions for a driving-time of 899.4 seconds and an autonomy of 79%. This is significantly lower than the autonomy of the reference paper which reported an autonomy of 98% on their test-set. This could be due to our dataset being more challenging to drive than the dataset used in the reference paper. Unfortunately as the dataset from the original paper is not publicly available this is hard to verify. We can however get an estimation of how difficult our dataset is by using same metric to calculate the autonomy of a human driver. If we apply the same definition of autonomy we used to measure the performance to the recorded driving in the dataset we find that the human driver actually performed worse then the waypoint neural net did, having 71 interventions for a 957.5s driving time with an autonomy of 56%. An overview of the results can be seen in Table 2 and in Table 3 you can see the results of the reference paper.

Figure 10: A visualization of the path driven by the drive model. Green squares represents recorded positions within the allowed path, red squares are outside of the recorded path. Blue circles represents points where we needed to intervene as the car drove too far of the path. A single intervention top left was outside of the bounds of the map.

Ped-Data dataset	Interventions	Time driven	Autonomy
Our Model	32	899.4 s	79%
Human in dataset	71	957.5 s	56%

Table 2: Performance of our model and the human driver.

End to End driving dataset	Autonomy
End to End driving	98%

Table 3: Performance of the reference paper. Only autonomy was cited in [17].

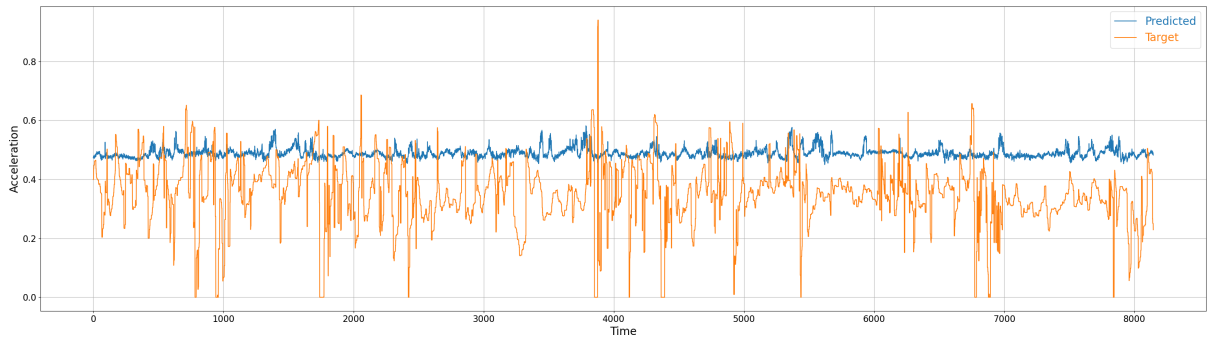

Figure 11: A graph showing predicted acceleration against the target acceleration

Next, we recorded a new round of driving done by a human, and tested how well the waypoint neural net was able to predict both steering and acceleration. The results can be seen in Figure 11 and 12.

As can be seen in these graph the behavior of the human driver is very different from the behavior the waypoint neural net learned. While the human driver mostly steers with large corrections the model uses smaller corrections held over a longer time. This discrepancy is rather surprising. Though the behavior of the model and the human driver seem to be very different, the car did manage to stay on the road during driving. In order to investigate the behavior of the waypoint neural net further, we created a graph where we smoothed the input of the human driver. This graph can be seen in Figure 13. The graph depicts cleary the correlation between the predicted steering angle and the target steering angle,as can be seen by the see peeks in the target steering input lining up with peeks in the predicted steering input.

During driving we did observe that when the waypoint neural net was driving it would sometimes sway across the road. This behavior can also be seen in Figure 10. Looking at the predicted behaviour in the graph this might explain this behaviour. The model seems to be less determined in its steering than the human driving from the dataset, only making small adjustments. These small adjustments are probably not enough to fully correct the steering angle. And as the car moves further from the best path the predicted steering angle increases until the car steers back towards to best path. Here the model's lack of determined steering causes it to miss the optimal path, over-shooting and continuing the swaying.

We also noticed that the predicted acceleration of the model is consistently higher than the target acceleration. This is probably due to inconsistency of the human driver. During the human test drive the mean acceleration was 0.34 while the mean acceleration in the dataset was 0.46. The mean predicted acceleration was equal to 0.49, which is a lot closer to the acceleration in the dataset than to the human driven acceleration. This difference in acceleration did not seem to matter for the quality of driving. During testing we never observed the car reaching a speed that would prevent it from making a turn with the wheels at its maximum steering angle. Figure 12 does show a lack of correlation between the predicted and target acceleration. This did not seem to cause an issue when driving autonomously.

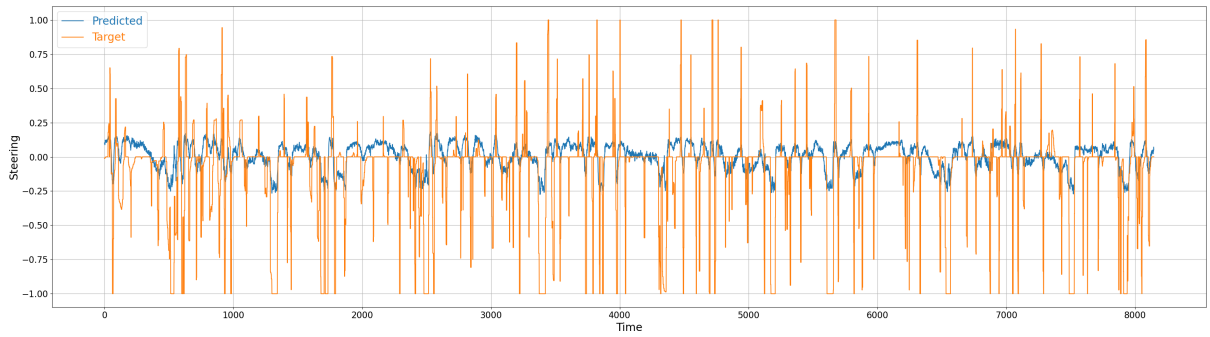

Figure 12: A graph showing predicted steering against the target steering

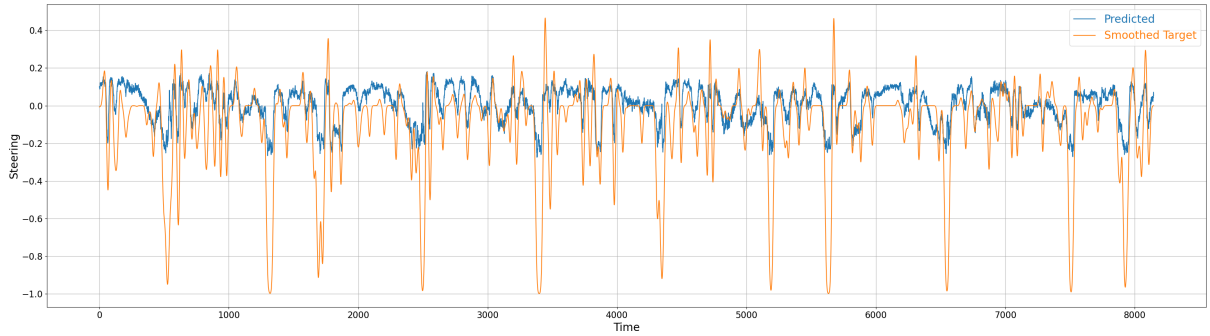

Figure 13: A graph showing predicted steering against the target steering smoothed using a gaussian filter.

5.2 Ego position estimation

In this experiment we trained our ego-position neural net to predict positions of the rover based on captured images. We then ran the model on a newly recorded sequence. A visualization of the predictions from the model on the validation set and the test set can be seen in Figure 14. An overview of the results can be seen in Table 4.

The ego-position neural net managed to achieve a mean absolute error of 112 centimeters on the validation-set, 95% of the predicted locations on the validation set have an error below 250 centimeters. The model performed significantly worse on the test set. On the test set the ego-position neural net achieved a mean absolute error of 374 centimeters and 95% of the predicted locations had an error below 1265 centimeters. This is relatively surprising, while in general it is expected that the results of the test set have lower accuracy then on the validation set due to indirectly fitting the training method to the validation set, the found discrepancy is unexpectedly high. One explanation for the difference might have been due to the difference in weather between the test set and the validation set. Most of the data of the validation and training set was recorded on rather cloudy days, while it was sunny during the recording of the test-set.

In order to validate this explanation we also recorded another test-set during cloudier weather. A visualization of the result of running the model on this new test-set can be seen in Figure 15. The ego-position neural net achieved on this new test set a mean error of 161 centimeters and 95% of the errors were below 398 centimeters. This result is much closer to the one achieved on the validation set and seems to indicate that weather has a large impact on the performance of the model.

	Mean Error	Median Error	95% quantile	75% ”	50% ”	25% ”
GPS Signal error	-	-	182 cm	-	-	-
Mobile Phone	490 cm	-	-	-	-	-
ZED-F9P Non-RTK	-	150 cm	-	-	-	-
Model validation	112 cm	91 cm	250 cm	132 cm	91 cm	59 cm
Model test cloudy	162 cm	122 cm	398 cm	201 cm	122 cm	76 cm
Model test sunny	374 cm	215 cm	1265 cm	340 cm	215 cm	142 cm

Table 4: Predicted position results.

Figure 14: A visualization of the predicted positions (pink boxes) against the ground truth positions (green boxes) from the validation-set (left), and test-set (right).

These achieved errors are significantly worse then the reference errors from the GPS signals, and a sufficiently good receiver might be able to outperform our trained ego-position neural net consistently, however our model performed better than mobile phones in both tests and also better than the ZED-F9P without RTK during the cloudy test. The ZED-F9P is a reasonably advanced GNSS receiver with good performance even without RTK, and so outperforming it is a significant achievement. Also this model did outperform the cited mean accuracy of GPS equipped smartphones, as both on the test set as well on the validation the mean accuracy of the trained ego-position neural was well below 4.9 meters.

6 Conclusions

In this paper we presented a rover platform, various tools such as UGS and Jepture, the platform, and the Ped-Data dataset we created. Using the dataset we conducted two experiments which show that the recorded data can be used to train models which can learn to autonomously drive in pedestrian spaces as well as predict the position of a vehicle in such a space. We adapted a method previously only applied to public roads for pedestrian spaces and showed that it is possible to use these adapted methods for pedestrian spaces. We trained the adapted models and showed that they were able to preform tasks like following a pre-learned trajectory. The conducted

Figure 15: A visualization of the predicted positions (pink boxes) against the ground truth positions (green boxes) from a test-set recorded during more cloudy weather.

experiments showed interesting preliminary results with regards to autonomous waypoint driving and ego-position estimation using existing Imagenet neural networks, such as Efficientnet B0, onboard our rover platform. The ego-position estimation experiments showed that NN models are able to predict positions from captured images with an accuracy exceeding that of common non-RTK GNSS devices. However, the achieved accuracy of these models proved to be dependend on similar environmental conditions as encountered during the dataset creation. Running the model during weather conditions that are not encountered during dataset creation resulted in a drastically lower accuracy.
Our final contributions in this paper are the following:

- A design and implementation of a rover platform applicable for use in autonomous driving and ego-position estimation research in pedestrian spaces.
- The Jepture library for efficient image capturing on the Jetson Nano.
- UBX GPS server, a set of tools for low latency interfacing and working with the ZED-F9P.
- The Ped-Data dataset captured with our rover platform and tools.
- A trained model and experimental results with applications in autonomous waypoint driving in pedestrian spaces.
- A trained model and experimental results with applications in ego position estimation.

7 Future research

Although we found some interesting results in this paper our experiments gave only preliminary results and insights. Possible future research could investigate the problems of autonomous waypoint driving and ego-position estimation.
Our preliminary experiment for autonomous waypoint driving was not able to properly incorporate waypoints into the experiment. The method was only trained to be able to follow roads. Further

research could look into incorporating waypoints into the model. A model may for example be extended to take a position relatively to the vehicle's position and the model should then make its way to the next waypoint while following roads and making the right turns at crossings.
The dataset we created is currently limited to a single location. Furthermore, from our experiments we found that the data also lacks diversity of weather conditions. This seems to significantly impact the performance of the trained models. In a future effort, when using our platform to capture a more comprehensive large scale pedestrian dataset for autonomous driving and position estimation research, data should be recorded during more diverse weather conditions and at different locations. Furthermore, while the RC car was useful in testing the platform, its applicability is limited. The tools we created could, relatively easily, be used in a different configuration by altering what the actuators of the platform control. A more applicable platform might for instance be a delivery robot, or a mobility scooter. If one uses a method to control the motor and steering of a mobility scooter and mount a camera unit, the methods described in this paper should be easily transferable to that new platform.

References

[1] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2015.

[2] N. Radwan, A. Valada, and W. Burgard, "Vlocnet++: Deep multitask learning for semantic visual localization and odometry," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 4407–4414, 2018.

[3] E. Rehder, J. Quehl, and C. Stiller, "Driving like a human : Imitation learning for path planning using convolutional neural networks," in *International Conference on Robotics and Automation Workshops*, pp. 1–5, 2017.

[4] D. Chen, V. Koltun, and P. Krähenbühl, "Learning to drive from a world on rails," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 15590–15599, 2021.

[5] "Tesla AI Day 2021." https://www.youtube.com/watch?v=j0z4FweCy4M Accessed on: 12-3-2022, 2021.

[6] A. Stocco, B. Pulfer, and P. Tonella, "Mind the gap! A study on the transferability of virtual vs physical-world testing of autonomous driving systems," *IEEE Transactions on Software Engineering*, 2021.

[7] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: a survey," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 737–744, 2020.

[8] N. T. S. B. US Government, "Global positioning system standard positioning service performance analysis report." https://www.nstb.tc.faa.gov/reports/2020_Q4_SPS_PAN_v2.0.pdf, Accessed on: 7-1-2023, 2021.

[9] D. Pomerleau, "ALVINN: An autonomous land vehicle in a neural network," *Advances in neural information processing systems*, vol. 1, 1988.

[10] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: object detection via region-based fully convolutional networks," *Advances in neural information processing systems*, vol. 29, 2016.

[11] V. Badrinarayanan, A. Handa, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixel-wise labelling," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2015.

[12] K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. B. Girshick, "Masked autoencoders are scalable vision learners," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16000–16009, 2021.

[13] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 652–660, 2016.

[14] C. R. Qi, W. Liu, C. Wu, H. Su, and L. J. Guibas, "Frustum pointnets for 3d object detection from RGB-D data," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 918–927, 2017.

[15] L. Yu, X. Shao, Y. Wei, and K. Zhou, "Intelligent land-vehicle model transfer trajectory planning method based on deep reinforcement learning," *Sensors*, vol. 18, no. 9, 2018.

[16] H. Shao, L. Wang, R. Chen, H. Li, and Y. Liu, "Safety-enhanced autonomous driving using interpretable sensor fusion transformer," in *Conference on Robot Learning*, pp. 726–737, PMLR, 2023.

[17] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, vol. abs/1604.07316, 2016.

[18] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J. Allen, V. Lam, A. Bewley, and A. Shah, "Learning to drive in a day," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8248–8254, 2018.

[19] A. Dosovitskiy, G. Ros, F. Codevilla, A. M. López, and V. Koltun, "CARLA: an open urban driving simulator," in *Conference on robot learning*, pp. 1–16, 2017.

[20] M. G. Bechtel, E. McEllhiney, and H. Yun, "Deeppicar: A low-cost deep neural network-based autonomous car," in *2018 IEEE 24th international conference on embedded and real-time computing systems and applications (RTCSA)*, pp. 11–21, 2017.

[21] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization*, pp. 109–123, 2019.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2015.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

[24] S. Kornblith, J. Shlens, and Q. V. Le, "Do better imagenet models transfer better?," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2661–2671, 2018.

[25] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, V. Vasudevan, W. Han, J. Ngiam, H. Zhao, A. Timofeev, S. Ettinger, M. Krivokon, A. Gao, A. Joshi, Y. Zhang, J. Shlens, Z. Chen, and D. Anguelov, "Scalability in perception for autonomous driving: Waymo open dataset," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2446–2454, 2019.

[26] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 11621–11631, 2019.

[27] "AWS Deepracer." https://aws.amazon.com/deepracer,Accessed on: 7-1-2023, 2021.

[28] "Donkey Car." https://www.donkeycar.com,Accessed on: 7-1-2023, 2021.

[29] Nvidia, jaybdub, tokk-nv, and adammconway, "Jetracer." https://github.com/NVIDIA-AI-IOT/jetracer, 2021.

[30] F. Camara, N. Bellotto, S. Cosar, D. Nathanael, M. Althoff, J. Wu, J. Ruenz, A. Dietrich, and C. W. Fox, "Pedestrian models for autonomous driving part i: low-level models, from sensing to tracking," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 10, pp. 6131–6151, 2020.

[31] Y. Luo, P. Cai, A. Bera, D. Hsu, W. S. Lee, and D. Manocha, "Porca: Modeling and planning for autonomous driving among many pedestrians," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3418–3425, 2018.

[32] E. Horváth, C. Pozna, and M. Unger, "Real-time lidar-based urban road and sidewalk detection for autonomous vehicles," *Sensors*, vol. 22, no. 1, p. 194, 2021.

[33] H. Karnan, A. Nair, X. Xiao, G. Warnell, S. Pirk, A. Toshev, J. Hart, J. Biswas, and P. Stone, "Socially compliant navigation dataset (scand): A large-scale dataset of demonstrations for social navigation," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 11807–11814, 2022.

[34] N. Carlevaris-Bianco, A. K. Ushani, and R. M. Eustice, "University of Michigan North Campus long-term vision and lidar dataset," *International Journal of Robotics Research*, vol. 35, no. 9, pp. 1023–1035, 2015.

[35] T. Taketomi, H. Uchiyama, and S. Ikeda, "Visual slam algorithms: A survey from 2010 to 2016," *IPSJ Transactions on Computer Vision and Applications*, vol. 9, no. 1, pp. 1–11, 2017.

[36] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE transactions on robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[37] Y. Wu and Y. Liu, "Position estimation of camera based on unsupervised learning," in *Proceedings of the International Conference on Pattern Recognition and Artificial Intelligence*, pp. 30–35, 2018.

[38] T. Eriksson. https://en.wikipedia.org/wiki/File:Real_time_kinematic.svg, 2020.

[39] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "CNN features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 806–813, 2014.

[40] "The scipy documentation `scipy.optimize.curve_fit`." https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html Accessed on: 4-12-2023, 2023.

[41] P. E. Frank, van Diggelen, "The world's first gps mooc and worldwide laboratory using smartphonse."," *Proceedings of the 28th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2015)*, 2015.

Appendices

Appendix A Description of the Ped-Data dataset

The following section describes the layout and data of the recorded Ped-data dataset.

A.1 Overview

The dataset is divided into several directories each containing data for a single recording session. Each recording session directory is named after the date the session took place. For example `01-12-22` for the recording session which took place on the first of December 2022.
A single session directory contains a set of subdirectories as well as a `tgt.hdf5` dataset file. The `tgt.hdf5` is the main file describing how the data was recorded and the entry point for using the dataset. The directories placed alongside the `tgt.hdf5` that are named `left_` or `right_` together with a timestamp contain the images recorded from the stereo camera of our rover platform. Every session directories contains a pair of left and right directories with the same timestamp for each recording sequence. During dataset recording we occasionally needed to reset our rover platform, when this happened a new pair of directories was created. Each image in the dataset is a jpeg image with as name a number. The `tgt.hdf5` files contain specific information about how to load the images in the right order as well as other sensor information recorded alongside the images.
The sensors on our rover record data at different intervals making it not feasible to store all the data in a single table. Instead the dataset is divided into several data-collections which contain data recorded at a specific interval. For instance: the car data-collection contains information about images and recorded driving input. This data was recorded at 30 frames per second. However, the GPS data-collection, which contains position information, was recorded at 10 frames per second. Because of this difference in recording interval the dataset does not directly specify how frames of different data-collections are related to one another. Instead the dataset provides timestamps as well as offsets between the specific epochs from which the timestamp was derived. It is up to the user to use the provided timestamps to match or interpolate the data.

A.2 The dataset in depth

The dataset consists of a number of directories each containing the data for a recording session. A directory containing a dataset file: `tgt.hdf5` and any number of image directories called `left_$stamp` and `right_$stamp` where `$stamp` is the timestamp the recording of that sequence of images started as a floating point seconds, e.g. `left_1673449657.825414`.

A.2.1 `tgt.hdf5`

Every directory containing a recording session contains a file called `tgt.hdf5`. The tgt.hdf5 file is a h5 file containing several tables forming three data-collections which contain all recorded sensor data with the exception of the recoded images. Note the h5 file format calls tables in their file formate datasets. We refer to these datasets as tables to avoid confusion.

A.2.2 Data-collections

The `tgt.hdf5` is split in several data-collections. Each collection can have multiple tables which encode the data. These tables are called `${name}_${tablename}` where `${name}` is the name of the data-collection and `${tablename}` is the name for this specific table. For example the table `car_frame_id` from the `car` data-collection.
Each of these tables has a single table which describes which frames of recorded data belong to a single sequence. A sequence is a collection of frames recorded without any interruptions. The table which has this sequence information is called the sequence table and is named `${name}_sequence`. For example, in the case of the `car` data-collection the sequence table is called `car_sequence`.
The sequence table is an array of an arrays containing 2 integers: the start-frame-index and end-frame-index of each recorded sequence in this collection. The start-frame-index is the index of the first frame of each sequence in that order. The end-frame-index is the index which is the first index to no longer be part of the sequence. So the frames belonging to a sequence range from the start-frame-index to, but not including, the end-frame-index. For example, the `tgt.hdf5` file for the `01-12-22` recording session has at sequence index 1 the values `[3259,4578]` in the `car_sequence` table. This means that the second sequence (with index 1) is from the frame with index 3259 to the frame with index 4577, 4578 not including.
The frames for each data-collection where recorded independent of each other. To allow matching the frames of one data-collection to another all data-collection have timestamp information as well as the recorded offset in timestamps.

A.2.3 Data-collection indices

In this dataset we differentiate between frame indices: `frame_idx` and sequence indices: `seq_idx`. A sequence index is the index of a specific sequence in a h5 file. The sequence index is used to index the sequence table and other tables which contain information about a specific sequence. A frame index is the index of a specific frame of data in a h5 file. The frame index is used to index a tables containing information about a specific frame. A sequence contains a number frames as specified by the sequence table. All tables in the h5 file are indexed with a frame index unless otherwise specified.
A `tgt.hdf5` file contains 3 different collections of data-collection: `gps`, `car`, `speed`.

A.2.4 Loading images

The image for the current frame can be loaded from the path
`${orientation}_${car_sequence_stamp[seq_idx]}/${car_frame_id[frame_idx]}.jpg` where `${orientation}` is either `left` or `right` for the left or right camera, `car_sequence_stamp`, and `car_frame_id` are tables from the `tgt.hdf5` file, `frame_idx` is the frame index, and `seq_idx` is sequence-index of sequence to which the frame belongs.
For example: for the recording session in directory `02-12-22` there are 6 sequences in the dataset. The sequence with index `1` is from frame `3259` until but not including frame `4578`. If we wanted to load the left image for the frame `3400` the `seq_idx` would be `1` and the `frame_idx` would be `3400`. If we look up the data in the dataset file we find that `car_sequence_stamp[seq_idx]` is `1669901792.3659706` and `car_frame_id[frame_idx]` is `4739` so the image to load is `./left_1669901792.3659706/4739.jpg`.

A.3 Dataset tables

The following section describes each of the tables in the `tgt.hdf5` file.

A.3.1 The `car` collection

The `car` data-collection contains data recorded from the car itself, this includes metadata of the images recorded as well as driving input recorded. The frames in this data-collection were recorded in 30 frames per second. Timestamps in this data-collection were recorded relative to the linux monotonic clock. This clock returns a timestamp which is reset every time the computer reboots. The datasets in this collection are `car_steer`, `car_accel`, `car_frame_id`, `car_proc_stamp`, `car_image_stamp`, `car_sequence_stamp`, and `car_sequence`.

car_steer `car_steer` contains the steering input for each frame for the car as a floating point value between -1.0 and 1.0, where steering fully right is encoded as 1.0.

car_accel `car_accel` contains the acceleration input for each frame for the car as a floating point value between -1.0 and 1.0, where fully accelerating encoded as 1.0 and fully braking while moving forward and then, once the rover has come to a stop, reversing is encoded as -1.0

car_frame_id `car_frame_id` contains the id for the images associated with the current frame. See loading images A.2.4.

car_proc_stamp `car_proc_stamp` contains the timestamp of when the frame was processed, i.e. when the steering and acceleration was recorded. The timestamp is a floating point value in seconds since an unspecified epoch.

car_image_stamp `car_image_stamp` contains two timestamps of when the images associated with the current frame where captured. The first timestamp corresponds to the left image the second timestamp corresponds to the right image. The timestamp is a integer of milliseconds since an unspecified epoch, the same epoch as `car\_proc\_stamp`.

car_sequence_stamp `car_sequence_stamp` contains the timestamp of when the sequences where recorded. This table is indexed by the sequence index. See loading image for usage (A.2.4).

car_sequence `car_sequence` contains the start and end for each sequence as described previously in section A.2.2. This table is indexed by the sequence index.

A.3.2 The gps Collection

The GPS collection contains data captured from the GPS system. The frames in this data-collection where recorded at 10 frames per second. The datasets in this collections are `gps_position`, `gps_accuracy`,`gps_stamp`, `gps_delay`, `gps_recv_stamp`,`gps_status`,`gps_sequence_position`.

gps_position `gps_position` contains the position data for each measurement of the GPS frames. Each frame consists of three integers containing the offset of the car with respect to a central point. The first value is the offset to the north, the second value is the offset to the east, and the third is the elevation offset downwards. All offsets are in centimetres. The position of the central point is only guaranteed not to move within the same sequence. The position is given in units of 1 cm.

gps_accuracy `gps_accuracy` contains the reported accuracy of the position measurement for the current frame. Each frame consists of three integers containing the accuracy of the measurement in three axis, north, east, and downwards, respectively. We tried to keep the accuracy of the gps measurements at 1 cm but during recorded accuracy might have degraded. The accuracy is given in units of 0.1 mm.

gps_stamp `gps_stamp` contains the timestamp for the current measurement. The stamp is in milliseconds since the GPS epoch. For more information see the iTOW section of the ZED-F9P integration manual.

gps_delay `gps_delay` contains the estimated delay between the GPS epoch and the unspecified epoch from the car. The delay is a floating point value delay in seconds from the car epoch to the GPS epoch. If one wants to retrieve the car time at the time of a GPS measurement, one should subtract the delay from the GPS measurement.

gps_recv_stamp `gps_recv_stamp` contains the time at which the car obtained the measurement from the GPS. The stamp is a floating point value in seconds since the unspecified car epoch.

gps_status `gps_status` contains the status of the measurement, specifically the position fix type of the GPS. If the GPS has no fix, the status equal to 0. If the GPS has a floating fix, the status is equal to 1. If GPS has full fix, the status is equal to 2. During the recording of the dataset we tried to ensure that the car had a full fix as often as possible. However during the recording, the fix might have degraded. For more information regarding a GPS fix see the ZED-F9P integration manual.

gps_sequence `gps_sequence` contains the start and end for each sequence as described previously in section A.2.2. This table is indexed by the sequence index.

gps_sequence_position `gps_sequence` contains a unique id for different base-station locations. The positions recorded in the `gps` collection are relative to the base station. Two similar coordinates in different sequences might not belong to a similar location as the base-station could have changed. This table contains a number which is a id for a location of the base-station. If a sequence has the same location id then we tried to place the base-station in that same location to the best of our abilities. Note that it is possible that there are small difference in base-station location even if the id is the same. This table is indexed by the sequence index.

A.3.3 The `speed` collection

The `speed` collection contains data captured from the RPM sensor. The frames in this data-collection where recorded at 30 frames per second. The datasets in this collections are `speed_measure`, `speed_sequence`, and `speed_stamp`.

speed_measure `speed_measure` contains the number of rotation recorded by the RPM sensor between the current and the last frame.

speed_stamp `speed_stamp` contains the timestamp of when the frame was processed, i.e. when the number of rotations was recorded. The timestamp is a floating point value in seconds since an unspecified epoch. This epoch is the same as the epoch of the `car` collection.

speed_sequence `speed_sequence` contains the start and end for each sequence as described previously in section A.2.2. This table is indexed by the sequence index.

Appendix B Platform Setup

The following appendix describes how to setup the Jetson Nano RC such that it can run the code provided with this thesis.

B.1 Car Setup

The first step is to install the Jetpack distribution onto the SD-card. At the moment of writing the last supported version of Jetpack for the Jetson Nano is 4.6.
After installing the Jetpack image onto a SD card by any preferred method boot up the Jetson and make sure Jetpack is fully upgraded by running the following commands:

```
$ sudo apt-get autoremove
$ sudo apt-get update
$ sudo apt-get upgrade
```

Install pip to be able to install python packages:

```
$ sudo apt-get install python3-pip
$ pip3 install -U pip
```

Then change the default python ubuntu:

```
$ sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.6 1
$ sudo update-alternatives --install /usr/bin/python python /usr/bin/python3.6 1
```

The installed version of `setuptools` is rather outdated so run the following command to upgrade `setuptools`.

```
$ pip3 install -U setuptools
```

In order to control the motors of the car from python we need to install some packages:

```
$ pip3 install Adafruit-PlatformDetect==3.14
$ pip3 install adafruit-circuitpython-servokit==1.2.2
```

Next install Pytorch and Torchvision. A wheel file for Pytorch can be download from the following thread: https://forums.developer.nvidia.com/t/pytorch-for-jetson/72048/14. I have only been able to get Pytorch to run with python 3.6. So I recommend installing the latest version of Pytorch compatible with that version of python, which is at the time of writing 1.10.0.
Torchvision can best be installed from source:

```
$ sudo apt-get install libjpeg-dev zlib1g-dev libpython3-dev
$ sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev
$ pip3 install -U pillow
$ git clone https://github.com/pytorch/vision.git --branch vSPECIFIC_VERSION_HERE
$ cd vision
$ export MAX_JOBS=2
$ python3 setup.py bdist_wheel
$ cd dist
$ pip3 install ./torchvision-SPECIFIC_VERSION-cp36-cp36m-linux_aarch64.whl
```

The next library required is Pycuda which can be installed from pip. However, the Pycuda installation script looks for `xlocale.h` which is not present in the Jetpack distro. As a work around run:

```
$ sudo ln -s /usr/include/locale.h /usr/include/xlocale.h
```

Then Pycuda can be installed with:

```
$ pip3 install --global-option=build_ext \
  --global-option="-I/usr/local/cuda/include" \
  --global-option="-L/usr/local/cuda/lib64" pycuda
```

A Bluetooth controller is used to be able to control the car during recording and testing. The controller library used by the scripts should be able to read most controllers but, as with most game controllers, getting controllers to connect properly might be somewhat of a hassle depending on the controller used. It is recommend to use a Bluetooth Xbox controller as we encountered the least problems with those controllers, but your millage may very.
The input library used by the scripts is `approxeng-input`. This library will generally work but the bindings for the controller buttons might be off. You can alter the button layout by editing the libraries config files. A config that has been used file for the Xbox controller I is given in Listing 1 at the end of this appendix. The config files for controller can be found in `/home/$USER/.local/lib/python3.6/site-pacakages/approxeng/input/yaml_controllers` after aprroxeng-input is installed.
Approxeng-input can be install via pip by running:

```
$ pip3 install approxeng.input
```

You can then check the functioning of your controller with the following command:

```
$ approxeng_input_show_controls
```

Finally the scripts require some more python libraries not available by default, that can be installed with the following commands:

```
$ pip3 install h5py
$ git clone https://github.com/DelSkayn/jepture.git
$ cd jepture
$ pip install ./
```

B.2 Create local WiFi

It might be useful to be able to connect to the car when not in range of a WiFi network. It is possible to configure the car to create its own WiFi network while still being able to connect to other networks by the use of `linux-wifi-hotspot` which can be found here: https://github.com/lakinduakash/linux-wifi-hotspot.
In order to setup a network install it following the instructions. Then alter the `/etc/create_ap.conf` file to preferred settings, A config file, that has been used, can be found in Listing 2. If `nat` is used as `SHARE_METHOD`, then install dnsmasq and iptables with the following command:

```
$ sudo apt-get install dnsmasq iptables
$ sudo systemctl enable create_ap.service
$ sudo systemctl start create_ap.service
```

B.3 Setup VNC server

While most of the scripts can be executed using SSH, which is the generally prefered method, it is useful to setup a VNC server to be able to check, for example, whether the cameras are properly focused. Tigervnc is prefered, which can be install with:

```
$ sudo apt-get install tigervnc
```

And then started with:

```
$ sudo vncserver
```

Listing 1: Approxeng-input config file for the Xbox Controller config file

```
axes:
  dx:
    code: 16
    invert: false
    max_value: 1
    min_value: -1
  dy:
    code: 17
    invert: false
    max_value: 1
    min_value: -1
  lt:
    code: 2
    invert: false
    max_value: 1023
    min_value: 0
  lx:
    code: 0
    invert: false
    max_value: 32767
    min_value: -32767
  ly:
    code: 1
    invert: true
    max_value: 32767
    min_value: -32767
  rt:
    code: 5
    invert: false
    max_value: 1023
    min_value: 0
  rx:
    code: 3
    invert: false
    max_value: 32767
    min_value: -32767
  ry:
    code: 4
    invert: true
    max_value: 32767
    min_value: -32767
buttons:
  circle: 305
  cross: 304
```

```
  home: 316
  l1: 310
  ls: 317
  r1: 311
  rs: 318
  select: 314
  square: 307
  start: 315
  triangle: 308
name: Xbox Wireless Controller
product: 654
vendor: 1118
```

Listing 2: The `create_ap.conf` file for `linux-wifi-hotspot`

```
CHANNEL=default
GATEWAY=10.0.0.1
WPA_VERSION=2
ETC_HOSTS=0
DHCP_DNS=gateway
NO_DNS=0
NO_DNSMASQ=0
HIDDEN=0
MAC_FILTER=0
MAC_FILTER_ACCEPT=/etc/hostapd/hostapd.accept
ISOLATE_CLIENTS=0
SHARE_METHOD=nat
IEEE80211N=0
IEEE80211AC=0
HT_CAPAB=[HT40+]
VHT_CAPAB=
DRIVER=nl80211
NO_VIRT=0
COUNTRY=
FREQ_BAND=2.4
NEW_MACADDR=
DAEMONIZE=0
NO_HAVEGED=0
WIFI_IFACE=wlan0
INTERNET_IFACE=eth0
SSID=jetson
PASSPHRASE=liacsjetson
USE_PSK=0
```