Towards an Absolute Depth Estimation Method for Small FPV Drones

Candace Do, '24

Submitted to the
Department of Mechanical and Aerospace Engineering
Princeton University
in partial fulfillment of the requirements of
Undergraduate Independent Work.

Final Report

January 17, 2023

Professor Anirudha Majumdar
Professor Luigi Martinelli
MAE 339
61 pages
Adviser Copy

This report represents my own work in accordance with University regulations.

Abstract

We are motivated by the problem of developing a fast monocular absolute depth estimation method for small first-person view (FPV) drones such as the Crazyflie. Depth estimation provides robots with key information about the environment which allows them to plan trajectories to navigate the environment. Current methods using Simultaneous Localization and Mapping (SLAM) or stereo cameras are not desirable because they require heavy hardware or many ground-truth inputs that are not feasible for the small drone application. We propose a modified version of `merged-depth`, a method that that combines the MiDaS, SGDepth, and Monodepth2 models for estimating depth from a monocular image, that uses a ground-truth input of the maximum depth in the environment. For experiments in five indoor environments using the RealSense camera, we find that our model has error rates of 5-12%, within the acceptable range of error. We conclude that our model could be a viable solution for monocular absolute depth estimation in well-lit, high-contrast environments where the bounds of the operating area are known.

Acknowledgements

I am extremely grateful to the Intelligent Robot Motion Lab for taking me on as an undergraduate researcher, and to Professor Ani Majumdar for his invaluable feedback. This endeavor would not have been possible without Alec Farid, who guided me through a field I had almost no experience in. I would like to extend my thanks to the Department of Mechanical and Aerospace Engineering for funding this project. I'd also like to acknowledge Nate Simon and MAE 345 course staff, whose work this builds on. Finally, many thanks to my friends and family for their support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Robot navigation is paramount for developing autonomous robotic systems that can do work in complex and dangerous environments. To navigate, a robot must be able to determine its own position relative to objects in its environment, then plan its path towards a goal. Some typical image-based approaches use stereo cameras to infer depth, but this method is difficult for very small drones because they do not have the load capacity for two cameras. An alternative is depth estimation using monocular images, where a robot uses a single camera to obtain an image of its environment, then estimates the distance from each of the objects in the image. Monocular depth estimation is challenging because one cannot determine the scale of objects in the image without making some assumptions (e.g. how large a particular object, such as a tree, is).

Small off-the-shelf first-person view (FPV) drones are desirable for robot navigation in complex environments because they are highly agile, lightweight, and cost-effective. We would like to develop a method that enables small FPV drones (such as the Crazyflie 2.1, which has a maximum payload mass of 15g, or about three nickels) to use monocular depth estimation to gain an understanding of their environment. Once a drone can map its environment, it can use this information to plan a path forward.

1.2 Related work

1.2.1 Approaches for robot navigation

Recently, advancements in the field of control, estimation, and trajectory planning have made full vehicle autonomy possible, and new methods push the bounds of using low-cost sensors [18]. For vehicle autonomy, robots must be able to navigate their environments. Gul et al. [7] report several methods for robot navigation. They discuss navigation in static environments, where objects do not move, and in dynamic environments, where objects do move. They further divide navigation methods into global (off-line) or local (on-line). In global methods, the robot has full information about the environment, while in local methods, the robot must use sensors to determine its environment. In this project, we will be focusing a static local method for navigation.

One popular local method for robot navigation is Simultaneous Localization and Mapping (SLAM) which typically uses LiDAR and RGB cameras to find and identify landmarks in the environment. SLAM is useful for larger robots that have the ability to carry both LIDAR devices and cameras (see e.g. [9], [4]), but our small FPV drones do not have the ability to carry such devices. Figure 1.1 shows the approximate scale of one small FPV drone, the Crazyflie 2.1 by Bitcraze.

Figure 1.1: Crazyflie 2.1 drone next to a quarter. Courtesy of Bitcraze [2].

1.2.2 Learning-based depth estimation models

Some image-based depth estimation methods use stereo cameras, which enables a robot to take advantage of parallax information, similar to human eyes. Recently, learning-based depth estimation methods for stereo cameras have arisen using deep learning [11]. While these have grown popular in the past decade for autonomous driving and augmented reality applications, the hardware limitations of small FPV drones prevent us from using a stereo camera setup since the drones cannot carry two cameras at a reasonable distance apart for stereo-based learning.

For small FPV drones, monocular depth estimation, which leverages some of the same deep learning tools as stereo depth estimation, is more viable due to fewer hardware requirements. Some progress has been made in advancing the accuracy of depth estimation models over the past few years (e.g. [22], [3]) using deep learning methods such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and generative adversarial networks [21]. These models are typically tested on well-known benchmark datasets, such as KITTI [5], NYU Depth [16], and Make3D [15]. However, the inference times from these models are too slow for them to be useful for fast-moving FPV drones. Some progress has also been made on lowering inference times such as the FastDepth [19] model, but FastDepth calculates the relative depth rather than the absolute depth which is not useful in most navigation methods. Saxena et al. [14] have been able to calculate absolute depth using a supervised learning model, but their model has only been tested on images of trees and forests and thus cannot be applied in more general situations.

1.3 Objectives

We would like our monocular depth estimation method to satisfy some key requirements. For successful robot navigation, it must

- be able to run at 10Hz, and thus have an inference time per frame of approximately 100 milliseconds or less on a GPU,
- predict depth with $\pm$ 20% accuracy with respect to the real depth,
- have an operating range of about 5 meters (approximately 16 feet)
- be implementable on small FPV drones (i.e. does not require hardware that is too heavy for a small drone).

Chapter 2

Methods

2.1 `merged-depth` overview

An abundance of monocular depth estimation models currently exist. We chose to use `merged-depth` [17], which combines five depth estimation models to create weighted average depth predictions for each pixel in an image. `merged-depth` uses AdaBins, DiverseDepth, MiDaS, SGDepth, and Monodepth2 to calculate its depth estimation. We chose to use `merged-depth` because it provided a tool leveraging some of the most powerful monocular depth estimation tools today in a format that could be used out-of-the-box. According to the original GitHub repository, the input to the `merged-depth` model is a single image or set of images, and the output is a depth map with the absolute depth in meters.

Below is a brief description of each of the estimation models that `merged-depth` uses:

- **AdaBins** [1] builds off a traditional convolutional neural network structure by adding a transformer-based architecture block that divides the total depth range into "bins." Each depth bin is estimated adaptatively for each image. AdaBins performed better than state-of-the-art models in terms of depth estimation, using the datasets NYU Depth v2 (for outdoor scenes) and KITTI (for indoor scenes).

- **DiverseDepth** [20] aims to improve depth estimation for a variety of generalized environments and to improve the high-level geometry in predicted scenes. Yin et al. created a Diverse Scene Depth dataset to train this model, which

performed better than state-of-the-art models for NYU, KITTI, DIW, ScanNet, Eth3D, and their new DiverseDepth datasets.

- **MiDaS** [12] also tackles the issue of diverse datasets. In a process called zero-shot cross-dataset transfer, Ranftl et al. make use of five diverse datasets, including 3D films, to train their model. They then use six different datasets, including DIW, ETH3D, Sinntel, TUM-RBGD, NYU Depth v2, and KITTI, to test their model.

- **SGDepth** [10] improves the depth estmation for frames with dynamic objects, i.e. objects that move frame to frame, such as cars or pedestrians. SGDepth outperformed state-of-the-art models on the KITTI Eigen split for monocular depth estimation.

- Finally, **Monodepth2** [6] improves on state-of-the-art monocular depth estimation with several key design choices that reduce systematic depth prediction errors. Monodepth2 outperforms other self-supervised monocular and stereo models on the KITTI Eigen split.

2.2 Modifications to `merged-depth`

Running `merged-depth` out of the box revealed that this engine is too slow for real-time depth estimation on a drone flying at maximum speeds of 20 ft/s, with almost 25 seconds per frame.

To reduce the time `merged-depth` takes to analyze each frame, we decided to remove some of the 5 models that `merged-depth` uses. The two slowest models were AdaBins and DiverseDepth. Removing these two models from the pipeline sped up analysis time considerably, as shown in the table below. Note that there are two separate AdaBins models, one for outside environments and one for inside environments. This timing data is the average time for each model to analyze each of 21 images from a FPV drone camera.

AdaBins NYU	9.217 ± 0.799
AdaBins KITTI	9.118 ± 0.816
DiverseDepth	5.564 ± 0.520
MiDaS	0.251 ± 0.100
SGDepth	0.574 ± 0.052
Monodepth2	0.370 ± 0.037
Total	$\mathbf{25.137 \pm 2.185}$

Table 2.1: Average time for each `merged-depth` model, in seconds, on Intel(R) Core(TM) i7-8550U CPU over 20 trials

Without AdaBins and DiverseDepth, the total analysis time would only be 1.195 seconds. These timing tests were run on a personal computer with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz. However, using a GPU speeds up the inference time considerably. We conducted timing tests with the MiDaS, SGDepth, and Monodepth2 models on a server with eight GeForce RTX 2080 Ti GPUs, with the results shown in Table 2.2.

MiDaS	0.045 ± 0.018
SGDepth	0.022 ± 0.005
Monodepth2	0.011 ± 0.003
Total	$\mathbf{0.078 \pm 0.026}$

Table 2.2: Average time for each `merged-depth` model, in seconds, on GeForce RTX 2080 Ti GPUs over 20 trials

An inference time of 78 milliseconds per frame (faster than 10 Hz) is sufficiently fast for the drone to process while flying.

Note that the three remaining models (MiDaS, SGDepth, and Monodepth2) still improve on earlier depth estimation models with diverse scene datasets (MiDaS) and general improvements to depth estimation error (SGDepth and Monodepth2). Additionally, the five models (AdaBins, DiverseDepth, MiDaS, SGDepth, and Monodepth2) are weighted 1:1:5:1:1 respectively, so AdaBins and DiverseDepth do not affect the final depth estimation much. We found that the full model (including AdaBins and DiverseDepth) results in a similar error rate to the modified model; the full model had an error of about 12% on Environment 5 while th modified model had

an error rate of about 11%.

2.3 Camera selection

2.3.1 Wolfwhoop WT05 AIO camera

Our preliminary experiments were conducted with the Wolfwhoop WT05 AIO Camera that is used in MAE 345 (Introduction to Robotics). This is an inexpensive and lightweight camera built for FPV Quadcopter Drones, and has the advantage of having a built-in video transmitter and antenna. However, the video quality is quite low at 600 television lines (TVL) as shown in the figure below, and we suspected that the low video quality was hindering accurate depth predictions. Below is a sample image captured in a dorm room. Note that the image is quite overexposed in the bright areas and colors are washed out and green-tinted.

Figure 2.1: Image from a Wolfwhoop camera

2.3.2 Intel RealSense camera

For further testing, we decided to use the Intel RealSense D415 Depth Camera. This camera has the advantage of having both a RGB sensor and a depth sensor, which means we can verify the accuracy of a `merged-depth` prediction using the depth data from the same frame. The RGB camera quality was also markedly better. Below

is a sample image from the RealSense's RGB camera, taken in a dorm room. The image is correctly exposed and colors are accurate. These properties conform to the datasets that `merged-depth`'s models are trained on.

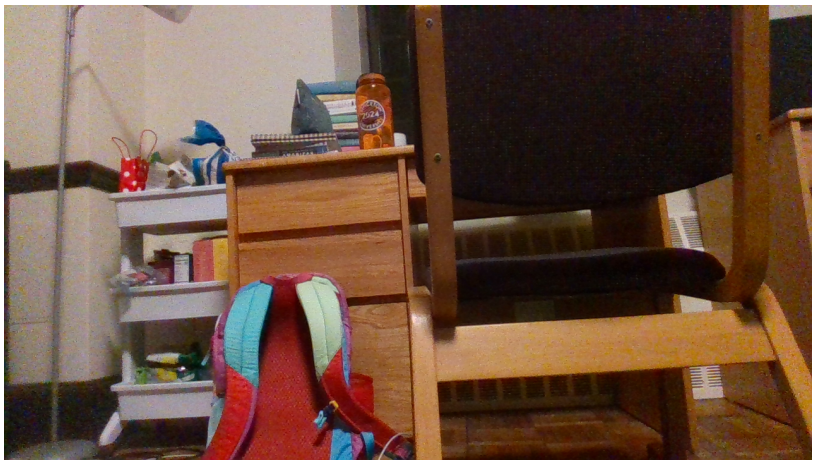


Figure 2.2: Image from a RealSense camera

See Appendix B.1 for the script used to capture RealSense data.

Note that future work with the Crazyflie drone will require a small FPV camera built for drones due to the load limit on the Crazyflie (10 grams). The Intel RealSense camera (62 grams) is much too heavy for a small drone to carry, but could potentially be used for larger drones.

2.4 Depth estimation pipeline

2.4.1 Image capture and pre-processing

The `merged-depth` engine does not require much pre-processing of images. The engine as published on GitHub currently accepts `.png` or `.jpeg` file formats, but can be easily modified to accept other file formats such as `.jpg`.

Image capture will look different for various cameras. On the Wolfwhoop camera (and other similar drone FPV cameras), individual frames can be captured easily from the video stream using the Python package OpenCV (`cv2`).

`cv2` can also be used to capture RGB frames (as well as depth data) from the RealSense camera, along with the package `pyrealsense2`.

2.4.2 Fisheye correction

As seen in Figure 2.1, many drone FPV cameras employ a wide-angle lens that leads to a fisheye effect. Since most depth estimation models are trained on datasets without fisheye effects, we were concerned that this would negatively affect the depth estimation accuracy. We performed a fisheye correction procedure using OpenCV according to Kenneth Jiang's blog post [8] on one set of images and compared the `merged-depth` results to the same set of images without the fisheye correction. As seen below, there seems to be marginal difference in the quality of depth predictions. While the fisheye-corrected image provides a more accurate depiction of the size and scale of objects, the depth values are similar. Depending on the particular camera used, fisheye correction may be useful, but it will add to the analysis time for each frame.

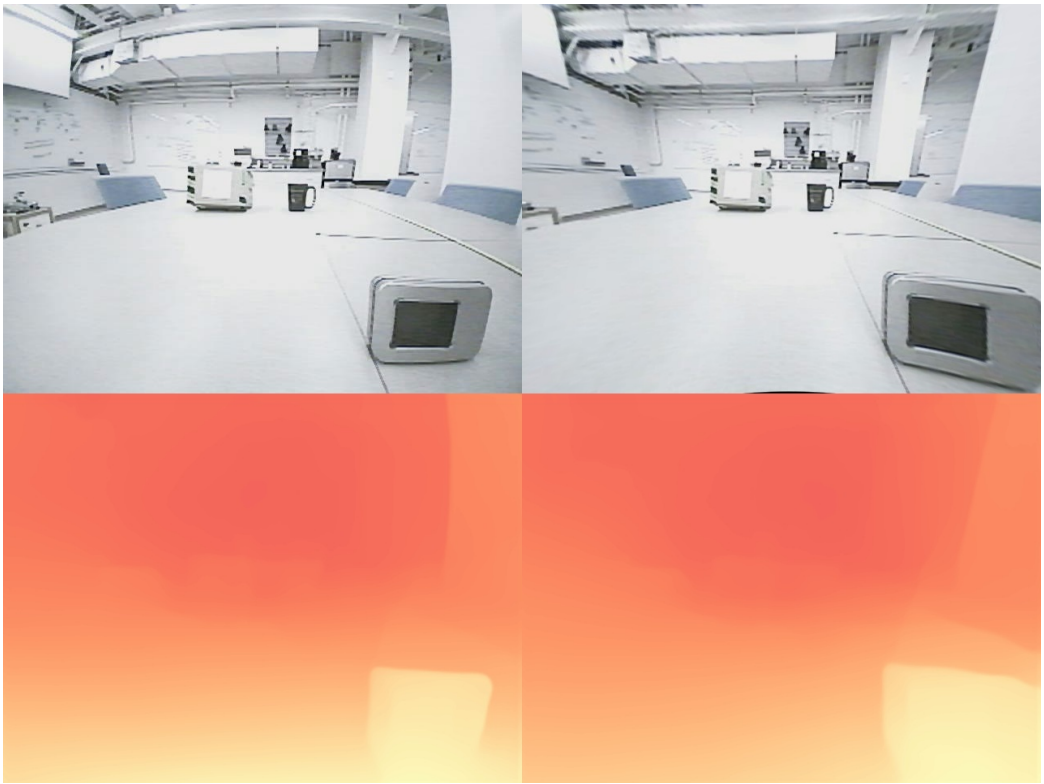


Figure 2.3: Comparison of `merged-depth` results before (left) and after (right) fisheye corrections. In the depth maps, orange areas are farther from the camera and white/yellow areas are closer.

.

The RealSense camera does not have a wide-angle lens, so it does not need any fisheye corrections.

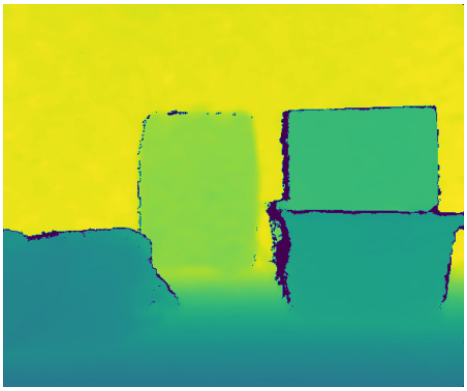
2.4.3 `merged-depth` processing

The `merged-depth` engine, after our modifications, runs the MiDaS, SGDepth, and Monodepth2 models, then weights each of these predictions in the ratio 5:1:1 respectively to provide the weighted average depth prediction for each pixel. After providing the correct input and output image folders to the script, `merged-depth` will output a set of depth maps (in `.npy` format), as well as a stacked image of the original image and the colorized depth map.

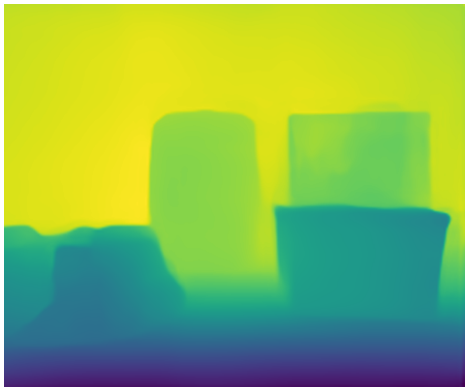
2.4.4 `merged-depth` correction

While `merged-depth` seems to provide a good visual depth map, upon further inspection, further corrections need to be made to the depth map output of the `merged-depth` engine. We ran several experiments where we moved the RealSense camera forwards in a scene with several objects to simulate the drone in its environment. We compared the `merged-depth` depth map with the depth map from the RealSense depth camera. Below are two depth map images from the RealSense `merged-depth`, as well as the original RGB image.

(a) Original RGB image



(b) RealSense depth map



(c) `merged-depth` depth map

Figure 2.4: RealSense and `merged-depth` depth maps for sample frame. In the depth maps, yellow areas are farther from the camera and blue areas are closer.

The dimensions for the RealSense depth map are not the same as the other images because there are a few hot pixels in the RealSense depth camera, so we cut off about 60 pixels from the left side of the frame. We then plotted each pixel's real (RealSense) depth value with its inferred (`merged-depth`) value, as shown in Figure 2.5.

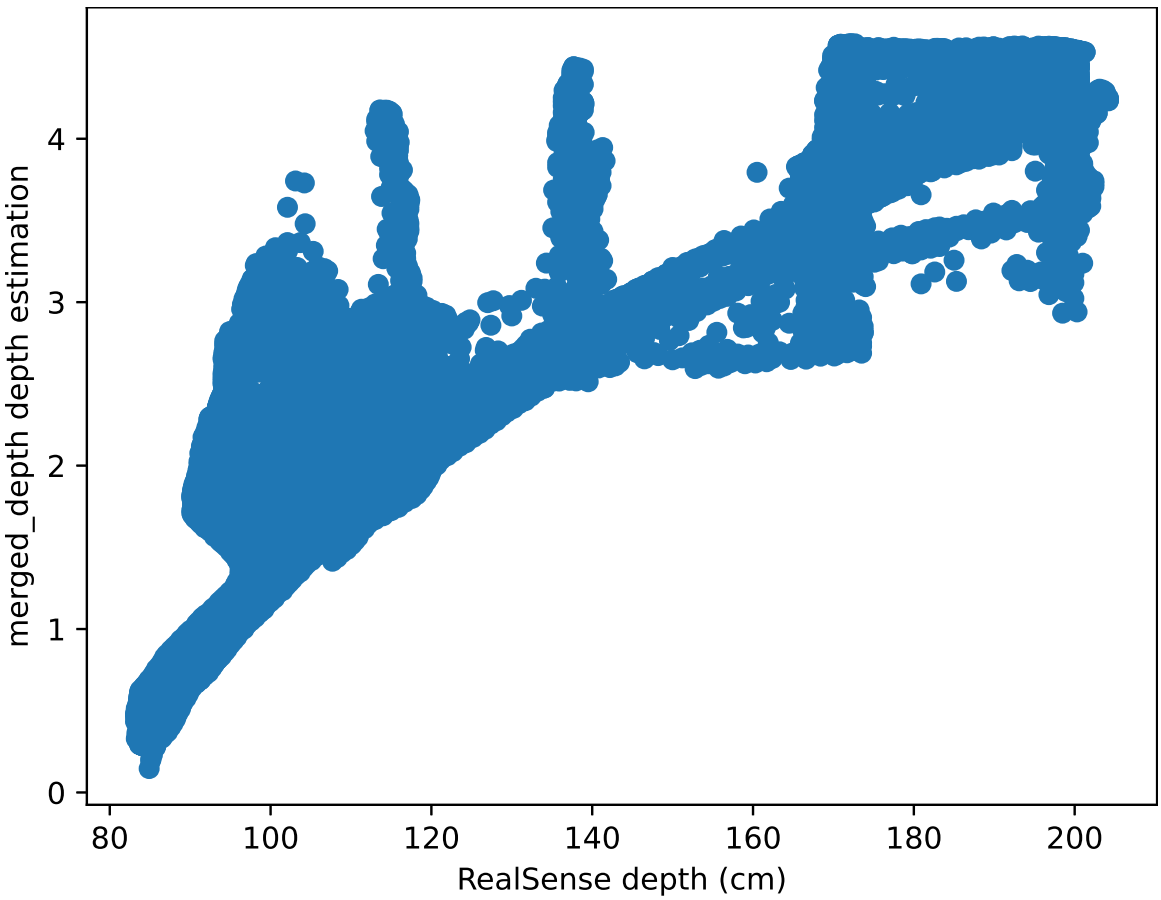


Figure 2.5: Real depth values vs. `merged-depth` predicted values

Since the RealSense depth camera outputs depth in increments of 1 cm, each real depth value corresponds to multiple inferred values. Additionally, since the inferred depth map is not entirely accurate, this is an expected result.

This plot of real vs. inferred value does look linear across datasets, and thus we decided model the function converting the inferred `merged-depth` to real depth as a linear function. This was done simply using the `polyfit` function from `numpy` to find a best-fit line across the data. For the particular frame shown, the correlation coefficient is was 0.93, indicating that a linear model is a reasonable assumption for this data. In the figure below, the depth points are in blue and the best-fit line is in red.

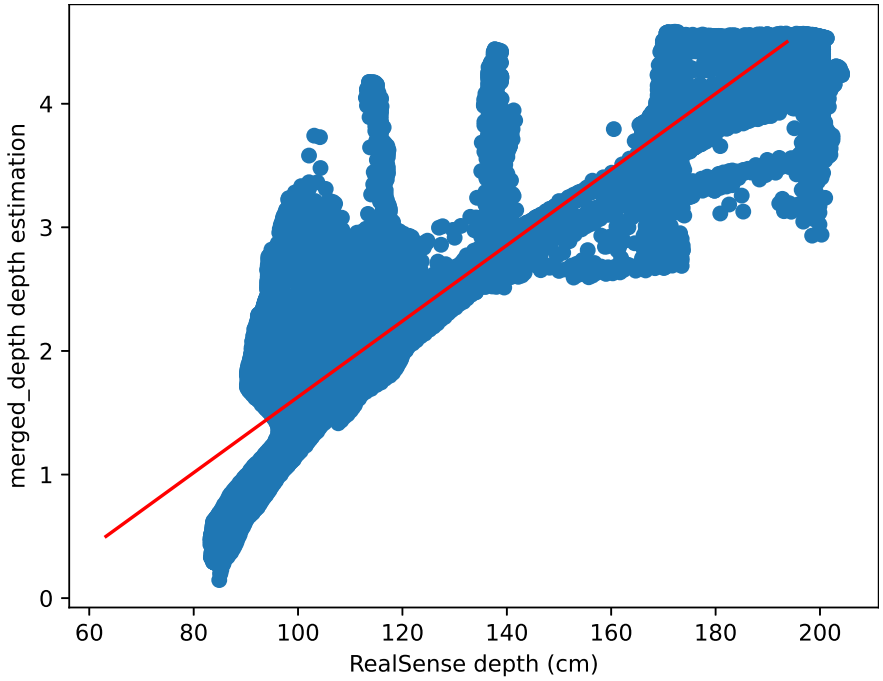


Figure 2.6: Linear model for real depth values vs. `merged-depth` values with correlation coefficient 0.93

To more easily see the linearity of the data, we can take the average of the `merged-depth` values corresponding to each real depth value, and plot them along with the linear model found earlier.

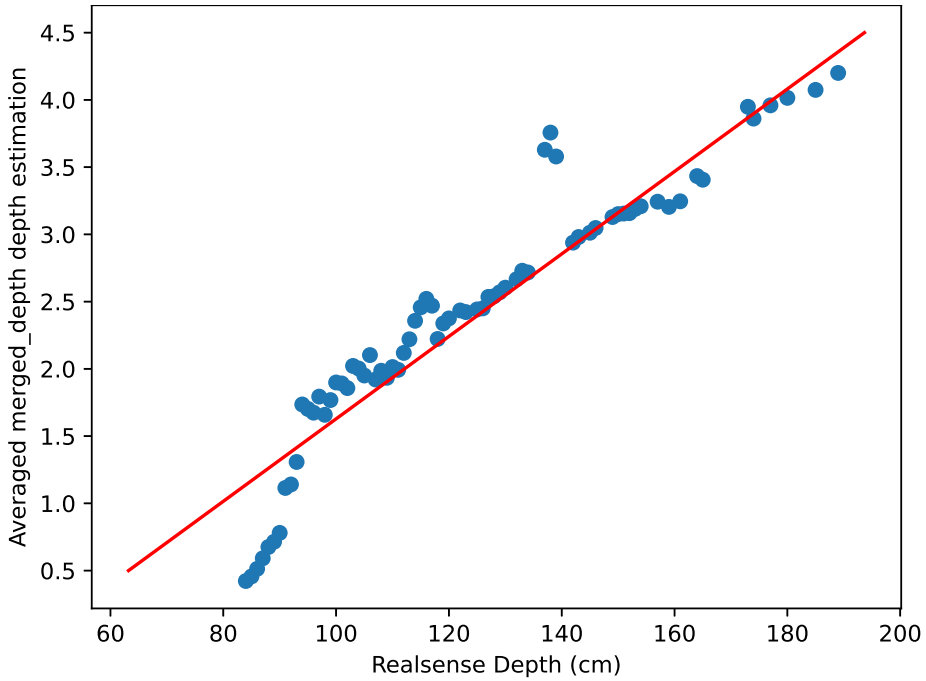


Figure 2.7: Real depth values vs. averaged `merged-depth` predicted values

When plotting several frames together, note that the plots shift to the left or right due to the camera moving, but the scale stays roughly the same.

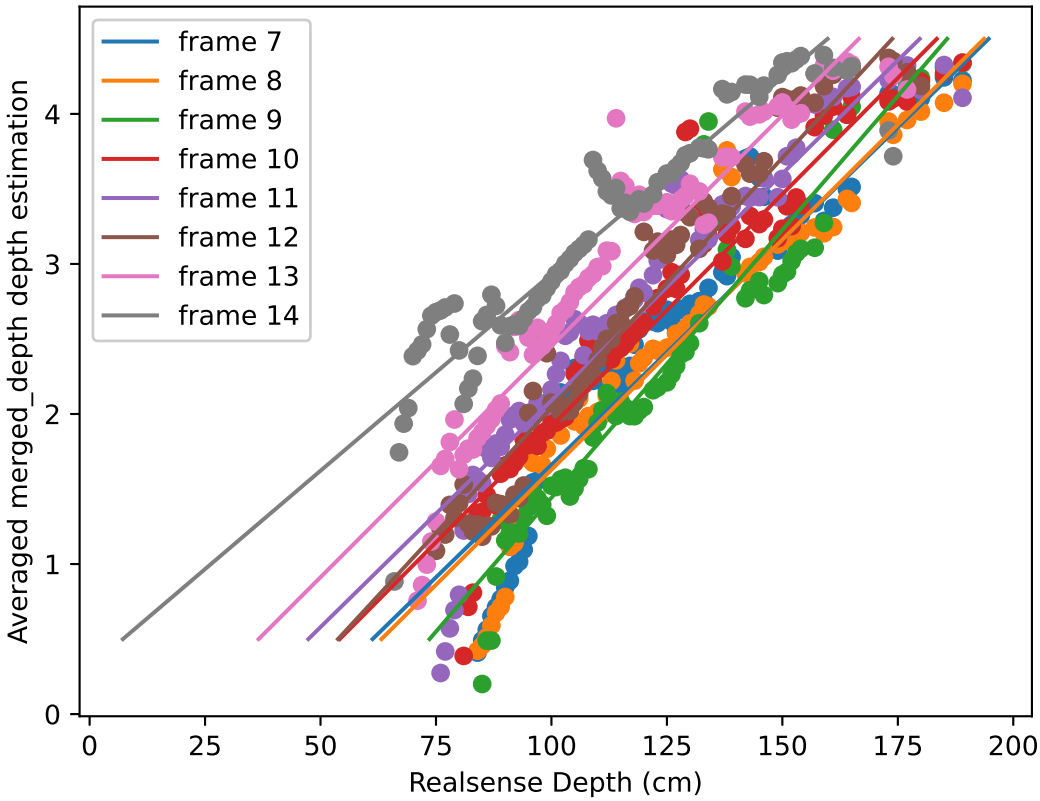


Figure 2.8: Real depth values vs. averaged `merged-depth` predicted values for 8 frames, along with best-fit lines for each frame

To account for discrepancy in the shift, we choose to determine the specific shift for the linear prediction on a frame-by-frame basis. This requires knowing the distance of a particular point in the image at all times. For convenience, we choose the far wall as our known point. Since the Crazyflie drone has localization abilities, it is a reasonable assumption that we can know how far away the wall is at all times for an indoor environment. This should be achievable if the bounds of the operating area are known (e.g. the size of the particular room).

To calculate the scale, we have two options. The first option is to gather depth data about the environment using the RealSense camera (or other depth sensor/camera), then use `merged-depth` to determine the average scale for this set of sample images. Then, use this average scale for any new images. The second option is to choose a

set scale. From the data collected from 5 indoor environments, the average scale was between 15-35. See Figures 3.1-3.5 for examples of the data collected.

To create the corrected depth map, we find or determine the scale from the `merged-depth` prediction to the real depth. Then, we use a point on the wall to determine the exact shift. Finally, we use this scale and shift to correct the original `merged-depth` prediction and attempt to match the RealSense depth map. Below is a sample image of the corrected `merged-depth` depth map.

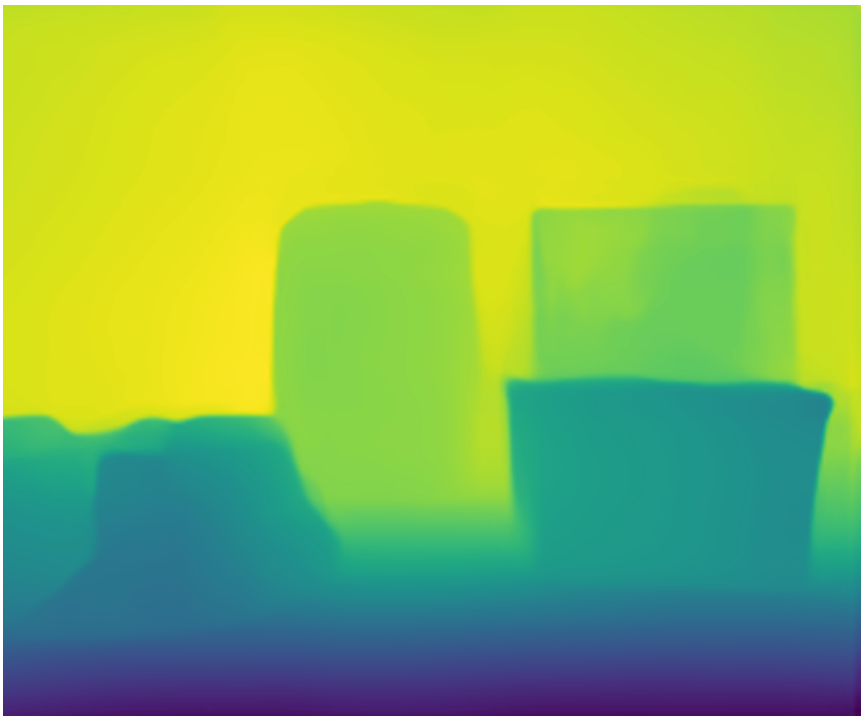


Figure 2.9: Corrected `merged-depth` map

See Appendix B.2 for the script used to create these plots.

2.4.5 Pipeline summary

To create a depth map for a new environment, we do the following:

1. Capture RGB image(s) of the environment.
2. Run the image(s) through the modified `merged-depth` algorithm.
3. Find or determine a scale from the `merged-depth` estimation to the real depth.
4. Use the known distance from some object in the frame(s), usually a wall, to calculate the shift.

5. Use the scale and shift to calculate the corrected depth predictions, using the following formula:

$$\text{corrected depth} = (\text{scale}) \times (\texttt{merged-depth}\ \text{prediction}) + \text{shift}. \tag{2.4.1}$$

Chapter 3

Results

3.1 Experiments

To test the depth estimation pipeline and determine its accuracy relative to the "ground truth" RealSense depth data, we collected images and depth data from five indoor environments with a variety of objects, similar to what we might expect an experimental drone to encounter in a lab environment. For each environment, we first captured a set of 20 images using the RealSense RGB camera and depth information using the RealSense depth camera; see `capture.py` in Appendix B. Each frame is 1/3 seconds after the previous. During image capture, we moved the camera forward at a rate of about 6 in/s to simulate the drone flying forwards slowly. Below are sample frames (RGB and depth) for each of the 5 environments. Note that the depth map is not as wide as the RGB image. Due to pixel hotspots on the depth camera, we cut off 20 pixel columns on the left side of the depth map image. For the depth maps, note that that the yellow areas are farther from the camera and blue areas are closer.

(a) RGB image **(b)** Depth map

Figure 3.1: RealSense data for Environment 1

(a) RGB image **(b)** Depth map

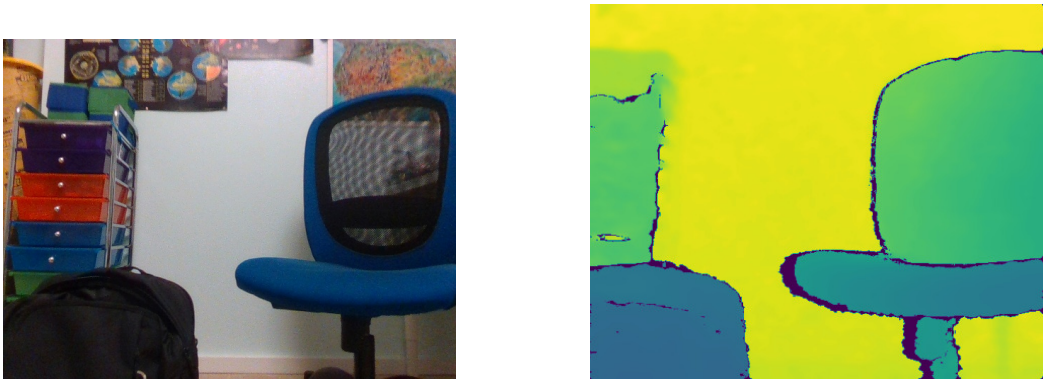
Figure 3.2: RealSense data for Environment 2

(a) RGB image **(b)** Depth map

Figure 3.3: RealSense data for Environment 3

(a) RGB image **(b)** Depth map

Figure 3.4: RealSense data for Environment 4

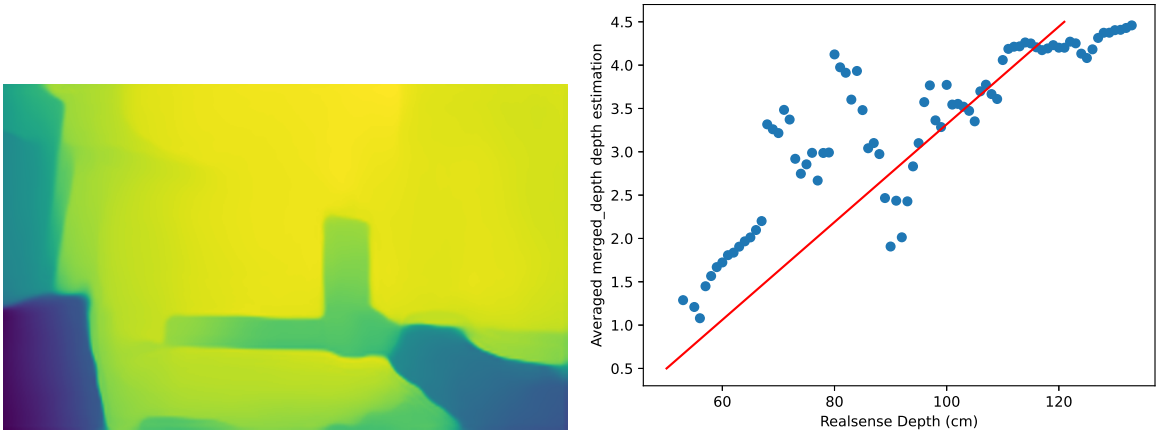


(a) RGB image **(b)** Depth map

Figure 3.5: RealSense data for Environment 5

The dark spots in the RealSense depth maps are pixels with value 0; they correspond to areas that the RealSense depth camera was not able to gather sufficient data from to create a depth prediction.

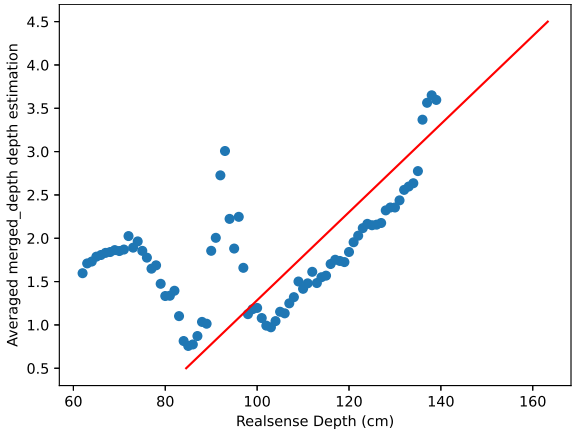
After capturing the image data, we ran each of the RGB images through the `merged-depth` engine and plotted the RealSense depth values versus the averaged corresponding `merged-depth` values for each frame, as described in Chapter 2. We also plotted the best-fit lines from the depth values prior to averaging. Below are sample `merged-depth` depth maps, scatter plots (in blue), and best-fit lines (in red) for each of the frames shown above.



(a) `merged-depth` depth map **(b)** Predicted vs. real depth values

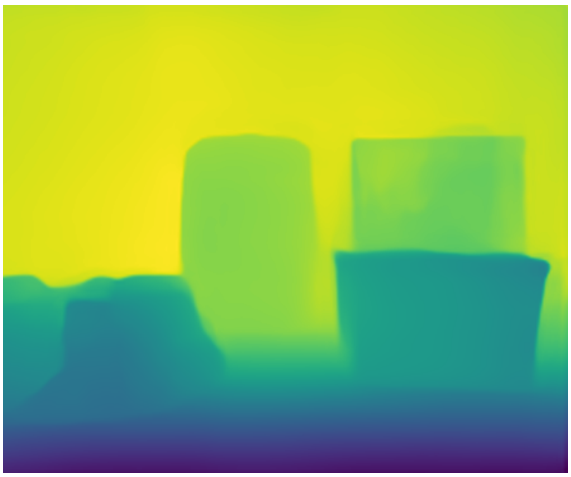
Figure 3.6: `merged-depth` output for Environment 1

(a) `merged-depth` depth map

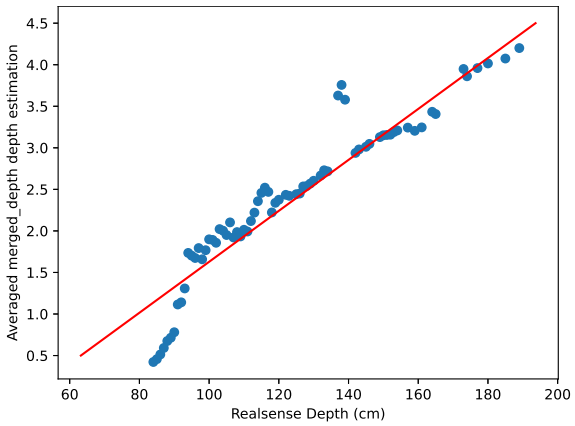


(b) Predicted vs. real depth values

Figure 3.7: `merged-depth` output for Environment 2

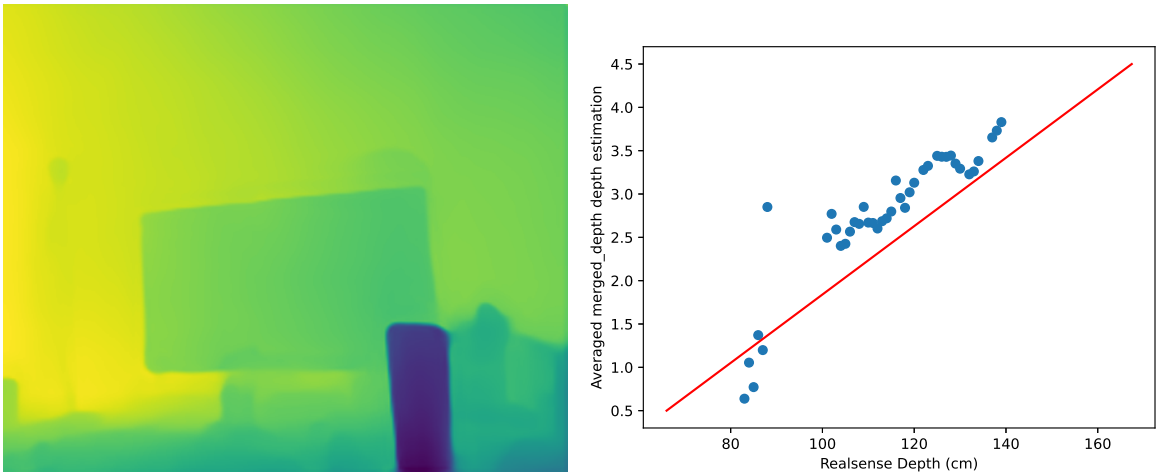


(a) `merged-depth` depth map



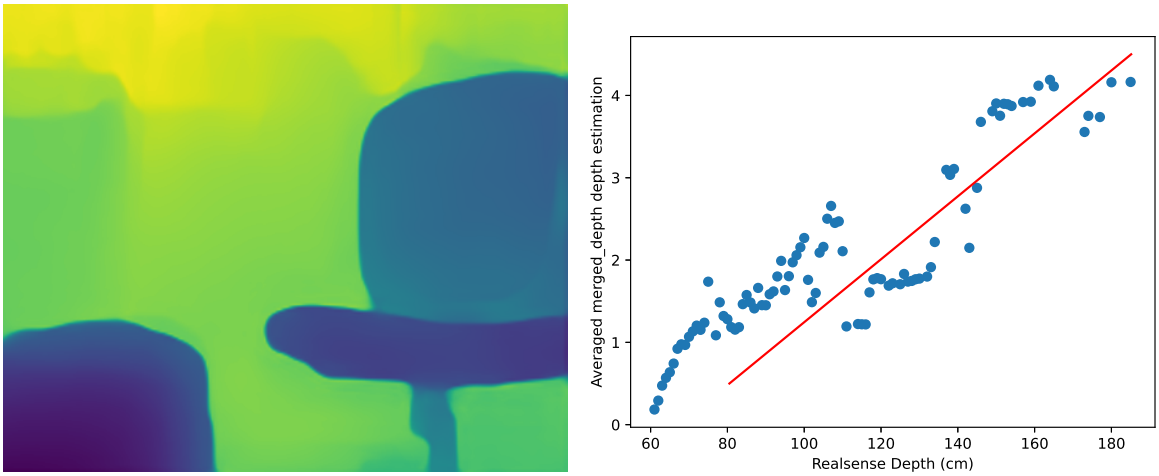
(b) Predicted vs. real depth values

Figure 3.8: `merged-depth` output for Environment 3



(a) `merged-depth` depth map **(b)** Predicted vs. real depth values

Figure 3.9: `merged-depth` output for Environment 4



(a) `merged-depth` depth map **(b)** Predicted vs. real depth values

Figure 3.10: `merged-depth` output for Environment 5

Then, for each environment, we picked a scale to correct the `merged-depth` prediction as described in Section 2.4.4 using the averaged scale determined from five frames in the set of images. Note that we would not have access to the real distances from the wall (i.e. ground truth information) on the real drone hardware platform; however, it is reasonable to assume that we can either get this ground truth information from the drone's localization information (which is possible on the Crazyflie) or we can obtain RealSense data before using the hardware platform.

We determined the shift for a particular frame by picking a point on the wall, determining the distance from the wall using the RealSense depth map, then calculating

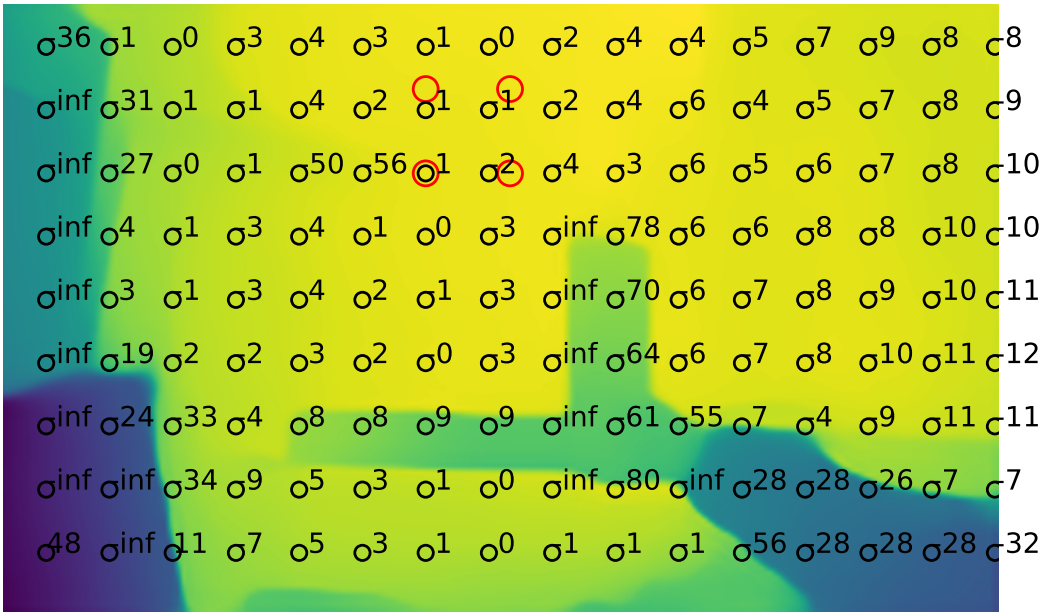
the shift from the `merged-depth` prediction to the real distance. Finally, we applied this linear correction (scale and shift) to all points in the `merged-depth` depth map to obtain the corrected depth predictions.

3.2 Depth prediction error

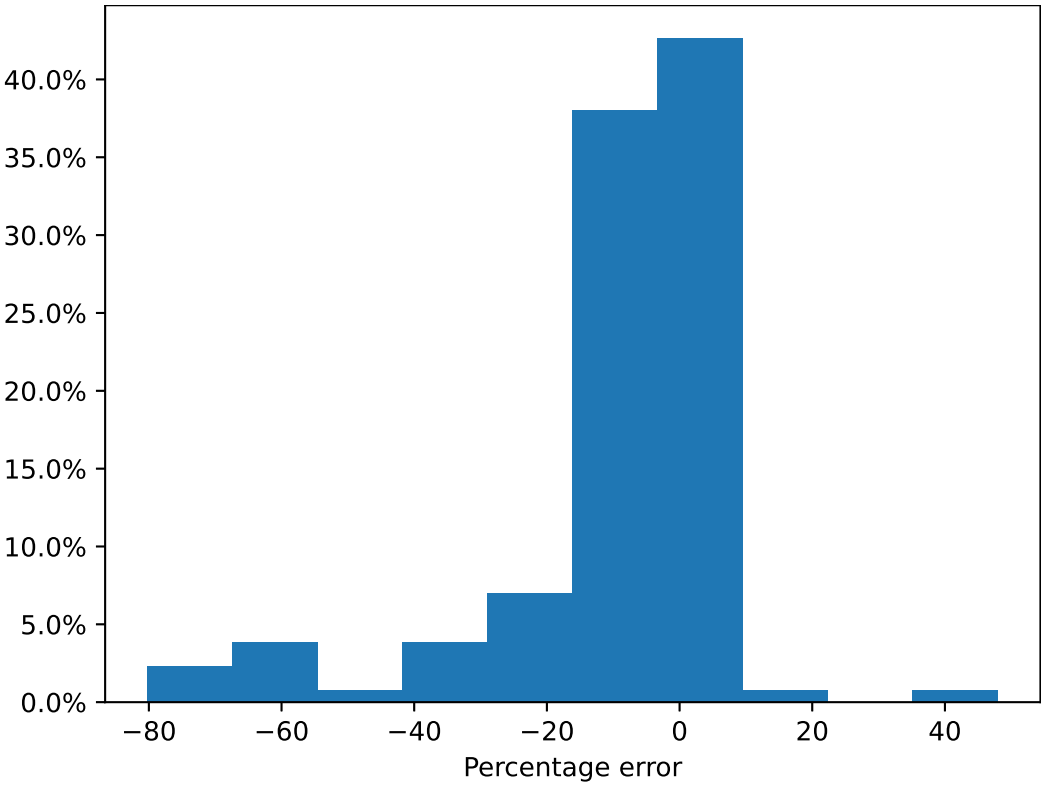
To quantitatively verify the depth predictions from `merged-depth`, we calculated the percentage error of the `merged-depth` estimate relative to the RealSense depth values using the following equation:

$$\% \text{ Error} = \frac{(\text{RealSense value}) - (\text{corrected } \texttt{merged-depth} \text{ value})}{\text{RealSense value}}. \tag{3.2.1}$$

For each image, we took sample points at equally spaced intervals in the image (circled in black), calculated the percentage error, and created a histogram of the errors. We also found the minimum, maximum, and average absolute error for the image. Below are the errors for each sampled point as well as error histograms for each sample image. On each depth map, the points with infinite error are points where the RealSense depth maps had a value of 0 (i.e. the depth camera was not able to determine their distances). The points on the wall we took to find the appropriate shift are circled in red.



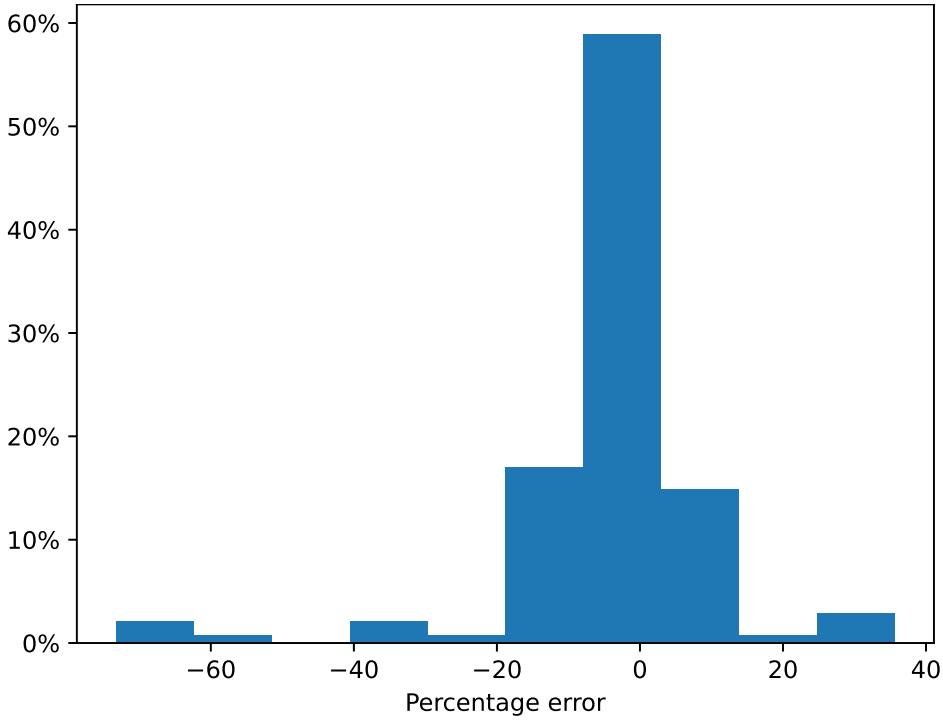
(a) Percentage errors



(b) Error histogram

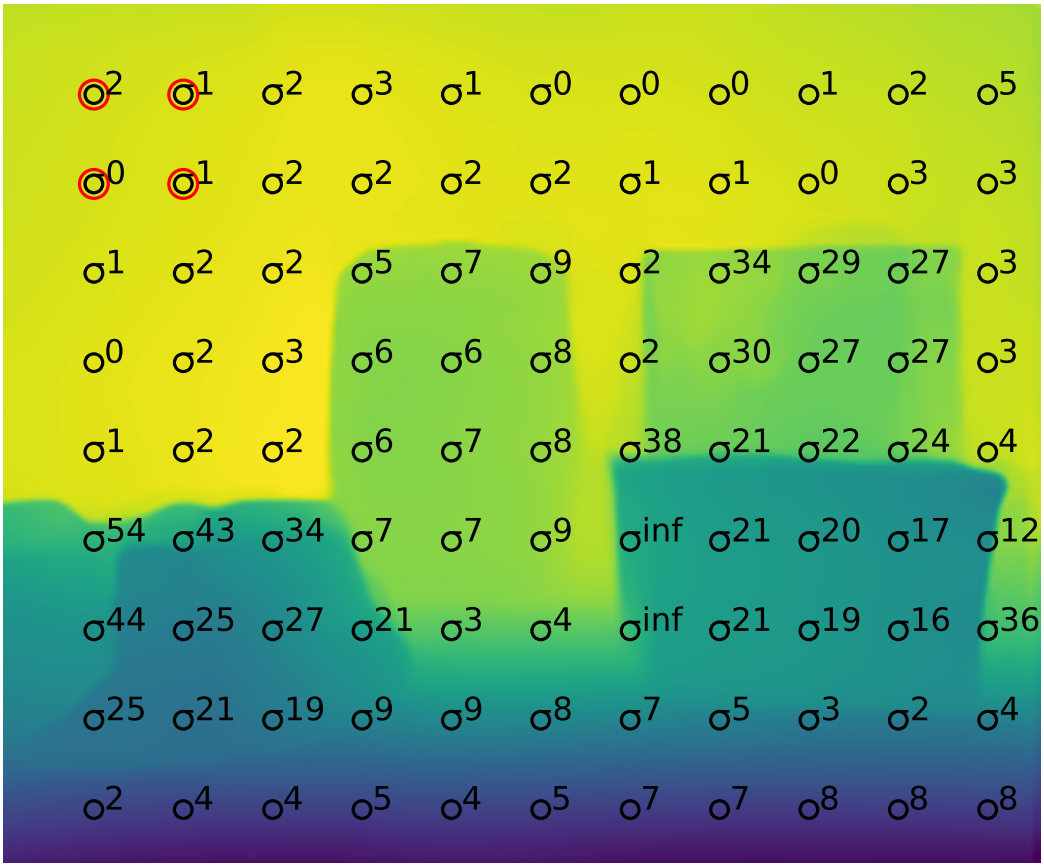
Figure 3.11: Calculated errors for Environment 1

(a) Percentage errors

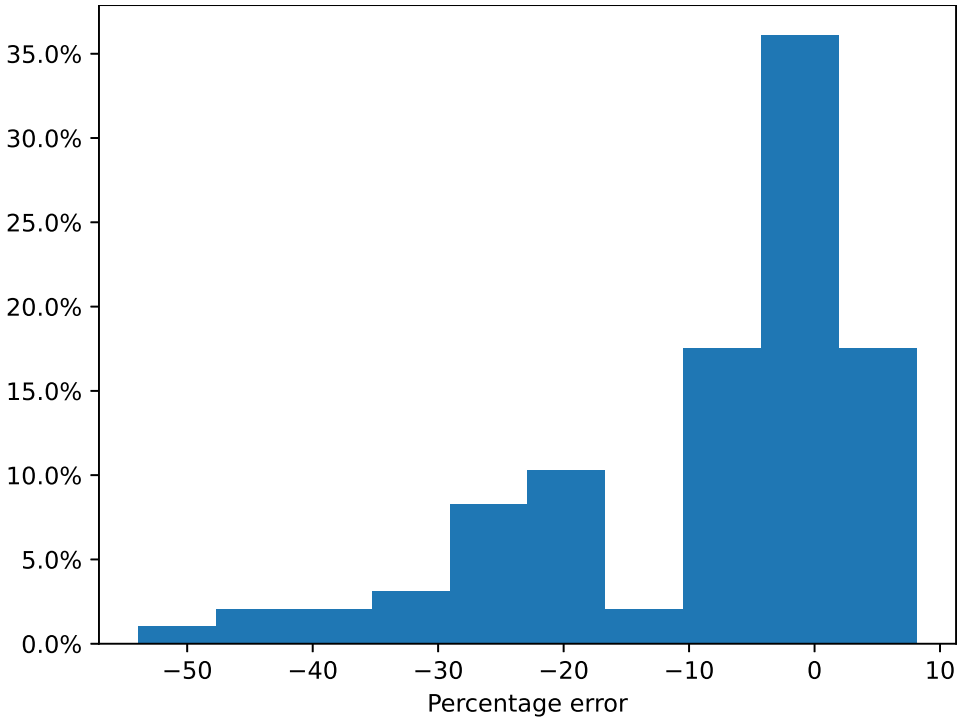


(b) Error histogram

Figure 3.12: Calculated errors for Environment 2

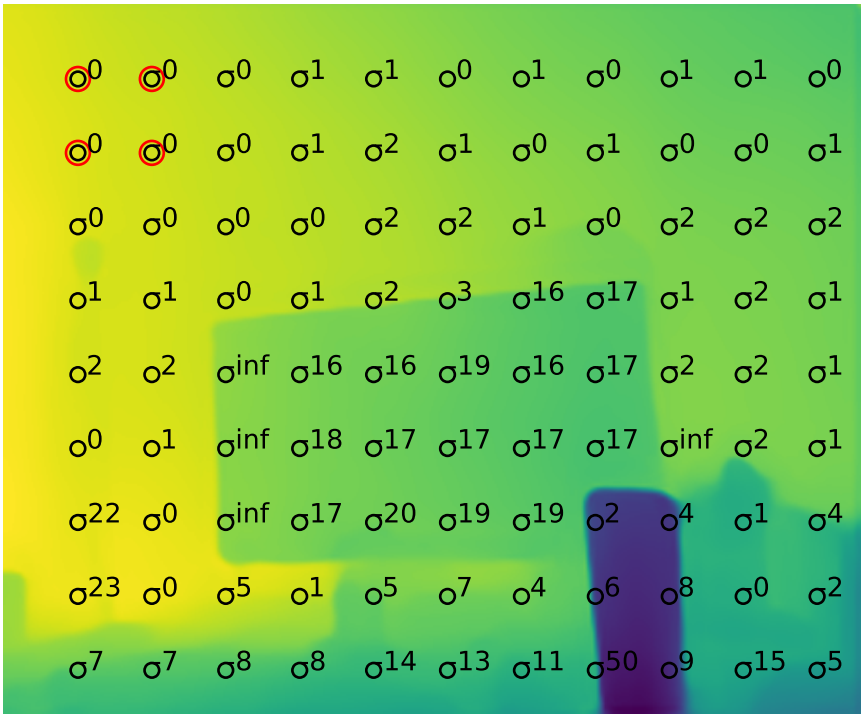


(a) Percentage errors

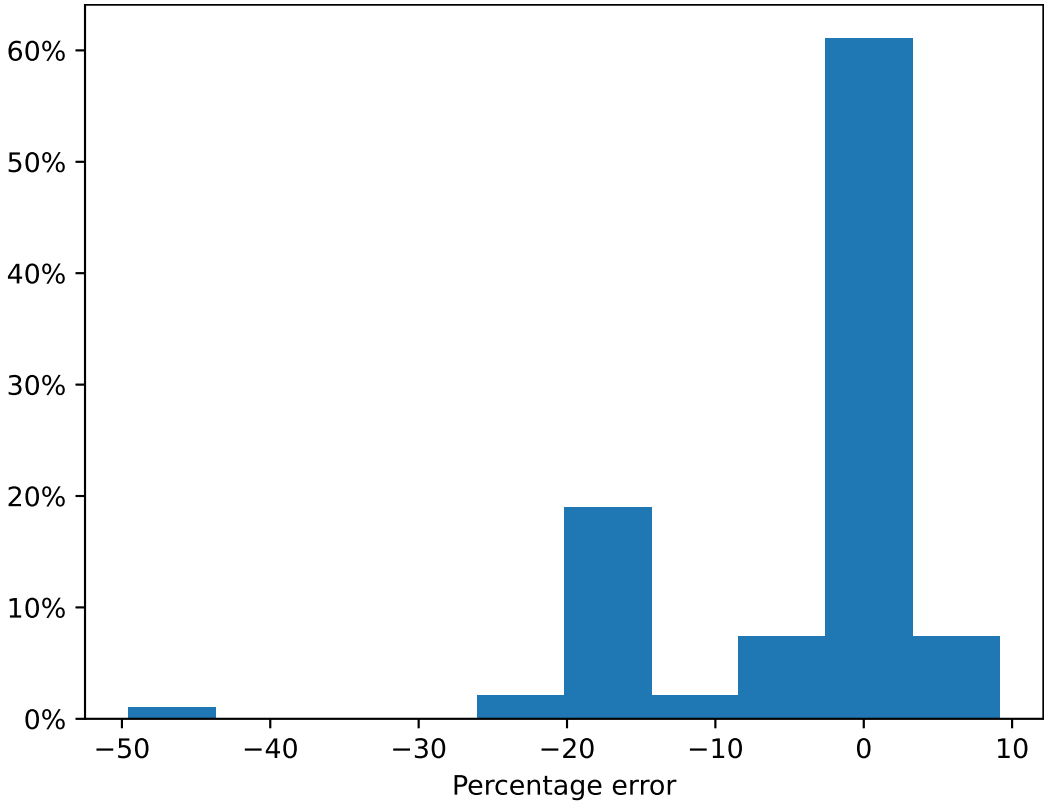


(b) Error histogram

Figure 3.13: Calculated errors for Environment 3

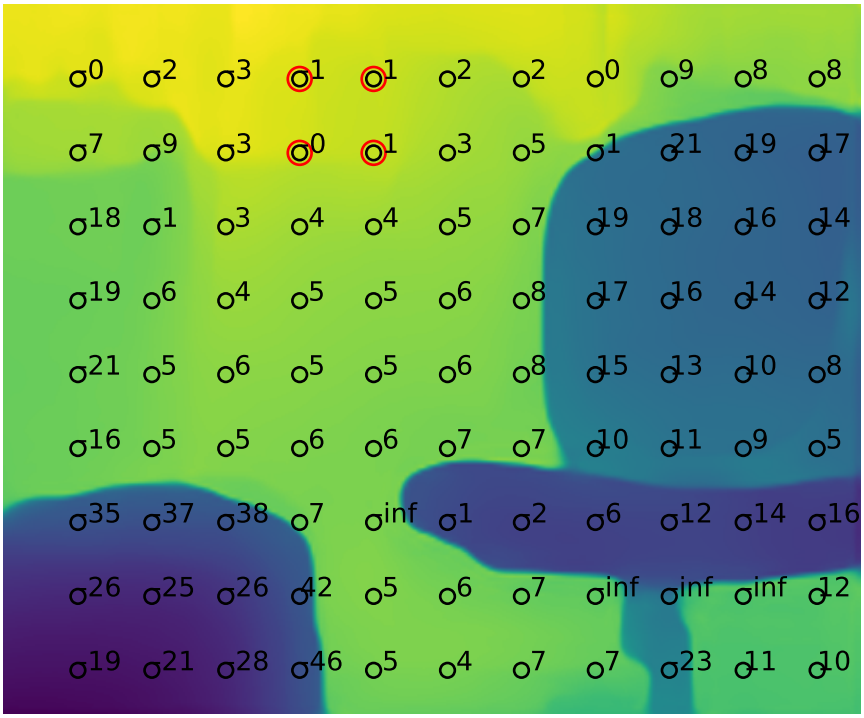


(a) Percentage errors

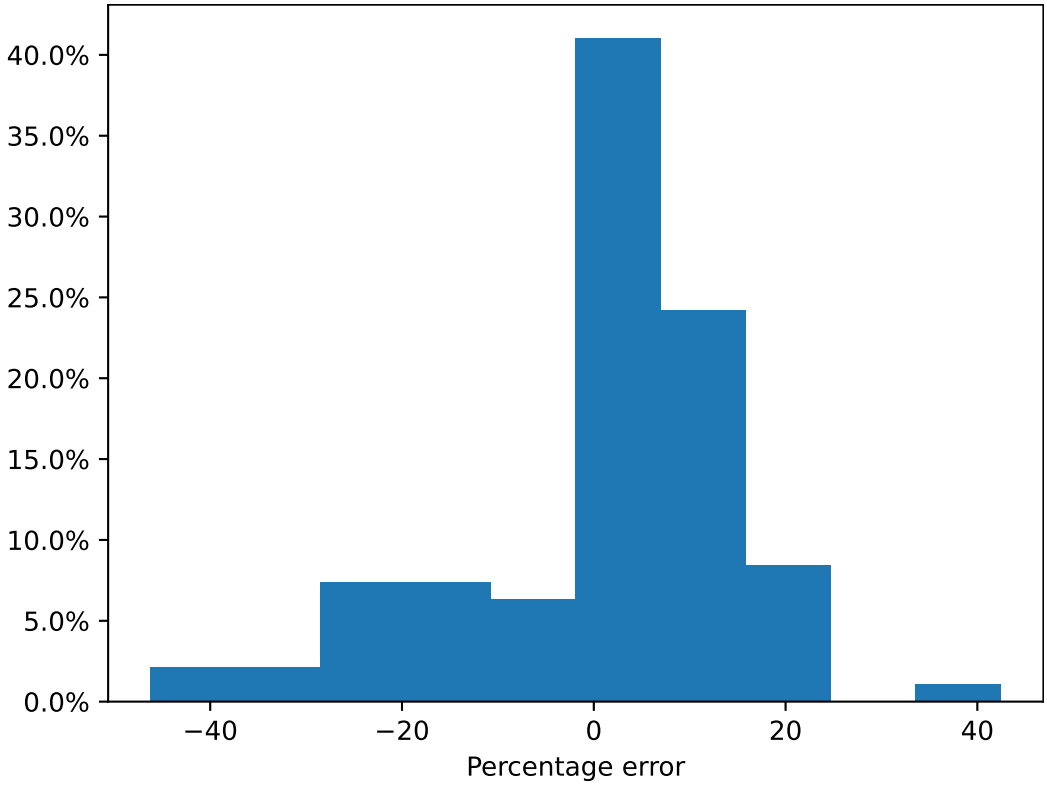


(b) Error histogram

Figure 3.14: Calculated errors for Environment 4



(a) Percentage errors



(b) Error histogram

Figure 3.15: Calculated errors for Environment 5

Table 3.1 contains a summary of the minimum, maximum, and average absolute errors, as well as the scales and shifts used for each image:

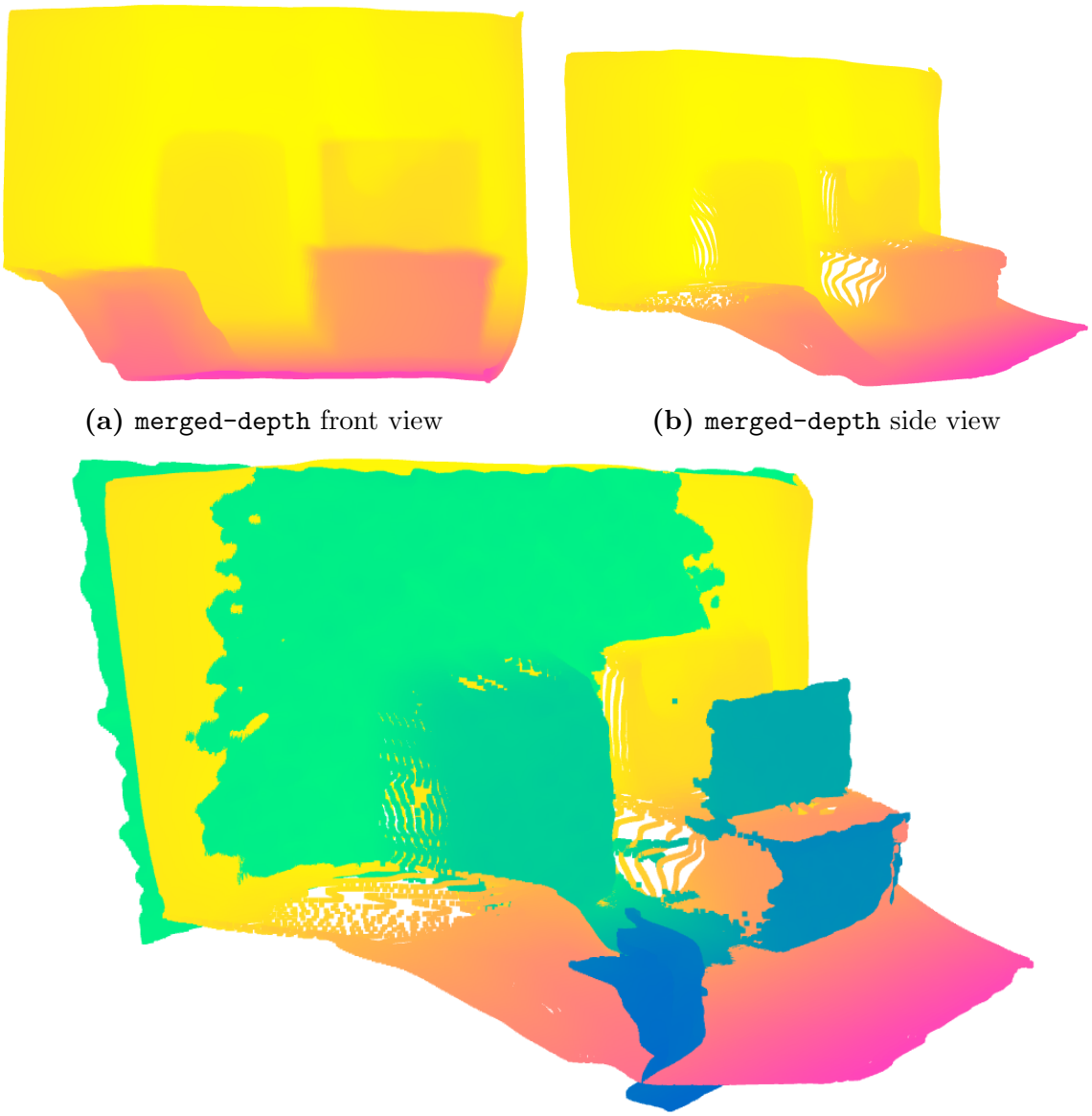
Env.	Scale	Shift	Min. Err. (%)	Max. Err. (%)	Avg. Abs. Err. (%)
1	17.73	47.25	-80.18	11.78	11.78 ± 16.91
2	19.68	74.35	-67.53	38.10	7.39 ± 12.52
3	32.61	56.42	-53.93	8.17	10.50 ± 11.81
4	25.34	58.97	-49.55	9.17	5.96 ± 8.12
5	26.16	64.79	42.46	-46.21	10.86 ± 9.58

Table 3.1: Error statistics for sample images from each environment

From these sample frames, we see that the scale used does vary across environments; however, the scale is relatively consistent within a single environment, as seen in Figure 2.8. The shift does vary across environments, as expected. In general, the error is negative, which suggests that `merged-depth` seems to overestimate the distance from the camera to the object for all of these environments; this is useful information for a real navigation algorithms since we can expect objects to be closer than `merged-depth` predicts. The average absolute error for all five environments was within 12%. The maximum distance in any environment was about 2 meters, so on average, the `merged-depth` prediction was about 24 cm (or just under 1 foot) off. However, we also note that the standard deviation of the error is high, between 8% and 17% across the five environments. We can see in the error plots from Figures 3.11-3.15 that the error tends to be higher for objects that are not the wall (generally about 20% error).

3.3 Point cloud comparison

To gain a qualitative understanding of the accuracy of the `merged-depth` predictions, we used the Open3D package to generate point cloud visualizations of the depth maps. We created point clouds from the corrected `merged-depth` predictions as well as the RealSense depth maps. As an example, Figures 3.16 and 3.17 show point cloud visualizations for the `merged-depth` predictions for Environments 3 and 4. We also overlaid the `merged-depth` and RealSense depth maps to compare them. Here, we've provided front and side views of the `merged-depth` point clouds; it is easier to see the predicted depth discrepancies in the side views for the overlaid maps.



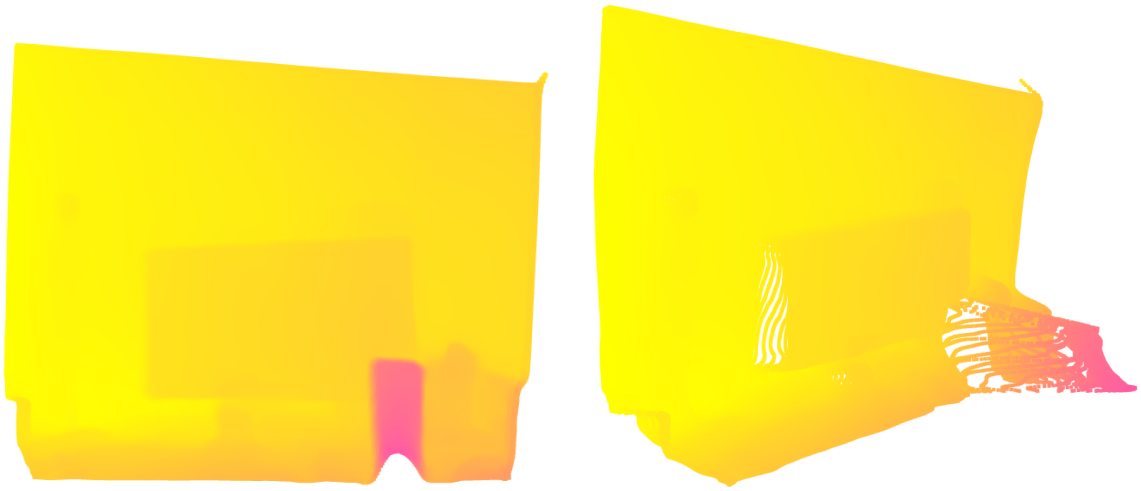
(a) `merged-depth` front view

(b) `merged-depth` side view

(c) Overlaid `merged-depth` and RealSense view

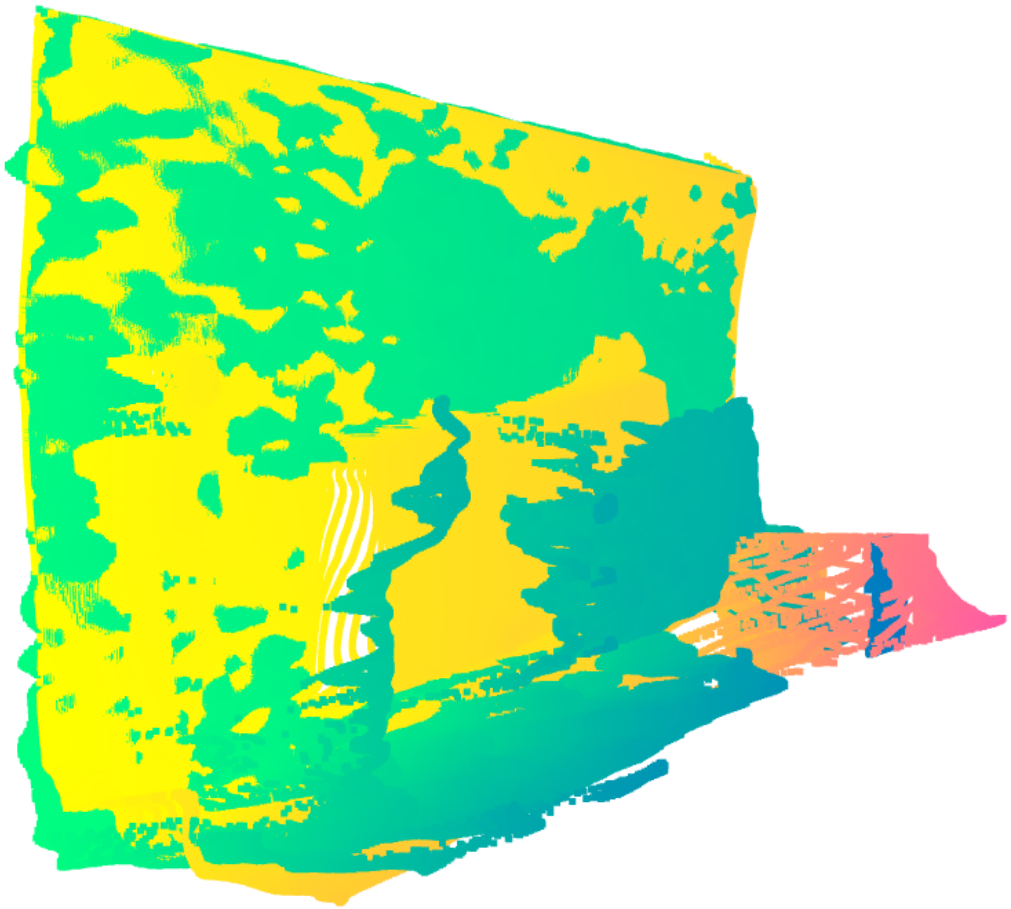
Figure 3.16: Point clouds for Environment 3. In the overlaid maps, the RealSense depth map is mapped in green-blue, and the `merged-depth` map is in pink-yellow.

Figure 3.17 shows similar point clouds for the sample frame from Environment 4.



(a) `merged-depth` front view

(b) `merged-depth` side view



(c) Overlaid `merged-depth` and RealSense view

Figure 3.17: Point clouds for Environment 4

In the overlaid point cloud, we can see that some objects in the the RealSense point cloud are closer to the viewer than `merged-depth` predicts they are; this is also

reflected in the quantitative errors we found earlier (see Figure 3.13). For instance, note the duffel bag in the bottom right corner in Figure 3.16c (see Figure 3.3a for RGB image). The `merged-depth` prediction (in orange/pink) is further away from the camera than the real depth (in blue).

See Appendix B.3 for the script used to calculate error and plot point clouds.

See Appendix A for point clouds for Environments 1, 2, and 5.

Chapter 4

Discussion

4.1 Comments on accuracy of `merged-depth`

As seen in Table 3.1, on average, the absolute error for the five tested environments was between 5-12%; this translates to less than 12 inches (30 cm) of error in these environments. However, the standard deviation of error was somewhat high, between 8% and 17%. Taking a closer look at the percentage error in Figures 3.11-3.15, we can see that foreground objects tend to have a higher error rate as `merged-depth` does not accurately predict how far they are from the background. This accuracy varied across environments. For instance, in Environment 1, we can see in Figure 3.11 that the water bottle in the center had particularly high error rates, around 70%. The arm in the lower right corner had error rates of about 28%. In Environment 2 (Figure 3.12), the error rates for objects are much lower; the recycling bin on the right has an average error of about 10%, while the fruit snacks box in the center-right has an error of about 7%. The couch (in the lower half of the image) has varying error; from about 5% on the left to 50% on the far right of the image. We can perform similar analysis on the images from Environments 3, 4, and 5, and we find that most objects have an average error rate of 10-20%, with some exceptions at 30-50%.

Additionally, in Environment 3 (Figure 3.13), note that the floor at the bottom of the image has a fairly low error rate, 5% on average. This suggests that the `merged-depth` model was able to accurately predict the closest points in the image, even though not all the intermediate objects had accurately predicted depths.

We also note that the `merged-depth` algorithm can be easily tricked particularly in situations where there is low contrast between different objects. For example, in

Environment 1 (see Figure 3.11 and Figure 3.1), the water bottle and wood detail on the wall are a very similar color, and `merged-depth` predicted that they would be the same distance away, though this is clearly not the case. This resulted in a high error rate for the water bottle (70%), though the wood detail has an error rate of about 9%. In the other environments, different objects had relatively high contrast in the RGB images, and `merged-depth` was successful at distinguishing them as separate objects.

Overall, this data suggests that our modified `merged-depth` model generally meets the objective of 20% error or less for most objects in the tested environments, given a ground-truth reference of the maximum depth in the environment. The model has an inference time of about 78 milliseconds on GeForce RTX 2080 Ti GPUs, faster than the 10 Hz requirement. The method is also implementable on any monocular camera; one example of a lightweight camera one could use is the Caddx Baby Ratel Starlight, which weighs 4.6 grams. Finally, we have shown that the model works for an operating range of 3 meters (the maximum range of the RealSense camera), though further testing is needed to ensure that the model is sufficient for an operating range of 5 meters.

4.2 Viability of `merged-depth` predictions

This `merged-depth` model has some limitations that prevent it from being a true absolute depth estimator. First and foremost, it requires a ground-truth input of the maximum depth in the environment. This was a reasonable assumption to make while constructing this model because the Crazyflie drone does have localization abilities and therefore we will assume it has information on how far it is from the bounds of the area of interest, but this may not be possible on all hardware platforms.

Secondly, `merged-depth` performs best in well-lit environments with high contrast. We expect that this is possible in most lab environments, but may not be possible in all operating environments. This challenge is not limited to our particular approach, but is a fundamental challenge in machine learning as it is difficult to generalize models to new environments, which is an active research area.

However, given these restrictions, the model is fast and reasonably accurate for the tested environments given only one ground-truth input. Improvements can certainly be made, and some of these are discussed in Section 5.2.

Chapter 5

Conclusions

5.1 Summary

Absolute depth estimation for small FPV drones is a relatively unexplored field of computer vision. In this project, we aimed to create a method for absolute depth estimation that was fast and reliable and uses lightweight hardware. To do so, we modified a version of `merged-depth` to reduce its inference time and tested the model using a RealSense camera to determine its accuracy over a range of indoor household environments. Our version of `merged-depth` relies primarily on the MiDaS, SGDepth, and Monodepth2 models. This model also requires knowing the maximum depth for every frame, but this is a reasonable assumption due to the localization abilities of small drones (e.g. the Crazyflie drone).

For the five environments we tested, the average absolute error between 5% and 12%, and the inference time per frame is estimated to be 78 milliseconds on GeForce RTX 2080 Ti GPUs. These results are within our objectives of 20% error and 100 milliseconds inference time. The model is sufficient for an operating range of 3 meters (tested on the RealSense camera) and is implementable on any high-resolution monocular camera, including small FPV cameras for lightweight drones. This version of `merged-depth` thus seems reliable in indoor environments similar to those we tested; that is, environments that are well-lit and have high contrast.

However, since our model was tested with the RealSense camera and not a true drone FPV camera, further work needs to be done to ensure that the model accurately predicts depth on an FPV camera within the desired operating range of 5 meters.

5.2 Future work

5.2.1 Improving depth estimates for foreground objects

As seen in Figures 3.11-3.15, the error for foreground objects tends to average around 20% error. This results in a relatively high standard deviation of the overall error in each frame, as seen in Table 3.1. We can improve these depth estimations by taking the average depth estimations over multiple frames from the drone's video feed as it flies. We expect that averaging the estimates will help filter out some of the high-frequency noise that leads to overestimating or underestimating the distance away from foreground objects.

5.2.2 Improving estimates of maximum depth

Our model requires knowledge of the maximum depth (i.e. distance to the wall) in any frame. This is currently a reasonable assumption due to the localization abilities of the drone. None of the three models in `merged-depth` (MiDaS, SGDepth, and Monodepth2) had accurate predictions of the maximum depth in the environments we tested. However, AdaBins seems to have more accurate predictions of maximum depth, but was one of the models we removed to reduce inference time. Training a new model to just predict maximum depth or improving one of our current models to better predict maximum depth would improve our model by removing the requirement for a maximum-depth input.

5.2.3 Improving drone camera hardware

As seen in our discussion about camera selection (see Section 2.3), we believe that significant accuracy improvements could be made if we were to find a better FPV drone camera than the Wolfwhoop. One option is the Caddx Baby Ratel Starlight, which has improved resolution at 1200 TVL, but is still very lightweight at 4.6 grams net weight. Higher-end drone cameras will likely provide higher-quality images that see improved accuracy in `merged-depth`, but may not perform as well as the Intel RealSense.

5.2.4 Improving inference time

It is not always viable to run these models on a GPU, but we need fast inference times in order for depth estimation to be useful for a flying drone. Fast depth estimation methods do exist, such as one by Poggi et al. [13], which is able to infer a depth map at 8 Hz or faster. Faster inference times may also allow better control and thus flight at higher speeds.

Other promising monocular depth estimation models do exist, such as FastDepth [19], which could be viable alternatives to `merged-depth`.

5.2.5 Onboard computing with Bitcraze AI Deck

In the future, it may be desirable to have onboard computing on the Crazyflie drone. Bitcraze's AI Deck is a good option for introducing AI-based autonomous navigation methods without the need to transmit and receive data from an external device.

Bibliography

[1] Shariq Farooq Bhat, Ibraheem Alhashim, and Peter Wonka. Adabins: Depth estimation using adaptive bins. *CoRR*, abs/2011.14141, 2020.

[2] Bitcraze. https://www.bitcraze.io/wp-content/uploads/2019/10/seek4.png.

[3] Vincent Casser, Soeren Pirk, Reza Mahjourian, and Anelia Angelova. Depth prediction without the sensors: Leveraging structure for unsupervised learning from monocular videos. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):8001–8008, Jul. 2019.

[4] Baofu Fang, Gaofei Mei, Xiaohui Yuan, Le Wang, Zaijun Wang, and Junyang Wang. Visual slam for robot navigation in healthcare facility. *Pattern Recognition*, 113:107822, 2021.

[5] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, 2012.

[6] Clément Godard, Oisin Mac Aodha, and Gabriel J. Brostow. Digging into self-supervised monocular depth estimation. *CoRR*, abs/1806.01260, 2018.

[7] Faiza Gul, Wan Rahiman, and Syed Sahal Nazli Alhady. A comprehensive study for robot navigation techniques. *Cogent Engineering*, 6(1):1632046, 2019.

[8] Kenneth Jiang. Calibrate fisheye lens using opencv — part 1. https://medium.com/@kennethjiang/calibrate-fisheye-lens-using-opencv-333b05afa0b0, 2017.

[9] Pileun Kim, Jingdao Chen, Jitae Kim, and Yong K. Cho. Slam-driven intelligent autonomous mobile robot navigation for construction applications. In Ian F. C. Smith and Bernd Domer, editors, *Advanced Computing Strategies for Engineering*, pages 254–269, Cham, 2018. Springer International Publishing.

[10] Marvin Klingner, Jan-Aike Termöhlen, Jonas Mikolajczyk, and Tim Fingscheidt. Self-supervised monocular depth estimation: Solving the dynamic object problem by semantic guidance. *CoRR*, abs/2007.06936, 2020.

[11] Hamid Laga, Laurent Valentin Jospin, Farid Boussaid, and Mohammed Bennamoun. A survey on deep learning techniques for stereo-based depth estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(4):1738–1764, 2022.

[12] Katrin Lasinger, René Ranftl, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *CoRR*, abs/1907.01341, 2019.

[13] Matteo Poggi, Filippo Aleotti, Fabio Tosi, and Stefano Mattoccia. Towards real-time unsupervised monocular depth estimation on cpu. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5848–5854, 2018.

[14] Ashutosh Saxena, Sung Chung, and Andrew Ng. Learning depth from single monocular images. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems*, volume 18. MIT Press, 2005.

[15] Ashutosh Saxena, Min Sun, and Andrew Y. Ng. Make3d: Learning 3d scene structure from a single still image. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):824–840, 2009.

[16] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. Indoor segmentation and support inference from rgbd images. In Andrew Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid, editors, *Computer Vision – ECCV 2012*, pages 746–760, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[17] Pranav Srinivas Kumar, Max Guo, and Michael Murray. Merged depth. `https://github.com/p-ranav/merged_depth`, 2021.

[18] Sarah Tang and Vijay Kumar. Autonomous flight. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(1):29–52, 2018.

[19] Wofk, Diana and Ma, Fangchang and Yang, Tien-Ju and Karaman, Sertac and Sze, Vivienne. FastDepth: Fast Monocular Depth Estimation on Embedded Sys-

tems. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

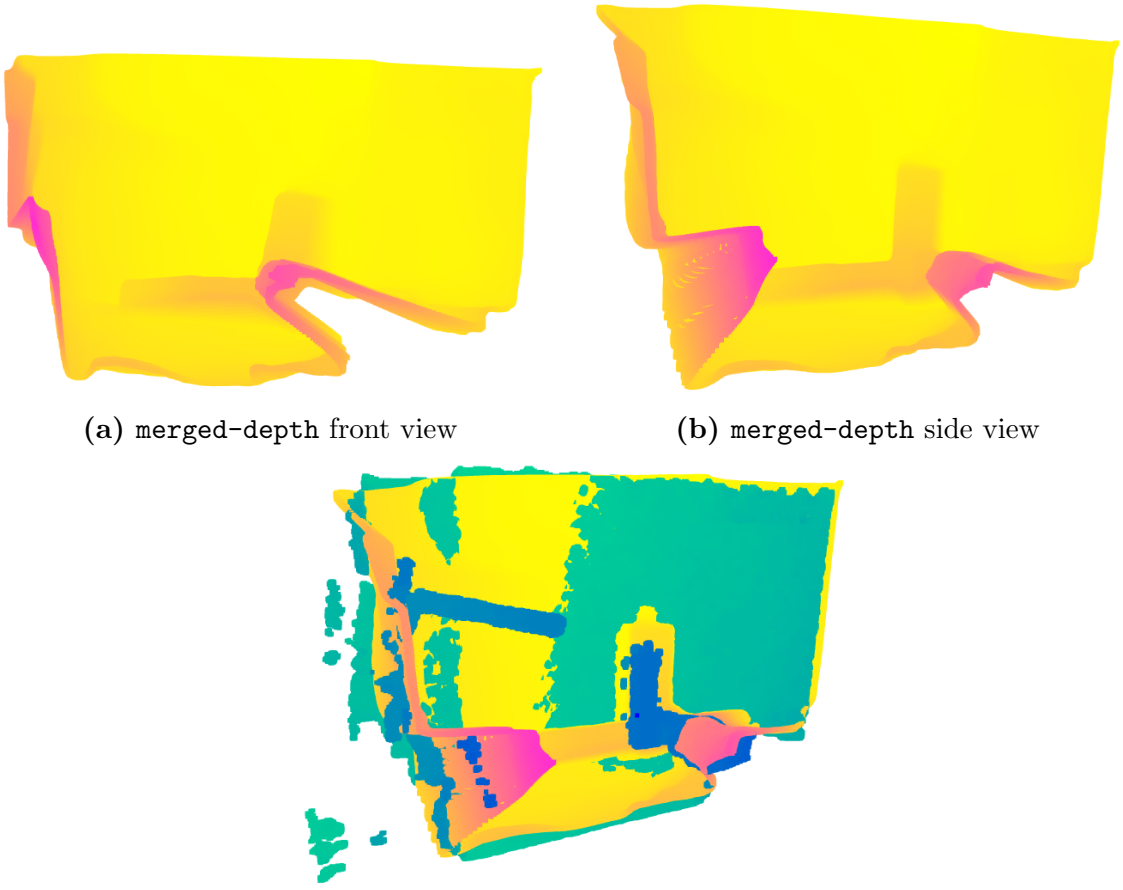
[20] Wei Yin, Xinlong Wang, Chunhua Shen, Yifan Liu, Zhi Tian, Songcen Xu, Changming Sun, and Dou Renyin. Diversedepth: Affine-invariant depth prediction using diverse data. *CoRR*, abs/2002.00569, 2020.

[21] ChaoQiang Zhao, QiYu Sun, ChongZhen Zhang, Yang Tang, and Feng Qian. Monocular depth estimation based on deep learning: An overview. *Science China Technological Sciences*, 63(9):1612–1627, Sep 2020.

[22] Tinghui Zhou, Matthew Brown, Noah Snavely, and David G. Lowe. Unsupervised learning of depth and ego-motion from video. *CoRR*, abs/1704.07813, 2017.

Appendix A

Additional Point Clouds

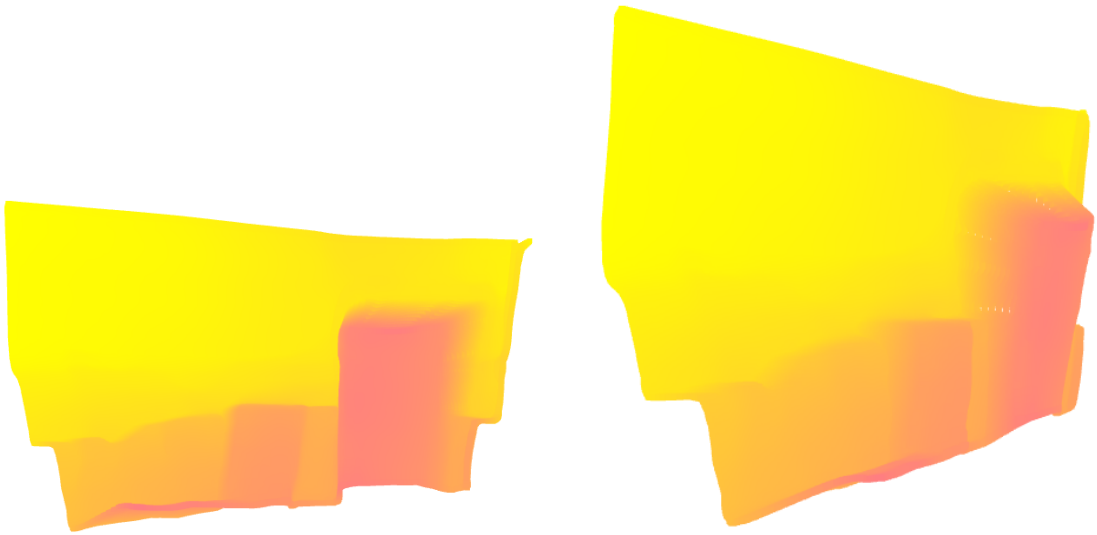


(a) `merged-depth` front view

(b) `merged-depth` side view

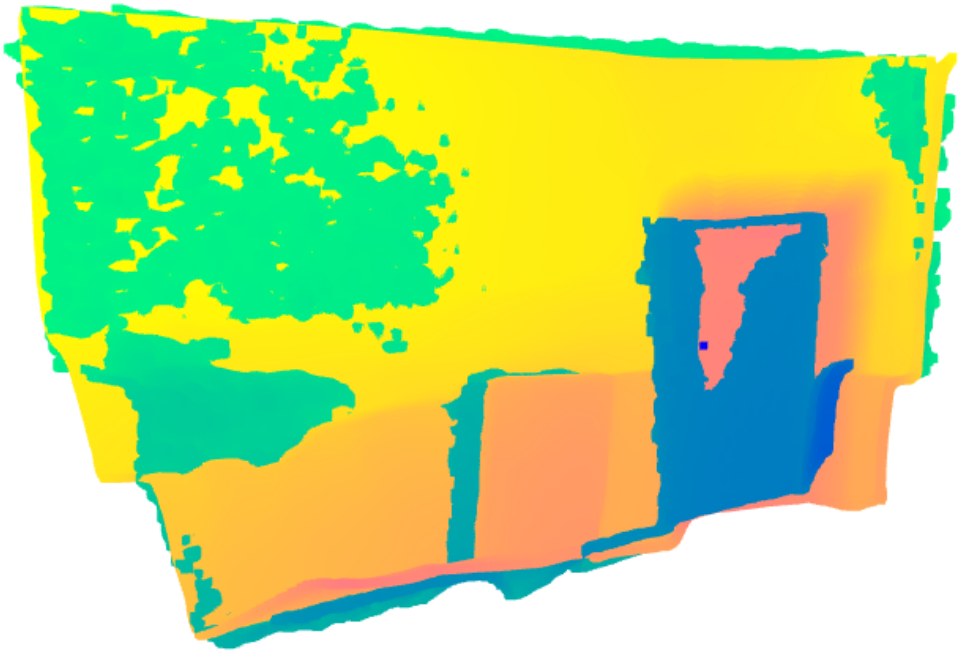
(c) Overlaid `merged-depth` and RealSense view

Figure A.1: Point clouds for Environment 1. In the overlaid maps, the RealSense depth map is mapped in green-blue, and the `merged-depth` map is in pink-yellow.



(a) `merged-depth` front view

(b) `merged-depth` side view

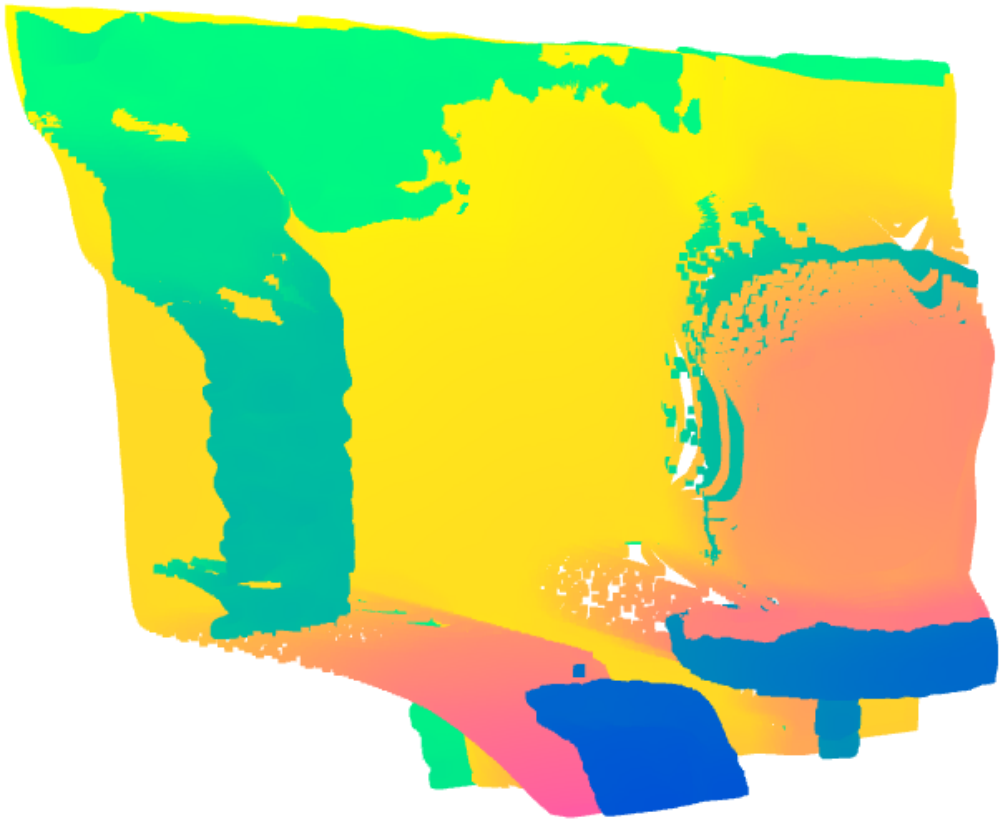


(c) Overlaid `merged-depth` and RealSense view

Figure A.2: Point clouds for Environment 2

(a) `merged-depth` front view

(b) `merged-depth` side view



(c) Overlaid `merged-depth` and RealSense view

Figure A.3: Point clouds for Environment 5

Appendix B

Code

B.1 capture.py

The following script captures RGB and depth data using the RealSense camera and saves it to a specified location.

```
import pyrealsense2 as rs
import numpy as np
import cv2

test = 6 # test number

# start realsense
pipe = rs.pipeline()
profile = pipe.start()
color_frames = [] # capture RGB data
depth_frames = [] # capture depth data

# capture frames and keep color/depth frames
print('capturing frames...')
try:
  for i in range(0, 200): # total number of frames taken
    frames = pipe.wait_for_frames()

    cf = frames.get_color_frame()
```

```
    cf.keep()
    color_frames.append(cf)

    df = frames.get_depth_frame()
    df.keep()
    depth_frames.append(df)

finally:
    print('finished capturing frames.')
    pipe.stop()

# number of frames
print('number of frames:', len(color_frames))
if (len(depth_frames) != len(color_frames)):
    print('unequal number of depth and color frames!')

# save color frames as jpgs
for i in range(len(color_frames)):
    if i % 10 == 5:
        image = np.asanyarray(color_frames[i].get_data())
        image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
        cv2.imwrite('./test{}/color_{}.jpg'.format(test, i), image)
   # color data filename
        # cv2.imshow('', image)
        # cv2.waitKey(0)
        # cv2.destroyAllWindows()
print('saved color data.')

# save depth data as npy
depth_data = []
for i in range(len(depth_frames)):
    if i % 10 == 5: # take every 10 frames
        df = depth_frames[i]
        depth = df.get_data()
        depth = np.asanyarray(depth)
        depth_data.append(depth)
```

```
55
56 depth_file = './test{}/test{}.npy'.format(test, test) # depth data
       filename
57 with open(depth_file, 'wb') as f:
58     np.save(f, np.array(depth_data))
59 print('saved depth data to {}.'.format(depth_file))
```

B.2 plot_depth.py

The following script plots the RealSense depth map and `merged-depth` map and calculates the linear correlation function between them.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  dataset = 3 # test number
5
6  for frame in [8]: # frame(s) to analyze
7      real_depth = np.load('test{}/test{}.npy'.format(dataset, dataset
       ))
8      real = real_depth[frame][:,60:1280] / 10 # remove pixel hotspots
       and convert from mm to cm
9
10     merged = np.load('test{}/color_{}5_depth.npy'.format(dataset,
       frame))
11     merged = merged[:,60:1280]
12
13     # display and save RealSense depth map
14     plt.figure(1)
15     plt.imshow(real)
16     plt.axis('off')
17     plt.savefig('env{}-{}_real.eps'.format(dataset, frame),
       bbox_inches='tight', pad_inches = 0)
18     plt.title('realsense depth')
19
20     # display and save merged-depth map
```

```
    plt.figure(2)
    plt.imshow(merged)
    plt.axis('off')
    plt.savefig('env{}-{}_merged.eps'.format(dataset, frame),
bbox_inches='tight', pad_inches = 0)
    plt.title('merged_depth')

    ## plot real depth vs. merged-depth
    frame_shape = real.shape
    real = real.flatten()
    merged = merged.flatten()

    # remove values where real depth is 0 or nan
    nonzero = np.where(np.abs(real - 0) >= 1)
    real = real[nonzero]
    merged = merged[nonzero]
    finite = np.where(np.isfinite(real))
    real = real[finite]
    merged = merged[finite]

    # calculate linear model from merged-depth to real depth
    coeffs = np.polyfit(merged, real, 1)
    linreg_m = np.linspace(0.5, 4.5)
    linreg_r = merged * coeffs[0] + coeffs[1] # to calcuate
correlation coefficient
    correlation = np.corrcoef(real, linreg_r)
    linreg_r = linreg_m * coeffs[0] + coeffs[1] # to plot
    print('correlation matrix:')
    print(correlation)

    # take average merged-depth value for every real value
    real2merged = []
    for depth_value in range (5, 200): # range of real depths
        loc = np.where(real == depth_value)
        merged_loc = merged[loc]
        avg = np.mean(merged_loc)
```

```
55          real2merged.append([depth_value, avg])
56      real2merged = np.array(real2merged)
57
58      valid = []
59      for i in range(len(real2merged)):
60          if not np.isnan(real2merged[i,0]) and not np.isnan(
    real2merged[i,1]):
61              valid.append(i)
62      real2merged = real2merged[valid]
63
64      avg_coeffs = np.polyfit(real2merged[:,1], real2merged[:,0], 1)
65      avg_linreg_m = np.linspace(0.5, 4)
66      avg_linreg_r = avg_linreg_m * avg_coeffs[0] + avg_coeffs[1]
67
68      # scatter plot of real depth vs. merged-depth w/ correlation
    line
69      plt.figure(3)
70      plt.scatter(real[::5], merged[::5])
71      plt.plot(linreg_r, linreg_m, 'r')
72      plt.xlabel('RealSense depth (cm)')
73      plt.ylabel('merged_depth depth estimation')
74      plt.savefig('env{}-{}_scatter.eps'.format(dataset, frame),
    bbox_inches='tight', pad_inches = 0)
75
76      # scatter plot of real depth vs. averaged merged-depth w/
    correlation line
77      plt.figure(4)
78      plt.scatter(real2merged[:, 0], real2merged[:, 1])
79      plt.plot(linreg_r, linreg_m, label='frame {}'.format(frame))
80      plt.ylabel('Averaged merged_depth depth estimation')
81      plt.xlabel('Realsense Depth (cm)')
82      plt.savefig('env{}-{}_scatter-avg.eps'.format(dataset, frame),
    bbox_inches='tight', pad_inches = 0)
83      print('linear coefficents:')
84      print(coeffs)
85  plt.legend()
```

```
plt.show()
```

B.3 find_error.py

The following script compares the corrected `merged-depth` map to the RealSense depth map and plots and outputs the error.

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.ticker import PercentFormatter
import open3d as o3d


dataset = 6 # test number
frame = 9 # frame for analysis
a = 26.16 # scale (from linear correlation)

# load data
real_data = np.load('test{}/test{}.npy'.format(dataset, dataset))
real = real_data[frame][:,60:1280] / 10  # remove pixel hotspots
merged = np.load('test{}/color_{}5_depth.npy'.format(dataset, frame)
    )[:,60:1280]

# real depth map
plt.figure(1)
plt.axis('off')
plt.imshow(real)

# calculate shift using maximum depth
points = [[50, 200], [100, 200], [100, 250], [50, 250]] # pixels
    where maximum depth is located
rows, cols = zip(*points)
wall_m = np.mean(merged[rows, cols])
wall_r = np.mean(real[rows, cols])
b = wall_r - a * wall_m
```

```
28 plt.scatter(cols, rows, s=80, facecolors='none', edgecolors='r')
29
30 # merged-depth map
31 plt.figure(2)
32 plt.imshow(merged)
33 plt.axis('off')
34 plt.scatter(cols, rows, s=80, facecolors='none', edgecolors='r')
35
36 # compare real values with merged depth predicted values, calculate
       error rates
37 errors = []
38 for i in range(50, merged.shape[0], 50):
39     for j in range (50, merged.shape[1], 50):
40         m = merged[i][j]
41         r = real[i][j]
42         r_ = a * m + b
43         err = (r - r_) / r * 100
44         # print("real: {}, estimated: {}, error: {:2.0f}".format(r,
       r_, err))
45         plt.scatter(j, i, s=30, facecolors='none', edgecolors='k')
46         if np.isfinite(err):
47             errors.append(err)
48         plt.annotate("{:2.0f}".format(err), (j, i))
49 rows, cols = zip(*points)
50 plt.savefig('env{}-{}_errors.eps'.format(dataset, frame),
       bbox_inches='tight', pad_inches = 0)
51
52 # rescaled merged-depth map
53 corrected_r = a * merged + b
54 plt.figure(3)
55 plt.imshow(corrected_r)
56 plt.axis('off')
57 plt.savefig('env{}-{}_corrected.eps'.format(dataset, frame),
       bbox_inches='tight', pad_inches = 0)
58
59 # error histogram
```

```
errors = np.array(errors)
errors = errors[np.isfinite(errors)]
plt.figure(4)
plt.hist(errors, weights=np.ones(len(errors)) / len(errors))
plt.xlabel('Percentage error')
plt.gca().yaxis.set_major_formatter(PercentFormatter(1))
plt.savefig('env{}-{}_hist.eps'.format(dataset, frame), bbox_inches=
    'tight', pad_inches = 0)

# print outputs
print("a: {}, b: {}".format(a, b)) # scale and shift
print("max error: {}".format(max(errors)))
print("min error: {}".format(min(errors)))
print("average absolute error: {} +- {}".format(np.mean(np.abs(
    errors)), np.std(np.abs(errors))))
# plt.show()

# pointcloud visualization
FX_DEPTH = 5.8262448167737955e+02
FY_DEPTH = 5.8269103270988637e+02
CX_DEPTH = 3.1304475870804731e+02
CY_DEPTH = 2.3844389626620386e+02

pcd_m = [] # merged-depth points
pcd_r = [] # real depth points
height, width = merged.shape
for i in range(height):
    for j in range(width):
        z_m = corrected_r[i][j]
        x_m = (j - CX_DEPTH) * z_m / FX_DEPTH
        y_m = (i - CY_DEPTH) * z_m / FY_DEPTH
        pcd_m.append([x_m, y_m, z_m])
height, width = real.shape
for i in range(height):
    for j in range(width):
        z_r = real[i][j]
```

```
        x_r = (j - CX_DEPTH) * z_r / FX_DEPTH
        y_r = (i - CY_DEPTH) * z_r / FY_DEPTH
        pcd_r.append([x_r, y_r, z_r])

# assign different colormaps
depth_m = np.array(pcd_m)[:,2]
depth_r = np.array(pcd_r)[:,2]
cmap_spring = mpl.cm.get_cmap('spring')(depth_m / np.max(depth_m))
    [:,:-1]
cmap_winter = mpl.cm.get_cmap('winter')(depth_r / np.max(depth_r))
    [:,:-1]

# plot merged-depth pointcloud
pcd_o3d_m = o3d.geometry.PointCloud()  # create point cloud object
pcd_o3d_m.points = o3d.utility.Vector3dVector(pcd_m)  # set pcd_np
    as the point cloud points
pcd_o3d_m.colors = o3d.utility.Vector3dVector(cmap_spring)

# plot real depth pointcloud
pcd_o3d_r = o3d.geometry.PointCloud()  # create point cloud object
pcd_o3d_r.points = o3d.utility.Vector3dVector(pcd_r)  # set pcd_np
    as the point cloud points
pcd_o3d_r.colors = o3d.utility.Vector3dVector(cmap_winter)

# show visualization
o3d.visualization.draw_geometries([pcd_o3d_m, pcd_o3d_r])
```

This paper represents my own work in accordance with University regulations.