Inferring Depth Maps from 2-Dimensional Laser Ranging Data in a Simulated Environment

Viljar Rúnarsson

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 24.4.2018

Supervisor

Prof. Ville Kyrki

Advisors

Dr. Francesco Verdoja

M.Sc. Jens Lundell

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Viljar Rúnarsson

Title Inferring Depth Maps from 2-Dimensional Laser Ranging Data in a Simulated Environment

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Prof. Ville Kyrki

Advisors Dr. Francesco Verdoja, M.Sc. Jens Lundell

Date 24.4.2018 **Number of pages** 39+10 **Language** English

Abstract

Depth estimation plays a key role in mobile robotics for applications including scene understanding, navigation and mapping. Recently, deep learning methods have proven effective in estimating depth maps from a combination of different sources such as 3D LiDAR or RGB images. However, they face two challenges; the lack of dense ground truth data and the depth input sparsity which ranges from 4-10% pixel density on an input image. This thesis explores the feasibility of inferring a full depth map from extremely sparse 2D LiDAR measurements via neural network. To address the lack of ground truth data, a simulation tool is created for data gathering. The results show that from our sparse input of 0.024% pixel density on input images, the tested network infers shapes but struggles with blurry boundaries on objects.

Keywords Depth Completion, Deep Learning, Robotic Simulation, Convolutional Neural Networks

Preface

I want to thank Professor Ville Kyrki and my instructors Dr Francesco Verdoja and Jens Lundell for their invaluable assistance and expertise.

Otaniemi, 24.4.2018

Viljar Runarsson

Contents

Symbols and abbreviations

Symbols

F_N	Coordinate frame with annotation N
T_{NK}	Transformation from coordinate frames N to K

Abbreviations

absREL	Absolute Relative Error
API	Application Programming Interface
CNN	Convolutional Neural Network
FOV	Field of View
fps	Frames per Second
iMAE	Mean Absolute Error of the inverse depth [1/km]
I/O	Input / Output
iRMSE	Root Mean Squared Error of the inverse depth [1/km]
LiDAR	Light Detection and Ranging
MAE	Mean Absolute Error [mm]
RGB	Red Green Blue
RMSE	Root Mean Squared Error [mm]
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
SRS	Software Requirements Specification
TF	Transformation
UAV	Unmanned Aerial Vehicle

1 Introduction

The term hallucination is defined as "an experience involving the apparent perception of something not present." [18]. This might not seem relevant to robots, seeing as hallucination has implicit human connotations. In the context of this thesis however, the definition seems appropriate. Imagine the world as humans perceive it; we are constrained by our senses such as vision, hearing and touch. These senses are the means by which we interact with our environment. Now imagine a world from the perspective of a robot; it too is constrained by its senses. In this context, the robots senses are comprised of the sensors given to it; cameras with different modalities or range finders. For example, a robot equipped only with a gray scale camera cannot see color; it does not exist from its perspective. But what if it could infer color? The hallucinating analogy stems from this case. This work aims to investigate if robots would be able to perceive something that, to them, is not there. Specifically, we want to explore whether inferring full resolution depth maps from 2D LiDAR scans is possible.

Most, if not all mobile robots come equipped with a range finding devices, usually LiDARs which can range from 2D to 3D versions. LiDARs have high accuracy and a long sensing range, and as such have been incorporated in a wide range of robots, autonomous vehicles and even UAVs [2,14]. The 3D version tends to suffer from low framerate [9] while the 2D versions boast a relatively fast scan rate, capable of much higher frames per second (fps) than regular cameras along with a 360° Field of View (FOV). Given that 2D LiDAR scanners are generally cheaper and have a lower data dimensionality, they have become the dominating device for localization, mapping and navigation on various indoor mobile ground platforms [3]. In tandem with LiDAR scanners becoming more common and with the advent of deep learning, research into ascertaining more information from them has also grown, specifically completing sparse depth images. The motivation for this research is autonomous scene recognition to use in obstacle avoidance, robot grasping and image classification. The conventional approach for robots to faithfully recognize their surroundings is to use either a camera in conjunction with a 2D/3D LiDAR, a high resolution depth camera or a stereo camera, although the latter two methods are expensive and less robust, respectively [9]. Information is then gathered through sensor fusion, using the different modalities to obtain a variety of information about a given scene. Due to the higher data dimensionality brought on by utilizing different sensor modalities and the information redundancy that goes along with it, the computational costs of processing this information rises.

The approach explored in this thesis aims to enable scene and shape detection to be used in indoor obstacle avoidance. It would greatly reduce the amount of data and the cost of sensing equipment currently required to safely navigate a mobile platform. In addition, the 360° FOV of 2D LiDARs would enable corresponding scene reconstruction which would otherwise be very cost prohibitive. Due to the low data dimensionality, this approach could also prove useful in tele-operation, since a single vector of distances could be used to reconstruct a scene as opposed to a

stream of images.

Recent works use deep learning to take on the sparse-to-dense task [1,2,3,14,15]. They use sparse 3D LiDAR depth data points, sometimes in conjunction with RGB images to complete a full resolution depth map of a scene. The main challenge faced here is the lack of dense ground truth data to train the deep learning networks on. We intend to both address the lack of ground truth data and attempt to complete depth images with a much lower input data density than has previously been tried. We will develop an indoor simulation software capable of generating training data and we will apply this data to an autoencoder and explore the performance.

1.1 Problem Statement

Many mobile robots use 2D LiDAR scans for navigation and obstacle detection. These scans are accurate, have a small data dimensionality and a large FOV [1,3]. Their main drawback however is their inability to provide distances to obstacles that have a greater occupancy than the 2D LiDAR can measure, such as glass panels, tables, chairs or plants [3]. This thesis intends to explore the feasibility of inferring a depth image from a corresponding 2D LiDAR scan for the purposes of indoor navigation and obstacle detection. We will utilize and adapt deep learning architectures which have been successful in sparse-to-dense depth prediction [1,3,4].

Deep learning methods require large amounts of data [7] and the selection of datasets that include 2D LiDAR scans along with dense depth images are few and far between. Gathering a large enough dataset with sufficient variance from the real world would be too time consuming. Thus, one of the key components of this thesis is developing a simulation environment capable of gathering the amount of data needed. A simulated robot (Care-O-bot 4) will be the platform used to gather data. The robot has a 2D LiDAR just above ground level which yields a 2D point map around its perimeter. The software tools that will be used include ROS, Gazebo and Python. The core problem for an automated simulation such as this is accurately representing an indoor environment while maintaining the necessary variance to generate good training data. The main software requirements then include random pathing of the robot and pseudo-random obstacle placement.

Depth prediction is a well studied problem in computer vision and has grown with the advent of deep learning [1,5,7]. The depth prediction problem is usually tackled by combining different sensor modalities such as a 3D LiDAR complimented by an RGB image, monocular camera image or stereo camera image [1, 2, 8] although sparse-to-dense methods do exist with only a sparse depth image as an input [5,12,13]. Since these methods are optimized and designed to generate images from images, we opt to project the LiDAR point rays onto the depth camera frame. This projection effectively creates a sparse depth image, making this an extreme case of the sparse-to-dense problem, allowing us to utilize existing neural network architectures that have proved effective in solving similar problems. It should be noted that this

problem is non-trivial since the pixel population of the depth images used in the aforementioned methods ranges from 4 to 10 % of the image [2,15] while our average pixel population rests at around 0.024 %.

To more accurately frame the problem, let us consider a 2D LiDAR and a depth camera positioned on a robot. The LiDAR takes readings in a horizontal disk around the perimeter of the robot. Let the coordinate frame of the robots base link be $\underrightarrow{F}_B$, let the coordinate frame of the LiDAR be $\underrightarrow{F}_L$ and let the coordinate frame of the depth camera be $\underrightarrow{F}_C$. The transformation from the coordinate frame of the camera to the coordinate frame of the LiDAR will then be a 3x4 transformation matrix T_{CL} consisting of a 3x3 rotation matrix R_{CL} and a 3x1 translation vector t_{CL}. Let us then define the output of the LiDAR that has N points as a vector $\boldsymbol{p}_L$ in the coordinate frame $\underrightarrow{F}_L$ such that

$$\boldsymbol{p}_L = [x_i, y_i, z_i]_{i=0}^{N} \tag{1}$$

Where the points x_i, y_i and z_i represent the euclidean coordinates of each point i in the lasers scan range. Our goal is to explore if a mapping $\mathcal{G}$ can be found that projects the LiDAR points $\boldsymbol{p}_L$ to fill an image frame that corresponds to our ground truth images. This will be discussed in more detail in 3.2.

To approximate this mapping $\mathcal{G}$, a deep regression neural network is proposed. Given the exploratory nature of this work, we are particularly interested in the generalization capabilities of the trained network. To this end, we will qualitatively compare performance in simulated situations that differ from the training environment. We will discuss the strengths and weaknesses of both the simulated environment and the neural network and analyze the possible reasons for the variations in performance. From this analysis, further augmentation of the network or simulation will be proposed.

In summary, this thesis:

- Discusses the problem of depth prediction and its various methods in detail.
- Contributes simulation and data processing tools to aid in data gathering for indoor environments, grating access to a diverse set of situations for both testing and training.
- Explores the feasibility of inferring depth maps from sparser data than state-of-the-art methods.
- Proposes a method for inferring depth maps from 2D LiDAR scans.
- Provides experiments to evaluate the performance of the method and analyzes its strong and weak points.

1.2 Structure of the Thesis

The thesis is structured as follows. Section 2 discusses related background literature, covering challenges and approaches in the sparse-to-dense completion field. Section 3 will discuss in detail the tool usage, software construction and various design concerns regarding the data acquisition in addition to introducing the tested neural network structure and metrics. Section 4 presents the experiment settings and results, consisting of tables containing error metrics, images to illustrate performance along with an analysis of the results. Finally, Section 5 will present conclusions based on the results analysis, literature review and software design. Suggestions will be given on which aspects improve performance.

2 Background: Depth Completion

Depth prediction without sparse depth measurements is well studied. For example, RGB-based depth prediction has been studied for over a decade while still being far from practical [1]. For the purposes of this review, we will focus on methods that include sparse depth measurements. A comparison will be made of performance of state-of-the-art methods and a discussion of the impact, strengths and limitations of augmenting sparse depth data with different sensor modalities. The discussion will include two types of depth completion. The former relies on combining depth data with either an RGB image or a monocular/stereo camera image. The latter type utilizes only sparse depth scans. Although most works include methods for both types, we will briefly discuss the impact of including RGB images. At this time, there are no works that attempt a full resolution depth prediction using data that has similar sparsity to ours and without a guiding RGB image. It is however important to review available methods that aspire to achieve the same thing as us; an accurate depth estimation from sparse data. In addition we will also look at the most common datasets used in the field, namely the KITTI, Make3D and NYU datasets. For reference, the top performing depth-completion methods with RGB-guidance and depth-only are listed in Table 1. Non-published methods are omitted. We will focus on comparing and evaluating approaches that are currently top performing on the KITTI and NYU datasets since they provide a good benchmark.

Table 1: Top performing published methods on the KITTI dataset [21]. Each algorithm performance is measured in terms of iRMSE (Root Mean Square Error of the inverse depth), iMAE (Mean Absolute Error of the inverse depth), RMSE (Root Mean Square Error) and MSE (Mean Absolute Error).

Ranking	**Method**	**iRMSE**	**iMAE**	**RMSE**	**MAE**	**RGB**
1.	Sparse-to-Dense (gd) [2]	2.8	1.21	814.73	249.95	yes
2.	HMS-Net v2 [14]	2.73	1.13	841.78	253.47	yes
3.	Spade-RGBsD [15]	2.17	0.95	917.64	234.81	yes
4.	HMS-Net [14]	2.93	1.14	937.48	258.48	no
5.	Sparse-to-Dense (d) [2]	3.21	1.35	954.36	288.64	no
6.	CSPN [10]	2.93	1.15	1019.64	279.46	yes

Note that the top methods propose approaches such that RGB-assistance is optional. Obviously, the RGB-assisted variants tend to outperform the depth-only ones, since the RGB images contain detailed shape information.

2.1 Challenges in Depth Completion

The task of predicting depth can be classified in to three categories; depth prediction, depth upsampling and depth completion. Depth prediction is the task of inferring the depth of a scene, most commonly an image, by utilizing sensor modalities that do not yield depth measurements. Depth upsampling is the term for completing non-sparse depth images. Finally, depth completion is the task of inferring the depth of a whole scene by utilizing sparse depth measurements, often referred to as inpainting. Both depth upsampling and depth completion may utilize RGB images to improve accuracy.
The top performing methods mentioned in Table 1 all use some variant of a CNN architecture to tackle the depth completion problem. These CNNs are classified as deep learning networks and, as such, require a large amount of data to be effective [7]. The biggest problem depth completion faces is the lack of dense ground truth data [1,2]. This is due to the nature of the most common sensors involved in registering depth information, namely LiDARs. These are active sensors which emit light and register the corresponding reflection, measuring travel time to obtain distance. Currently, state-of-the-art sensors such as the Velodyne HDL-64e only generates around 5% pixel density when its output is projected onto an image plane as described in 3.2 [12]. Specifically, it only has a vertical and angular resolutions of 0.4°and 0.8°, respectively. Furthermore, such sensors are expensive with the aforementioned Velodyne sensor costing $75k [2]. In addition to the inherent sparsity of depth data, these sensors typically have a long scanning time; a typical LiDAR will produce detailed images at 10 fps as opposed to 60 fps on most cameras [9]. As a consequence, the availability of dense ground truth depth data is scarce since it is expensive and time-consuming to gather.

The sparse depth points also tend to be irregularly spaced [2,5]. Since common CNNs do not discriminate between pixels that contain actual information or not, they tend to be sensitive to sparse inputs, specifically they do poorly when finding a depth map that is consistent with the sparse input [5]. Recent works have provided solutions to this problem by developing sparsity invariant convolution methods [13, 14, 15]. Conversely, [2, 10] propose using classical CNNs with guiding frameworks to solve the sparsity sensitivity while [12] proposes implementing a CNN with normalized convolutional layers in tandem with traditional convolutions to deal with sparse depth data and dense RGB images, respectively.

2.2 Methods in Depth Completion

With the advent of deep learning facilitated by increased computing power in latter years, depth prediction via RGB has improved immensely since one of the first proposed methods in 2006 [17]. However, we will constrain our review to methods that utilize depth data since they outperform any RGB-only method and are more relevant to this work. We will systematically review the top methods utilizing RGB and sparse depth images in Table 1.

Currently, [2] is the top performing method and proposes a deep regression model. Its architecture is that of a bottleneck autoencoder where the encoder consists of four 34-layer ResNet [11] blocks and the decoder is made up by six 3x3 transposed convolution blocks. Unlike in our structure, the output of each ResNet block is skip-connected to its corresponding decoder block [2]. If RGB images are available, Ma et al. also propose a self-supervised training framework which uses an additional spatially nearby RGB image to provide supervision signals. Because of the network depth and skip connections, it acquires greater detail than other top performing methods [2].

Instead of a guiding framework, [15] proposes a convolution that explicitly evaluates observed pixels, i.e. pixels containing information, and shows that it outperforms regular convolutional layers across sparsity levels. This convolution can be formulated as in [14] such that output features at each location $z(u, v)$ are generated by

$$z(u,v) = \frac{\Sigma_{i,j=-k}^{k} m_x(u+i, v+j) w(i,j) x(u+i, v+j)}{\Sigma_{i,j=-k}^{k} m_x(u+i, v+j) + \epsilon} + b \tag{2}$$

where w is the convolution kernel, b is a bias vector, trained via back-propagation. Then x is a sparse depth image and m is a trained sparsity mask that records the existence of input features at each location (u, v). In addition to the sparse convolution operator, [14] also proposes additional sparsity invariant operations such as summation, upsampling and concatenation. In their work, [14] attains the second best ranking on the KITTI dataset, but it is to be noted that [2] uses additional temporal information in their supervision scheme and [14] outperforms [2] when utilizing depth-only approaches. For our work, both approaches seem to have utility. Since using extra temporal information decreases the sparsity of data used to generate each full depth image and the sparsity invariant operations address it directly.
The approach presented in [12] proposes implementing a CNN with normalized convolutional layers in tandem with traditional convolutions to deal with sparse depth data and dense RGB images, respectively. They also claim that traditional CNNs cannot generalize when the sparsity level is changed, which is challenged by [15].

The fifth ranked method on the KITTI benchmark however claims that the sparsity invariant method, more specifically the mask input, might be redundant and propose a solution which uses generic CNNs, claiming that they can simply learn where valid input data exists [15]. They state that the masking approach is that since valid outputs are scaled, the mask becomes entirely valid at deeper layers in the network, making it redundant. The method [15] proposes is a slightly adapted version of NASNet [16] and incorporates a sparse training strategy that varies sparsity of the inputs during training. At the time of publishing, they were ranked first on the KITTI benchmark. This approach is interesting since it shows that conventional CNNs can be effectively utilized while even preserving sharp edges in predicted images while being simpler to design.

The eighth ranked method proposes a slightly different approach from the rest, where the bulk of their information comes from an RGB image. They introduce a

convolutional spatial propagation network which embeds sparse depth data into the propagation process, preserving input depth values in the final predicted depth map [10]. This method attained a significant 30% improvement, at the time of publishing, over the state-of-the-art methods while also being faster. Currently, this method is outperformed in both accuracy and speed.

Completing depth map via 2D LiDAR has only been attempted with the aid of RGB images [8]. Thus, relevancy of the background literature stems from solutions presented specifically to tackle sparsity of depth inputs. In [8], a novel approach for depth estimation via 2D laser scanner is introduced. Although this work is old compared to the state-of-the-art methods discussed, it is still worth a note since it is the only method that proposes a 2D LiDAR, albeit in conjunction with RGB images, to estimate a depth map. Based on the approach in [17] the method proposed in [8] stretches the sparse depth values vertically over the whole image and concatenates it with an RGB image before running the inputs through a ResNet-based autoencoder. They showed a substantial improvement over using previous RGB-only methods.

2.3 Impact of RGB images

As can be seen in Table 1, we see that using an RGB image in tandem with the sparse depth data consistently outperforms versions that use depth only. This makes sense intuitively since a full resolution RGB image contains a lot of information regarding shape. As such, all of the methods discussed use RGB for guidance or have a framework that offers the option. This, however may not always be the case. Results from [1] indicate a decreasing performance gap between RGB guided- and depth only versions of their network as the number of depth samples increases, showing that RGB+depth and depth-only variants have the same performance at around 1.5% pixel density of the sparse depth input. This finding is corroborated by [15] who show a similar performance gap decrease when performing their own ablation study. They do however, introduce early- and late-fusion schemes in which RGB images are incorporated into their network and the results show that an early fusion of RGB causes the network to almost completely ignore the information from an RGB image [15].

3 Tools and Methods

This chapter discusses development of the simulation software, its auxiliary scripts and the chosen network structure. We will discuss the tool choices, justify them and propose viable alternatives if they are available. The data processing will also be discussed along with some methods and assumptions that make training the network easier. A comprehensive list of the tools used and their functions can be found in Table 2.

Table 2: Software tools and their function

Care-o-bot 4	A mobile robot designed as an assistant. We use a simulated model equipped with virtual sensors such as a 2D LiDAR and depth camera.
Gazebo	A robot simulation software capable of simulating user-created environments, complete with a physics engine, graphical interface and a programmatic interface for ROS. Used to build environments and populate them with objects.
Matlab	Numerical computation software. Used for pre-processing inputs to the simulation environment.
Python	High-level programming language with which the simulation and post-processing are built.
PyTorch	An open source machine learning library for Python.
ROS	A framework for writing robot software. Used for data gathering, environment mapping and automatic path planning within the simulation.

The open source Robotic Operating System (ROS) is the de-facto standard framework for robotics. Gazebo is a simulation environment developed to work directly with ROS. We decided to use these tools for the breadth of functionality and popularity within the robotic community. However, the main drawback of these tools is their volatility; some APIs have functions that are not supported, the documentation can be poor and interfacing with the python language is unstable.

ROS has support and APIs for C++, Python and Matlab. Since there are no real-time concerns with this software, which would have made C++ the more sensible choice, we chose to use python because it is easy-to-use and more flexible than Matlab. Pytorch was chosen for its ease of use compared to the TensorFlow library which is its main alternative. Particularly, Pytorch was chosen because of its data loading scheme which allows for easier customization and manipulation of training data.

The products of this thesis include many scripts with different functionalities. For reference, they are listed below in table 3. For more a detailed user guide, control flow scheme and API, see Appendix A.

Table 3: Created scripts, their inputs/outputs and functions

Name	Input	Output	Description
`extract_from_bag.py`	.bag file	I/O tuples of depth- and projected laser point images	Reads depth images and laser scans, synchronizes them and projects the laser points on the depth camera image frame.
`make_train_csv.py`	Paths to depth and laser image locations	.csv files containing paths to I/O tuples for training and validation	Splits depth and laser image tuple locations into training and validation .csv files.
`run_robot.py`	.csv files containing allowed coordinates for goals and objects	N/A	Core of the simulation control flow. Handles object spawning, goal broadcasting, camera facing and the robot state
`prepmap.m`	.pgm picture of a map	dilated .pgm picture of the map	Dilates boundaries and colors pixels within them.
`getcoords.m`	.pgm picture marked with allowable nav coordinates and .yaml file	.csv file	Converts marked pixels to spatial coordinates for navigation in ROS and Gazebo

3.1 Simulation Software Description

The reason for building a simulation software stems from the lack of comprehensive indoor data sets with appropriate sensor modalities and the prohibitive time cost of gathering such data with the real Care-o-Bot 4. An advantage of a simulation environment such as this is the access to near on-demand testing environments. This is particularly useful when investigating the performance of neural networks that are trained in this type of simulation. The software allows a user to generate and customize differing environments in order to see if a trained network has generalized and also facilitates user-testing. A drawback is that the training data and resulting networks are not immediately transferable to the real-world. This problem concerns the discipline of transfer learning, but is out of the scope if this work.

In this context, it is worth mentioning the recently released Gibson environment [20]. It includes detailed scans of over 1400 indoor floor spaces and 572 full buildings with numerous sensor modalities including dense depth images. It does not include LiDAR scans but this can be worked around by sampling the depth image to create a "fake" LiDAR scan. In our case, we utilize 2D LiDAR scans and project the points on to the image plane as is described in 3.2. The Gibson environment is detailed and comes with a variety of robot models but it lacks customization potential as the environments are scanned from the real-world and are static. The Gibson environment boasts many enticing qualities, but we do not use it since the release was after the start of this work and support for it is lacking because of how new it is.

3.1.1 Requirements and Challenges

This Section will discuss the software design, covering functionality, usage and challenges for all components of the system. The functionality is such that given a known map to navigate, the software shall spawn a number of obstacles with either pseudo-random or completely random placements. An option for a static, predetermined map configuration shall also be available. The robot shall then be given a random point on the map to navigate towards, keeping its camera facing the same direction it is driving. Once the robot either reaches its goal or stops moving, the current obstacles shall be deleted and new ones spawned with pseudo-random or completely random placements. The robot shall then be given a new random point on the map to navigate towards.

To facilitate these requirements, we utilize ROS in tandem with Gazebo and a Python script issuing commands. The main issues that had to be solved when making this system include random goal points, random obstacle placement, the robot's camera facing, catching instances where the robot gets stuck in the simulation due to collisions and working around the limitations of the built-in functions. Below, we will discuss these problems, their cause and how they were ultimately solved.

Random goal points

ROS offers a rudimentary automatic path planner which enables broadcasting a set of coordinates for the robot to navigate towards. The path planner consists of a global and local planner where the global one plans a preliminary path to the goal

and the local planner continually plans a path in the robots near vicinity, re-planning according to obstacles. The global path planner requires the robot to have complete information about the map to be navigated. Because of this, a user must build a map in Gazebo and manually navigate the robot through it using SLAM to acquaint the system with the map. This SLAM mapping produces an image of the map along with a .yaml file that specifies coordinate origin and resolution.

If the path planner is given an unreachable goal, it will throw an error and send the robot into a time consuming recovery mode, overriding all other commands while it is being executed. Since we want the robot to move while gathering data we can never issue coordinates that the robot cannot reach. To solve this, we made a Matlab script that takes in the outputs of the initial SLAM mapping mentioned above. It dilates boundaries on the image to close off open doors or paths and then translates all pixels that are inside the boundaries to coordinates using resolution information from the .yaml file. This script also facilitates a user manually marking allowed pixels (using GIMP for example) which can be advantageous if more customization is required.

Random obstacle placement

The goal here is to facilitate obstacles spawning randomly in the simulation. Given the context in which we are designing the software, the problem here is completely random obstacle placement. Such placement is not realistic and would not produce good training data. Instead, we want the obstacle placement to be random, but within reasonable bounds. This is an ambiguous requirement but we can define it as random placement of objects within areas where humans would put them, accompanied by a few completely random placements for variance of obstacle views. So the problem becomes: how to identify areas in which furniture and obstacles are likely to appear? One option is to have a software learn object placements from floor maps. The unavailability of such maps, however, prompted us to have a human stipulate where furniture and obstacles should appear. This is done by having a person mark the desired places and feeding the resulting image to the `getcoords.m` script shown in Table 3. After some preliminary testing we noted that objects had a tendency to spawn inside walls and each other. This is because Gazebo sets the default object state to static which means that the objects become unmovable in the simulation causing collisions to have no effect. This is a problem since both cases represent unrealistic scenarios, i.e. stacked furniture is rare enough to discard as a possibility and a table passing through a wall is unlikely (although the probability is non-zero [19]). Setting the object state to non-static allows the physics engine to resolve collisions and causes objects to bounce off each other in case they occupy the same space. To prevent any objects from spawning on top of each other, we simply keep track of all positions and force the software to re-generate a position if it is too close to any already-chosen positions.

Camera Facing

ROS does not incorporate functionality that keeps robot models facing toward their trajectories, save for non omni-directional robot models. For this problem, consider

the xy-plane to be the floor and the robot facing to be an arbitrary vector with an angle around the z-axis. To solve this we employ a thread that runs when the robot has been issued a goal. This thread periodically registers the robot position x_r, y_r and the robot facing θ_r. It then calculates a vector spanning from the previous position to the new position. This vectors angle around the z-axis is then calculated and compared to the current robot facing θ_r. A yaw-command is then issued accordingly. One thing to note is that while the yaw-command is being issued, the robot stops all other movement. This happens because any user-issued movement command overrides the path planner.

Collisions
During initial testing of the software we noted that a robot-to-obstacle collision would cause a positional mismatch between ROS and Gazebo. This is likely due to ROS using the wheel rotation for position while Gazebo uses the robot models' absolute position. This causes the global path planner to fail since the robot can no longer localize itself. In order to avoid colliding, ROS utilizes a cost map which takes in appropriate sensor modalities and creates an area around detected obstacles which the robot will try to avoid. Simply put, the path planner receives a numeric cost for any path segment it plans, depending on how close it goes to an obstacle. By default, the only sensor used to create a cost map is the 2D LiDAR. In our case, we noted that a cost map consisting of only the 2D LiDAR would fail when navigating past obstacles that have a greater occupancy than the laser can detect. The laser is positioned at around shin-height and will only see the legs of tables for example. According to a cost map generated with only this info, the robot might plan a path between the legs of a table. Simply increasing the cost map dilation is insufficient since it closes off otherwise accessible paths leading to the same global path planner failure that we want to avoid. The solution is incorporating the robots depth camera into the cost map. This requires modifying the navigation configuration of the virtual Care-o-bot 4 model. The cost map can include different layers and this solution simply adds a voxel layer which takes its observation source as the robots depth camera. We will not go into the details of this modification. For reference, all parameter modifications are marked as such in their respective .yaml files and a further information about the modifications can be found in Appendix A.

Built-in Function Limitations
This simulation uses an in-built client (`SimpleActionClient`) to publish navigation goals to the robot. This client has a major drawback in that it does not handle instances where the global path planner fails. This is because the method that publishes a goal to the robot initiates a thread that blocks all other execution until the robot has reached its goal. Thus, it does not handle situations where the robot is blocked from navigating towards its goal, i.e. a corridor may be blocked by an obstacle. In order to solve this, we created our own version of a blocking thread. This thread is the same one as is mentioned above under Camera Facing. The exit-criteria for this thread is simply to exit if the robots current and previous positions have not diverged over a two second period. For reference, a simple illustration of the

control-flow can be found in Appendix A.

Limitations of Gazebo
Two major drawbacks encountered when setting up simulations in Gazebo are the lack of support for obstacle shapes and its Python API volatility. Namely, Gazebo freezes Python when calling the `delete_model` service. This produces no error and the software simply freezes for an indefinite time. This is still an open issue on the ROS forums. Also, most complex models found for Gazebo do not present correctly in simulation when scanned by the LiDAR. For all of the non-stock models tested, the LiDAR simply sees their bounding boxes and not the actual legs of tables or chairs.

3.2 Data Collection

In this work, we gather data from two virtual sensor modalities, full-resolution depth images and 2D LiDAR scans. These two data types have different dimensions, 3D and 2D, respectively. The sparse-to-dense problem is conventionally tackled by utilizing neural network structures that map images to images. The depth information used is usually obtained from a conventional (3D) LiDAR or by sampling a full-resolution depth image, barring the approach in [8] which uses a 2D LiDAR scanner. A well performing neural network structure could be rewritten to take a vector containing all distances from a 2D LiDAR as an input. This approach would require time-consuming modification of any network structure that is to be tested in addition to manually fitting the LiDAR's FOV to the camera's. Also, in this case, the network would have to learn how to project the laser scan points onto an image plane since the LiDAR output is just a vector with numbers representing ranges from each scan angle. So the network would have to learn the depth camera parameters and the transformations from the depth camera coordinate system to the LiDAR coordinate system. So the above mentioned approach is not only time consuming but also makes the neural network training more difficult.

Luckily, we have access to the information needed to project the laser points onto an image that corresponds to the images taken by the depth camera. Consider a single laser scan p_L given in eq.(1) that consists of 3D points in the laser coordinate frame $\underrightarrow{F}_L$. Let us also define the parameters of the depth camera with a coordinate frame $\underrightarrow{F}_C$ as a pinhole model where the camera matrix A is given as

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

where f_x, f_y and c_x, c_y are the cameras focal lengths and principal point offsets, respectively. The distortion coefficients d for the camera are assumed to be zero. Recall that the transformation from the camera coordinate frame to the LiDAR coordinate frame is $T_{CL} = [R_{CL}|t_{CL}]$. The projection of the 3D points p_L onto the

depth camera image plane is then given by

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = AT_{CL} \begin{bmatrix} p_L & 1 \end{bmatrix}^T \tag{4}$$

Where u, v are the pixel coordinates of the image and s is some arbitrary scaling factor. Note that the raw data from the laser comes in the form of a vector containing the range in meters where each cell increment corresponds to a rotation of the laser beam around the z-axis. We can then simply insert the range value into their corresponding pixel coordinates as the intensity and normalize the image. The projection described in eq. (3) is facilitated by the cv2 library for python. All parameters necessary for this transformation can be obtained via the ROS tf-tree and camera topics which are recorded during simulation.

In this context, it is worth mentioning that the simulation gathers laser scans at a much higher frequency than depth images. The `extract_from_bag.py`script used to get the images gathers time stamps of each depth image and laser scan to synchronize them. Because of the higher frequency of the laser scan it might be feasible to associate each depth frame with more than one laser frame for training a neural network. For example associating a ground truth depth image with ten laser scans, gathered before and after the depth image. Given the extreme sparsity of the input data, an approach like this would multiply available input data points by up to tenfold. This approach is however, out of the scope of this work.

3.3 Neural Network Structure

The network we train was chosen from [1]. At the time of publication, it achieved good performance on the KITTI and NYU benchmarks with RGB images augmented with very sparse depth measurements. The motivation for this choice of network is twofold; It has to be deep enough to learn meaningful associations and shapes from sparse data and it has to be simple enough to provide a baseline for performance. In essence, it is better to choose a simpler solution and then evaluate if further augmentation is needed.

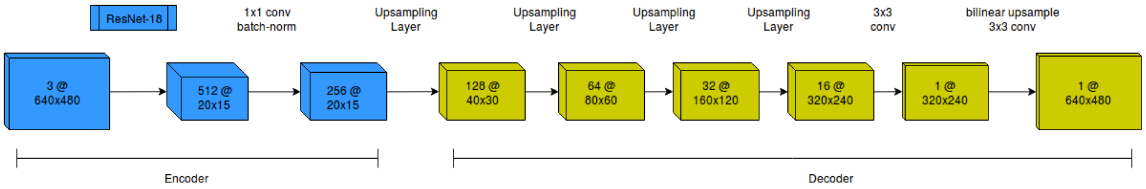

Figure 1: Architecture of the CNN. Each block is represented as features @ height x width.

The network has a bottleneck architecture consisting of an encoder and decoder. The encoding layer consists of an 18-layer ResNet [11] with a convolution layer at the end. Note that, as per [1], the final average pooling- and linear transformation layers of the ResNet are removed. The decoding layer is comprised of 4 upsampling layers with a bilinear upsample and a smoothing convolution layer at the end.

3.3.1 Metrics

Here, we will briefly go over and define the numerical metrics used to evaluate the results. These include MSE, RMSE, absREL, MAE and the delta function described in [1]. The MSE, RMSE and MAE metrics are documented because they are the standard by which top performing methods are ranked by, as seen in Table 1. The δ metric is chosen to give a greater breadth to the assessment criteria. For each following equation, we consider ground truth and predicted images with n pixels defined as y_i and $\hat{y}_i$, respectively, where i is the ith pixel in the image.

$$MSE = \frac{1}{n}\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5}$$

$$RMSE = \sqrt{MSE} \tag{6}$$

$$absREL = \frac{|\Sigma_{i=1}^{n}\hat{y}_i - y_i|}{Y} \quad Y = \Sigma_{i=1}^{n}y_i \tag{7}$$

$$MAE = \frac{1}{n}\Sigma_{i=1}^{n}|y_i - \hat{y}_i| \tag{8}$$

These metrics are widely used and well defined. This work utilizes one more metric which is described as "δ_k: percentage of predicted pixels where the relative error is within a threshold" [1]. Defined as

$$\delta_k = \frac{card(\{\hat{y}_i : max\{\frac{\hat{y}_i}{y_i}, \frac{y_i}{\hat{y}_i}\} < 1.25^k\})}{card(\{y_i\})} \tag{9}$$

where *card* is the cardinality of a set. δ_k ranges from 0 to 1 where 1 is the best prediction [1].
These metrics are documented during training and validation for each ground truth and prediction image pairs. The MSE criterion was used during training. Section 4.3 will discuss and analyze the efficacy of these and attempt to shed light on which metric performs well or poorly in different situations.

4 Experiments and Results

Up until now we have reviewed state-of-the-art methods in depth completion, defined and discussed the simulation, data processing and network structure. Let us explore the performance and ascertain whether it is feasible to infer depth maps from the extremely sparse data we use. This chapter will discuss the network training on a dataset gathered via eleven simulation runs, each lasting for 12 hours. We will also evaluate the performance on two validation sets, utilizing the simulation software's flexibility to quickly build new and different environments.

These experiments are made to answer questions of performance. Particularly, we want to gain insight into how the network structure handles the sparse input data and if the training set is sufficient in both detail and scene variance. To this end, we will use the metrics described in Section 3.3.1 and evaluate predicted images in the context of visual inspection, i.e. we will look at the predicted images, determine if the metrics are good or poor and analyze if some of them fail when compared to visual appraisal. Furthermore, we will inspect chosen cases where the predicted images are either poor or good and give an analysis based on both the metrics and network structure.

4.1 Experiment Settings

The training data was gathered in the simulation software described in Section 3. A map layout was designed in Gazebo to mimic an indoor office space, hereafter referred to as the TUAS world. The exploratory nature of this work pushed us to gather training data in a static environment to ascertain whether the auto encoder could even learn a simple setting. The simulations yielded 59741 I/O tuples consisting of time-synchronized projected laser and depth images with a resolution of 640x480. The set was randomly split into training and validation tuples numbering 53841 and 5900, respectively. The training tuples were then randomly horizontally mirrored during training, effectively doubling the set. An epoch represents one forward and one backward pass of all training I/O tuples. This means that with a batch size of 10 and roughly 50 thousand I/O tuples, one epoch would take 5000 iterations to complete. After preliminary runs of the network, we found that the metrics (MSE, RMSE, MAE, etc.) plateaued after around 170 epochs as can be seen in Figure 3. The chosen network parameters are presented in Table 4.

Table 4: Network parameters. *Learning rate decays by a factor of 10 every 40 epochs.

Number of epochs	200
Batch size	10
Number of workers	4
Learning rate	0.01*
Momentum	0
Weight decay	0.0001
Criterion	MSE
Optimizer	Adam
ResNet layers	18
Decoder	upconv

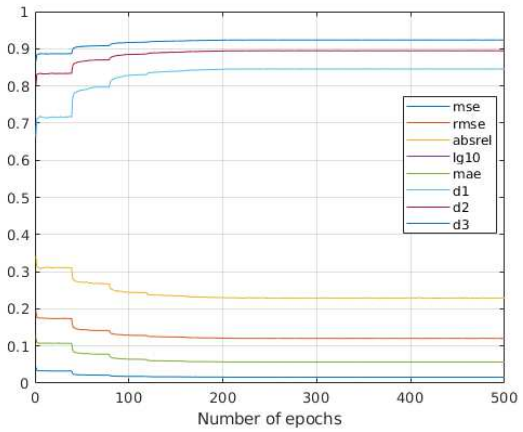

Figure 2: Metric progression during training.

To analyze the network performance, two validation testing sets were generated. One was generated in the same manner as the training set with different robot goals and trajectories. The other validation set, hereafter referred to as the Sandbox world, was generated in a smaller map layout with different obstacle placements and corridor width.

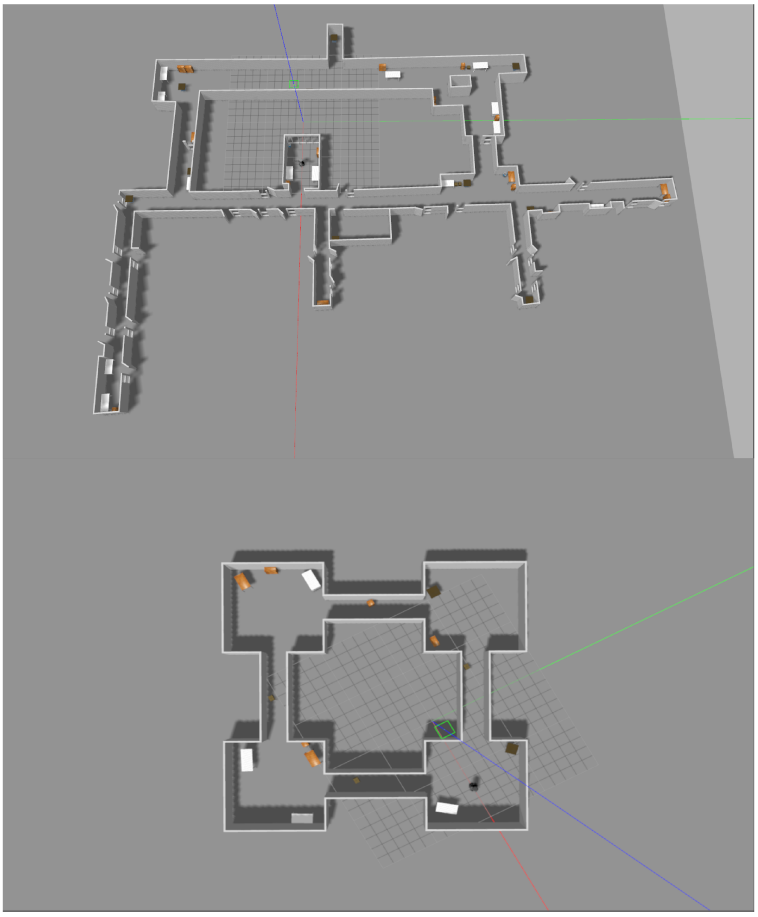

Figure 3: TUAS training map (top) and Sandbox validation map (bottom)

4.2 Results

This section presents the numerical and visual results of the network when applied to two different environments; one known and the other unknown. Below, we illustrate results in a number of figures. For better overview and comparison, Table 5 summarizes the error metrics in each figure.

4.2.1 Validation on a Known Environment

The trained network was validated on an image set generated by running a new simulation in an unchanged training environment. This was to ensure that validation would not be performed on the same path trajectories as were made while generating the training set. This validation set consists of 3507 I/O tuples. We recorded errors for every image in the validation set, giving us concrete numerical metrics for each image to compare during visual inspection. For overview, Table 6 shows the averages and variances for each metric over the whole set along with worst and best values. The averages indicate a good overall performance, with the RMSE error being around

Table 5: Summary of error metrics for each figure

fig	MSE	RMSE	absREL	MAE	$\delta 1$	Visual
4	0.2730	0.5225	2.5208	0.4422	0.2425	Bad
5	0.0786	0.2803	5.5107	0.2418	0.1268	Bad
6	0.0046	0.0681	0.0286	0.0208	0.9766	Good
7	0.0331	0.1820	0.2994	0.1115	0.5121	Very bad
8	0.0056	0.0748	0.0555	0.0353	0.9261	Good, details lacking
9	0.0266	0.1631	0.3024	0.1055	0.5805	Ghost obstacle
10	0.0201	0.1417	0.0855	0.06019	0.8602	Overdraw
11	0.0618	0.2486	0.3047	0.1423	0.7611	Very bad
12	0.0065	0.0805	0.0736	0.0305	0.9065	Good
13	0.0090	0.0947	0.0265	0.0193	0.9850	Good
14	0.0104	0.1018	0.0907	0.0585	0.9270	Low contrast
15	0.0060	0.0774	0.0842	0.0381	0.8792	Training artifact

10 cm which is a good margin given the environment scale. The variances are all an order of magnitude lower than the average except for the absREL metric. This indicates that the environment might be too monotonous such that scenes are very similar. The worst and best values however indicate that there is at least one scene that the network cannot handle. Indeed, these cases are more than one and will be discussed below.

Table 6: Results for TUAS validation

Metric	average	variance	worst	best
MSE	0.0154	9.4885e-04	0.2730	2.1247e-04
RMSE	0.0992	0.0056	0.5225	0.0146
absREL	0.1876	0.2212	5.5107	0.0112
MAE	0.0528	0.0045	0.4422	0.0052
$\delta 1$	0.8603	0.0409	0.0051	0.9997

Below, we will present predicted depth frames with the input and ground truth for reference. The images are chosen both by the numerical metrics and by visual appraisal such that both good and bad cases are shown. Every image contains, from left to right; input, ground truth, prediction.

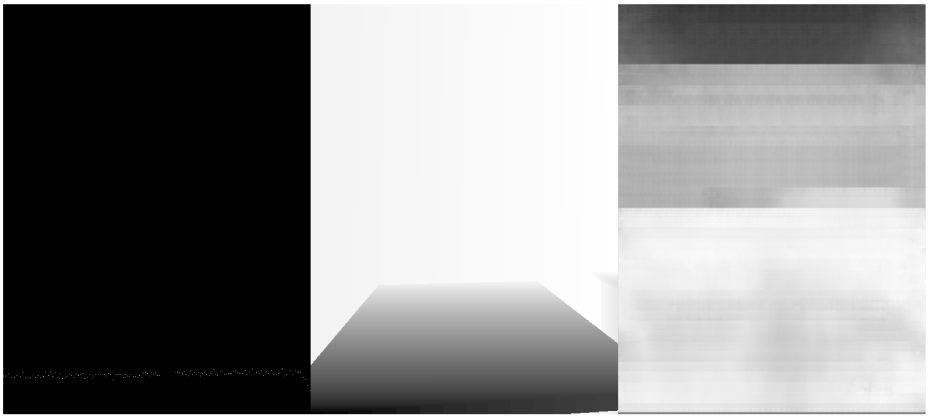

Figure 4: TUAS environment; worst prediction according to MSE, RMSE and MAE. absREL=2.5208, $\delta 1$=0.2425 consult 6 MSE, RMSE and MAE.

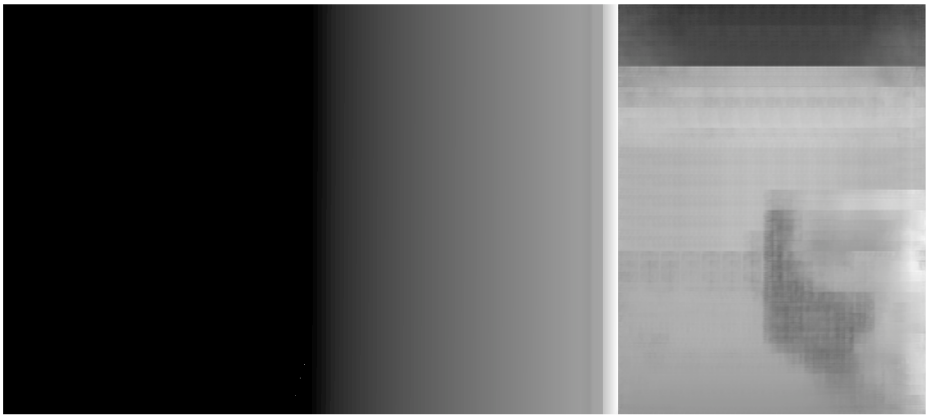

Figure 5: TUAS environment; worst prediction according to absREL. MSE=0.0786, RMSE=0.2803, MAE=0.2418, $\delta 1$=0.1268.

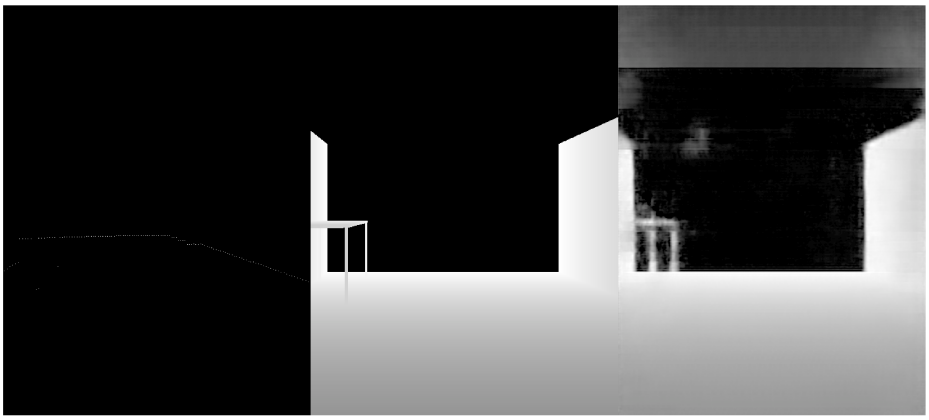

Figure 6: TUAS environment; a visually good prediction. MSE=0.0046, RMSE=0.0681, absREL=0.0286, MAE=0.0208, $\delta 1$=0.9766.

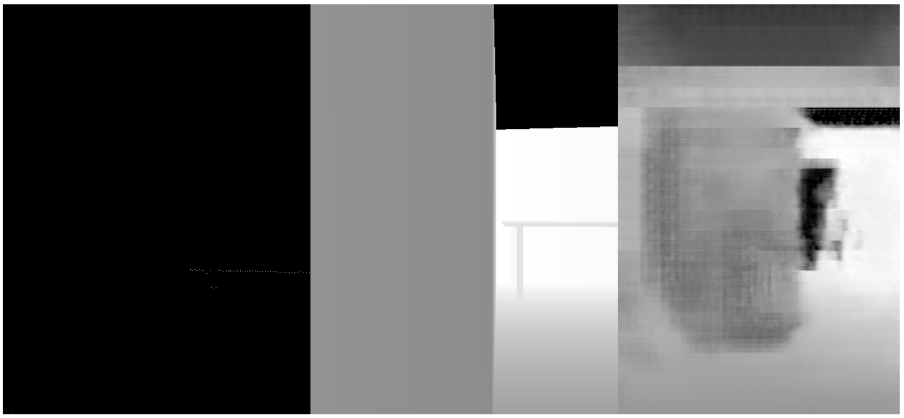

Figure 7: TUAS environment; a visually poor prediction. Note the lack of depth points on the left hand side. MSE=0.0331, RMSE=0.1820, absREL=0.2994, MAE=0.1115, $\delta 1$=0.5121.

Figure 8: TUAS environment; missing details. MSE=0.0056, RMSE=0.0748, absREL=0.0555, MAE=0.0353, $\delta 1$=0.9261

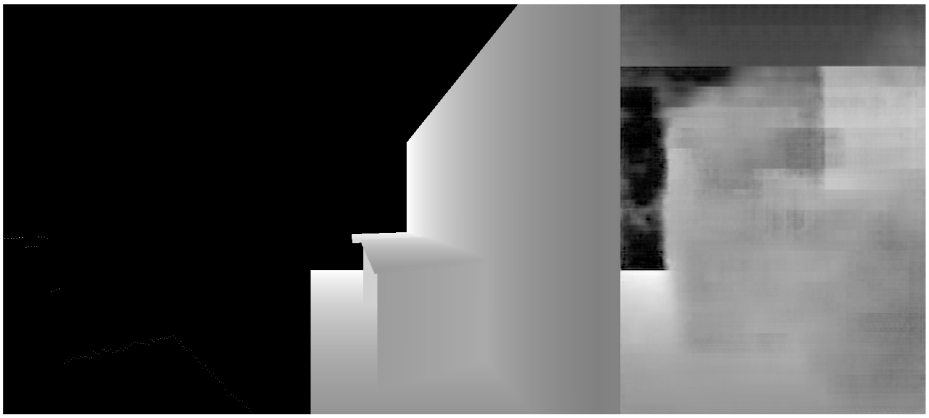

Figure 9: TUAS environment; case of a ghost obstacle being predicted. MSE=0.0266, RMSE=0.1631, absREL=0.3024, MAE=0.1055, $\delta 1$=0.5805

To further investigate, the error metrics and their respective image indices were sorted from best to worst. This way, a visual confirmation is obtained that the best and worst images are the same case. The top 20 or so predictions all contain open areas, which the network has learned very well. The top 20 worst predictions are all cases where the robot is extremely close to either a wall or obstacle. This will be discussed in 4.3.

4.2.2 Validation on an Unknown Environment

The trained network was also validated on an image set generated by running a new simulation in an unknown training environment with differing corridor width, obstacle placement and layout. This is intended to explore how well the network can generalize from the training set. This validation set consists of 6648 I/O tuples. We recorded errors for every image in the validation set, giving us concrete numerical metrics for each image to compare during visual inspection. Table 7 shows the averages and variances for each metric over the whole set along with worst and best values.

Table 7: Results for Sandbox validation

Metric	average	variance	worst	best
MSE	0.01	2.5978e-04	0.2871	1.9776e-04
RMSE	0.0874	0.0024	0.5358	0.0141
absREL	0.1093	0.0815	4.1106	0.0119
MAE	0.0416	0.0016	0.5014	0.0060
$\delta 1$	0.8996	0.017	0.0078	0.9999

In the unknown environment, the worst and best cases are exactly the same situations as in the known environment. The best predictions are from open areas with no

obstacles and the worst predictions happen when the robot is extremely close to walls/obstacles. Visually, these predicted images do not differ from the ones presented in 4.2.1. so we will omit these and reference Table 7 for extreme values. Instead, we will focus on visually interesting images and inspect their metrics.

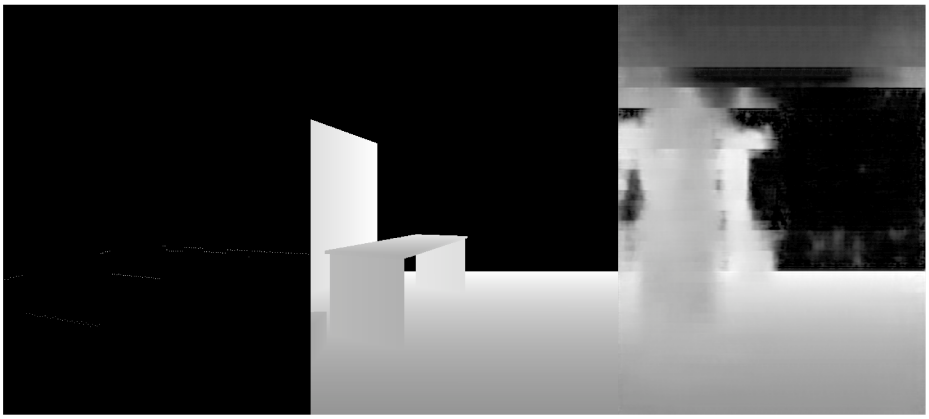

Figure 10: Sandbox environment; overestimation of obstacle occupancy. MSE=0.0201, RMSE=0.1417, absREL=0.0855, MAE=0.06019, $\delta 1$=0.8602

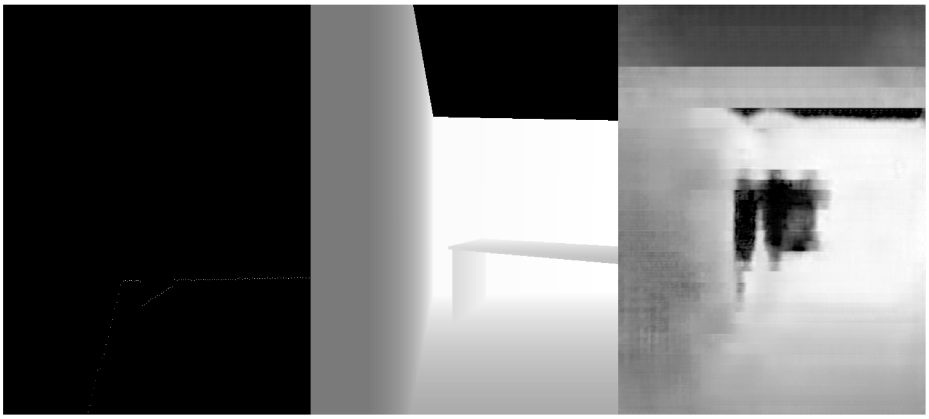

Figure 11: Sandbox environment; Visually poor prediction. MSE=0.0618, RMSE=0.2486, absREL=0.3047, MAE=0.1423, $\delta 1$=0.7611

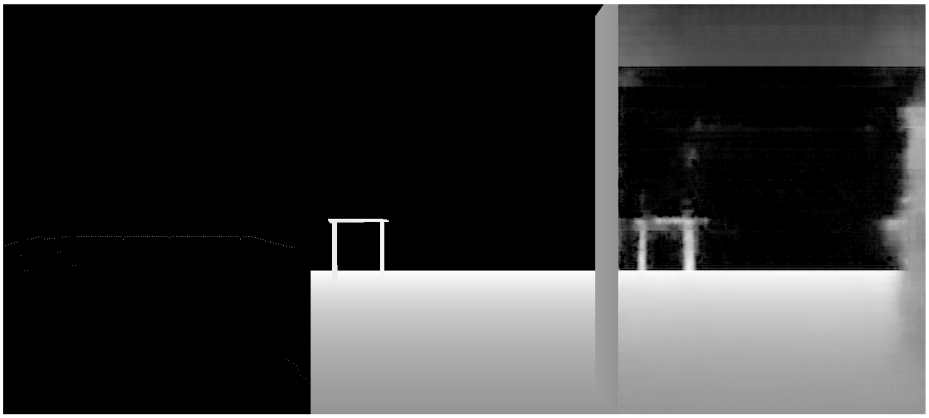

Figure 12: Sandbox environment; Visually good prediction. MSE=0.0065, RMSE=0.0805, absREL=0.0736, NaN, MAE=0.0305, $\delta 1$=0.9065

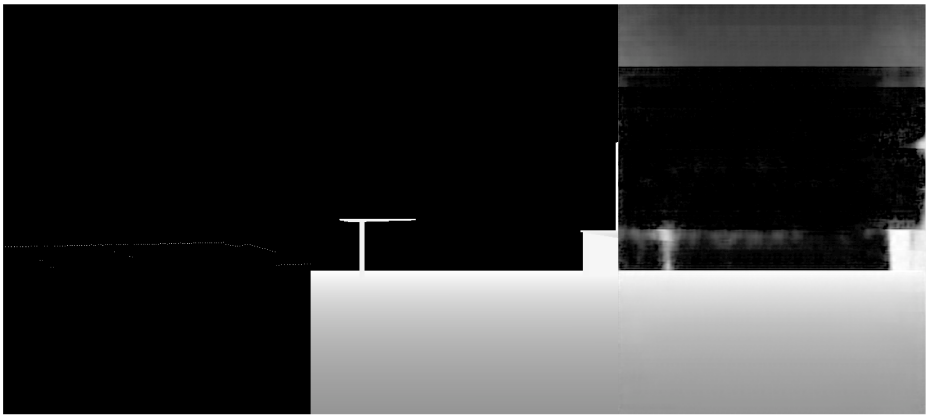

Figure 13: Sandbox environment; A good prediction of obstacles. MSE=0.0090, RMSE=0.0947, absREL=0.0265, MAE=0.0193, $\delta 1$=0.9850

Figure 14: Sandbox environment; Low contrast case, note the shadows where the table legs are. MSE=0.0104, RMSE=0.1018, absREL=0.0907, MAE=0.0585, $\delta 1$=0.9270

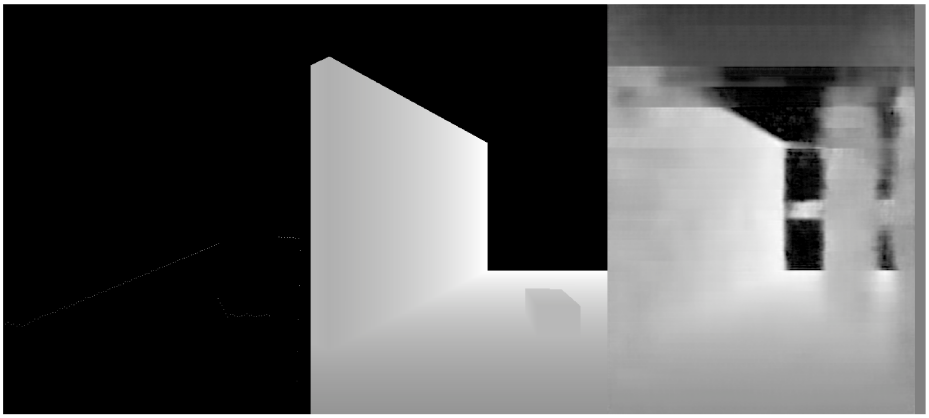

Figure 15: Sandbox environment; an artifact from training. MSE=0.0060, RMSE=0.0774, absREL=0.0842, MAE=0.0381, $\delta 1$=0.8792

4.3 Discussion

Overall, considering the input data sparsity, the performance is surprisingly good. Expecting a fully detailed depth map from such a sparse input with no guidance for shape is a tall order. Since this approach was tested on a custom generated dataset, we can make no meaningful comparison to other works. Instead we will attempt to evaluate the network performance by the metrics defined in Section 3.3.1 and by visual appraisal. We also need to consider the context in which we evaluate the predictions. So we need to consider differences in criteria for example tele-operation where shape is more valuable versus autonomous navigation where accurate depth estimation in each pixel might be more useful. For the purposes of this discussion, we will focus on the tele-operation case, since the current network output is too noisy for autonomous application.

By far, the most striking result is that validation on an unknown environment outperforms the original training environment in almost every metric as can be seen in Tables 6 and 7. This is highly irregular since the network should obviously have learned its training environment better than an unknown one. By sorting the error metrics by performance, we can take a look at the cases that produce the worst errors. All of the used error metrics show that the worst results occur when the robot is very close to walls and obstacles, see Figures 4 and 5. The TUAS environment consists, by large, of long and tight corridors. When the robot turns in place or around corners, it is more likely that this close-up case happens. To show this numerically, we can look at a histogram of the MSE of both the TUAS and Sandbox environments.

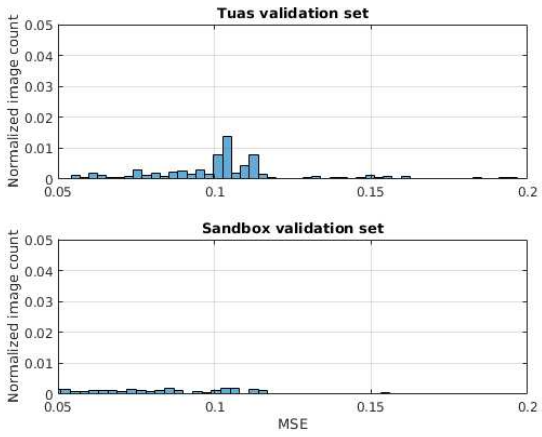

Figure 16: Histogram of the MSEs of TUAS and Sandbox environments. The image count has been normalized.

Figure 16 shows that the high error cases do occur more frequently in the TUAS environment. To confirm that these are close-up cases, we can inspect points of high error in the MSE for the TUAS environment.

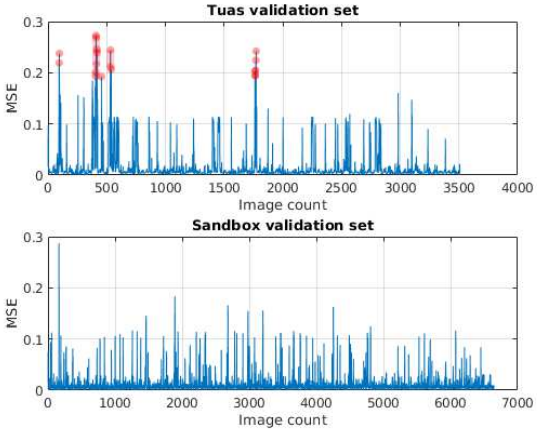

Figure 17: High MSE error cases in the TUAS environment.

The images belonging to peaks marked red in Figure 17 were inspected and confirmed to be cases of close-up images which produce this high error. We also see a large number of images with 0.1 MSE errors. These images are numerous so an exhaustive visual check cannot be done on all of them. Looking at a few of them however, we can hypothesize them to be scenes of corridors. So we have shown that the difference in performance has a reasonable explanation.

Looking at the images, we can see that the network is able to generate walls and corridors well, and, given that the laser's range is longer than the camera's, it has also learned that points extending a certain range are to be interpreted as black or

unknown. In high contrast cases such as this (a black background), the network is able to infer an approximate occupancy of obstacles as seen in Figures 6, 12 and 13, since it knows that everything above a certain pixel value is out of range. Although in low contrast cases, it performs more poorly as seen in 14. The explanation for this might stem from the training environment not containing many of these cases since obstacles were placed alongside walls and the robot always moves parallel to them, rarely obtaining data points that encompass a view like this. The low contrast issue is also characterized by blurry boundaries, [1] suggests that fine details are lost in the bottleneck architecture of the network. This indicates that a greater breadth of data is required and that skip connections might be warranted since they preserve details in bottleneck architectures [3].
Figures 7 and 11 illustrate the visually worst kind of errors; prediction of gaps in a solid background. Indeed, the RMSE is a good metric to go by when catching bad predictions like this, since it penalizes large pixel errors more. It does however fail when appraising an object that is close to the robot such that the difference between a black background and the falsely predicted object becomes small.
In this context it is worth mentioning the nature of our ground truth data; the virtual depth camera has a limited range and represents out-of-view parts of its output as black pixels. Thus, it will produce images that contain a large portion black pixels when viewing an open area for example. The implication of this is that the error metrics include missing information, i.e. they calculate the difference between a predicted pixel and a ground truth pixel that essentially contains no information. This is a problem when calculating an error of false predictions that are very close to the robot but where the ground truth has a black background. The prediction is false, but the error becomes very small. This case is illustrated in Figure 15, note the below average RMSE. This is also a problem when the network produces noise and artifacts where the background is supposed to be black. This indicates that representing unknown pixels as zeros is inadequate for this application, since it causes any metric to produce low errors for close, falsely predicted obstacles.

Note that none of our error metrics take into account that some errors are worse than others, i.e. overdrawing an obstacle into a column is not that bad for tele-operation, but incorrectly placing black/unknown pixels into walls or obstacles is worse. Here, an error metric that is sensitive to the error direction (i.e. is the predicted value greater or lesser than the ground truth) might be warranted. However, placing black holes or tears such as in Figure 12 in a prediction is likely a result of the training environment where such features were inserted freely to mimic windows and open doors. To strengthen the claim that a more diverse dataset is in order and that the network has over fitted some characteristics of the training environment, Figure 15 shows a remnant of the training data set. Here we can see a prediction of windows and doorways in the TUAS environment, which are not present in the unknown environment. This case happens numerous times in the unknown validation set.

In light of the above discussion, we can say that the models reliability rests on the training environment and the constraints of the laser range. The model con-

sistently produces correct predictions of hallways and corridors. Given unobscured views, it also infers obstacle occupancy to a certain degree. Although it does not predict obstacle occupancy to such a degree that even a human operator would never collide. The model also struggles where a part of the view contains no information because of the minimum range of the LiDAR, as seen in Figure 12. In the simulations, we did not incorporate any model for noise in the LiDAR measurements. So this model would not be transferable to the real world. The model also produces artifacts from the training environment when applied to the unknown environment. From this we gather that it generalizes poorly since it was trained on a static environment and as such has low external validity. This indicates that a greater breadth of environments and scenes is warranted.

5 Conclusions

In this thesis, we went over the motivations and problem structure for depth completion via 2D LiDAR. We reviewed relevant literature and assessed the state-of-the art methods according to performance on two well known benchmarks. A simulation environment was built, capable of flexible indoor data gathering and customization. We tested an existing deep regression network, with a few modifications, to explore the feasibility of inferring such complex scenes from such sparse data. The results were evaluated and based on them, conclusions and suggestions regarding future direction will now be discussed.
To answer the original question; is it feasible to infer depth maps from 2D LiDAR data in a simulated environment? For a static environment, the answer is yes. The results also indicate that this approach could be generalized to more varying environments. The network tested in this thesis is relatively simple when compared to state-of-the-art networks and still performed better than expected; it could learn shapes of obstacles to an impressive extent when the input data sparsity is considered. There are still some challenges to consider though.
Our results indicate that the network has over-learned some characteristics which is not surprising considering that it was trained on data from a static environment for exploratory purposes and because of the software limitations discussed in Section 3.1.1. We also note that our ground truth images have a limited range and set unknown pixels to zero, while a lower pixel value indicates a shorter distance. Thus, common error metrics cannot correctly identify possibly serious errors. Finally, the blurriness of obstacle boundaries suggest that the bottleneck architecture of the network may be discarding detail.

To address these challenges, let us discuss some propositions. The obvious solution to over-fitting is increasing the data set variance. Sadly, the chosen software, namely Gazebo does not facilitate this as is discussed in Section 3.1.1, also prior to starting this work, we had no knowledge of how unstable the python API is. When reviewing the current build of the software, it might be feasible to rewrite the simulator script in C++. To circumvent these challenges, there are three possible solutions: building custom obstacles that are compatible with virtual laser scanning in Gazebo, using a more powerful simulator than Gazebo or utilizing the recently released Gibson environment [20]. To address the unknown pixel value issue we suggest inverting the ground truth images during training and then convert the outputs back to the conventional format afterwards. Finally, to address the lost details in the bottleneck network architecture, we add skip connections between each corresponding encoding/decoding layers [3].

Given the increased interest in depth sensing for mobile platforms and the wide availability of cheap 2D LiDARs, the motivation to delve further into this topic is apparent. This thesis has provided a precursor to the infrastructure required for it and a heading regarding the methods needed. We are still a ways off to practical implementation but who knows? Maybe future robots will all hallucinate via laser.

References

[1] F. Ma and S. Karaman, "Sparse-to-dense: Depth prediction from sparse depth samples and a single image," *IEEE International Conference on Robotics and Automation*, pp. 1–8, 2018.

[2] F. Ma, G. V. Cavalheiro, and S. Karaman, "Self-supervised sparse-to-dense: Self-supervised depth completion from lidar and monocular camera," *arXiv preprint arXiv:1807.00275*, 2018.

[3] J. Lundell, F. Verdoja, and V. Kyrki, "Hallucinating robots: Inferring obstacle distances from partial laser measurements," *arXiv preprint arXiv:1805.12338*, 2018.

[4] B. Yang, S. Rosa, A. Markham, N. Trigoni, and H. Wen, "Dense 3d object reconstruction from a single depth view," *IEEE transactions on pattern analysis and machine intelligence*, vol. abs/1802.00411, 2018.

[5] N. Chodosh, C. Wang, and S. Lucey, "Deep convolutional compressed sensing for lidar depth completion," *arXiv preprint arXiv:1803.08949*, 2018.

[6] A. Eldesokey, M. Felsberg, and F. S. Khan, "Propagating confidences through cnns for sparse data regression," *arXiv preprint arXiv:1805.11913*, 2018.

[7] X.-W. Chen and X. Lin, "Big data deep learning: challenges and perspectives," *IEEE access*, vol. 2, pp. 514–525, 2014.

[8] Y. Liao, L. Huang, Y. Wang, S. Kodagoda, Y. Yu, and Y. Liu, "Parse geometry from a line: Monocular depth estimation with partial laser observation," in *IEEE International Conference on Robotics and Automation*, pp. 5059–5066, 2017.

[9] L.-K. Liu, S. H. Chan, and T. Q. Nguyen, "Depth reconstruction from sparse samples: Representation, algorithm, and sampling," *IEEE Transactions on Image Processing*, vol. 24, no. 6, pp. 1983–1996, 2015.

[10] X. Cheng, P. Wang, and R. Yang, "Depth estimation via affinity learned with convolutional spatial propagation network," in *European Conference on Computer Vision*, pp. 108–125, Springer, Cham, 2018.

[11] 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, 2016.

[12] J. Hua and X. Gong, "A normalized convolutional neural network for guided sparse depth upsampling.," in *International Joint Conferences on Artificial Intelligence*, pp. 2283–2290, 2018.

[13] J. Uhrig, N. Schneider, L. Schneider, U. Franke, T. Brox, and A. Geiger, "Sparsity invariant cnns," *arXiv preprint arXiv:1708.06500*, 2017.

[14] Z. Huang, J. Fan, S. Yi, X. Wang, and H. Li, "Hms-net: Hierarchical multi-scale sparsity-invariant network for sparse depth completion," *arXiv preprint arXiv:1808.08685*, 2018.

[15] M. Jaritz, R. De Charette, E. Wirbel, X. Perrotton, and F. Nashashibi, "Sparse and dense data with cnns: Depth completion and semantic segmentation," in *2018 International Conference on 3D Vision*, pp. 52–60, IEEE, 2018.

[16] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *Computing Research Repository*, vol. abs/1707.07012, 2017.

[17] C. Cadena, A. R. Dick, and I. D. Reid, "Multi-modal auto-encoders as joint estimators for robotics scene understanding.," in *Robotics: Science and Systems*, vol. 12, pp. 1–9, 2016.

[18] "Google dictionary." https://www.google.com/search?client=ubuntu&hs=T1&channel=fs&q=Dictionary#dobs=hallucination. Accessed: 2018-10-20.

[19] D. Griffiths, *Introduction to Quantum Mechanics*. Pearson international edition, Pearson Prentice Hall, 2005.

[20] F. Xia, A. R. Zamir, Z.-Y. He, A. Sax, J. Malik, and S. Savarese, "Gibson env: real-world perception for embodied agents," in *Computer Vision and Pattern Recognition*, IEEE, 2018.

[21] "Kitti depth completion ranking." http://www.cvlibs.net/datasets/kitti/eval_depth.php?benchmark=depth_completion. Accessed: 2018-10-24.

A Appendix

This Appendix is intended to document the software functions, classes and usage. The general structure of the software will be covered along with usage guidance and pitfalls. To use the software, the following step-by-step guide is presented along with an illustration shown in Figure A1. The reader is assumed to have rudimentary knowledge of ROS and Gazebo.

1. Design a map layout in Gazebo. This can be done via drawing freely or importing an image as a template. Save the resulting map as a .world file.

2. SLAM the .world file in Rviz. This is done by loading the map in Rviz and manually navigating the robot around the environment until it has gathered data points that encompass its operating space. Save a .png picture of the map and a .yaml file that contains resolution and coordinate origin information.

3. Make copies of the .png picture from the SLAM and mark allowable coordinates for goals and different obstacle types using software such as GIMP. Run the Matlab script `getcoords.m` to get .csv files that contain allowable coordinates. Conversely, if the user wants an automatically generated list of goal coordinates, run the script `perpMap.m` and then `getcoords.m`.

4. Launch Gazebo and Rviz with the .world, .png and .yaml files. A launch file `env_world.launch` is provided. Once the simulators are running, provide the `run_robot.py` script with paths to the coordinate .csv files and run it via the `rosrun` command for testing or via launch file `record_data.launch`. It will run until interrupted.

5. Given that the launch file `record_data.launch` was used to initialize, an output .bag file is available after simulation. Provide the script `extract_from_bag.py` with appropriate paths to the .bag file, topics to monitor and output directories. It will extract and synchronize depth images and laser data projected onto an image plane that corresponds to the depth images.

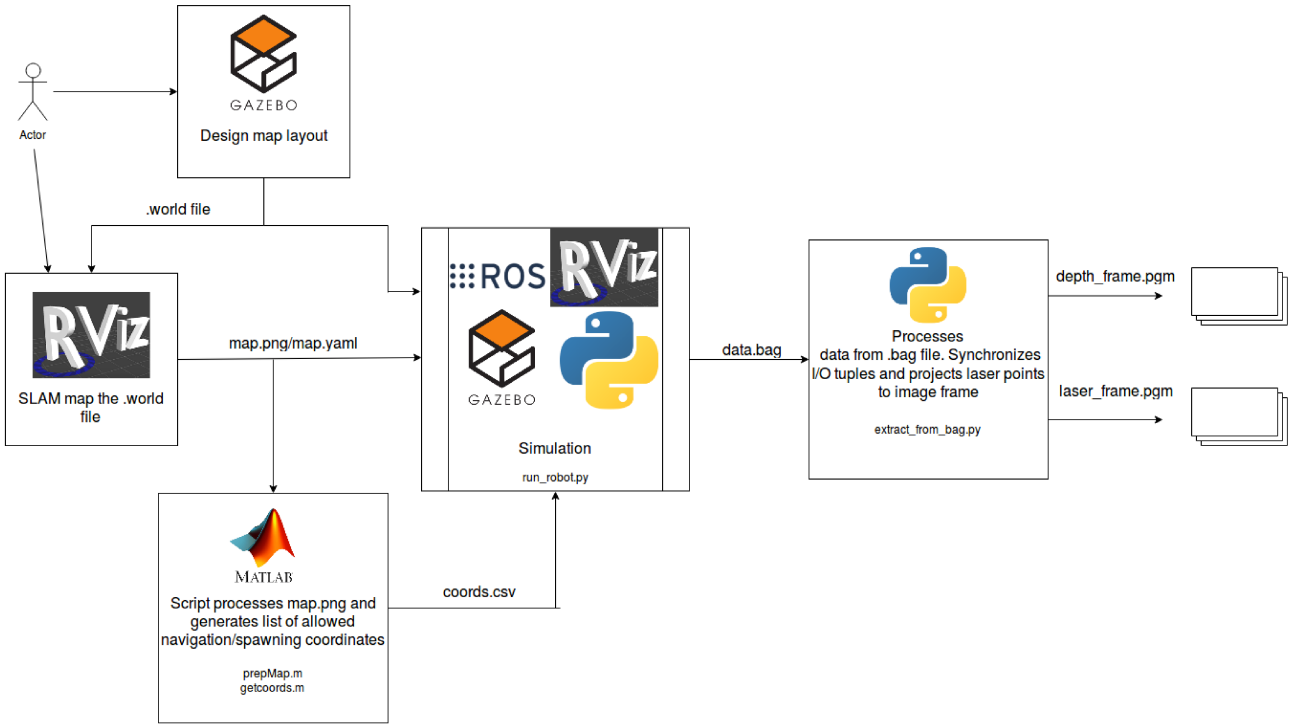

Figure A1: Illustration of a use-case to run a simulation and gather data.

The control flow is presented in Figure A2.

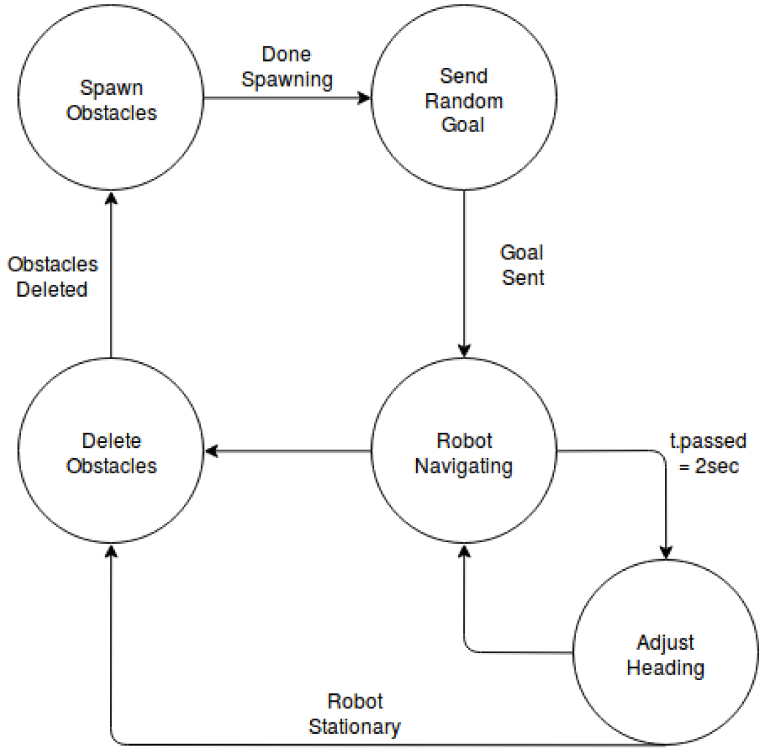

Figure A2: Control flow of simulation for data gathering.

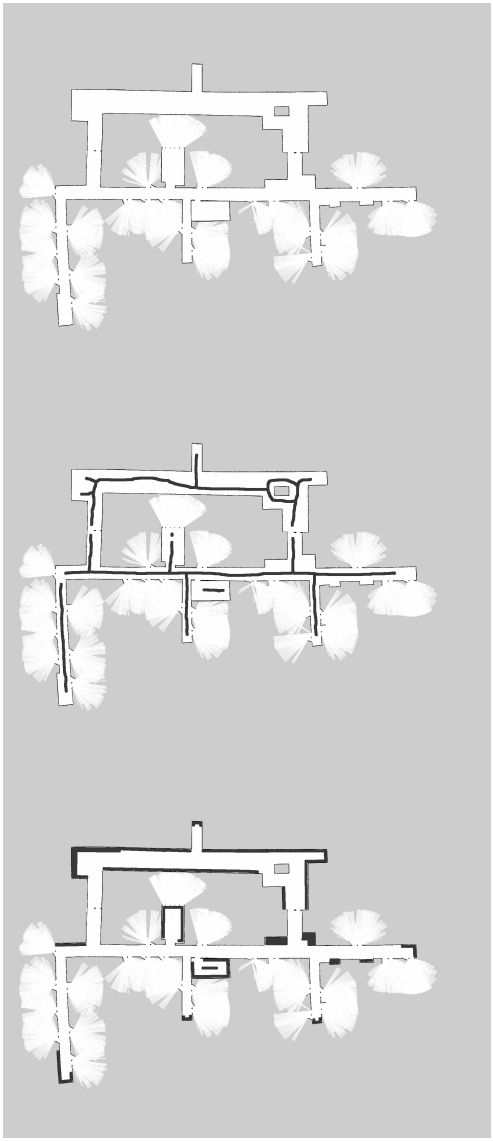

Figure A3: Examples of marking coordinates on SLAM mapped image. Top: original map, middle: marked goals, bottom: marked furniture placements.

run_robot.py

This script automates goal broadcasting and object spawning/deletion for a simulation in ROS/Gazebo. It needs to be supplied with paths to .csv files of allowable robot goal coordinates and any coordinates for various obstacle types such as tables, chairs, clutter, etc. These .csv files are generated via the `getcoords.m` script, described below in this document.

class Block:

Encapsulates format for models in Gazebo.

- **Class functions:**
- def ___init___(self, name, relative_entity _name)
  - USAGE: Block(name, relative_entity _name)
  - INPUT: name is a valid model name as it appears in Gazebo. relative_entity_name is a custom name.
  - OUTPUT: Constructs a Block class, encapsulating a format for models in Gazebo.

class ModelPos:

Class that gets and manages model coordinates and orientation in Gazebo. Add models by inserting a valid Gazebo model name into the _blockListDict list.

- **Class functions:**
- def show_gazebo_models (self)
  - USAGE: models = show_gazebo_models ()
  - INPUT: N/A. _blockListDict must contain at least one valid model
  - OUTPUT: Returns the x- and y- coordinate position (in meters) and orientation (around the z-axis in degrees) of all models in _blockListDict such that [x,y,θ]

class Blocker:

Class that encapsulates position and orientation change of the robot and initializes a thread conditioned on those parameters.

- **Class functions:**
- def ___init___(self)
  - USAGE: Blocker()
  - INPUT: N/A.

  - OUTPUT: N/A. Initializes a threading condition

- def pathcatcher(self, goal)
  - USAGE: catch = pathcatcher(goal)
  - INPUT: A list where the first two values list[0] and list[1] contain x- and y- coordinates, respectively, in meters.
  - OUTPUT: Blocks execution of other functions until exit. Returns False if robot is mobile. Returns True if robot is immobile.

Global functions

- def movebase_client()
  - USAGE: run = movebase_client()
  - INPUT: N/A
  - OUTPUT: N/A. Selects random goals and coordinates for robot and obstacles, publishes them. Repeats this operation everytime the robot stops moving.

- def spawn_items(item_name, prodXml, numItems)
  - USAGE: models = spawn_items(name, prodXml, numItems)
  - INPUT: name is a list of model names. prodXml is a list of corresponding model Xml file paths. numItems is either the number of objects to be spawned or None. If None, the GENERALIZE?

- def delete_items(item_name)
  - USAGE: delete_items(item_name)
  - INPUT: A list of item names.
  - OUTPUT: N/A. Deletes models contained in input list item_name.

- def getrandompoint()
  - USAGE: goal, idx = getrandompoint()
  - INPUT: N/A. Reads a global list of allowed goal coordinates for the robot.
  - OUTPUT: selected random goal and its list index.

- def spawn_items()
  - USAGE: spawnedModels = spawn_items(item_name, prodXml, numItems)

  - INPUT: item_name is a list containing names of items, prodXml is a list containing paths to those items, numItems is the number of items desired. If numItems is None, all items in item_name will be spawned.
  - OUTPUT: A list of items spawned. This function will not spawn items of the same type in close proximity to one another. It will also spawn 20% of objects in abnormal coordinates, which is a .csv file that the user provides.

- def getobstaclepoint()
  - USAGE: goal, idx = getobstaclepoint()
  - INPUT: N/A. Reads a global list of allowed coordinates for a given model type.
  - OUTPUT: selected random obstacle coordinates and their list index.

- def proximitycheck(idx, selection)
  - USAGE: isclose = proximitycheck(idx, selection)
  - INPUT: idx is a list containing coordinate list indices of already-spawned models, selection is an index from the same coordinates list.
  - OUTPUT: True if selection is close to any index in idx. False otherwise.

extract_from_bag.py

This script extracts 2D LiDAR readings and depth images from a rosbag, synchronizes them and returns corresponding image tuples. It contains only one method and is used in the following way:

```
python extract_from_bag.py arg1 arg2 arg3
```

Where `arg1` is the path to a .bag file to be extracted, `arg2` is the name of the depth camera ros topic and `arg3` is the name of the laser ros topic.

- def find_nearest(array, value)
  - USAGE: idx = find_nearest(array, value)
  - INPUT: array is a sorted array of numbers and value is a number.
  - OUTPUT: idx is the index of the number in array that is closest to value.

getcoords.m

This script is intended to generate a .csv file of xy-coordinates (in meters) from a .pgm image of a map and a corresponding .yaml file. The script will read a chosen pixel value which is marked by the user via Gimp and translate each pixel to coordinates. To run this script, supply it with the following items:

- A path to a .pgm image of a map that is manually marked with allowable coordinates (This .pgm image is an output from SLAM in Rviz)
- The resolution and origin coordinates from a .yaml file that corresponds to the map (Also an output from SLAM in Rviz)
- Pixel value of the gray scale color used to mark allowable spots on the .pgm image
- Name of output .csv file

prepMap.m

This script is intended to automatically identify a bounding box for an indoor floorplan and generate a set of coordinates withing that bounding box. It will generate a .pgm image containing the allowable coordinates of the map provided. To run this script, supply it with the following items:

- A path to a .pgm image of a map with no manual markings (This .pgm image is an output from SLAM in Rviz)
- The resolution and origin coordinates from a .yaml file that corresponds to the map (Also an output from SLAM in Rviz)
- Name of output .csv file

Modifications

This section touches on the costmap modifications made to avoid obstacle collision in simulation. By default, the costmap that ROS generates for the Care-o-bot 4 includes only its 2D LiDAR. Below is a list of parameter files that were modified such that the costmap also considers the depth camera input. This enables the robot to avoid colliding with objects such as tables. In each of these files, the changes are clearly commented.

Table A1: Modified configuration files

/cob_ws/src/cob_navigation/cob_navigation_config/robots/cob4/nav/costmap_common_params.yaml
/cob_ws/src/cob_navigation/cob_navigation_global/config/global_costmap_params.yaml
/cob_ws/src/cob_navigation/cob_navigation_global/config/local_costmap_params.yaml
/cob_ws/src/cob_navigation/cob_navigation_local/config/global_costmap_params.yaml
/cob_ws/src/cob_navigation/cob_navigation_local/config/local_costmap_params.yaml