Lidar Simulation for Robotic Application Development: Modeling and Evaluation

Abhijeet Tallavajhula
CMU-RI-TR-18-8

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics.

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
May 2018

Thesis committee
Alonzo Kelly, CMU RI (Chair)
Martial Hebert, CMU RI
Michael Kaess, CMU RI
Peter Corke, Queensland University of Technology

Abstract

Given the increase in scale and complexity of robotics, robot application development is challenging in the real world. It may be expensive, unsafe, or impractical to collect data, or test systems, in reality. Simulation provides an answer to these challenges. In simulation, data collection is relatively inexpensive, scenes can be procedurally generated, and state information is trivially available. Despite these benefits, the use of simulators is often limited to the early stages of application development. In this work, we take steps to close the gap between simulation and reality, for Lidar simulation.

We adopt the perspective that the eventual purpose of a simulator is a tool for robot application development. Our framework for sensor simulation consists of three components. The first is sensor modeling, which describes how a sensor interacts with a scene. The second is scene generation, needed to construct simulated worlds corresponding to reality. The third is simulator evaluation, based on comparing real and simulated data. We formalize the intuition that application performance must be similar in simulation and reality, using an application-level loss. Our framework is broadly applicable to simulating sensors other than Lidars.

We instantiate our framework for two domains. The first domain is planar Lidar simulation in indoor scenes. We construct a high-fidelity simulator using a parametric sensor model. We show how application development paths for our simulator are closer to reality, compared to a baseline. We also pose sensor modeling as a case of distribution regression, which leads to a novel application of a nonparametric method that adapts to trends in sensor data. The second domain is Lidar simulation in off-road scenes. Our approach is to build a library of terrain primitives, derived from real Lidar observations. These are shown to generalize, resulting in an expressive simulator for complex off-road scenes. For this domain as well, we quantitatively demonstrate that our simulator is better for application development, compared to a baseline.

Our work suggests a generic approach to building useful simulators. We view them as predictive models, and perform thorough tests on real data. We evaluate them with an application-level loss, which supports their greater use in the development cycle.

Acknowledgements

I first thank my advisor, Alonzo Kelly, for supporting me during the Ph.D. This work is in large part his vision. Al has valuable lessons to teach, and I hope I have learned them well. I am grateful to my committee members, Peter Corke, Martial Hebert, and Michael Kaess, for their time and comments. I am grateful to Barnabás Póczos for inputs on distribution regression. This work would have been incomplete without guidance from Çetin Meriçli, whose help I could always count on. I thank Brad Lisien for assistance in collecting data from Gascola. This research was supported by the National Science Foundation under Grant Number 1317803.

Interactions with the faculty of the Robotics Institute, and CMU, have provided many learning opportunities. I thank Sidd Srinivasa for the opportunity to come to RI. I am grateful to the staff of the Robotics Institute, and the OIE. I especially thank Suzanne, who looks out for students, and helped me navigate timelines. Finally, I would like to thank my family, and my friends. I am fortunate in that I would have enjoyed their support regardless of whether I was doing a Ph.D. or not.

Contents

Abstract i

Acknowledgements ii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Outline of approach . . . 7

2 Background 10
- 2.1 Sensor modeling . . . 10
- 2.2 Scene generation . . . 11
- 2.3 Simulator evaluation . . . 14
  - 2.3.1 Data-driven simulator training and test . . . 14
  - 2.3.2 Observation-level loss . . . 15
  - 2.3.3 Application development . . . 16
  - 2.3.4 Application-level loss . . . 18
- 2.4 Assumptions . . . 22
- 2.5 Related work . . . 23

3 Indoor planar Lidar simulation 26
- 3.1 Introduction . . . 26
- 3.2 Related work . . . 29
- 3.3 Statistics background . . . 31
  - 3.3.1 Density estimation . . . 31
  - 3.3.2 Regression . . . 32
  - 3.3.3 Distribution regression . . . 33
- 3.4 Approach . . . 34
  - 3.4.1 Real and simulated scenes . . . 34
  - 3.4.2 Sensor modeling . . . 36
    - 3.4.2.1 Parametric method . . . 37
    - 3.4.2.2 Nonparametric method . . . 39
  - 3.4.3 Simulator evaluation . . . 42
- 3.5 Experiments . . . 44
  - 3.5.1 Observation-level simulator evaluation . . . 44
  - 3.5.2 Application-level simulator evaluation . . . 47
  - 3.5.3 Magnetic field sensor . . . 56

4 Off-Road Lidar simulation **59**
4.1 Introduction . . . 59
4.2 Related work . . . 64
4.3 Approach . . . 67
4.3.1 Real and simulated scenes . . . 67
4.3.1.1 Ground segmentation . . . 69
4.3.1.2 Scene elements . . . 70
4.3.1.3 Scene segmentation and labeling . . . 74
4.3.1.4 Primitives and scene construction . . . 75
4.3.2 Simulator evaluation . . . 76
4.4 Experiments . . . 79
4.4.1 Observation-level simulator evaluation . . . 79
4.4.2 Application-level simulator evaluation . . . 86
4.4.3 Datasets generated . . . 89

5 Conclusion **90**
5.1 Summary of contributions . . . 90
5.2 Future work . . . 91
5.2.1 Nonparametric sensor modeling . . . 91
5.2.2 Off-road Lidar simulation . . . 92
5.2.3 Simulator evaluation . . . 95
5.3 General sensor simulation outline . . . 96

Chapter 1

Introduction

1.1 Motivation

The scale of robotics is increasing, evidenced by the growth in fields such as self-driving cars, and unmanned aerial vehicles. This is accompanied by an increasing challenge of developing software for robots. In fact, the difficulty of working with real robots can be experienced even in a simple lab setting. Testing robots in real-world conditions can be expensive, unsafe, or impractical. We illustrate these points using a few examples.

FIGURE 1.1: Simulation is useful in education, as an aid for students. Pictures are from the Mobile Robot Programming Lab course, CMU. On the left is a Neato robot, retrofitted as a fork-truck. On the right is a competition task between student groups.

The first example is related to education. At the Mobile Robot Programming Laboratory (MRPL)[1], a course at Carnegie Mellon University (CMU), students program all the basic building blocks of mobile robots. Working through a series of assignments, they learn about and implement position control, and trajectory planning. Students also use planar laser range sensors on the robots to perform localization and object detection.

[1] http://www.frc.ri.cmu.edu/~alonzo/teaching/16x62/16x62.html

The course culminates in a competition between groups. This example comes from first-hand knowledge, from our involvement in teaching the course. Despite the apparent simplicity, students face non-trivial problems when working with robots. To start with, the robots are a limited resource, which different groups must coordinate to share. Students quickly face the quirks of working with real data, such as sensor noise. In addition, deploying code-in-progress directly on robots can be unsafe. For example, we have observed that, when testing their Proportional-Integral-Derivative (PID) code, students might forget to reset integral error to 0. The result is a robot that accelerates suddenly, which can lead to damage. Not only students, but also maintainers of the course have dealt with hardware issues. For example, an intended motor upgrade to the robots malfunctioned, putting a number of them out of service. Labs were delayed be a week, an expensive amount of time during a semester.

FIGURE 1.2: Simulation is useful for outdoor mobile robots, where real-world tests are expensive. On the left is the Crusher robot. On the right is an unmanned ground vehicle from the Pegasus project. Both projects are from NREC, CMU.

Our second example is the domain of outdoor mobile robots. We refer to projects from the National Robotics Engineering Center (NREC), CMU [2]. Researchers at NREC develop software, such as perception for off-road navigation [1], and virtualized reality [2], for robots in challenging outdoor scenarios. Validating systems by testing in the real world can be very expensive in this domain. For projects with the Crusher [3] robot, which went on for 5 years, field tests were conducted every second week. An average of ten people accompanied each field test. In the worst case, the cost for a week was $\$250,000$. Unforeseen issues such as weather conditions may interrupt data collection plans. Hardware failure on large systems can delay testing, incurring further cost.

A third example of the challenge of working with real systems is in machine learning for robotics. Learning-based methods are widely used in robotics, especially in perception. The challenges may be illustrated by considering classification, a common supervised

[2] http://www.nrec.ri.cmu.edu/
[3] https://www.nrec.ri.cmu.edu/solutions/defense/other-projects/crusher.html

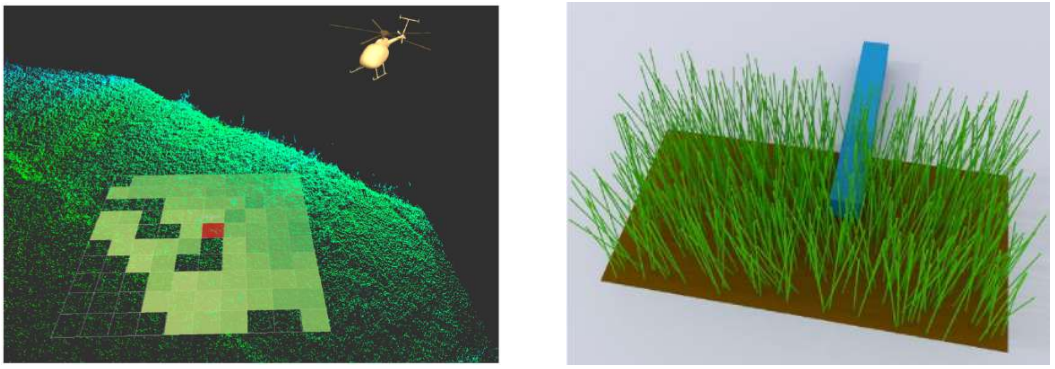


FIGURE 1.3: Simulation is useful for learning algorithms, where collecting and annotating large volumes of real data is impractical. Pictures are from the work on Maturana and Scherer [3] on helicopter landing zone detection. The picture on the left shows predictions from a CNN. The right depicts simulated grass used to generate synthetic training data.

learning problem. In the work of Maturana and Scherer [3], the classification problem was to detect landing zones from autonomous helicopters. A convolutional neural network (CNN) was used as the classification algorithm, and the input was Lidar range data. CNNs were chosen for their high performance, and fast predictions. Deep learning methods often require large amounts of training data. This was difficult to obtain in [3], as the source of real data was expensive flight tests. Even after obtaining real data, annotation of a large dataset can be daunting. Further, for multi-class classification, it may be impractical to arrange for sufficient diversity of classes in a real dataset. A related problem is domain adaptation. For example, assume that the classifier in [3] was trained in sunny weather conditions, but the landing detection was then required to work on a helicopter operating in a different season.

FIGURE 1.4: Some of the many modeling tools, and simulators.

Developing application algorithms for robot systems that operate in complex, real-world conditions is thus challenging. Methods which simplify the process of application development will thus be of wide-ranging impact to the robotics community. Simulation is one such solution. Simulated data is cheaper to obtain than real data. In a simulated environment, safety of systems and other hardware issues are not a concern. Simulation worlds can also be generated (up to representation accuracy) to specification, allowing tests in conditions that would be impractical to target in reality. Simulators for different

FIGURE 1.5: Examples of the recent interest in simulation. From left to right, the Gazebo simulator for the DARPA Robotics Challenge [4], simulation for visual tracking [5] using Unreal Engine 4, and AirSim for aerial robotics [6].

phenomena have been developed (Figure 1.4). These include physics (such as simulating contacts and friction between soft and rigid bodies), graphics (such simulating the interaction of light with different materials and textures), and various sensors (such as simulating GPS, IMUs, cameras). The components are further bundled into specialized packages, such as simulators for manipulation, aerial robots, and robot soccer, to name a few. The domain of this work is range sensor simulation. While we mostly refer to and work with Lidars, the techniques and language we use in this thesis are broadly applicable to simulating other sensors. The motivation for focusing on Lidars is that they are a key technology that have enabled advances in outdoor robots. They are often used to provide a three-dimensional picture of the world to the robot. Applications are routinely developed to process Lidar data, such as algorithms for scan matching, object detection [7], and mapping [8]. Applications must deal with the noise and large scale of Lidar data. They must be robust and have high performance, qualities which are essential for good decision-making in the higher levels of autonomy. Lidar simulation has been, or could be, used to address the challenges in each of the real-world anecdotes mentioned above, as we briefly discuss next.

We built a simulator for the MRPL course mentioned earlier (Figure 1.1), which proved useful in a number of ways. Working in simulation allowed students to make progress even when shared real robots were occupied. Prototyping software in simulation added a layer of safety. With regards to the mobile robots example, a Lidar simulator was used during the Pegasus project at NREC [4], during the design of an unmanned ground vehicle, see Figure 1.2. The decision of where to place a Lidar sensor was arrived at by running simulations. This saved time for the hardware team, which would otherwise have had to test different sensor configurations in reality. In the CNN-based landing zone detection work of Maturana and Scherer [3], the authors used synthetic training data in the absence of sufficient real data, see Figure 1.3. This use of a Lidar simulator enabled the selection of architectures and hyperparameters and for the CNN. Synthetic data is attractive for machine learning, since a large amount of data may be generated

[4] https://www.cmu.edu/news/stories/archives/2014/june/june3_robotteam.html

procedurally. Since state information is trivially available in simulation, annotations can be generated automatically. Other biases with real datasets, such as class imbalances and inadequate diversity, can be overcome in simulation, since desired worlds can be directly specified.

Simulation is closely tied with application development in robotics. In this thesis, by application we refer to an algorithm to be deployed on the robot to perform some task. By development we refer to the process of tuning and optimizing the algorithm. Simulation is beneficial for application development in a number of ways, in addition to those mentioned above. First, simulation is repeatable. The same tests can be run repeatedly, even as an application algorithm is modified. Achieving this in reality is difficult. Second, desired worlds as test environments can be be specified in simulation. For example, we might want to test perception algorithms in a dense forest. Locating and scheduling such a test in reality may be expensive, but is easier in simulation. Third, state information can often be trivially obtained in simulation. The same information may not be available in reality. This allows simulated testing of applications in ways, and using metrics, not possible otherwise. For example, consider the application of visually tracking a target. The tracking algorithm may internally estimate the velocity of a target. In reality, the state of the target may not be known, and a metric such as bounding box overlap may be used to evaluate the tracking algorithm. In simulation, however, the target state will be known. This information can be used to compute additional metrics for the algorithm. It may lead to insights, such as poor performance because of errors in target velocity estimation.

In fact, simulators are already part of the application development timeline in robotics. However, their role may be limited. Simulators may be used in the early stages, to test initial implementations, and catch obvious bugs. As development proceeds and an application matures, simulation may be set aside. Testing in reality may be favored. Why is this the case, given the benefits of simulation outlined? Our answer is that a useful simulator has three components: sensor models, scene construction, and simulator evaluation. The sensor models must be high-fidelity. Scenes constructed in simulation must be realistic, or more strictly, must correspond to real scenes. Evaluation of simulators must be thorough, and meaningful in the context of robot application development. A lack of any of these leads to a simulator of limited value. We discuss these, once again, in the context of the simulation anecdotes.

When developing a Lidar simulator for the MRPL course, we found that a vanilla ray-tracing sensor model was not good enough. Students preferred to work with real robots, because the simulated data was too clean. As an example, consider the situation in Figure 1.6, which is similar to one of the course assignments. The task is to detect a

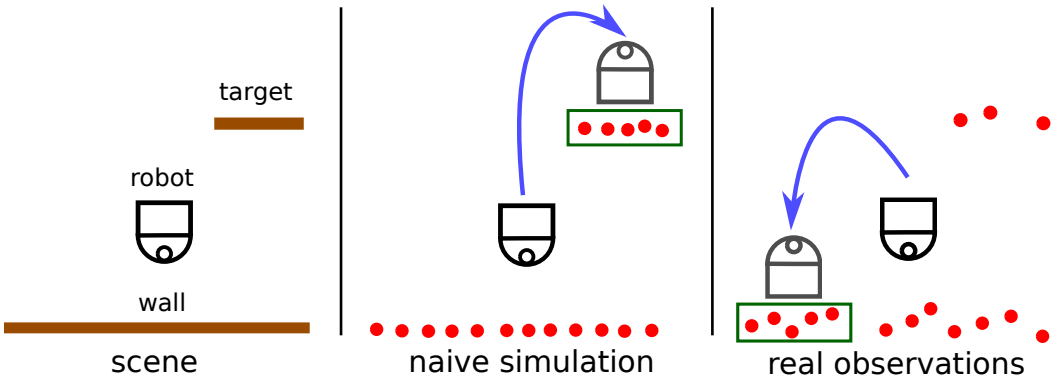


FIGURE 1.6: A detection task. The first panel shows the scene. The target is to be detected from range data. The second panel shows range data form a naive simulator. In this data, it is easy to locate and navigate to the target. The third panel shows the real range data, which is noisy and has missing data. The same algorithm would find a false positive, and cause the robot to drive into the wall.

target, which the robot has to approach and pick up. Behind the robot is a wall. In a naive simulation, developing software to detect the target, while ignoring sensor data from the wall, is simple. In reality, there are breaks in the sensor data. A detector developed on the naive simulator might detect a false positive, and the robot drives straight into the wall. While the aim of simulation is to ease the burden of working with reality, this should not come at the cost of simplifying real data. Because the real Lidar sensor was noisier than the vanilla simulator, applications developed by students which performed well in simulation would degrade in reality. They switched to working with real data once simulation stopped being a useful source of feedback. This observation led us to use high-fidelity sensor models in our simulator.

We mentioned in the outdoor mobile robot anecdote (Figure 1.2) that a Lidar simulator was used to decide the placement of a sensor. While useful, the drawback with the simulator in that case was that it was 'geo-specific'. This was because the Lidar simulator was constructed by mapping a real site. Lidar data could be simulated in only one simulated scene, which was the mapped version of the real scene. While simulated data was high-fidelity, the simulator was inexpressive. We mentioned earlier that simulators are useful because they allow the specification of desired worlds. But this ability is limited by the expressiveness of the simulator. Outdoor missions might require testing in a number of real environments. The ability to support the creation of simulated scenes corresponding to complex real world scenes is important for simulation.

Finally, in the work of Maturana and Scherer [3] on autonomous landing zone detection (Figure 1.3), a problem noted was Lidar simulator evaluation. First, it was difficult to create simulated worlds corresponding to real landing zones. An expressive simulator,

where the real world can be reconstructed, is thus important even for simulator simulation. Second, it was observed that the performance of algorithms in simulation was an optimistic estimate of performance in reality. Admittedly, simulation was not the centerpiece of [3]. Even in other projects developing applications for autonomous systems, simulation may not be the focus. But unless simulators are evaluated in a principled way, for the purpose they are intended for, trust in simulation will remain low.

1.2 Outline of approach

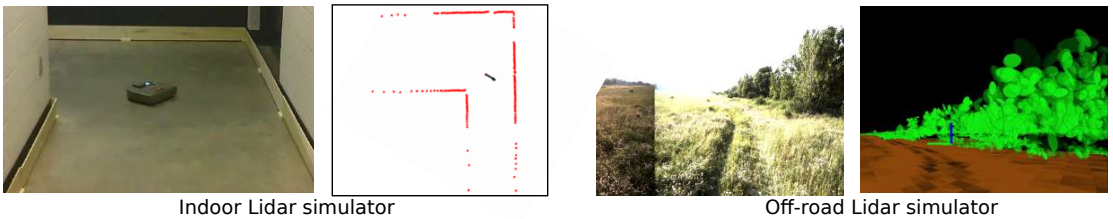


FIGURE 1.7: Real and simulated scenes from our work. The left figure depicts the indoor Lidar simulator. The right figure depicts the off-road Lidar simulator

The potential benefits of simulation, and the drawbacks of current approaches, are motivations to build better Lidar simulators. At a high level, our approach is to consider each component of simulation individually: sensor models, scene generation, and simulator evaluation. We instantiate the high-level approach in the construction of two Lidar simulators. The first simulator is for a planar Lidar sensor operating in indoor environments. The second simulator is for a 3D Lidar sensor operating in off-road environments. In either case, we demonstrate an improvement over existing simulation methods. The outline of the thesis document is as follows.

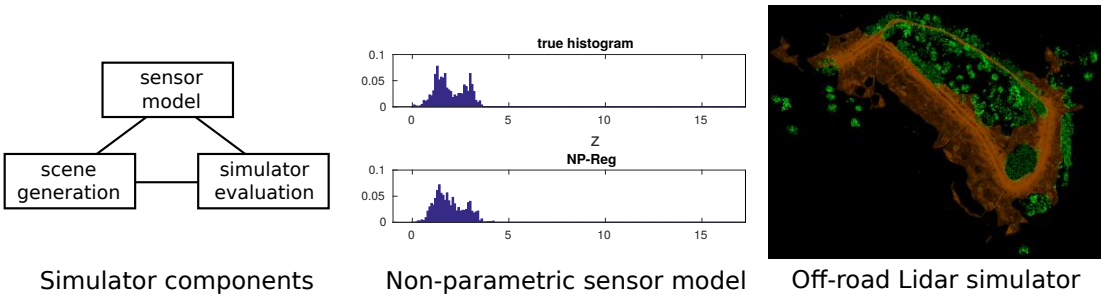


FIGURE 1.8: Our work includes a formulation for general sensor simulation (Chapter 2), non-parametric sensor models (Chapter 3), and complex off-road scene generation (Chapter 4).

- Chapter 2 contains the terminology used in this thesis. It is the common language which we use to speak about both indoor and off-road Lidar simulation. We describe our notation and mention assumptions. We approach evaluation in a

data-driven manner, and show how this approach requires the construction of realistic scenes in simulation. We specify what is meant by application development, and note the difference between observation-level and application-level simulator evaluation. The benefit of introducing our terminology is that past approaches to evaluating simulators can be discussed in common terms. We can view them as specific cases of our more general approach. While our work is in Lidar simulation, we believe that our high-level approach is broadly applicable. It can be used to guide the construction of any sensor simulator.

Note that there are other aspects of simulators: their software architecture, communication between modules, user interfaces, and others. While important, these are not the focus of this thesis.

- Chapter 3 details the simulation of a planar Lidar in indoor scenes. The motivation was a classroom simulator. The simulated scenes in this case are simple, and the contributions are in the sensor model. We first use a specific parametric model which is able to reproduce quirks in real sensor data. Then, in response to the question of whether it is possible to simulate sensors directly from raw data, we reply in the positive with a completely non-parametric approach. This is a novel application of a recently developed non-parametric regression method to modeling sensors. We then perform application-level simulator evaluation. We demonstrate the approach via two fundamental mobile robot tasks: object detection and state estimation.

  The non-parametric regression method we use is conceptually simple. It is efficient to apply to modeling any sensor with unidimensional output. We demonstrate this by modeling a probe used in medical applications.

- Chapter 4 details the simulation of a 3D Lidar in off-road scenes. The motivation was off-road autonomy missions. The sensor model in this case is consists of hybrid geometric elements. It is a representation well-suited to simulating Lidars interacting with off-road terrain, such as vegetation. Our contribution is the introduction of expressiveness to such a simulator. We extract terrain primitives from training data, which results in a primitive library optimized for simulation. The primitives can then be used to construct new simulated scenes. We evaluate the simulator in complex, real-world scenes of interest. In this case, we perform application-level evaluation for scan matching.

  Our approach can also be used to simulate arbitrary scenes, and to efficiently map off-road sites. We have generated labeled datasets in the course of our work on simulation. These may be of independent interest, e.g. for Lidar semantic segmentation and object detection.

- Chapter 5 concludes the thesis. We highlight lessons learned and summarize the contributions. We list short-term directions for improving the simulators constructed in this thesis. We also discuss long-term paths for future work.

Discussion of related work is included in each chapter. For work on the individual simulators, we also include extra notation where necessary.

Chapter 2

Background

This chapter contains the terminology used in this thesis. We describe our notation and mention assumptions. Broadly, we view a Lidar simulator as a predictive model, consisting of three parts: a sensor model, a scene generator, and an evaluation method.

2.1 Sensor modeling

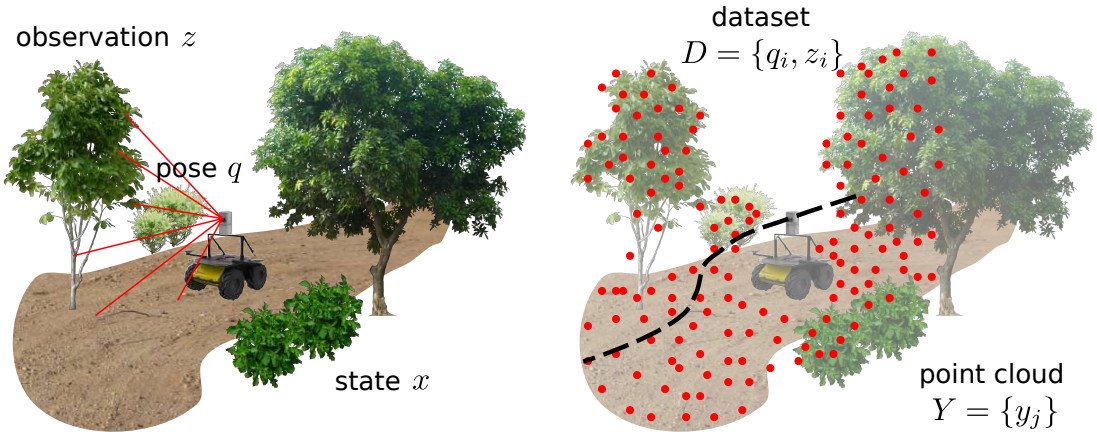


FIGURE 2.1: Some of the notation illustrated for a Lidar sensor in an off-road environment. On the right, the dotted line represents the path taken by the vehicle. Information gathered along the path is the dataset.

The state of the world in reality is denoted by x. We consider all information that affects sensor observations as part of the state. This includes the pose of the sensor, q_{sensor}, and the sensor's environment. The state space is denoted by $\mathcal{X}$. The sensor observation is denoted by z. The observation can be a vector. For a Lidar sensor, the observation is a set of range returns $z = \{z^j\}$, where the subscript j denotes coordinates of the observation vector. The returns in an observation may correspond to a sensor-specific

scanning (or firing) pattern. A return is just a real number in the case of a hit, and a 'nothing' value in the case of a miss. Sensor observations are typically noisy, so no two observations at a particular state will be equal. We take this into account by saying that the true behavior of a sensor is represented by the observation distribution, $p(z|x)$, which specifies the probability (density) of an observation at a particular state.

It is unlikely that the real world will be recreated exactly in a simulator. Therefore, we denote the world state in simulation by $\hat{x}$. The corresponding space of simulated world states is $\hat{\mathcal{X}}$. Simulated observations are denoted by $\hat{z}$. A sensor model is a model of the true observation distribution, and denoted by $\hat{p}(z|\hat{x})$. Note that the sensor model depends on the simulator state. A simulator does not have access to the real state x. Simulated observations are samples drawn from the sensor model, $\hat{z} \sim \hat{p}(z|\hat{x})$. The sensor model can be deterministic, as in the case of a simple ray-trace simulator.

The sensor model may have parameters which can be tuned to optimize the model. In this work, we consider data-driven sensor models, which are optimized on real data. This requires a dataset D, which is a set of state-observation pairs $D = \{(x_i, z_i)\}$. In practice, we might only have access to a part of the real state. We overload the dataset notation and make its contents clear, where required. For example, in the off-road case, the dataset is a set of sensor pose-observation pairs, $D = \{(q_{\text{sensor},i}, z_i)\}$. Since lidar observations are naturally viewed as points in space, we will often convert a set of observations into a point cloud. A range return that results in a hit, when transformed into the world frame, using the sensor pose q_{sensor}, results in a point denoted by $y \in \mathbb{R}^3$. A set of points in the world frame, often accumulated over multiple observations, results in a point cloud denoted by $Y = \{y_j\}$.

2.2 Scene generation

By scene generation, we refer to the construction of simulated worlds in which to query sensor observations. Our reasons for paying attention to scene generation require a notion of simulator evaluation. Evaluation is the subject of Section 2.3 below. For now, however, we work with the understanding that a simulator is optimized by matching simulated sensor observations to real observations. First, consider the simplest method of selecting a sensor model, classic calibration. In calibration, the sensor is placed in a controlled setup. For example, a range sensor may be placed on a measuring table, pointing at a plain target such as a flat surface (Figure 2.2, top). The distance to the target, which is also the state x, can be accurately measured. Observations are logged at a state, and the data collection is repeated at different states, resulting in a dataset of state-observation pairs, $D = \{x_i, z_i\}$. The simulated state is simply the distance to the

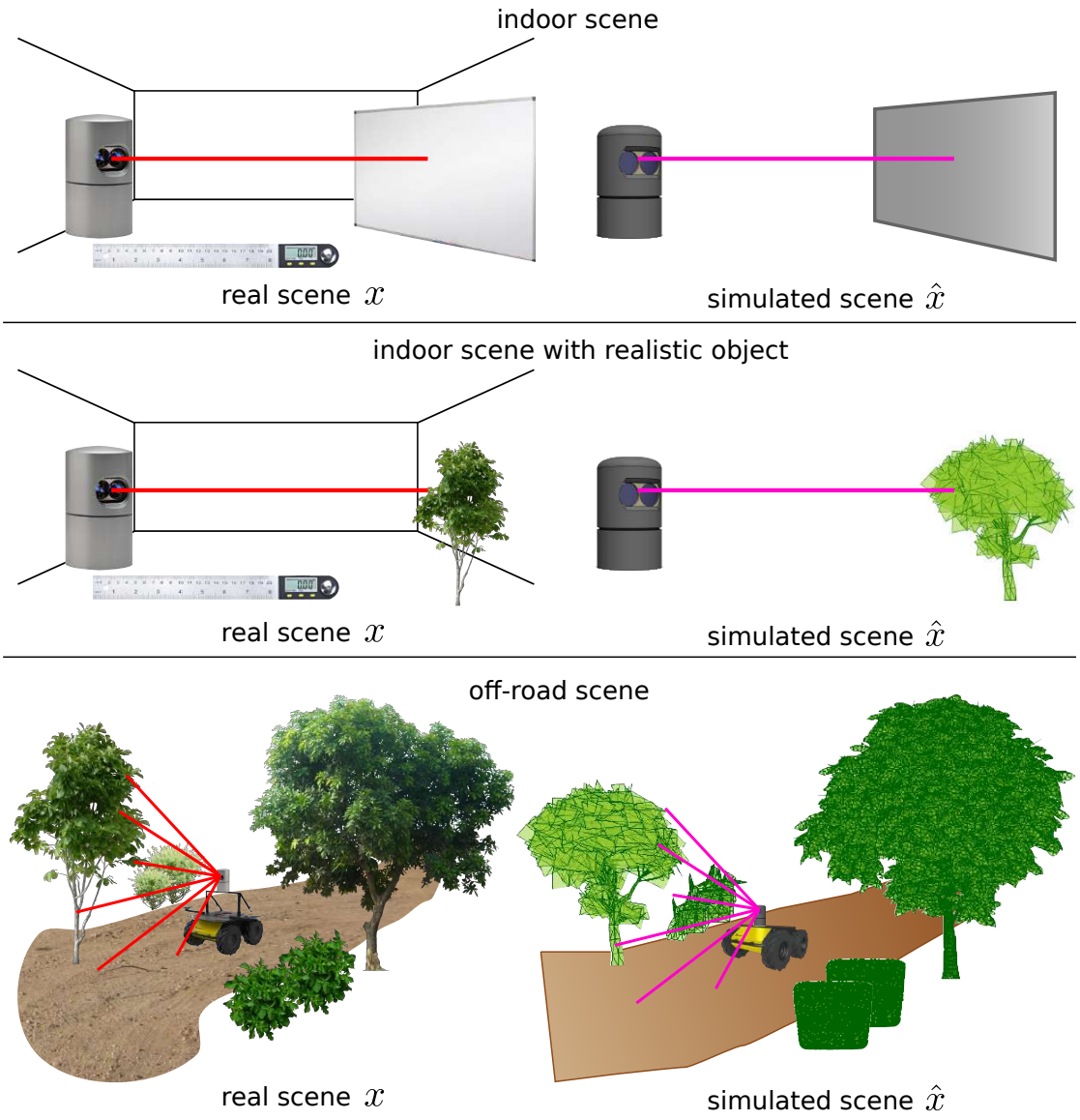


FIGURE 2.2: The progression of simulator training from laboratory indoor scenes to complex off-road scenes.

target, which has been measured in reality, $\hat{x}_i = x_i$. The simulator in this case is simply the sensor model, $\hat{p}(z|\hat{x})$. Matching simulated observations $\hat{z}_i$ to real observations z_i leads to an optimized sensor model.

Of course, our sensor's operating condition may not be a sanitized laboratory. Suppose we want to construct an off-road Lidar simulator. The objects that the Lidar looks at might more likely be uneven ground and vegetation. To this end, we could replace the flat surface in the calibration setup with a real plant (Figure 2.2, middle). To generate corresponding simulated observations, the simulated scene would now have to contain a simulated plant. The simple distance-to-target state description is not enough. Scene generation is more difficult in this case. The simulated plant may not match the real plant in terms of position, shape, and so on.

After selecting a sensor model with the sample plant, the next step would be to place the sensor on a robot and take it outdoors, into the type of real environment that we eventually want to simulate in (Figure 2.2, bottom). It is important that we take this step forward to train simulators in complex scenes. Otherwise, the simulator may be dismissed as good only for toy worlds, leading to a gap between simulation and reality. The problem of scene generation is immediately apparent here. To simulate observations that can be compared with real data, we have to reconstruct a complex real scene in simulation. Note the difference between realistic and reconstructed simulated scenes. A realistic simulated scene does not correspond to a real scene. Although it may be similar (according to some metric) to the real world. This may be a common use case for a simulator: hallucinating worlds to test applications in. A reconstructed simulated scene is an analogue of a real scene. The latter is more challenging, but the payoff is a principled evaluation of the simulator. Hence the connection between evaluation and scene generation. Once a simulator has been validated for reconstructed scenes, its use in realistic scenes is justified.

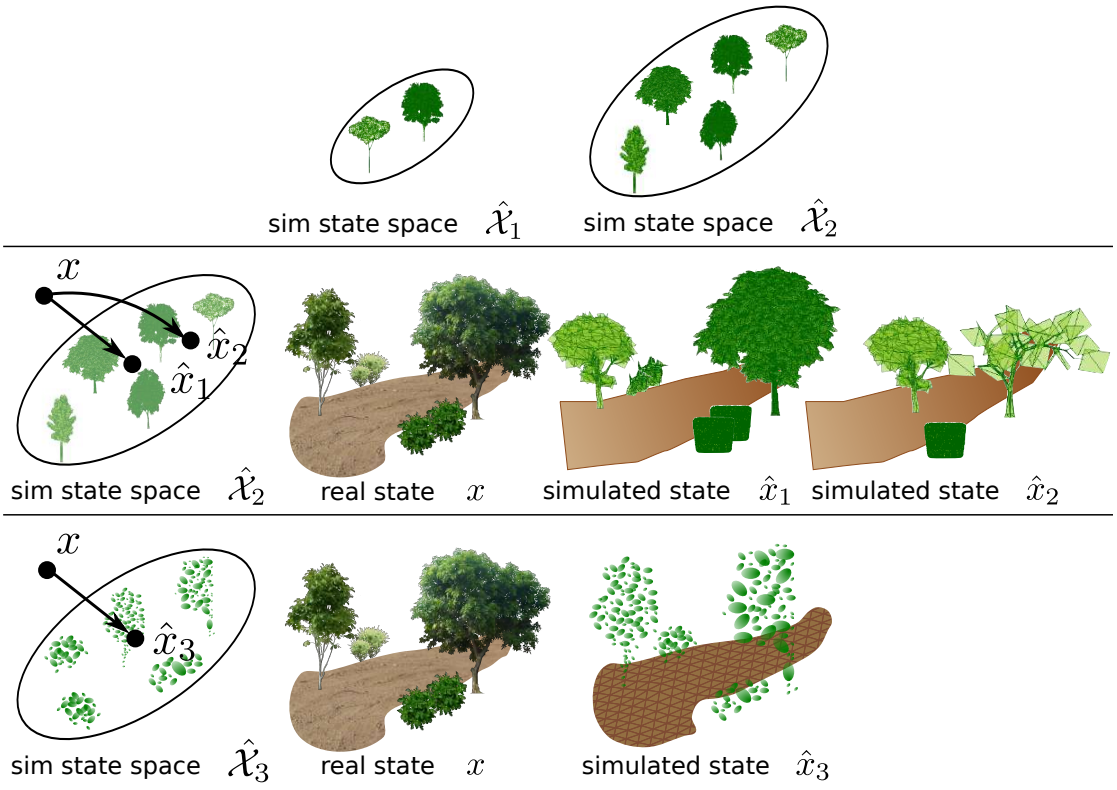


FIGURE 2.3: Generating a simulated world state corresponding to a real world state can be thought of as a projection onto the simulator state space.

The space of simulated states is $\hat{\mathcal{X}}$. We can informally interpret the 'size of $\hat{\mathcal{X}}$' as the expressiveness of the simulator. For example, suppose that the library of objects in simulator 2 is a superset of the library in simulator 1 (Figure 2.3, top). Simulator 2 is a more expressive simulator, since more diverse real scenes can be reconstructed in

simulation . Given a real state x and some simulator with space, $\hat{\mathcal{X}}$, however, we still need to map x to a simulated scene $\hat{x}$. This can be thought of as a projection operation, that is (i.e.), selecting the simulated scene closest to the real scene. For simulator 2, a good scene projection leads to simulated scene with state $\hat{x}_1$ (Figure 2.3, middle). A poorer scene projection results in a scene with state $\hat{x}_2$, and $\hat{x}_1$ is a better reconstruction of $\hat{x}$. Figure 2.3, bottom, depicts a scene generation in a different simulator. While simulator scene space $\hat{\mathcal{X}}_2$ uses mesh objects, $\hat{\mathcal{X}}_3$ uses Gaussian distributions to model objects. In our work, the projection step is manual. It can, however, be automated using well-known Lidar processing tools, as discussed in Section 5.2.2. Details of the scene representation chosen for our indoor and off-road Lidar simulators are in Chapters 3 and 4, respectively.

2.3 Simulator evaluation

In Section 2.2 we said that a simulator is evaluated by comparing simulated data with real data. Borrowing language from learning theory [9], we compare them using a quantity called the loss l. The loss, averaged over a dataset, is called the empirical risk $\hat{R}$. By simulator evaluation, we formally mean the calculation of the empirical risk over a dataset. We first discuss simulator training, and then different choices for the simulation loss.

2.3.1 Data-driven simulator training and test

For evaluation, we first collect a dataset in a real scene, $D = \{x_i, z_i\}$. Although we write the real dataset as containing the real state, typically only partial state information will be available. We generate a simulated scene corresponding to reality. It is assumed that the real dataset contains the sensor pose information, $q_{\text{sensor},i}$. We query observations at the sensor poses, to generate a corresponding simulated dataset, $\hat{D} = \{\hat{x}_i, \hat{z}_i\}$. Writing the simulated dataset as containing the full simulated state is exact. The simulator can then be evaluated by comparing the real and simulated data, using the simulation risk.

The simulator typically has parameters θ. These could include parameters for the sensor model and the scene generation. For simulator training, a training dataset is collected. The parameters θ are then tuned to reduce the risk on the training dataset, resulting in an optimized simulator, say with parameters θ^*. The evaluation process is then repeated on a separate test dataset. The simulator is not to be tuned on the test dataset. The empirical risk on the test dataset is only calculated for a single set of parameters θ^*, allowing it to serve as an unbiased estimate of the simulator's performance.

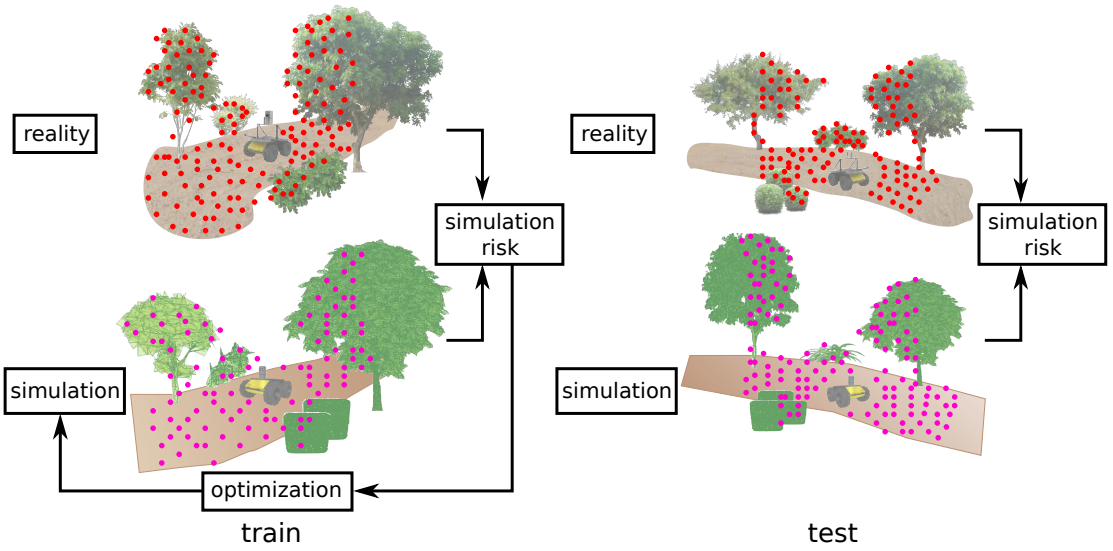


FIGURE 2.4: Real data is compared with simulated data using the simulation risk. The simulator is optimized to minimize the risk during training. The risk is calculated on a test scene for an unbiased estimate of simulator performance.

2.3.2 Observation-level loss

An observation-level loss compares simulated and real data at the level of raw observations. For a Lidar sensor, where the observation is a vector of range returns, $z = \{z^j\}$, we can define the loss and risk to be

$$l(z, \hat{z}) = \frac{1}{M} \sum_{j=1}^{M} |z^j - \hat{z}^j|, \; R = \mathbb{E}\, l(z, \hat{z}), \tag{2.1}$$

where M is the number of returns in an observation. In practice, a modification to the loss may be necessary to account for the hits and misses among the range returns. The risk is an expectation of the loss, calculated with respect to some distribution of states and observations $p(x, z)$. For a dataset of size N, with data sampled from the distribution $p(x, z)$, we calculate the empirical risk

$$\hat{R} = \frac{1}{N} \sum_{i=1}^{N} l(z_i, \hat{z}_i). \tag{2.2}$$

The simulator training procedure can be rewritten as $\min_\theta \hat{R}(\theta)$. The averaging step in the risk means that, even if risk were low, the simulator may infrequently simulate observations that do not agree with reality. Where infrequently is with respect to the distribution $p(x, z)$. The risk may be defined as the worst-case loss, $R = \max l(z, \hat{z})$, with the maximum taken over all real observations. However, this quantity is impractical to work with. Defining the risk as an average makes it tractable to compute. It also

allows the provision of guarantees under the framework of statistical learning theory [9]. In practice we do not deal with the distribution $p(x, z)$ directly. It is implicit in the data collection procedure. The kind of generalization we expect from the simulator is specified through the distribution. For example, we may want to build a Lidar weather simulator, with high-fidelity simulation of Lidar data across conditions such as clear skies, snow, and rain. Our target distribution would then be over scenes with varying weather conditions. It must be kept in mind that low risk on one distribution does not directly imply good simulator performance on another distribution. These are standard caveats with any learning-based method.

2.3.3 Application development

Informally, a low observation-level risk means that, on average, simulated data agrees with reality. However, in robotics, the aim of simulation is not just the generation of sensor-realistic observations. A simulator is a tool used to develop an application to be deployed on a real system. Intuitively, a useful simulator is one in which application development proceeds similarly to how it would in reality. In this section, we specify what is meant by application development.

An application algorithm is denoted by A. When clear, we will use the terms application and algorithm interchangeably. An algorithm A is associated with tunable parameters α. These are unrelated to the simulator parameters θ. The performance of the algorithm is measured using an algorithm objective J, with a low objective indicating high performance. The algorithm objective $J(\alpha, D)$ is calculated for a set of algorithm parameters α, over a dataset D. As an illustration, consider 2D localization as a hypothetical application, and the algorithm A to be scan matching (Figure 2.5, top). Lidar range data is to be matched to a map of the environment. The range data is depicted as red points, and the map is depicted as blue lines. The solid arrow represents the true pose of the robot. The output of the algorithm, the estimated pose after scan matching, is depicted as a dashed arrow. The Lidar data is transformed to points based on the estimated pose. Real sensor noise is exaggerated for illustration. For simplicity, the algorithm has a single parameter α, the maximum inlier distance to the map. The closest distance of Lidar points to the map is computed, in the input robot frame. If this distance exceeds α, then the point is marked as an outlier and not considered for scan matching. Each instance i in the dataset D is a scan matching problem, consisting of a map, the initial pose, and the ground-truth pose of the robot, say q_i. Denote the output of the scan matching algorithm by q_i'. The ground-truth pose is not input to the scan matching algorithm. It is only used to compute the error of localization. The algorithm objective can be chosen to be the following, where the dependence of J on the algorithm

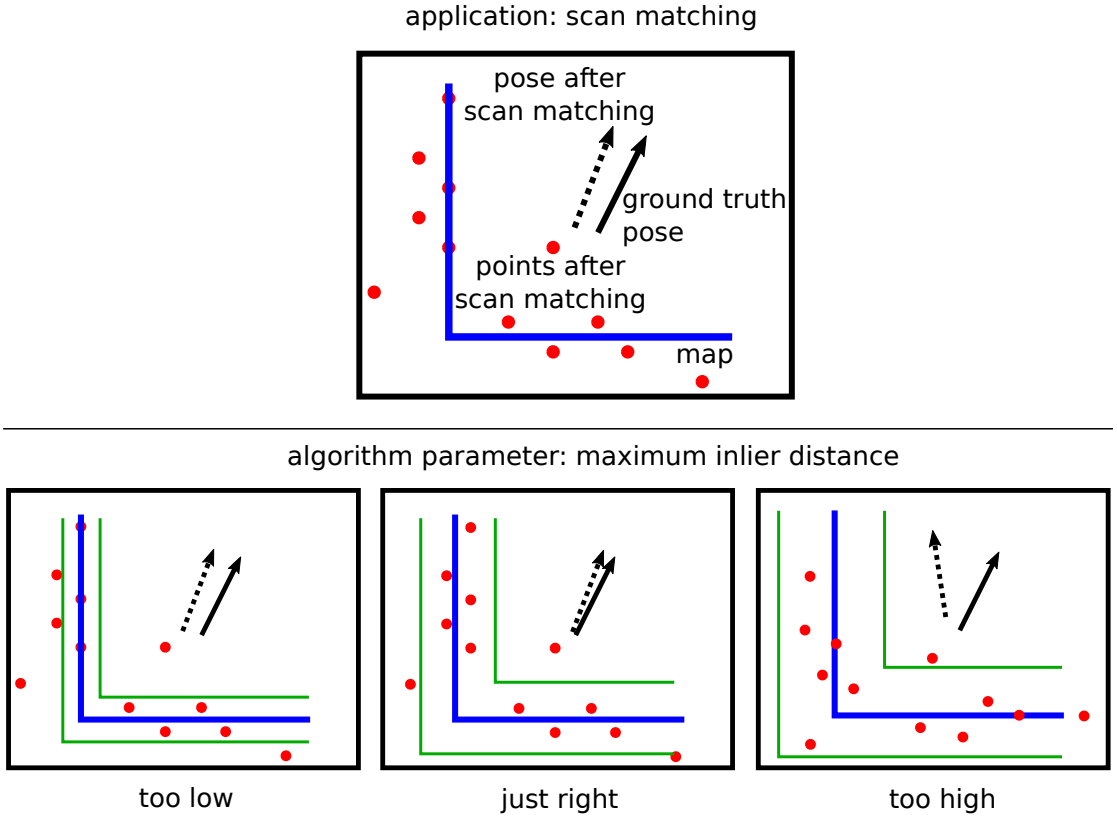


FIGURE 2.5: A hypothetical scan matching application to illustrate application-level loss. There is one parameter, the maximum inlier distance, which is to be tuned on data.

parameters α is implicit,

$$J(\alpha, D) = \frac{1}{N} \sum_{i=1}^{N} \|q_i - q_i'\|.$$

We can also choose the performance to take into account other metrics of interest, such as the number of times a particular bug (such as a system crash) occurs. By application development, we refer to tuning the algorithm parameters to minimize the objective (or maximize performance). For example, for the scan matching example, the algorithm parameter α is the maximum inlier distance. A suitable value for this parameter is not known *a priori*. In an ideal world all points would be inliers, but in reality, outliers will exist due to sensor noise, unmapped objects in the world, and other reasons. The optimal value for α will require a process of development. Given dataset D, we write the iterations of application development as $\{(\alpha_i, J_i)\}$. We attempt to improve the performance of the algorithm in successive development iterations, as depicted for the scan matching example in Figure 2.6. The dataset consists of a number of maps and scans. Initially, the algorithm performance is poor, as depicted by the estimated poses (dashed

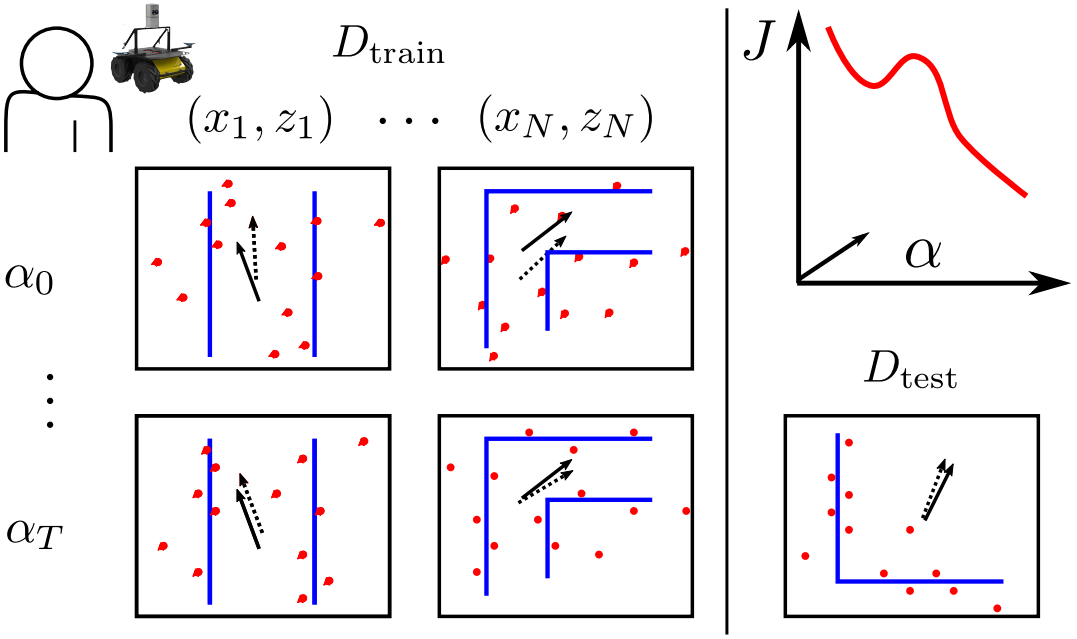


FIGURE 2.6: Application development is the process of tuning algorithm parameters α to minimize the algorithm objective J on a dataset. The algorithm performance may finally be calculated on a test dataset.

arrows) being far from the true poses (solid arrows). As development proceeds, the estimated poses line up with the true poses. At the termination of algorithm development, the algorithm is ready to be deployed on the robot. We denote this terminal point as (α^*, J^*), where $J_i = J(\alpha_i, D)$, and $J^* = J(\alpha^*, D)$. Application development can thus be viewed as tracing a path in the space of algorithm parameters and objective, (α, J). The algorithm objective may be finally calculated on a test dataset, serving as an unbiased estimate of the algorithm performance.

Application development is agnostic to the source of data. For the scan matching example, assume we had access to a Lidar simulator. When it is difficult to collect real data, application development may first occur in simulation. The illustration for the scan matching example, Figure 2.7, is analogous to development in reality, Figure 2.6. This results in a development path, $\{(\hat{\alpha}_i, \hat{J}_i)\}$, where $\hat{J}_i = J(\hat{\alpha}_i, \hat{D})$.

2.3.4 Application-level loss

Our application-level loss connects the evaluation of a simulator to its purpose, and is intended to be a meaningful evaluation of the usefulness of a simulator. The algorithm objective $J(\alpha, D)$ defined in Section 2.3.3 is agnostic to the data source. Now, for simulator evaluation, we are given a real dataset D, for which we simulate a corresponding dataset $\hat{D}$. For the same algorithm parameters α, we can compute the algorithm objective both in reality and simulation. These are $J(\alpha, D)$ and $J(\alpha, \hat{D})$, respectively.

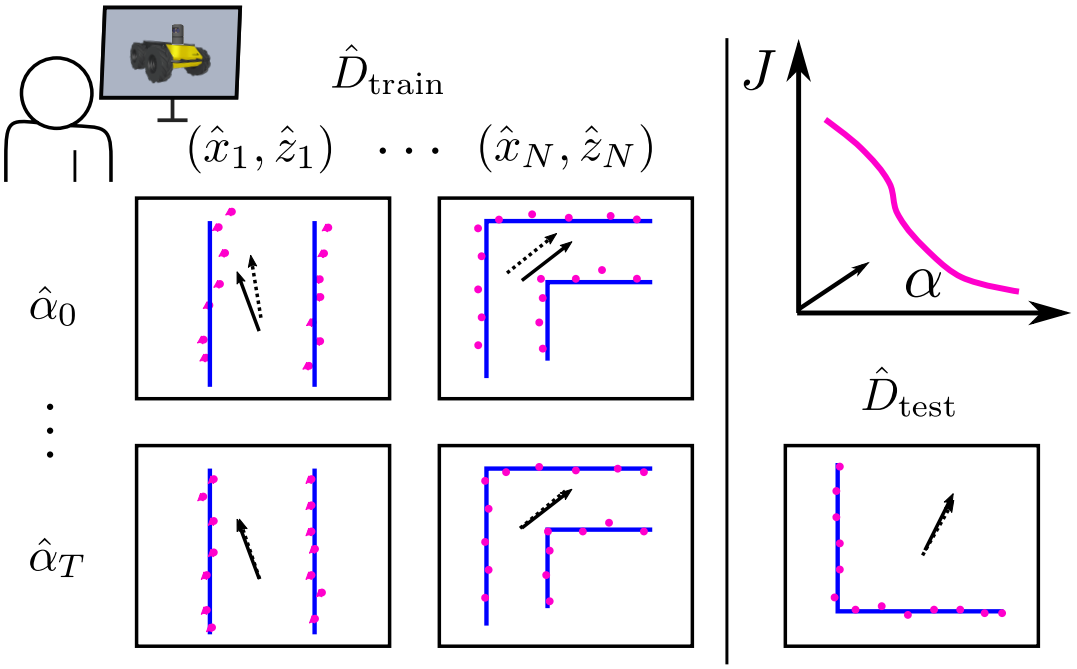


FIGURE 2.7: Application development for the same scan matching algorithm, but in simulation.

Consider a simulation loss defined as

$$l(\alpha) = |J(\alpha, D) - J(\alpha, \hat{D})|. \tag{2.3}$$

This application-level loss can be thought to measure the gap between simulation and reality, from the perspective of the application. However, this loss is a function of the algorithm parameters. Since these are not fixed, it leads to the question of which parameters to calculate the loss for. Answers are suggested by the process of application development.

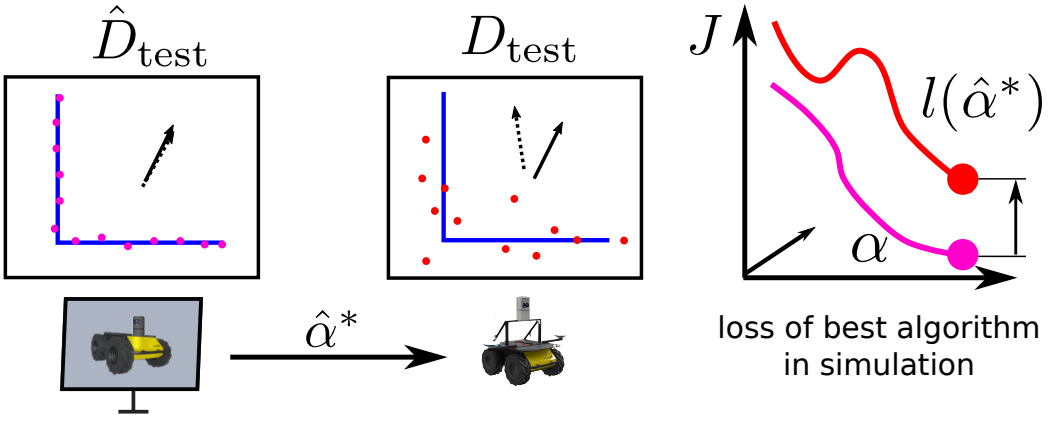


FIGURE 2.8: An application algorithm trained in simulation may degrade when transferred to reality. The difference can be quantified by the simulation loss.

Say the process of application development in simulation resulted in the path $\{(\hat{\alpha}_i, \hat{J}_i)\}$. If we only cared about the best performing algorithm, we would compute the loss for $\hat{\alpha}^*$, $|J(\hat{\alpha}^*, D) - J(\hat{\alpha}^*, \hat{D})|$. Further, consider that the Lidar simulator did not capture the non-idealities of the real data, which are high variance in range readings, sensor dropouts, and outliers. It is often observed that the performance of an algorithm degrades when it is transferred to reality. The loss for $\hat{\alpha}^*$ enables calculation of this performance gap during transfer. For the scan matching example, since the simulated data is relatively 'clean', increasing the maximum inlier distance improves performance. Lower objectives are achieved in simulation than in reality, and the performance gap is depicted in Figure 2.8.

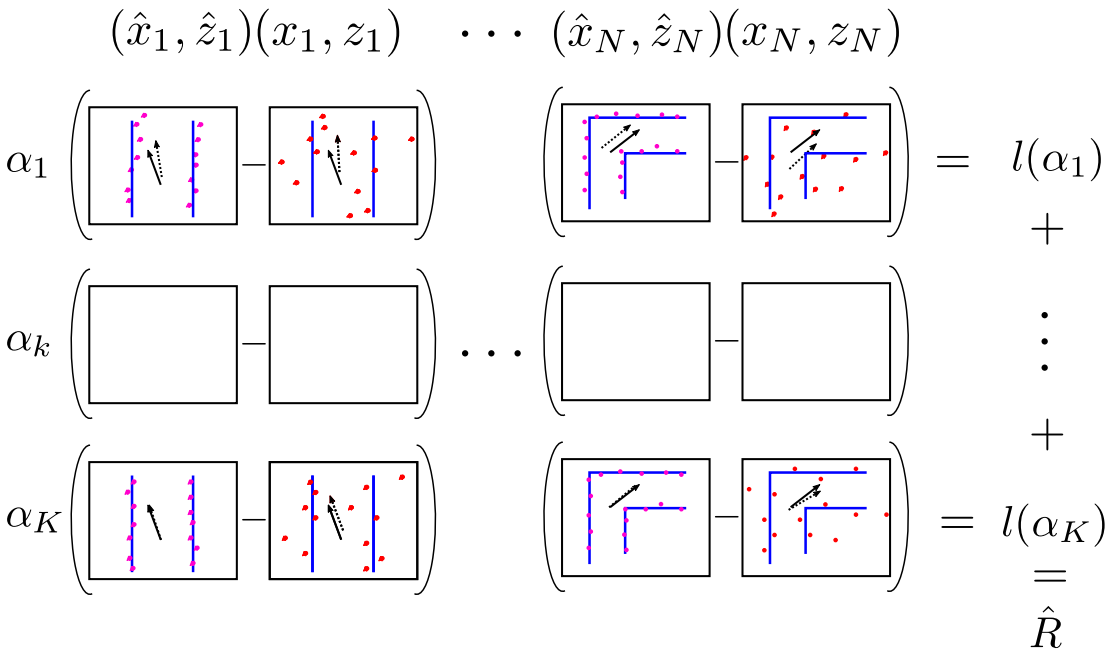


FIGURE 2.9: Illustration of the application-level risk. Each row is a particular set of algorithm parameters. Each column represents a real scene and corresponding simulated scene. Each bracket entry is the difference in performance between simulation and reality. The (empirical) simulation risk is the sum of losses over the α_k sampled from distribution $p(\alpha)$.

In cases where we are concerned only with the best performing algorithm parameters, application development is simply an optimization. In these cases, we can take the simulation risk to be $l(\hat{\alpha}^*)$. However, we can also consider cases where the losses at parameters other than $\hat{\alpha}^*$ are of interest. For example, suppose the scan matching algorithm was under development by students in a classroom setting. They may manually explore the parameter space in order to understand the effects of different parameters on the algorithm performance. In application development, roadblocks encountered when working with real data can provide useful insight into a system's operation. We would like simulation to be useful in this broader sense: it should challenge application development in a manner similar to reality, resulting in similar bugs, and leading to similar

design decisions. We could take the simulation risk as the maximum loss over the space of algorithm parameters, but this quantity is not tractable to compute. Instead, we define the empirical risk as a mean over parameters $\{\alpha_k\}$, $k = 1 : K$ drawn independently from a distribution $p(\alpha)$,

$$R = \frac{1}{K}\mathbb{E}_{p(\alpha)} l(\alpha), \tag{2.4}$$

$$\hat{R} = \frac{1}{K}\sum_{i=1}^{K} l(\alpha_i). \tag{2.5}$$

For the scan matching example the risk is visualized in Figure 2.9. Informally, a low application-level risk means that, on average, algorithm performance on simulated data agrees with reality. We would expect the simulator to provide similar feedback as reality, even when algorithm performance is poor. This definition of the risk does not depend on a particular algorithm development path.

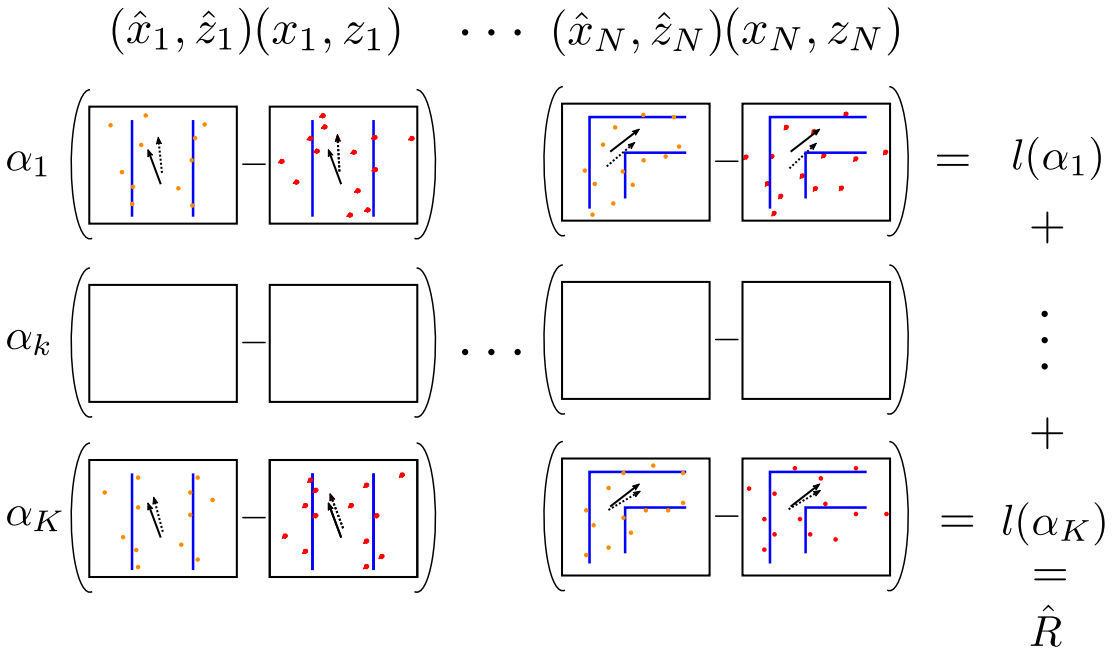


FIGURE 2.10: Upto now we considered a single simulator. We can calculate the application-level simulation risk in a similar manner for another Lidar simulator. This second Lidar simulator is better at modeling the real sensor noise.

The simulator risk gives us a precise way of comparing two simulators. For the scan matching example, suppose we had a second, more realistic simulator, Figure 2.10. If $\hat{R}_2 < \hat{R}_1$, then we can make the claim that simulator 2 is better for the purpose of developing some application than simulator 1. See Figure 2.10.

A related, qualitative way of comparing simulators is by comparing application development paths. In a typical setting, initial application development may occur in simulation, and suppose that this development path was $\{(\hat{\alpha}_i, \hat{J}_i)\}$, where $\hat{J}_i = J(\hat{\alpha}_i, \hat{D})$. In our

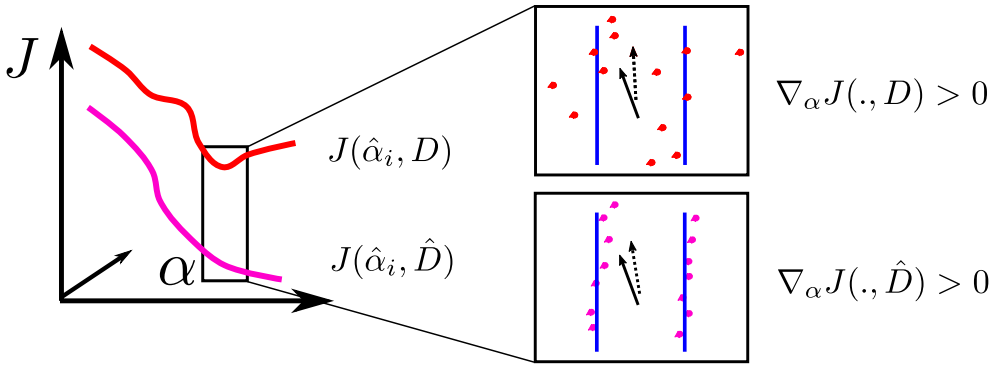


FIGURE 2.11: Comparing algorithm development paths in simulation and reality.

case, since we have a real dataset corresponding to the simulated one, we can compute the objective for these parameters on the real dataset as well. This results in a path $\{(\hat{\alpha}_i, J(\hat{\alpha}_i, D))\}$ representing performance in reality, which is invisible during application development in simulation. For the scan matching example, this is depicted in Figure 2.11. Imagine that the scan matching algorithm is trained in simulation. The developer working in simulation sees the pink curve. The red curve shows the corresponding objective of the same parameters in reality. At a point in the development path, we may zoom in, and analyze the difference in feedback provided by real and simulated data. For example, on simulated data, it may benefit to keep increasing the maximum inlier distance. On the other hand, the real data has outliers. The pose estimate becomes worse if the outliers are considered for registration.

2.4 Assumptions

We summarize the assumptions made in the formulation.

- We assumed that the real dataset D contains the sensor pose as part of the state. Localization robots in off-road environments is a challenging problem in itself, which we assume has been solved before turning attention to simulation. This choice simplifies the formulation used.
- Time is not explicitly part in our formulation, but is implicit in our assumption that the pose is known. We assume that the scene is stationary during data collection. Whether the sensor is stationary or dynamic depends on the domain. For the indoor Lidar simulator, we work with training data in the form of sensor histograms at each state. This requires data collection while the sensor was

stationary. For the off-road Lidar simulator, in contrast, training data was collected while a Lidar sensor mounted on a ground vehicle was being driven. For the purpose of simulation, scenes can be stationary or dynamic.

- We assumed, for application development, the availability of a real dataset D. An argument can be made that under this assumption, D can be used for application development directly, ignoring simulators. And that a simulated dataset $\hat{D}$ will be used primarily where D is difficult to collect. This argument misses key points. First, our aim is to evaluate simulators in a principled, data-driven manner. A simulator can be a predictive model as complex as an application developed on it, and there is no reason to expect that such evaluation will require less data than application development does. Second, it is such data-driven evaluation based on D that motivates application development in simulation in place of reality, as we expect the simulator to generalize to similar datasets. Third, a final iteration of application development for robots will likely require real data, and this can be later utilized to evaluate the simulator.

- We assumed that an algorithm A we develop is 'open-loop', that is, the dataset of states and observations is not dependent on the output of the algorithm. This assumption allows us to collect a real dataset and use it off-line for simulator evaluation.

2.5 Related work

We discuss related work for application-level simulator evaluation in this section. Related work for sensor models and scene generation are more appropriately discussed with each simulation domain. They can be found in Section 3.2 and 4.2 for indoor and off-road Lidar simulation, respectively. In prior work on general-purpose robotics simulators, such as Gazebo [4, 10], and Player/ Stage [11], the focus was on aspects such as software architecture, interfaces, and speed. In other work [12–14] there was comparison among simulators, but not between simulation and reality. Simulators such as BlenSor [12] and VANE [15] often make use sensor models for different sensors. These may be validated individually [6, 16] in controlled environments, similar to standard calibration experiments. The models may then be tested in increasingly complex and realistic environments. For example, [17] evaluated range finders in smoky environments. A challenge in this progression is to create simulated worlds corresponding to real worlds. In work on VANE [15], GPS data from a real environment was compared to simulated data from a 'geo-typical' simulated world. How well the real and simulated worlds matched was unknown. In [18], an outdoor Lidar simulator was evaluated in real scenes of interest.

Another way of stating that evaluation of simulators be performed on the kinds of scenes we are interested in developing applications in is to say that the train and test domains of simulators must be the same. Otherwise we need to deal with the additional problem of domain adaptation [19]. Domain randomization [20] is a recent technique to assist in the transfer of deep neural networks from simulation to reality. In our work, we perform evaluation on simulated scenes which exactly correspond to complex real-world scenes. Our work can be thought of as a validation step, the positive outcomes of which can be used to ground more complex steps such as that presented in [20]. While most of these evaluations were observation-level, some used an application-level loss. In work on AirSim [6], real and simulated data obtained from executing some trajectories on a quadrotor were compared. The simulator USARsim [21] was also evaluated at an application-level, for HRI studies. In the language of our approach, the simulation loss was computed for particular algorithm parameters in [6, 21]. The notion of application development was not further used in evaluation, while we define a simulation risk.

The use of simulation in application development has been studied in [22, 23]. Simulation can be used to generate a synthetic dataset with the size, variety, and annotation detail that a real datasets lacks. Simulated data can also be varied in a controlled way to uncover performance trends of algorithms in simulation. In [22], extensive testing of aerial tracking algorithms on a synthetic dataset was performed, with insights such as sensitivity of performance to target scale variation. In [23], simulated data was used to investigate visual recognition and SLAM algorithms, with insights such as performance trends with respect to the time of day. How well such insights transferred to reality was unaddressed in [22], and limited in [23]. Evaluating the simulator itself was not the focus, as both relied on Unreal Engine 4. In addition, how well the simulated worlds in [22, 23] mapped to real worlds was unknown. Simulator evaluation in our approach requires data from a real world, and a corresponding simulated world. Our work can be seen as complementary. A simulator can be evaluated at the level of an application, and then be used to investigate algorithms in ways not practical in reality.

There has been much work making use of simulation in computer vision [24–26], to augment real training data. For example, [27] presented a synthetic dataset for urban semantic segmentation. The simulated dataset, generated in Unity, was designed to improve on real datasets in terms of object classes, variability, and so on. The utility of simulation was demonstrated by showing that a neural network, trained on a combination of real and synthetic data, performed better than one trained on real data alone. Simulator evaluation was not the focus, and simulated worlds were generated heuristically. In contrast, in [28], simulated indoor scenes in were created using statistics of real scenes. A virtual KITTI dataset was presented in [5], in which simulated worlds were created to correspond to real KITTI videos. Simulation was evaluated, for the

application of object tracking algorithms, by comparing performance of an algorithm in simulation and reality. This quantity, in our approach, is just an application-level simulation loss. We also take a broader view of development, quantified using the simulation risk. In addition, we compare multiple simulators, while [5] evaluated only the one that was utilized. We further note that, in [5], simulation was evaluated by calculating a simulation loss for an algorithm parameter α which had been optimized, in part, to reduce the loss. We believe our loss is more representative of robot applications, since the parameters we use are not aware of the real dataset, making our risk is a more unbiased measure.

Chapter 3

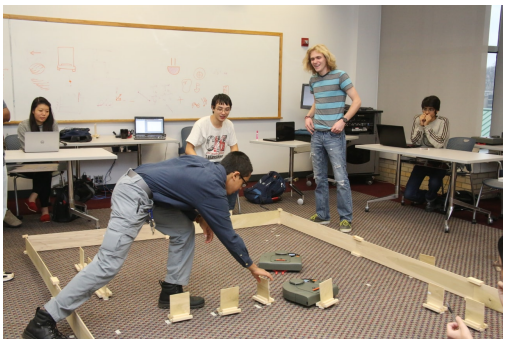
Indoor planar Lidar simulation

3.1 Introduction

We study the problem of building a planar Lidar simulator for indoor scenes. We are motivated by a desire to reduce the cost and complexity of robot application development, while increasing the quality of the resulting software. Perception systems that operate on data from rangefinders need to be robust to noise in the sensors. The algorithms in the pipeline often preprocess the data, and have parameters for thresholds such as outlier checks. The conventional wisdom is that robot applications developed in simulation will need significant rework before they work on hardware. We adopt the perspective that if simulators were only good enough, this would no longer be the case. Indeed, if the data produced by a simulator were as poor in quality as real data, then application developers could address the right issues earlier in the development cycle. They could do so using better debugging tools, and produce a higher quality result, with less effort.

Our focus in this chapter is on sensor modeling. The sensors we work with in robotics are often noisy, and operate in varied environments. Sensor models, which model the distribution of observations given state, help deal with these challenges. They are a widely used and integral part of robotics. To note two examples, they are a component of state estimation algorithms [29], used in updating state based on sensor readings. Observation distributions are also the workhorse of simulation [30], used to generate samples.

One of our motivations was to build a simulator for the rangefinder on a Neato robot (Figure 3.1b). The rangefinder is a low-cost planar laser range sensor, and further details regarding the construction can be found in [31]. A number of retrofitted Neato



(A) Mobile Robot Programming Lab, CMU

(B) Neato robot

robots were used in the Mobile Robot Programming Lab, CMU (Figure 3.1a). In the course, students develop software for mobility and perception. Providing the class with a simulator was one of the original motivations for this work. Despite its simplicity, there is sufficient richness in this task to illustrate the benefits of the approach we take. The Neato sensor's behavior is non-ideal in a number of ways. In Figure 3.2, a range scan from an L-shaped corridor is shown. Certain readings have large biases, which may be peculiar to the particular sensor. The readings have a variance which grows with obstacle distance. There are sensor dropouts, which means that certain readings consistently miss, even when an obstacle is in the range of the sensor. A vanilla sensor simulator, as may be found e.g. in Gazebo [10], is unable to generate realistic data. In addition, we are interested in a simple and principled procedure to acquire a sensor model, which practitioners can implement for their own sensors.

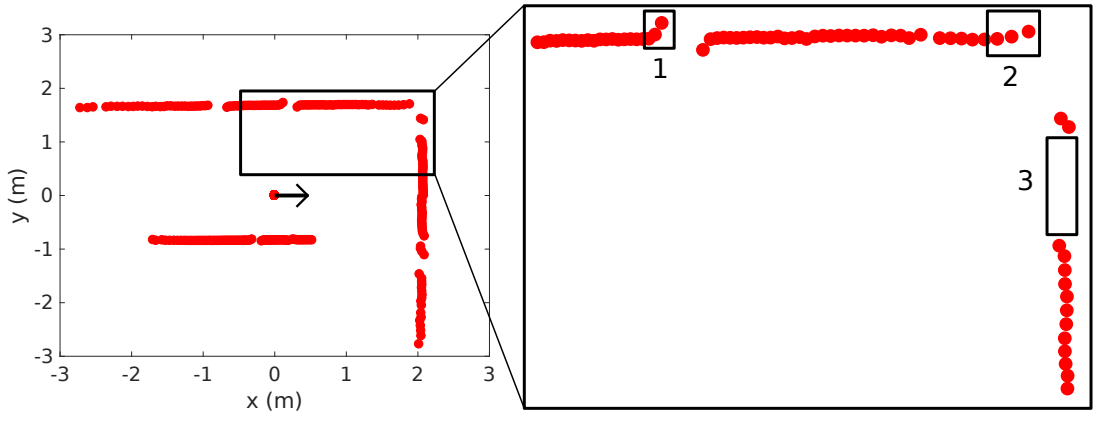


FIGURE 3.2: Non-idealities in the real sensor. Systematic bias (1), increased variance with increased range and angle of incidence (2), dropouts (3). The black arrow depicts the 2D pose of the sensor.

In Section 3.4.2.1 we describe our first approach, a parametric sensor model. The parametric sensor model is selected to capture non-idealities in the real sensor. We evaluate the sensor model by comparing not only raw predictions (Section 3.5.1), but also overall

performance of applications in simulation versus reality (Section 3.5.2). Our parametric approach outperforms the baseline simulator, but it is specific to the Neato Lidar sensor.

Observation distributions approximated by parametric models are limited in their expressiveness, and may require careful design to suit an application. However, sensors can behave noisily in commonplace situations, such as a compass on a mobile robot whose readings are affected by motor current. Instead of a search for the right parametric family for each new situation encountered, a principled and simple nonparametric approach to modeling and predicting distributions from data can be of wide benefit to robotics. In Section 3.4.2.2, we propose nonparametric distribution regression as a procedure to model sensors. It is a data-driven procedure to predict distributions that makes few assumptions on the data source. We apply the procedure to model raw distributions from real sensors (Section 3.5.3), and also demonstrate its utility to a mobile robot state estimation task (Section 3.5.2). We show that nonparametric distribution regression adapts to characteristics in the training data, leading to realistic predictions. The same procedure competes favorably with baseline parametric models across applications. The results also help develop intuition for different sensor modeling situations. Our procedure is useful when distributions are inherently noisy, and sufficient data is available.

This chapter contains work from the following publications.

Tallavajhula, Abhijeet, and Alonzo Kelly. "Construction and validation of a high fidelity simulator for a planar range sensor." ICRA 2015.

Tallavajhula, Abhijeet, Barnabs Pczos, and Alonzo Kelly. "Nonparametric distribution regression applied to sensor modeling." IROS 2016.

3.2 Related work

Existing open-source simulators are not well-suited for real-world sensor modeling. The Lidar model in Gazebo [10] is simplistic, consisting of random Gaussian noise added to the output from ray-tracing. The noise parameters are uniform and apply to all rays. The Lidar model in Blensor [12] is more fine-grained, with adjustable noise parameters for individual rays. However, even in the case of available parameters, there is no procedure provided as to what the noise settings must be, or how to determine them. A sensor model requires an associated training procedure.

Since our work is about acquiring a sensor model and using it for simulation, an associated field is calibration. Sensor calibration is a classic problem that has been repeatedly studied [32], [16]. Calibration is often stated as characterizing the error in a model. Starting with some representation of the error, the sensor is placed in a carefully controlled environment where precise and accurate measurements can be taken. Residuals are calculated from the difference between the true and the expected readings, and the error model is fit to the residuals. Given identified errors, simulating is simple: clean data is generated to which noise is added, which is what is done in the simulators [10, 12]. In contrast, we make no distinction between a model and noise. We choose parameters to directly represent the entire sensor distribution. We then use data-driven methods to learn how these parameters vary with state.

One way to characterize approaches to modeling for simulation is based on the assumed knowledge of state. In a controlled calibration environment, state may exactly be known. Results from calibration can then be used to predict sensor output at any state within experiment limits. [18] solves the problem of Lidar simulation in a different setting: training data is in the form of Lidar scans taken outdoors, with no map, but accurate pose information is available via a high-quality inertial navigation system (INS). The focus is on modeling the environment, which is represented by a combination of fit planes and permeable ellipsoids. Realistic simulation in the vicinity of training poses is possible, but this solution does not generalize across environments. Our work, conducted and applicable indoors, considers the case that both the map and sensor pose are known nominally, within reasonable error. We incur a loss in prediction power, but gain significant ease in data collection. This approach was motivated by the observation that while exact localization is difficult, current localization algorithms result in fairly good state estimates.

In sensor modeling, there are broadly two problems to be considered. The first is the problem of modeling distributions, given samples. There is a long history of parametric modeling, which approximates the observation distributions with a parametric one.

While these methods may be sufficient when the approximations are good, they are limited in their expressiveness. Dealing with complex distributions involves incorporating more parameters, and increasingly elaborate models. For example, [33] presents results of detailed parametric modeling for sonar sensors. The second problem is to predict observation distributions. Data, such as logged states and observations, will be available only at certain states, from which we may want to predict a distribution at an unseen state. With the parametric models, the approach is to allow the parameters to depend on state. Sensor models for range sensors, for example, vary in complexity from the simple raycast model in [29], to a Bayesian network model developed in [34].

The majority of sensor models are parametric. For example, [35] proposes a model for small sonar sensors. [16] characterizes a parametric model for ladar devices. Other work fits a parametric model to sensor readings, but estimates parameters at new states using nonparametric methods. [36] considered Gaussian distributions as observation models, but estimated parameters using Gaussian processes. Similarly, [37] used Gaussian process regression to predict models for multiple beams of a range sensor. [38] regressed covariances of a Gaussian distribution using nonparametric kernel smoothing. In contrast, we take a finer-grained approach to sensor modeling, performing even density estimation in a nonparametric manner.

There has been recent work in machine learning which extends the standard case of real-to-real regression to distribution-to-real regression [39], distribution-to-distribution regression [40], and other variants. Importantly, these procedures are nonparametric. The case that is relevant to sensor modeling is real-to-distribution regression, or regressing from state to observation distributions. We will henceforth refer to real-to-distribution regression as simply distribution regression. Nonparametric distribution regression is conceptually simple, consisting of two steps that can be abstracted as density estimation, followed by regression. Nonparametric density estimation allows us to deal with complex distributions. Nonparametric regression allows us to deal with complex functions of state.

Validation of simulators is a topic that receives little attention. The experimental results in [12] are comparisons between simulators, not a comparison of simulation with reality. [41] introduces a synthetic benchmark for the evaluation of vision algorithm that use RGB-D data. Algorithms are compared on clean and noisy simulations, but there are no results showing how this correlates with real performance. Part of the reason is the difficulty in recreating environments in simulation. In [18] extensive validation of lidar simulators is performed. Quantitative measures for comparing lidar readings are defined and computed for simulated versus real readings over a number of runs.

3.3 Statistics background

Our approach to sensor modeling is data-driven. It builds on concepts in statistics, which we review in this section. Some of the notation used is local to this section. Further details on these topics may be found in the references [9, 42].

3.3.1 Density estimation

Density estimation is the problem of estimating the density $p(X = x)$ of a random variable $X \in \mathbb{R}^d$, given independent and identically distributed (i.i.d) samples $\{X_i\}$, $i = 1 : n$ drawn from $p(X)$. Let the estimate be $\hat{p}(x)$. The objective is to minimize the risk, which can be taken as the expected L_2 loss, $\mathbb{E}(\int (p(x) - \hat{p}(x))^2)$.

In parametric density estimation, the density is assumed to depend on parameters β, expressed as $\hat{p}(x, \beta)$. The task is to estimate the parameters β. In nonparametric density estimation, it is not assumed that the density has a specific functional form. An example is kernel density estimation. For a scalar input, an example kernel is the Gaussian kernel,

$$K(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right), x \in \mathbb{R}. \tag{3.1}$$

A scalar kernel with a bandwidth δ is defined as $K_\delta(x) = \frac{1}{\delta} K(x/\delta)$. A kernel in higher dimensions can be defined in terms of a scalar kernel. We use a positive definite bandwidth matrix, Δ,

$$K_\Delta(x) = |\Delta|^{-\frac{1}{2}} K\left(\Delta^{-\frac{1}{2}} ||x||\right), x \in \mathbb{R}^d.$$

The bandwidth is a hyperparameter, which must be tuned for optimal performance. In practice, it can be selected on training data using cross-validation. Or it may be selected on a separate validation dataset. Once a bandwidth is selected, the error of the estimator is calculated on a test dataset. The hyperparameters are not tuned on the test dataset, because then the test error will no longer be unbiased. Other nonparametric density estimators are histograms, and orthogonal series estimators. Theoretical guarantees for nonparametric density estimation are well-understood. Consider the Hölder class of functions $\Sigma(k, L)$. For integer k, functions in this class have their $k - 1$th derivative Lipschitz continuous, with constant L. The Hölder class is a way of imposing regularity conditions on the unknown density. For densities in $\Sigma(k, L)$, the error goes to zero at rate $n^{-\frac{2k}{2k+d}}$. This is also known to be the minimax error rate.

3.3.2 Regression

Regression is the problem of predicting an output Y given input X. Typically the output is a scalar, $Y \in \mathbb{R}$, while the input may be a vector $X \in \mathbb{R}^d$. The joint distribution $P(X, Y)$ is unknown, but we have a dataset $D = \{(X_i, Y_i)\}, i = 1 : n$ of i.i.d samples from the unknown distribution. Our predictor is $f(X)$, which depends on the dataset. The objective is to minimize the expected squared error, $\mathbb{E}(Y - f(X))^2$. The expectation is over (X, Y) as well as the data which is used to select the predictor $f(X)$. Consider the squared error conditioned on the input, $\mathbb{E}[(Y - f(X))^2|X = x]$. A useful decomposition is as follows

$$\begin{aligned}\mathbb{E}[(Y - f(X))^2|X = x] &= \mathbb{E}[(Y - \mathbb{E}(Y|x) + \mathbb{E}(Y|x) - f(X))^2|x] \\ &= \mathbb{E}[(Y - \mathbb{E}(Y|x))^2|x] + \mathbb{E}[(\mathbb{E}(Y|x) - f(x))^2] \\ &= \sigma^2 + (\mathbb{E}(Y|x) - \mathbb{E}f(x))^2 + \mathbb{E}(\mathbb{E}f(x) - f(x))^2.\end{aligned}$$

Above, σ^2 is the variance of the output Y from its mean $\mathbb{E}(Y|x)$ at x, and is not in our control. From the second line, it is seen that the optimal prediction is $f(x) = \mathbb{E}(Y|x)$. Therefore, regression can be seen as an approximation of the conditional mean of the output given the input. The term $(\mathbb{E}(Y|x) - \mathbb{E}f(x))^2$ represents the predictor bias. The term $\mathbb{E}(\mathbb{E}f(x) - f(x))^2$ represents the variance of the predictor. Typically we will pick from a class of predictors. Choosing a larger class may reduce bias, since it may approximate the true conditional mean better. But it may increase the variance, due to the fact that we are learning from finite samples. Variance is controlled by techniques such as regularization. This observation is often referred to as the bias-variance tradeoff.

The class of predictors may be parametric. For example, the class of linear predictors, $f(x) = w^T x$. By contrast, in nonparametric regression, a fixed functional form is not assumed for $f(x)$. An example is kernel smoothing,

$$f(x) = \frac{\sum_{i=1}^{n} K_\Delta(||x - X_i||)Y_i}{\sum_{i=1}^{n} K_\Delta(||x - X_i||)},$$

which is similar to kernel density estimation. A combination of a linear predictor and kernel smoothing is the local linear predictor, in which weights w are chosen to minimize the error

$$\sum_{i=1}^{n} K_\Delta(||x - X_i||)(Y_i - w^T X_i)^2. \tag{3.2}$$

In contrast to a parametric linear predictor, the weights $w(x)$ depend on the query input. Similar higher-order local polynomial fits are possible. They may help reduce bias, at

the cost of increasing variance. Other nonparametric regression methods include RKHS regression, and Gaussian process regression. Nonparametric regressors may have their own hyperparameters θ, such as regularization weights. These can be selected using cross-validation, or separate validation data. The theoretical guarantee is similar to density estimation. For functions in the Hölder class $\Sigma(k, L)$, a nonparametric regression method such as kernel smoothing has error rate $n^{-\frac{2k}{2k+1}}$.

3.3.3 Distribution regression

This review in this section is based on the reference [40]. Standard regression considers both inputs and outputs to be real vectors. Distribution regression extends the setting to allow both inputs and outputs to be distributions themselves. Instead of a function, a functional f must be regressed. Denote the input distribution by $P \in \mathcal{P}$, and the output distribution by $Q \in \mathcal{Q}$. The input-output distribution samples are $\{P_i, Q_i\}, i = 1 : n$. In standard regression, the samples are observed. In this case, the distribution samples are not observed. Rather, data sampled from these distributions are. The available data is the set of pairs $\{(S_i^X, S_i^Y)\}, i = 1 : n$, where $S_i^X = \{X_{ij}\}, j = 1 : m^X$, and X_{ij} are i.i.d samples from P_i. Similarly, $S_i^Y = \{Y_{ij}\}, j = 1 : m^Y$, and Y_{ij} are i.i.d samples from Q_i. The number of samples from each input or output distribution, m^X and m^Y, are allowed to depend on index i.

The first step in distribution regression is to estimate the input and output densities, $\{(\hat{p}_i, \hat{q}_i)\}$. This step uses the techniques from Section 3.3.1. A functional from inputs to outputs is then regressed. This step adapts function regression techniques from Section 3.3.2. A number of nonparametric function regressors, including kernel smoothing, can be interpreted as linear smoothers. The prediction of a linear smoother at a query input is a weighted sum of the outputs in the training data. The weights themselves are a function of the input. Similarly, a functional regressor can be considered which has the form

$$f(p) = \sum_{i=1}^{n} W(p, p_i) q_i,$$

where W denotes the weights. In implementation, the estimated densities $\hat{p}, \hat{p}_i, \hat{q}_i$ will be input to the regressor. The theoretical analysis in [40] is carried out for orthogonal series estimation of the output densities. If the input distribution space is a doubling dimension, it is proved that the rate of convergence of the risk is polynomial in n, m^X, m^Y.

3.4 Approach

3.4.1 Real and simulated scenes

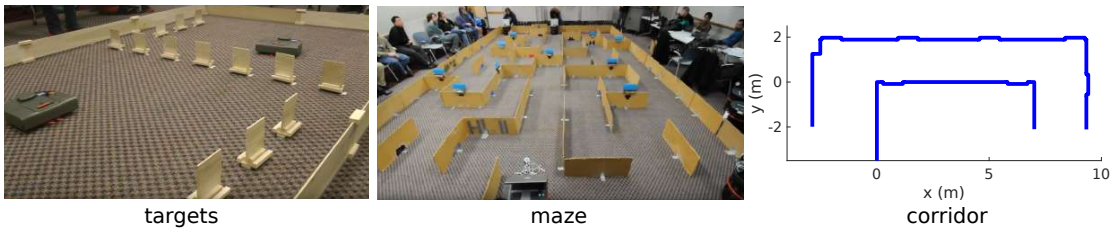


FIGURE 3.3: Objects made of line segments can represent a number of indoor scenes. The image on the right is the plan of a real corridor.

We consider planar scenes with objects that consist of line segments. The objects include piecewise-smooth lines, and closed polygons. One reason for this choice is the kind of scenes used in the MRPL course. In the course, scenes are constructed using straight segments (made of wood) of known length. A number of desired objects and environments can be constructed using the segments, such as

- walls with corners, and straight corridors. Such environments make it simple to specify a map to localize against.
- Obstacles, which robots must avoid.
- Targets, which robots must track and drive up to.
- Mazes, which robots must use path planning and search to get out of.

Line segments are also an accurate representation of indoor environments. For example, corridors, for which floorplans may be available. Some environments are depicted in Figure 3.3. Sensor modeling is simpler in such a scene compared to one represented by occupancy map. Further, a description of a scene in terms of line segments can be extracted from range data, as reviewed in [43, 44].

In symbols, a scene, denoted by S, is a set of objects, $S = \{o_k\}$. An object $o = (M, q_{\text{object}})$ consists of an object model M, and the object pose, q_{object}. The object model M may be taken to be the x and y coordinates of the endpoints of the line segments. The coordinates are specified in any coordinate system whose origin is fixed to the object. Our object model includes only geometry, and not variables such as reflectivity. In practice this simplification was enabled by constructing real scenes with the same material. The coordinates of the object model are specified in the identity frame. The object is transformed to the world frame via the object pose, q_{object}. The

real state $x = (S, q_{\text{sensor}})$ consists of the scene along with the pose of the sensor. All poses in planar scenes are in $SE(2)$.

Due to the simplicity of the real scenes, they can be specified exactly in simulation. This means that the real and simulated scenes spaces are identical, or $\hat{\mathcal{X}} = \mathcal{X}$. We know object models $\{M_k\}$ since they were constructed using planks of known lengths. We conducted experiments in an indoor lab space, where it was also simple to measure the pose of the objects $\{q_{\text{object},k}\}$. We localized the robot using either an optimized registration algorithm, or using a motion capture system. The registration itself was validated using the motion capture. Since both the real scene and sensor pose were known, we could reconstruct the real state exactly, or $\hat{x} = x$. In the rest of this chapter, we only refer to the real state x.

An interesting question is: can a useful sensor model be constructed when there is noise in the state information? Suppose that true state in the data is unknown and only estimates $\{x'_i\}$ were available. For example, we may only have a nominal map and the approximate sensor pose. Under such uncertainty, an Expectation-Maximization (EM) approach comes to mind. For example, consider a sensor mounted on a robot for which we had the control and observation history $\{u_i, z_i\}$. We could try to frame the simulator parameters β as the maximizing parameters and states $\{x'_i\}$ as the hidden variables. Given a sensor model with parameters β, the E-step would involve state estimation (which makes use of the sensor model). Given the distributions over state at each timestep, the M-step would be to optimize the sensor model, as $\beta = \arg\max_{\beta'} P(\{z_i\}|\{x_i\}, \beta')$. However, we are interested in simulator parameters that vary with state. We do not assume a functional form for $\beta(x)$, but use nonparametric regression. In this case, states are part of the data points that define the learned model. They can no longer be considered hidden variables, violating the conditions of the EM algorithm.

There may be further concerns that we equate x and $\hat{x}$, as the real state may not be exactly known in simulation. We primarily use the simulated state information to train regressors, so we address these concerns by appealing to results in statistics. The general problem of regression in the presence of measurement error has been studied in the literature on error in variables. Nonparametric prediction with input noise is in general a hard problem. Advanced methods [45, 46] allow input errors to be drawn from arbitrary distributions and construct unbiased estimators when multiple noisy observations of the same input are available. The resulting procedure is computationally intensive, however. A single prediction involves evaluating an integral as part of an inverse Fourier transform, a laborious task for a range simulator which needs to make hundreds of prediction at each state. Instead, we rely on ideas from regression calibration [47], a procedure in which the true state x is replaced by the noisy state x' and standard regression is performed.

This simple procedure is most useful when errors are small, a characteristic we use to our advantage. Observing that localizing state within $1cm$ takes reasonable effort, but doing so within $1mm$ takes much more, this procedure of working with approximate state relieves some of the burden of model-building. It is important to keep in mind the framework of the procedure. Regression calibration requires that predictions are made on inputs observed with the same error as the training inputs. Strictly then, the range simulator does not estimate $p(z|x)$, but $p(z|x')$. For a range sensor, approximate state knowledge during training implies that bias in the readings and pose errors are interrelated. A bias predicted in range could be partly due to error in the training pose.

3.4.2 Sensor modeling

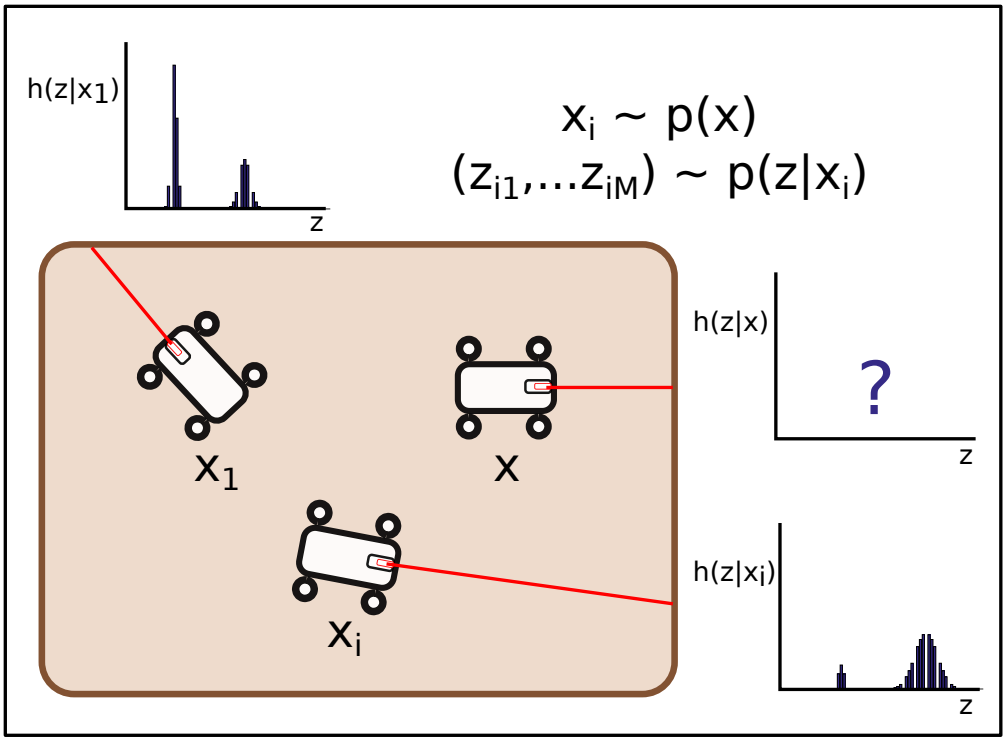


FIGURE 3.4: The problem of sensor modeling. Given data in the form of states and observations at those states, we want to model the observation distribution at unseen states.

Given our choice of indoor scenes, the problem of building a simulator reduces to that of sensor modeling. Standard regression would predict an observation $\hat{z}$ at a state x. Since sensors are noisy, this is not very useful. Instead, we estimate the true distribution of observations $p(z|x)$ by $\hat{p}(z|x)$. Since we estimate a distribution, sensor modeling is a special case of distribution regression:one in which the inputs are real vectors, but the outputs are distributions. We assume that datasets are of the form $D = \{(x_i, z_{ij})\}, i = 1\colon N, j = 1\colon M$. States are i.i.d samples drawn from a state distribution $p(x)$. The test distribution is assumed to be the same as the training distribution:

a standard assumption made by most learning procedures. Intuitively it means that we will encounter states similar to those trained on. Observations are i.i.d samples from the observation distribution at that state, $z_{ij} \sim p(z|x_i)$. It is not necessary that the same number of observations are logged at each state x_i. We take them all to be M for simplicity. Multiple observations at each state are required for good density estimates. This form of dataset is easy to collect in a number of cases, including our planar indoor Lidar sensor. With the sensor mounted on a mobile robot, as in Figure 3.10, the robot drives to a new pose, logs sensor data, and repeats the process.

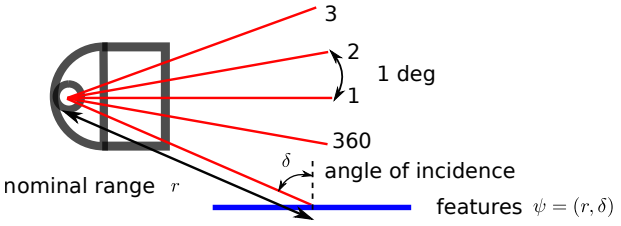


FIGURE 3.5: The laser has 360 bearings. They are spaced at 1 deg (not drawn to scale). We model each independently. Features used for regression are the nominal range and angle of incidence to an obstacle.

The Lidar sensor we work with has 360 range returns. Their bearings are equally spaced in $[0, 2\pi)$. For regression, we do not use the state x directly, but instead work with features $\psi(x)$ computed from the state. For a query ray in a scene, we use the nominal range, and angle of incidence to the target as features, $\psi = (r, \delta)$. This is depicted in Figure 3.5. We continue to write the sensor model as $\hat{p}(z|x)$, while understanding that the dependence is more accurately represented as $\hat{p}(z|\psi(x))$.

3.4.2.1 Parametric method

In this approach, the estimated distribution belongs to a parametric family, $\hat{p}(z|x) = \hat{p}(z|x, \beta)$. The β_i compress the information in the observations $\{z_{ij}\}$. Given training data D, parametric distribution regression involves two steps. First, parameters are estimated from the observations, resulting in a reduced dataset $D^\beta = \{(x_i, \hat{\beta}_i)\}, i = 1\colon N$. At a query state x, we predict the distribution by estimating $\hat{\beta}$ via regression with D^β, and constructing the distribution. The approach is summarized in Procedure 1. Note that the parametric only refers to the output distributions. The regressor to predict β itself can be non-parametric.

A parametric model was selected to capture the non-idealities we came across when working with the sensor. A coin is flipped, and with probability P_{null}, the sensor returns a null reading. With probability $1 - P_{\text{null}}$, a range reading is sampled from a Gaussian $\mathcal{N}(\mu, \sigma)$. Given samples $\{z_j\}, j = 1 : M$, we use the maximum likelihood estimators for

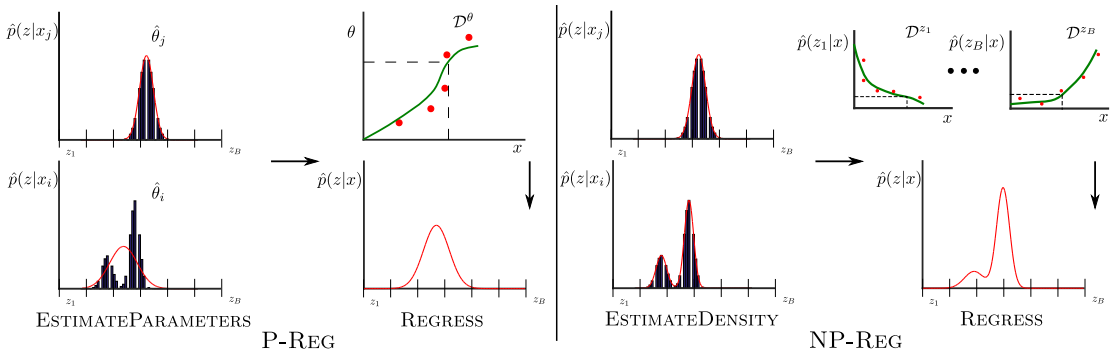


FIGURE 3.6: Illustration of the two regression procedures. In P-REG, parameters β_i are estimated for the training data. At a new state x, β is estimated via regression. In NP-REG, distributions are estimated nonparametrically. At a new state x, each $\hat{p}(z_b|x)$ is predicted independently via regression.

Procedure 1: P-REG

Input : Query x, Dataset $D = \{(x_i, z_{ij})\}$
Output : Predicted distribution $\hat{p}(z|x)$

1 $\hat{\beta}_i \leftarrow$ EstimateParameters$(\{z_{ij}\})$
2 $D^\beta \leftarrow \{(x_i, \beta_i)\}$
3 $\hat{\beta} \leftarrow$ Regress(x, D^β)
4 $\hat{p}(z|x) \leftarrow \hat{p}(z|x, \hat{\beta})$

EstimateParameters

$$\hat{P}_{\text{null}} = \frac{\text{num null readings}}{M},$$
$$\hat{\mu} = \frac{1}{M}\sum_{j=1}^{M} z_j,$$
$$\hat{\sigma} = \frac{1}{M}\sum_{j=1}^{M}(z_j - \hat{\mu})^2.$$

The resulting sensor model requires the prediction of three parameters, $\beta = (P_{\text{null}}, \mu, \sigma)$. As discussed in Section 3.2, we do not explicitly model an error in the signal. The systematic bias, for example, will be contained in the trend of the mean, μ. We used local linear smoothing (Equation 3.2) with a Gaussian kernel for Regress. Each component of β was regressed independently. At a query feature ψ, the prediction is $w^T\psi$, where the weights w are given by

$$w = (\Psi^T W \Psi)^{-1}(\Psi^T W Y),$$

Procedure 2: NP-REG

Input : Query x, Dataset $D = \{(x_i, z_{ij})\}$

Output : Predicted distribution $\hat{p}(z|x)$

1 $\hat{p}(z|x_i) \leftarrow \text{ESTIMATEDENSITY}(\{z_{ij}\})$

2 $D^z \leftarrow \{(x_i, \hat{p}(z|x_i))\}$

3 $\hat{p}(z|x) \leftarrow \text{REGRESS}(x, D^z)$

where $\Psi \in \mathbb{R}^{N \times \dim(\psi)}$ is a matrix where each row is a feature vector from the training dataset. The weight matrix $W \in \mathbb{R}^{N \times N}$ is a diagonal matrix, with entries

$$W_{ii} = \frac{K_\Delta(\|\psi - \psi_i\|)}{\sum_{j=1}^{N} K_\Delta(\|\psi - \psi_j\|)}.$$

The vector $Y \in \mathbb{R}^{N \times 1}$ is $(P_{\text{null},i})$, $(\hat{\mu}_i)$, or $(\hat{\sigma}_i)$, depending on which sensor model parameter is being regressed. We modeled the each of the 360 bearings in an observation $z = z^k$, independently. The hyperparameters of the model were the bandwidths in the local linear smoothers. We used bandwidths common to each Lidar bearing, but different across the parameters being estimated. These were tuned to minimize error on a validation dataset. Note that the simulator parameters θ in this case were the hyperparameters of REGRESS. The parameters β were not related to θ; they were estimated, and used to model the observation distribution.

3.4.2.2 Nonparametric method

We now discuss the nonparametric approach to sensor modeling. Given training data $D = \{(x_i, z_{ij})\}$, it involves two familiar steps. First, since we only have samples $\{z_{ij}\}$ from $p(z|x_i)$, we construct nonparametric density estimates $\hat{p}(z|x_i)$. This results in datasets $D^z = \{x_i, \hat{p}(z|x_i)\}, i = 1\colon N$. At a query state x, we predict the distribution by regressing at each z with D^z. We call this nonparametric regression procedure NP-REG, and it is summarized in Procedure 2. Note that, for both P-REG and NP-REG, $\hat{p}(z|x)$ has to be computed for each z of interest, although we have suppressed this in the input to the procedures. If we are performing state estimation, $\hat{p}(z|x)$ only at the observation currently being processed is of interest. If we are performing simulation, we need $\hat{p}(z|x)$ over the domain of z in order to draw samples.

For EstimateDensity, we implemented kernel density estimation. Let K_z denote the observation kernel, and δ_z the observation kernel bandwidth, then

$$\hat{p}(z|x_i) = \frac{1}{M}\sum_{j=1}^{M} K_{z,\delta_z}(||z - z_{ij}||), \tag{3.3a}$$

$$K_{z,\delta_z}(||z - z_{ij}||) = \frac{1}{\delta_z} K_z\left(\frac{||z - z_{ij}||}{\delta_z}\right). \tag{3.3b}$$

Sensor observations in our case take values from a discrete set, $z \in \{z_1 \colon z_B\}$. In such cases, we integrate the distribution to obtain the observation probabilities, and construct a histogram $h(x) \in \mathbb{R}^B$. The histogram has B bins, and stores probability values in each bin, $h^b(x) = P(z_b|x), b = 1 \colon B$. Instead of kernel density estimates integrated to histograms, we directly use histogram estimates when we want to speed up predictions. With $\mathbb{I}$ denoting the indicator function, the histogram probability estimate in bin b is given by

$$\hat{h}^b(x_i) = \frac{1}{M}\sum_{j=1}^{M} \mathbb{I}(z_{ij} = z_b),\ b = 1 \colon B. \tag{3.4}$$

For Regress, we use kernel smoothing. Let K_x denote the state kernel, and Δ_x the state kernel bandwidth. $\Delta_x \in \mathbb{R}^{p \times p}$ is a matrix, where p is the state dimension.

$$\hat{p}(z|x) = \frac{\sum_{i=1}^{N} \hat{p}(z|x_i) K_{x,\Delta_x}(||x - x_i||)}{\sum_{i=1}^{N} K_{x,\Delta_x}(||x - x_i||)} \tag{3.5a}$$

$$K_{x,\Delta_x}(||x - x_i||) = |\Delta_x|^{-\frac{1}{2}} K_x\left(\Delta_x^{-\frac{1}{2}}||x - x_i||\right) \tag{3.5b}$$

The kernel bandwidths δ_z and Δ_x are the hyperparameters of the procedures. They were selected to minimize the empirical risk $\hat{R}$ on validation data. The observation kernel bandwidth δ_z is a scalar, but the state kernel bandwidth Δ_x is a matrix. It is common to choose a scaled identity matrix, $\Delta_x = \delta_x I_p$, where I_p is the $p \times p$ identity matrix. But not all dimensions of state have the same scale. To balance computational overhead with expressiveness, we use a diagonal matrix, $\Delta_x = \text{diag}(\delta_{x^1} \ldots \delta_{x^p})$. The hyperparameters are selected using a grid search. We found that the choice of kernel was not crucial. Using a different kernel, such as the boxcar kernel, only led to different bandwidth values.

Other choices exist for the component procedures, such as series estimation [48] for EstimateDensity, and Gaussian processes [49] for Regress. With our choices, prediction using NP-Reg is efficient. Let $H \in \mathbb{R}^{B \times N}$ denote the matrix of estimated histogram values, where $H_{bi} = \hat{h}^b(x_i)$. Suppose there are Q query states, $\{x_1 \colon x_Q\}$.

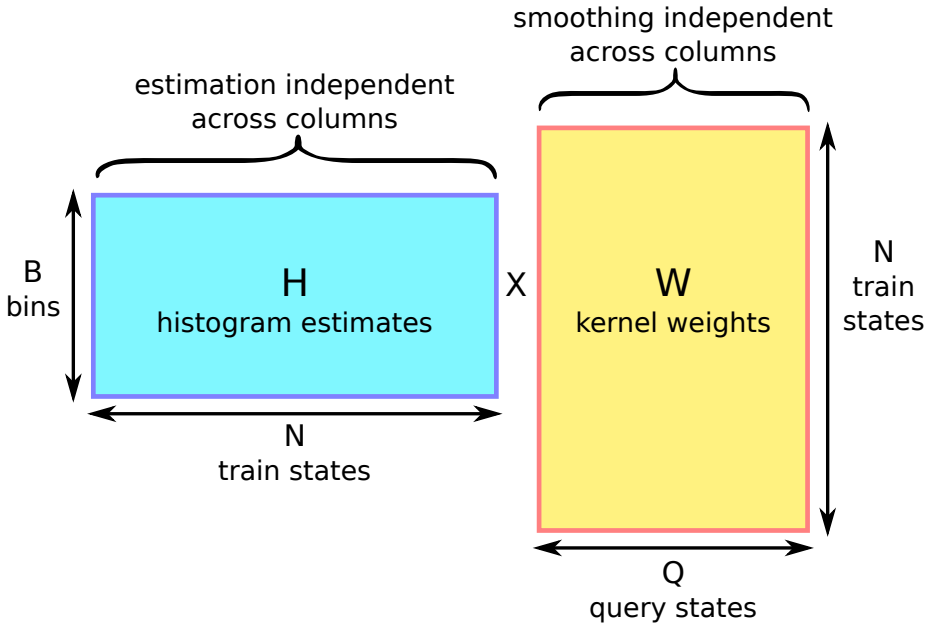


FIGURE 3.7: With histogram estimation and kernel smoothing, prediction is an efficient matrix product.

Let $W \in \mathbb{R}^{N \times Q}$ denote the matrix of weights derived from the state kernel, where $W_{iq} = K_{x,\Delta_x}(||x_q - x_i||)/(\sum_{i=1}^{N} K_{x,\Delta_x}(||x_q - x_i||))$. Then the $B \times Q$ matrix of predicted histograms at the query states is HW (Figure 3.7). Note that the histograms from the training data are linearly smoothed to make a prediction at a query state. This is a recurring feature of nonparametric methods. With a histogram density estimator, $N = 10^3$ training states, $M = 100$ observations and $Q = 200$ query states, prediction takes about $0.5s$ when implemented in MATLAB. Histogram estimation is independent across training data, and linear smoothing is independent across queries.

NP-REG is obviously useful when $p(z|x)$ is inherently noisy. We use 'noisy' in an informal sense, to mean that the observation distribution is hard to approximate parametrically. We point out another case. Let us expand the state to $(x,\ y)$. The part y of state is unmodeled. If the information y were available, it may be possible to build a good parametric model $\hat{p}(z|x, y)$ of the true distribution $p(z|x, y)$. But our model is $\hat{p}(z|x)$, and the effect of the unmodeled states on this distribution is increased apparent noise.

Instead of modeling the effect of unmodeled states in the observation model, an alternate approach is to directly model y. The choice depends on the application. Consider the operation of a range sensor. Let the unmodeled states y be dynamic objects in the environment, which are not explicitly represented. Careful calibration and reasoning may result in a satisfactory parametric model $p(z|x, y)$. In a sense, physics-based methods refine models by reasoning about y. Enumerating models for a few materials is easy, but if there are a large number of material types in the environment, it is difficult to

obtain information about all. If the distribution $p(z|x)$ is noisy, it is suited for nonparametric treatment. Limiting the size of x may also be desirable in applications such as simulation. An important point is that the distribution of the unmodeled states $p(y)$ must be the same during training and test. The predicted observation model $\hat{p}(z|x)$ will represent observations due to different materials in a statistically, if not physically, realistic manner.

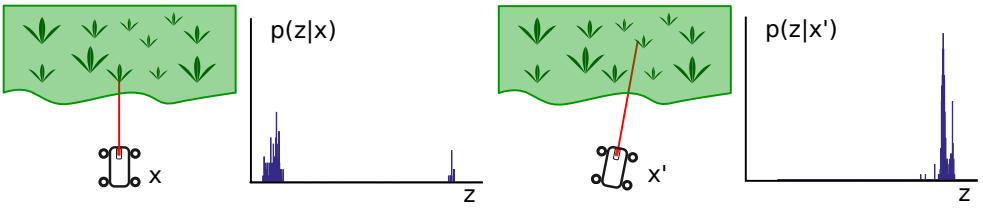


FIGURE 3.8: Illustration of when sensor modeling is fundamentally difficult. Observation distributions vary widely for small changes in pose. $p(z|x)$ is not only noisy but also non-smooth in state.

Finally, there are situations in which sensor modeling is fundamentally hard. This is because both the parametric and nonparametric procedures rely on a REGRESS step. Regression is hard when there is little smoothness of outputs in the inputs. Consider a range sensor again, operating outdoors, pointed at a patch of grass (Figure 3.8). The state x is just the pose of the sensor. $p(z|x)$ may be hard to model parametrically at any state, because of effects such as pass-throughs (e.g. when lidar randomly senses the range to foreground or background surfaces). Nonparametric methods can model distributions at any two states x and x'. However, between x and x', the distributions themselves may vary in a non-smooth manner, because blades of grass are spatially discontinuous.

3.4.3 Simulator evaluation

We only discuss the observation-level loss here. Details of the application for the application-level loss are can be found in the experiment sections. There are a number of methods to evaluate observation models (see [18] for a discussion), such as the likelihood of observations. We use the formal concept of risk from learning theory [9]. For the parametric approach, the loss is the squared error between $\hat{\beta}$ and β. The empirical risk is the mean of the loss over a dataset.

For the nonparametric approach, we need to specify the histogram metric used to compute the loss l of prediction. The Euclidean norm $l(h, \hat{h}) = \sum_{b=1}^{B}(h^b - \hat{h}^b)^2$ is not a good choice. It favors placing probability mass in the same bins, which may be low even with placing mass arbitrarily in other bins. This point is illustrated in Figure 3.9. Instead we use the histogram match, $l(h, \hat{h}) = \sum_{b=1}^{B} |\sum_{c=1}^{b} h^c - \sum_{c=1}^{b} \hat{h}^c|$. It is the $L1$ norm

of the cumulative sums, and takes into account the position and distribution of probability mass. The component algorithms in Procedures 1 and 2 (EstimateDensity etc.) will have free parameters, which are hyperparameters. During training, these hyperparameters were selected to minimize empirical risk.

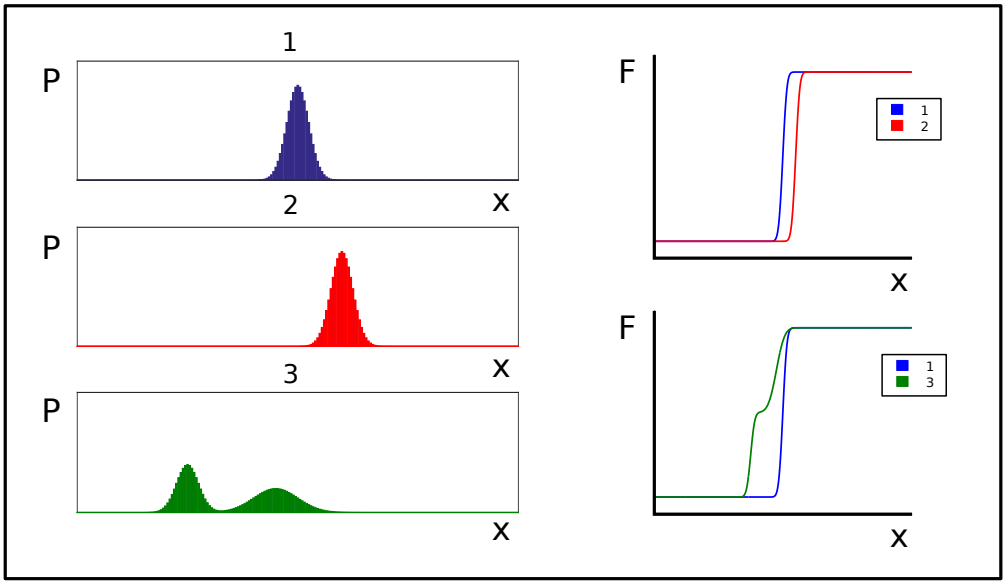


FIGURE 3.9: The plots on the left are the histograms. Plots on the left show corresponding cdfs. Under the Euclidean metric, histogram 1 matches 3 more closely than 2. Under the histogram match metric, histogram 1 matches 2 more closely than 3.

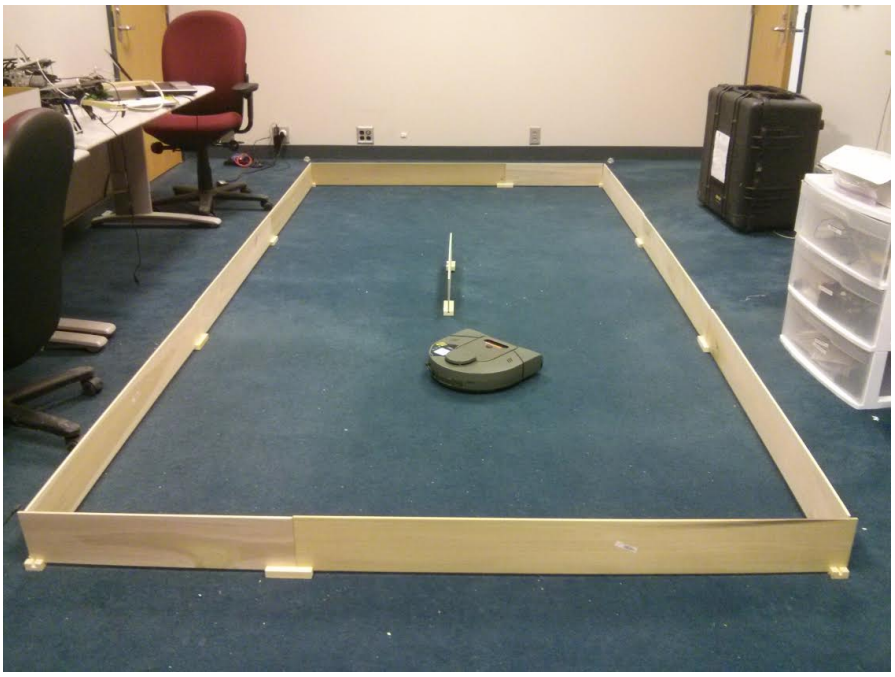


FIGURE 3.10: A Neato in the data collection environment

Field	Value
Frequency	5 Hz
Number of bearings	360
[Min range, Max range]	$[0, 4.5]$ m
Resolution	0.001 m
Number of histogram bins B	4501

TABLE 3.1: Sensor properties

3.5 Experiments

3.5.1 Observation-level simulator evaluation

The maximum range, resolution, and other details are listed in Table 3.1. Datasets for learning were collected by driving the robot around inside a known map, shown in Figure 3.10. The environment was static during data logging. The number N of sensor poses in the training, validation and test datasets was 100, 30, and 30, respectively. We chose the number of observations M to be logged at each state based on the histogram loss metric (Section 3.4.3). A large number of observations, 250, were collected at a representative state. The resulting histogram, say $\hat{h}_{250}$, was treated as ground truth. We then computed the estimated histogram, $\hat{h}_M$, and the loss $l(\hat{h}_{250}, \hat{h}_M)$. As expected, the loss decreases with increasing M. For data logging, we then picked the M at which the loss fell below a threshold (0.5). The value picked was $M = 70$. See Figure 3.14a.

Parameter	P-Reg error	Physics-based model error
μ	0.015 m	0.018 m
σ	0.001 m	0.002 m
P_{null}	0.0076	–

Table 3.2: Sensor model parameter prediction error

For comparison of our data-driven procedure with a physics-based sensor model, we use the following formula for the variance, derived in [50] and used in [16, 51],

$$\sigma^2 = K \frac{r^2}{\cos \delta}. \tag{3.6}$$

The proportionality constant $K = 1e - 3$ was determined from a grid search on the training data. Predicted error in the sensor model parameters for the physics-based approach and our data-driven parametric approach (Section 3.4.2.1) are in Table 3.2. Example real data, and corresponding simulated data, is depicted in Figure 3.12. The advantage to learning parameters for each bearing is that systematic biases can be reproduced. Certain bearings consistently record null readings, another behavior that is well-learned. The non-null readings seem geometric in nature, while the null readings are likely caused by physical obstructions to some bearings. Regression surfaces for P-Reg are shown in Figure 3.11.

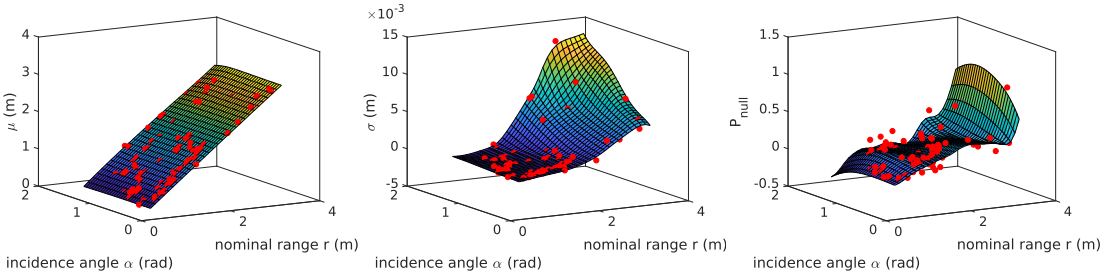


Figure 3.11: Regressed functions for μ, σ, and P_{null}.

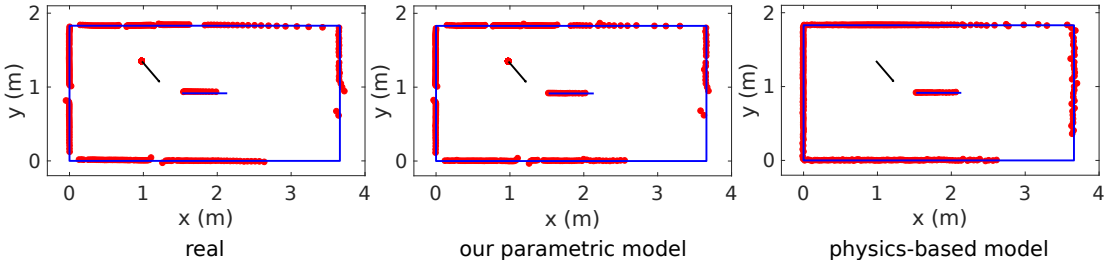


Figure 3.12: Real and simulated scans.

Empirical risk versus test size for the nonparametric and parametric methods are shown in Figure 3.14b. Sample histogram predictions are shown in Figure 3.13. NP-Reg makes noisy predictions, which is appropriate since the true histogram is also noisy at

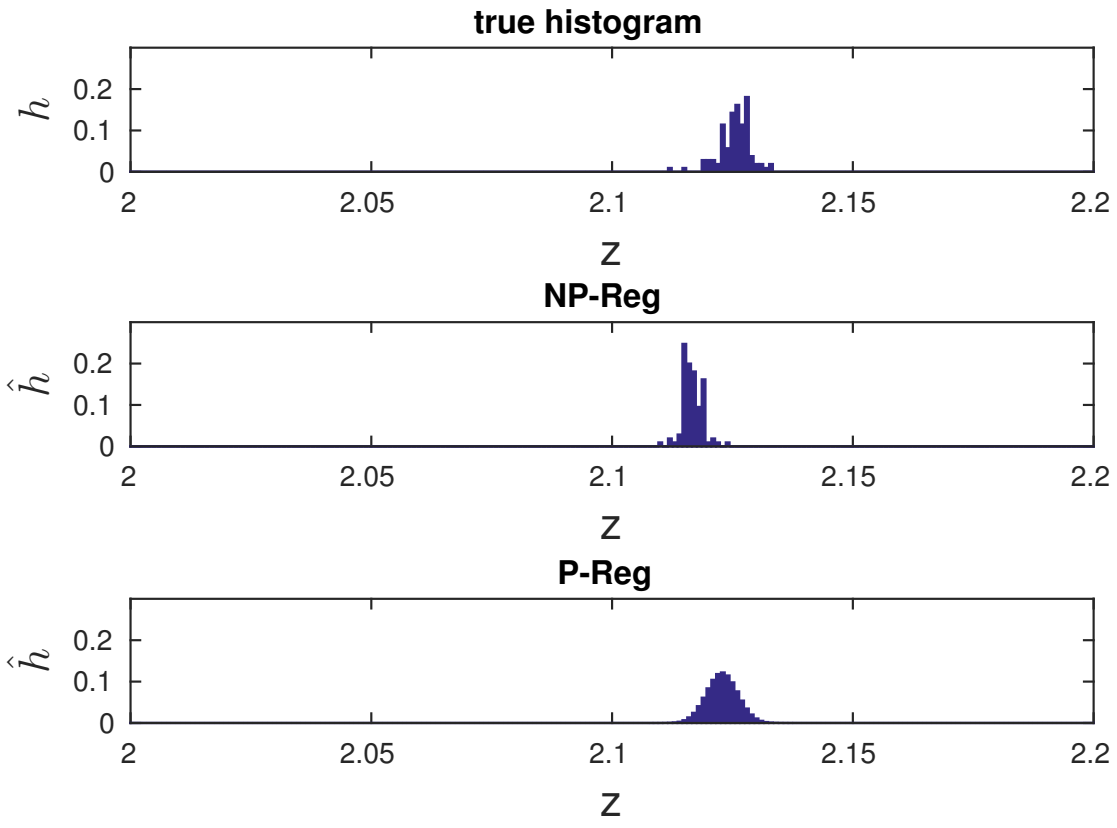
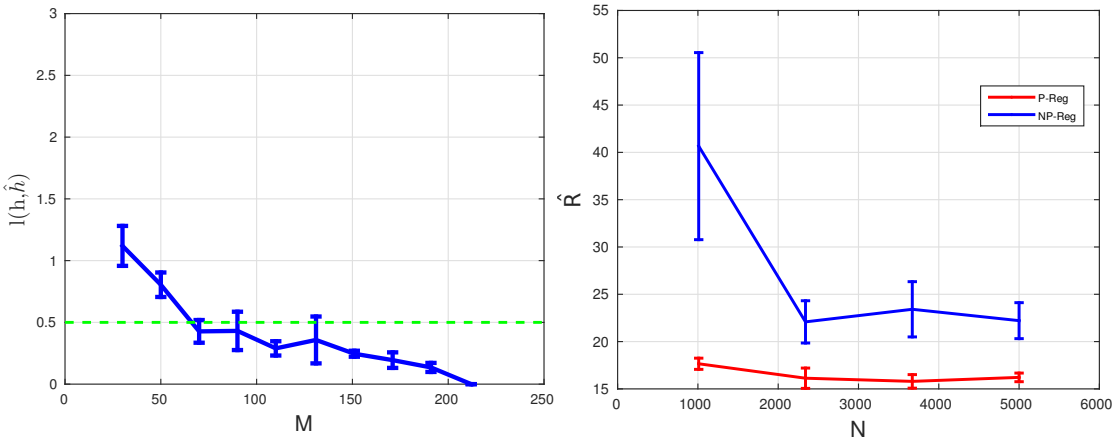


FIGURE 3.13: Example histogram predictions.

a similar scale. P-REG makes smooth predictions because of the Gaussian parametrization. Overall however, P-REG performs better in terms of risk. This result illustrates an important point. The main advantage of NP-REG over P-REG is the increased expressiveness. Parametric assumptions are not made during modeling, and the theory justifies regression at the level of distributions. This gain comes at a price, as nonparametric procedures generally require more training data. This case supports the intuition that a parametric model is preferable when it is a good approximation. Because we made a careful choice of the sensor model parameters, P-REG requires less data to learn. The risk is low to begin with. NP-REG starts at high risk, and converges slower. It will eventually achieve low risk with more data, but the increased expressiveness is unnecessary. Another reason for P-REG's better performance is the bearing-specific regression. For NP-REG, there was not enough data for bearing-specific regression, and so we pooled together data from all bearings. This is visible in that, while the histogram predicted by NP-REG is similar in shape to the real histogram, it is slightly shifted along the horizontal axis, accounting for the error.



(A) Loss $l(h, \hat{h})$ versus observations M. (B) Empirical risk $\hat{R}$ versus number of states N.

FIGURE 3.14

3.5.2 Application-level simulator evaluation

We present the utility of the sensor models for developing applications in this subsection.

Detection

For this application we used the parametric sensor model optimized on training data from Section 3.5.1. As a baseline Lidar simulator, we used the physics-based sensor model (Equation 3.6).

For the detection application, we considered a setup in which the scene of objects S consisted of an object representing boundary walls, o_{walls}, targets $\{o_{\text{target},j}\}$, and obstacles $\{o_{\text{obstacle},j}\}$. The boundary walls were in the shape of a rectangle. The sensor was at the center of the rectangle, and pointing in the direction of the $x-$axis. The targets and obstacles were straight lines. All targets were lines of the same length (0.61 m), while obstacles were of different lengths. The application task was to detect all the targets in the scene. The task was representative of finding simple features in range scans. Since the robots we used functioned as forktrucks, the targets detected represented objects to be picked up, while avoiding obstacles. We generated a set of scenes $\{S_i\}$, $i = 1 : N$ as follows. In simulation, we generated a large number (10^6) of candidate scenes by sampling from a distribution of scenes. From these, we selected $N = 50$ diverse scenes, using non-maximal suppression. We set these scenes up in reality, and logged observations to obtain the real dataset, $D = \{x_i, z_i\}$. The sensor pose $q_{\text{sensor},i}$ was the same at all scenes. A simulated dataset $\hat{D} = \{x_i, \hat{z}_i\}$ was generated using the Lidar simulator. We generated simulated datasets both for our approach, and the baseline.

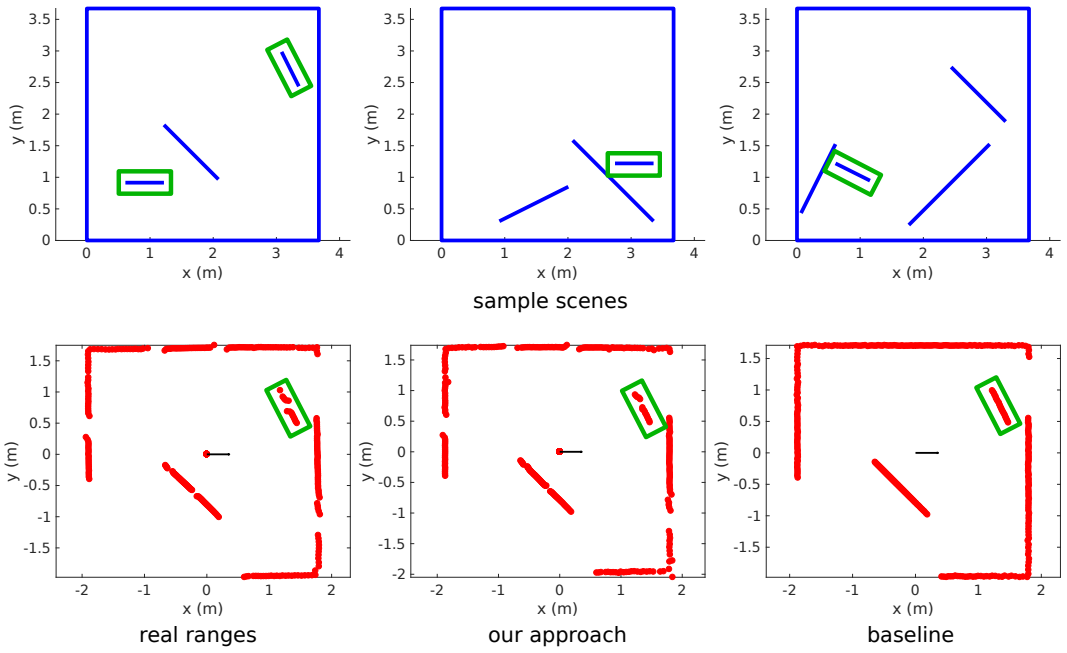


FIGURE 3.15: Top, sample scenes in the detection dataset. Below, sample scans in the detection dataset. The target is the green line. The black arrow shows the robot pose.

We are particularly interested in how simulators compare to reality for naive algorithms than high quality ones. We want the simulator to rapidly guide developers towards algorithms that will work well on the hardware with less (or no) changes. The detection algorithm A was a line-finding algorithm that starts at a point and grows a line in either direction. The algorithm parameters α included the maximum inlier distance of a point to a line, and the minimum number of points in a candidate line. The algorithm objective $J(\alpha, D)$ was the negative of the $F1$ score. The $F1$ score is a standard metric for detection tasks, and is defined in terms of the precision and recall. These quantities are calculated over a dataset as

$$\text{precision} = \frac{\#\text{ true positives detected}}{\#\text{ positives detected}}, \text{ recall} = \frac{\#\text{ true positives detected}}{\#\text{ positives in the dataset}},$$
$$F1 = \frac{2\,\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}, \; J = -F1.$$

We computed the application-level loss for two algorithm parameters, shown in Figure 3.16. The loss for our simulator is uniformly lower than that of the baseline. We can quantitatively make the claim that our simulator is better for developing the detection algorithm than the baseline.

We also wanted an insight into specific ways in which our simulator is better for application development. For this purpose, we performed another, qualitative evaluation. We compared application development trajectories in simulation and in reality, as was

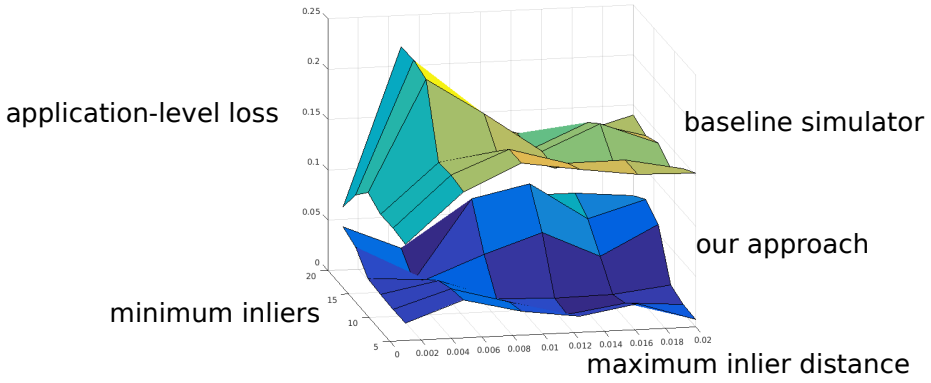


FIGURE 3.16: Application-level simulator loss is uniformly lower for our approach.

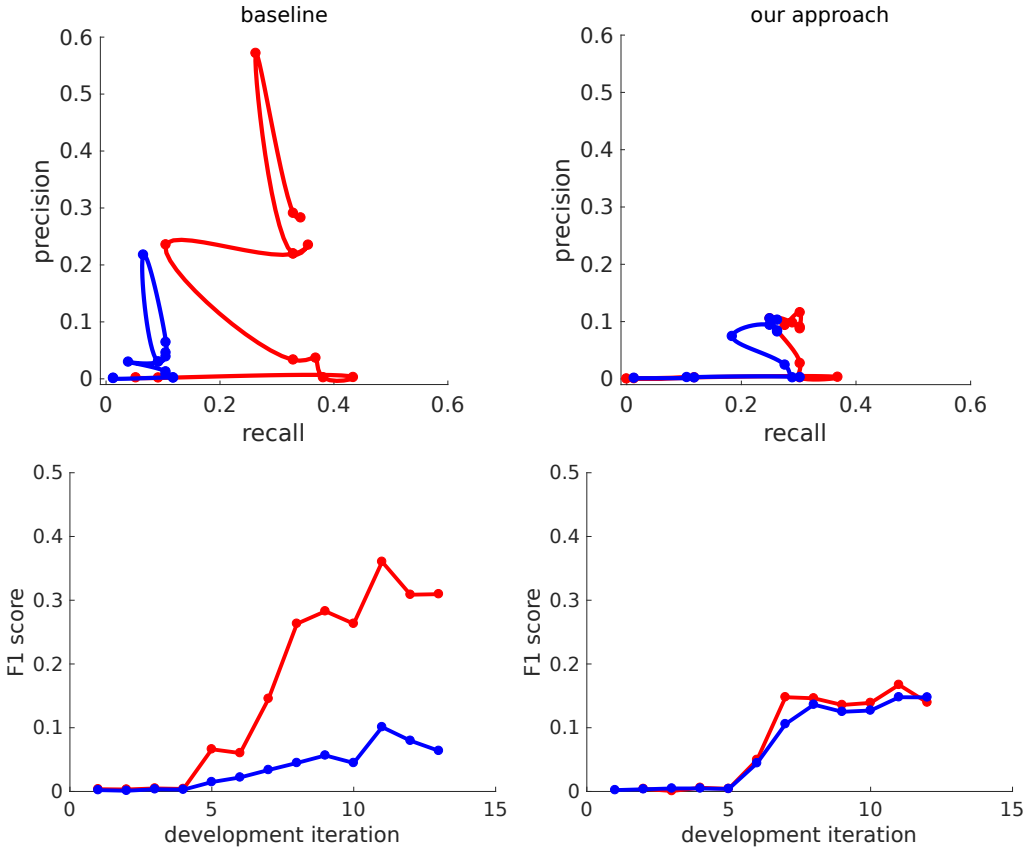


FIGURE 3.17: Comparing development paths between reality and simulation, for the baseline and our approach. Plots in red are for simulated data, while plots in blue are for real data.

also described in Section 2.3.3. A start configuration, α_0, of the detection algorithm was chosen with all checks for false positives disabled. This was meant to mimic a first implementation. We then tuned parameters based on the following strategy: first increase recall, and then increase precision (at some cost to recall). At any stage, the performance of the algorithm is a point in the precision-recall curve. With continuing development, the algorithm can be imagined to trace out a trajectory in the precision-recall space. This trajectory can be plotted for simulated and real datasets and is a visual aid. Figure 3.17 shows these trajectories for an algorithm developed on the baseline simulator. As is evident, performance on the baseline is overly optimistic. We repeated the exercise by performing development on our learned simulator. The trajectories are plotted in Figure 3.17. Although it starts at the same point, the trajectory on real data (the blue curve) is different in either case, because algorithm development proceeds differently. For example, preprocessing the range data before detection was a crucial step for higher recall, a step suggested by the learned simulator but not by the baseline. Similarly, tolerance parameters for outlier rejection chosen on the baseline were too aggressive on the learned simulator.

The application development paths resulted from a specific choice of a strategy. For detecting lines in range scans, we of course know how to construct a robust algorithm. The point of this exercise is that we can expect the learned simulator to be equally valuable on problems for which the developer is less skilled, or for which less is understood about the nature of a good solution. The benefit of our simulator over the baseline is quantitatively demonstrated for other algorithm parameters in Figure 3.16.

Registration-based localization

For this application we used the parametric sensor model optimized on training data. The baseline simulator is the same as that described in Section 3.5.2.

For the registration application, we considered a setup in which the scene S consisted of a L-shaped corridor. We selected velocity controls that commanded the robot to follow a non-trivial trajectory $\{u_t\}$, $t = 1 : T$ for T timesteps. At each timestep, an observation $\{z_t\}$ is received. An extended Kalman filter (EKF) was used to obtain pose estimates $\{q'_t\}$. The motion update of the EKF leaves the pose unchanged, but adds noise to the pose uncertainty. The noise covariance R for the motion update was experimentally

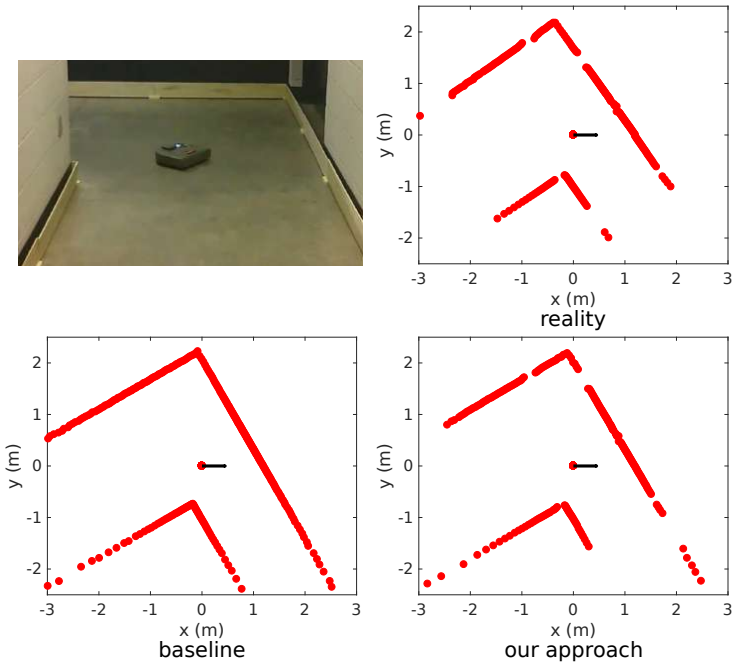


FIGURE 3.18: Range scans in the real and simulated corridors.

calculated to be

$$R = (V\Delta t)R_1 + (\omega\Delta t)R_2,$$
$$R_1 = \begin{pmatrix} 1.8 & -0.56 & 2.2 \\ -0.56 & 1.2 & -1.8 \\ 2.2 & -1.8 & 5.9 \end{pmatrix} \times 10^{-4}, R_2 = \begin{pmatrix} 0.62 & -0.072 & 0.038 \\ -0.072 & 0.79 & 0.57 \\ 0.038 & 0.57 & 5.7 \end{pmatrix} \times 10^{-4}.$$

Where V, ω are the linear and angular velocities in the robot's local coordinate frame. We chose a form that depended on the robot's dynamics. The noise is proportional to the linear and angular displacements. Such a form ensures that there is no contribution to the noise if there is no displacement in some time interval. The observation update performs the pose estimation, by registering the Lidar points z_t to the map of the corridor.

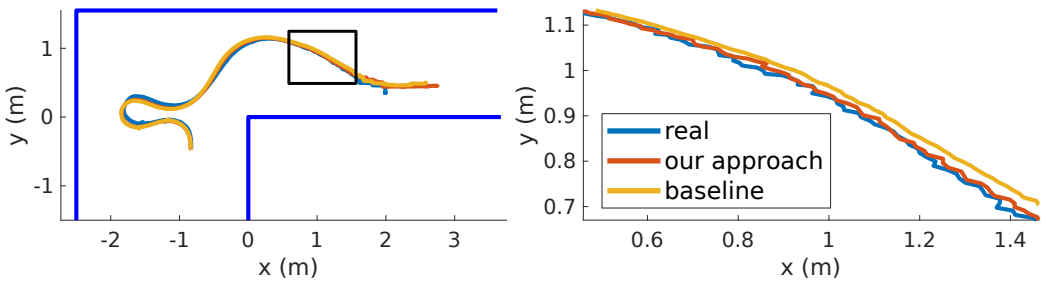


FIGURE 3.19: Example trajectory estimates from the registration-based filter. Detail of a section is also shown.

A run of the robot and filter results in the trajectory and its estimate, $\Xi = \{(q_t, z_t, q'_t)\}$. For each run, the corridor scene and the control commands $\{u_t\}$ were the same. The initial position was $q_{0,i}$ was different for each run i. This was accounted for by the initial position uncertainty, which was determined experimentally to be $\text{diag}(0.28\,0.28\,3.4) \cdot 10^{-3}$. The true pose of the robot was obtained using an OptiTrack[1] motion capture setup. This resulted in a real dataset $D = \{\Xi_i\}$, $i = 1 : N$. We collected data for $N = 20$ runs. We then generated a simulated dataset, $\hat{D} = \{\hat{\Xi}_i\}$. The true pose is trivially available in simulation. The same EKF was run in simulation and reality to generate pose estimates. We generated simulated datasets both for our approach, and the baseline simulator. Examples of estimated trajectories are in Figure 3.19.

The registration algorithm A worked by running an optimizer to minimize the error between observations z and the map. Error for a single Lidar point is defined as the nearest distance to the map. Error for a scan is the sum of errors for the Lidar points. In our implementation the optimizer was gradient descent. The optimization variables were the robot pose, and gradients were calculated numerically. The output of the registration algorithm was used in the observation update of the EKF. There are a number of parameters in the error computation. However, for tuning real-time localization, we picked the algorithm parameters α to be the number of optimization iterations. Each instance of optimization runs for a maximum number of refining iterations. The optimization may converge in fewer iterations. There is a trade-off here: a large number of refining iterations results in a better pose estimate, but the robot will have moved more and the estimate gets delayed. The algorithm objective J was the pose estimation error, averaged over the runs:

$$\text{error}(\Xi) = \frac{1}{T}\sum_{t=1}^{T} \|q_t - q'_t\|,$$

$$J = \frac{1}{N}\sum_{i=1}^{N} \text{error}(\Xi_i).$$

Plots for the simulation loss $l(\alpha)$ versus α are shown in Figure 3.20. Once again, performance on the learned simulator is a better approximation to real performance, than performance on the baseline simulator. The application-level risk for our simulator is 0.008 and 0.014 for the baseline. We also calculated the average time for the registration algorithm A to converge. Plots of the convergence time versus α are shown in Figure 3.20. Time taken by scan matching remains low on the baseline simulator, but increases on the learned simulator and real data because of increasing refining iterations. Encoder information arrives at $0.03s$ on the robot. If the baseline simulator were trusted and the

[1] https://optitrack.com/

refining iterations set to 100, scan matching would take $0.1s$ on average, and a number of encoder messages would queue up during the computation in a single thread.

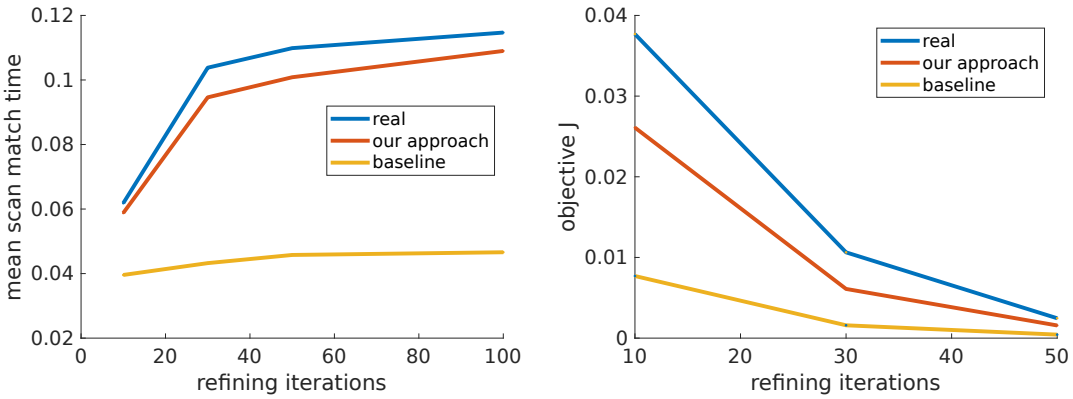


FIGURE 3.20: Algorithm performance measures versus the refining iterations of the scan matching algorithm.

Particle filter-based localization in a dynamic environment

In the observation-level simulator evaluation for the planar Lidar sensor, we saw that a well-chosen parametric sensor model is better than the nonparametric model. The nonparametric model performs well when the observation distributions are complex with unmodeled state, as this application demonstrates.

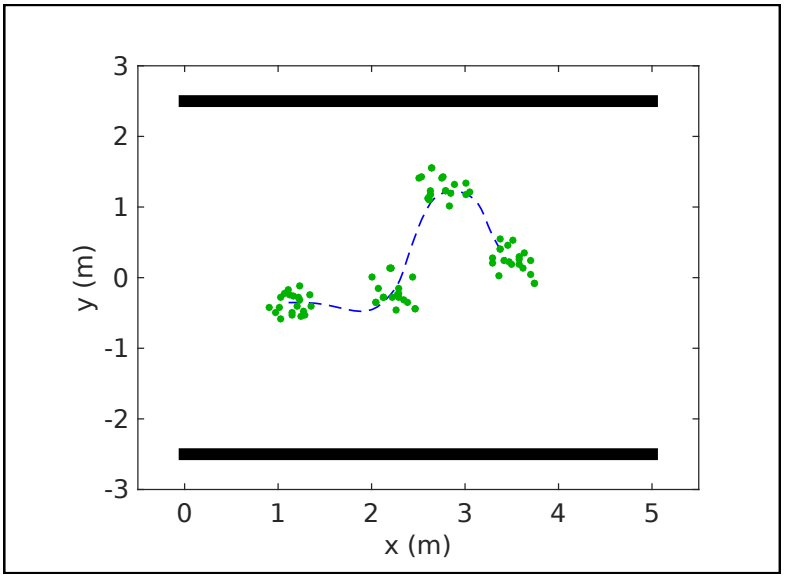


FIGURE 3.21: Snapshots from the operation of a particle filter. The nominal map is shown in black. Unmodeled objects are not shown. The true trajectory is a blue dotted path. Particles are shown in green.

Unlike the previous two sections where we compared simulation against reality, in this section we evaluate sensor models for an application. Our experiments for this section

were conducted in simulation. The application was particle filter-based localization in a dynamic environment. The setup was as follows. The scene S consists of a straight, wide corridor of dimensions 5×5m. In Figure 3.21, the walls of the corridor is depicted as solid black lines. The scene also consisted of unmodeled objects $\{o_{\text{unmodeled},t}\}$. By unmodeled, we mean that the map used by the particle filter does not contain the $\{o_{\text{unmodeled},t}\}$. However, they affect any sensor observations logged in the scene. The unmodeled objects represent dynamic objects, such as moving people, which can occur in indoor environments. The motion of the unmodeled objects were modeled as random walks. While the unmodeled objects affect sensor observations, it may be impractical to track the pose of all such objects. Therefore, one approach to state estimation is to include the effect of unmodeled objects in the sensor model. The unmodeled objects are depicted in gray in the top half of Figure 3.21.

We used NP-Reg as a sensor model, and compared it with the sensor model described in [29]. We refer to the latter as classic raycast model. In the classic raycast model, the probability density of an observation z at state x is defined as

$$p(z|x) \propto \eta \exp\left(-\frac{(z-r)^2}{2\sigma^2}\right) + (1-\eta)\exp(-\lambda r). \tag{3.7}$$

Where r is the nominal range to a target in the map. The classic raycast sensor model is thus a mixture of a Gaussian distribution, and an exponential distribution, with mixing weight η. The Gaussian distribution has standard deviation σ, and represents random noise around the nominal range. The exponential distribution has rate λ, and represents observations from dynamic objects. The complete sensor model described in [29] also contains terms representing null readings, and uniformly distributed readings. Since these effects were absent in the sensor used in our simulated scenes, we did not include them in the classic raycast model used as the baseline.

To optimize the sensor models, we obtained training data in simulation as follows. We set up the corridor with the dynamic objects in simulation, and logged sensor observations. We obtained data at a total of $N = 3660$ features. The features used were those described in Section 3.4.2, the nominal range and angle of incidence to a target. We simulated $M = 50$ observations at each feature, and the observations included the effects of the unmodeled (and dynamic) objects. We simulated 5-10 dynamic objects each, in the bottom and top half of the corridor. Both sensor models were optimized on the same training data. For NP-Reg, the hyperparameters were the kernel bandwidths, as described in Section 3.4.2.2. For the classic raycast model used as baseline, the parameters were (η, σ, λ) (Equation 3.7).

Observation model	Classic raycast model	NP-Reg
Mean pose error (m)	0.004 ± 0.041	0.002 ± 0.020

Table 3.3: Filter errors

Having described the sensor model, we describe the particle filter used as the algorithm A. A particle filter maintains a set of particles and associated weights, $\{(q_i, w_i)\}$, $i = 1 : N_p$, that are updated in response to sensor readings. When an observation is received, an observation update is performed. The poses of the particles are unchanged, but the weights are updated. Weights are proportional to the probability of the observation, $w_i \propto P(z|x_i)$. The observations depend on the state x, which includes the pose of the sensor and the map, but not the unmodeled objects. Updating the weights is followed by a resampling step. Further details of the particle filter can be found in [29].

To test the particle filter with either sensor model, we considered a sensor traveling along an S-shaped path, depicted in Figure 3.21. The path was 4.3m in length, and 52sec in duration. We logged 50 sensor observations along the path. We also logged the motion control commands (linear and angular velocities) input to the sensor. These control commands were available to the particle filter. Simulated noise in the motion was the source of uncertainty in the sensor pose. We initialized the particle filter with 200 particles. These were sampled from a uniform distribution, of radius 1m, and 10deg around the true initial pose. For the resampling step after the observation update, a low-variance resampler was used. Since the particle filter is stochastic, we ran the filter 30 times with either sensor model. Errors in estimating position are shown in Table 3.3.

The particle filter run with our sensor model had lower error, and we briefly explored why this was so. Overall particle filter performance is a non-trivial function of a number of variables, in addition to the observation model. As a simplification, we focused on a snapshot of particles, for a single time instant, shown in Figure 3.22, top. The true robot pose is a red arrow, and a set of particles is shown in green. Of these, we focused on a single particle, whose pose is also shown in green. This particle's pose was close to the true pose, and we would expect it to be assigned a high weight. We further concentrated on a single reading, at a specific bearing of the range sensor. As depicted, the observed range (red dotted line) was less than the nominal range (green dotted line), due to an unmodeled obstacle. The bottom half of Figure 3.22 shows the predicted distributions. The observation had a low probability under the classic model, despite the exponential term. On the other hand, the distribution predicted by NP-Reg had peaks at both the nominal and observed reading. Note that the same data (consisting of unmodeled objects) was used to train both models. It is possible to reason about dynamic obstacles in more detail and build a better parametric model, as in [34]. The point of the example is that NP-Reg required no modifications for this specific

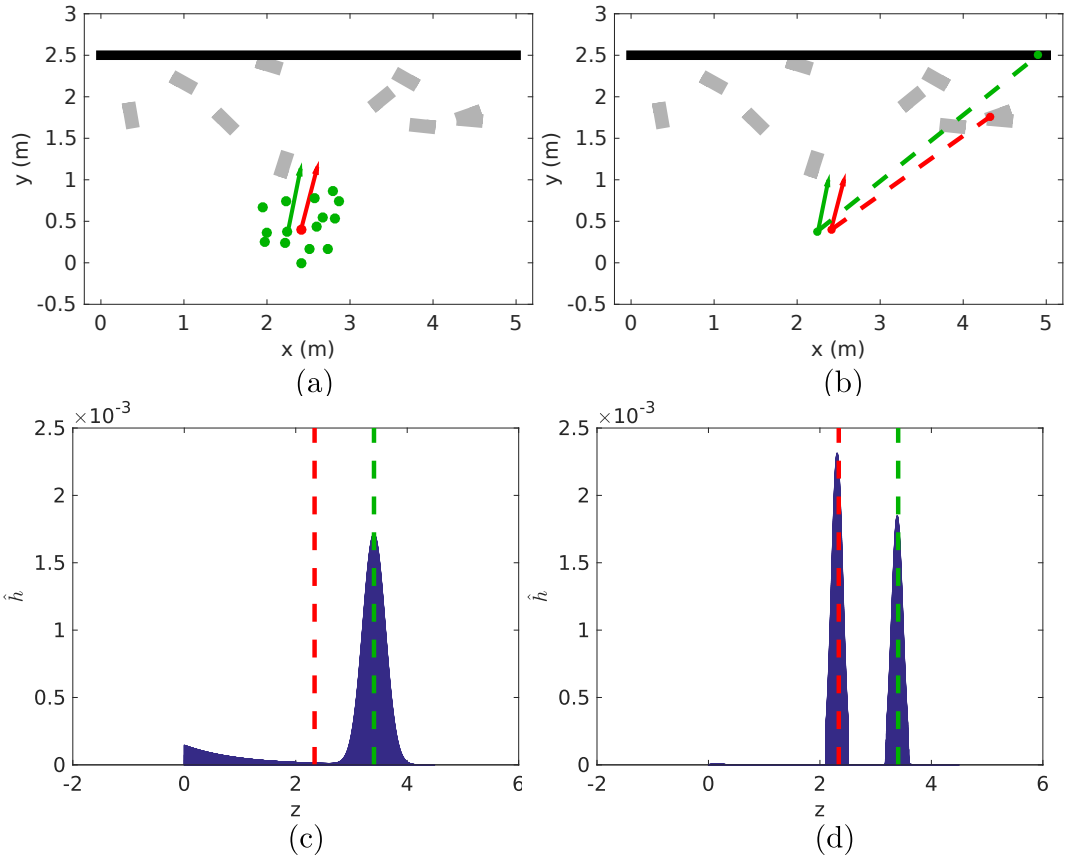


FIGURE 3.22: Observation probabilities under the different models. Figure 11(a) depicts the environment. We focus on a specific particle. Figure 11(b) depicts the nominal and observed range readings. Figure 11(c) depicts the histogram predicted by the classic model. Figure 11(d) depicts that predicted by NP-REG.

field	value
frequency	100 Hz
number of bearings	1
[min range, max range]	$[0, 0.2]$ m
resolution	$1e-4$ m
number of histogram bins B	2001

TABLE 3.4: Sensor properties

situation. Being a nonparametric procedure, it adapts to the training data to predict realistic distributions.

3.5.3 Magnetic field sensor

To illustrate the flexibility of NP-REG, we also modeled a magnetic field sensor [2]. It consists of a small probe whose pose is measured when it is placed within a generated

[2] https://www.ascension-tech.com/products/trakstar-drivebay/trakstar/

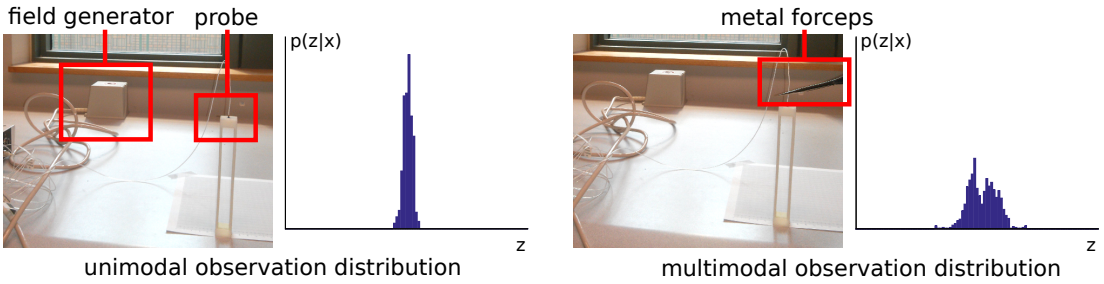
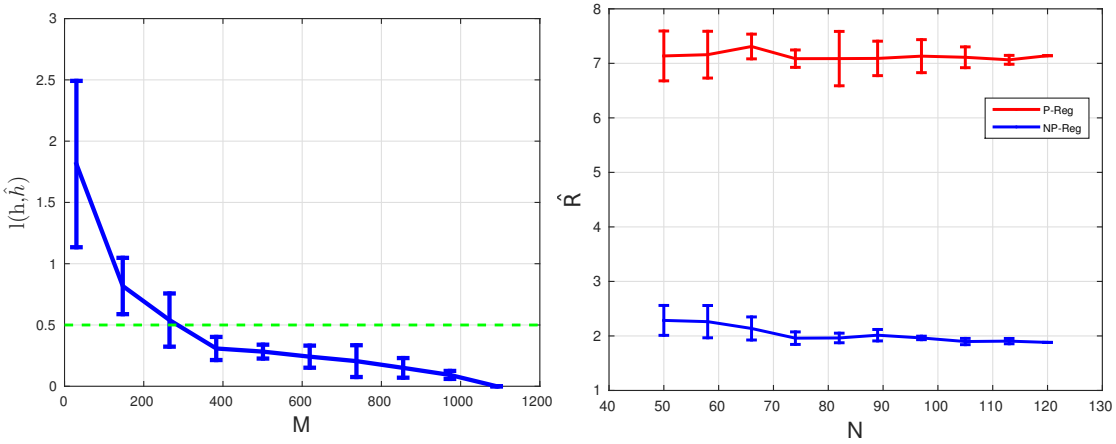


FIGURE 3.23: The effect of unmodeled state on the observation distribution. Bringing a metal object near the sensor affects the field, making the observations noisier and non-Gaussian.

magnetic field. The small size of the probe makes it suitable for medical applications. For simplicity, we consider only one axis of operation. Other details are in Table 3.4. The state x was the coordinate value along the axis, and the observation z is the state measurement. Data was collected by moving the sensor along the axis. Even within a workspace of 20 cm, the sensor shows interesting behavior. Referring to Figure 3.23, when the environment around the sensor is stationary, the observation distribution is well captured by a Gaussian. In reality, however, there could be metal objects around the probe that disturb the magnetic field, resulting in noisy distributions. These could be, for example, metallic surgical tools that are brought in the vicinity of the probe. It is difficult in practice to account for all such foreign objects in the state x, so they can be considered the unmodeled states y, which affect the observation distribution as noise. In our experiments, we deliberately introduced unmodeled states. A pair of metallic forceps was waved near the sensor while collecting data.



(A) Loss $l(h, \hat{h})$ versus observations M. (B) Empirical risk $\hat{R}$ versus number of states N.

FIGURE 3.24

The number of observations per state M was chosen in a similar manner as described for the laser sensor (Section 3.5.1). The loss in estimation versus M is shown in Figure 3.24a.

Reaching the threshold required about 300 samples. The more complicated distributions of the field sensor require a larger M for nonparametric modeling than the laser.

The choice of a baseline P-Reg was a Gaussian distribution, with a kernel smoothing procedure for the regression step. For both procedures, empirical risk on test data versus N is shown in Figure 3.24b. Not only does NP-Reg have lower risk in this case, it continues to improve with increasing data. The risk for P-Reg converges quickly, to a higher value. The hard-to-model distributions benefit from the nonparametric modeling. Sample histogram predictions are in Figure 3.25. NP-Reg comes closer to capturing the multi-modality of the observation distribution.

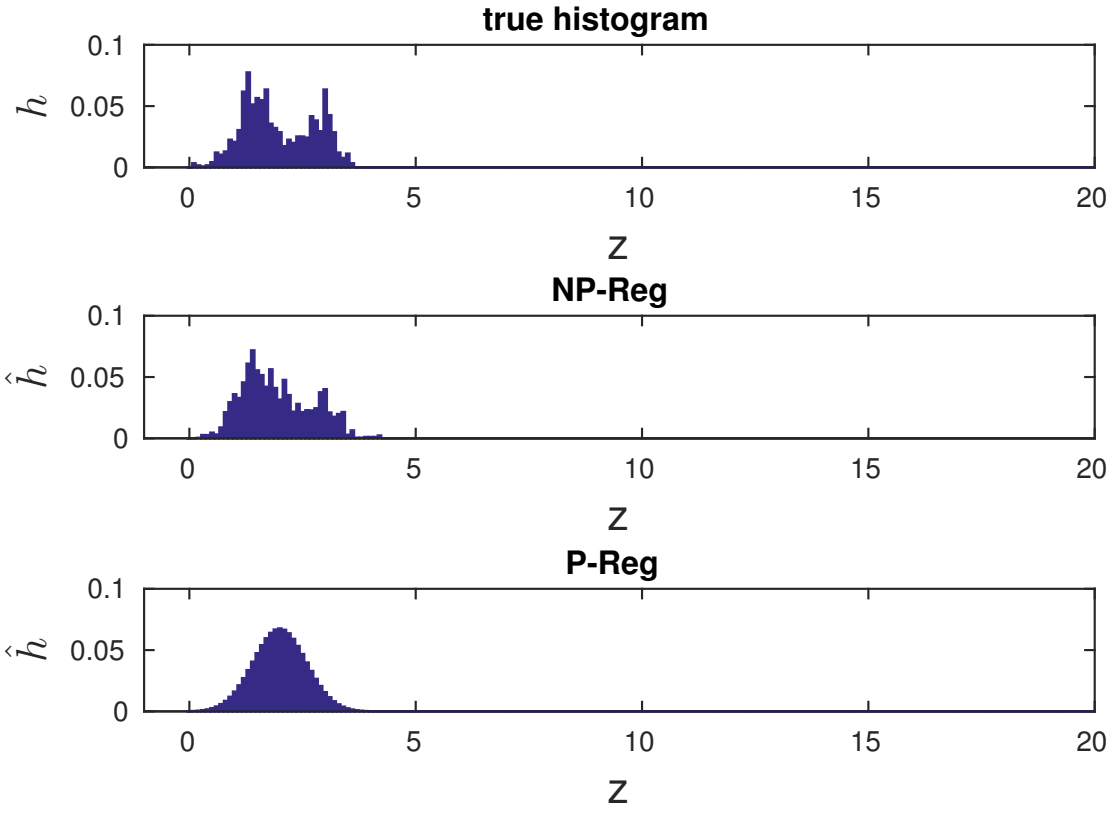


Figure 3.25: Example histogram predictions.

Chapter 4

Off-Road Lidar simulation

4.1 Introduction

Lidars are a key technology that have helped advance achievements in outdoor robots. As sensors, they provide robots with observations of the environment. A large amount of software is developed to process Lidar observations, for example, algorithms for state estimation and recognition. Such software should be able to deal with the noise and scale of Lidar data, among other requirements. The reliable and high-quality performance of perception software is essential for good decision making in the higher levels of autonomy. As the scale of robot operations grows, so do the challenges of software development. It may be difficult, unsafe, or expensive to develop software on enough real-world environments. Simulators have attracted attention as a solution to these problems. They are generative, and so conditions that are difficult or expensive to test in reality may be queried in simulation. Full state information is available in simulation, which is not the case in reality.

In this chapter, we focus attention on the off-road case. We present some motivating examples for building high-fidelity off-road Lidar scene simulators. Consider developing software, for tasks such as terrain modeling [1] and virtualized reality [2], for off-road mobile robots. Real-world development can be very expensive, with potential delays due to unforeseen weather and hardware issues. Further, some tests amount to repeated robot runs under different software or hardware configurations. Such problems are an ideal fit for solution by simulation. Another example is the use of Lidars in autonomous helicopters. In [3], a convolutional neural network (CNN) was trained to detect landing zones from Lidar observations. Real data was obtained from expensive flight tests. Synthetic data was obtained in [3] using a simple simulator. It was noted that *a priori* modeling of Lidar returns from vegetation was difficult, and that simulator evaluation

was a challenge. In this work, we build realistic Lidar models from data. We also perform data-driven evaluation of simulators, comparing against real data from complex scenes. Synthetic data from the simple simulator in [3] was found to be useful in parameter selection; a high-fidelity Lidar simulator can be of greater value in training learning algorithms.

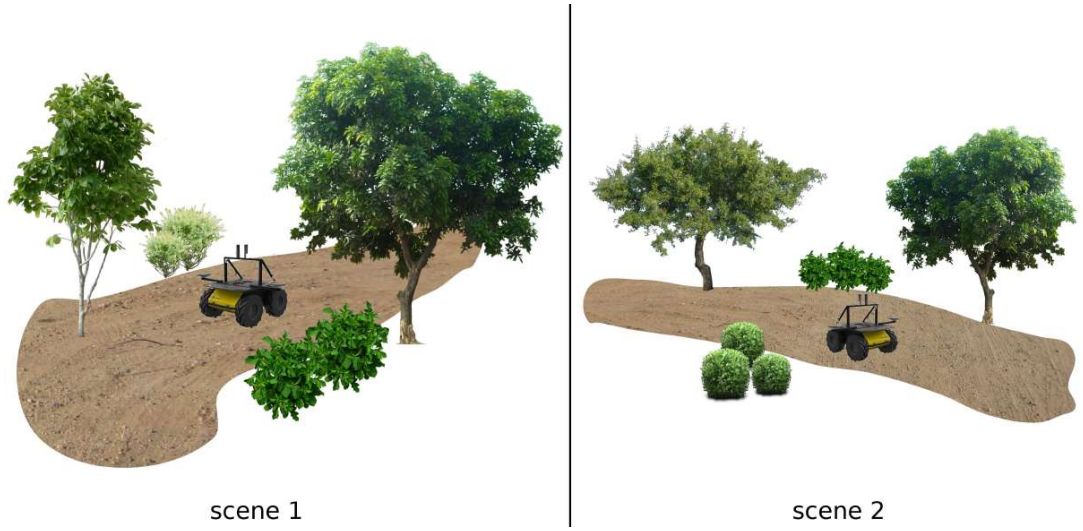


FIGURE 4.1: The domain of our work is off-road Lidar scene simulation. We may collect training data in scene 1, and want to simulate in a query scene 2.

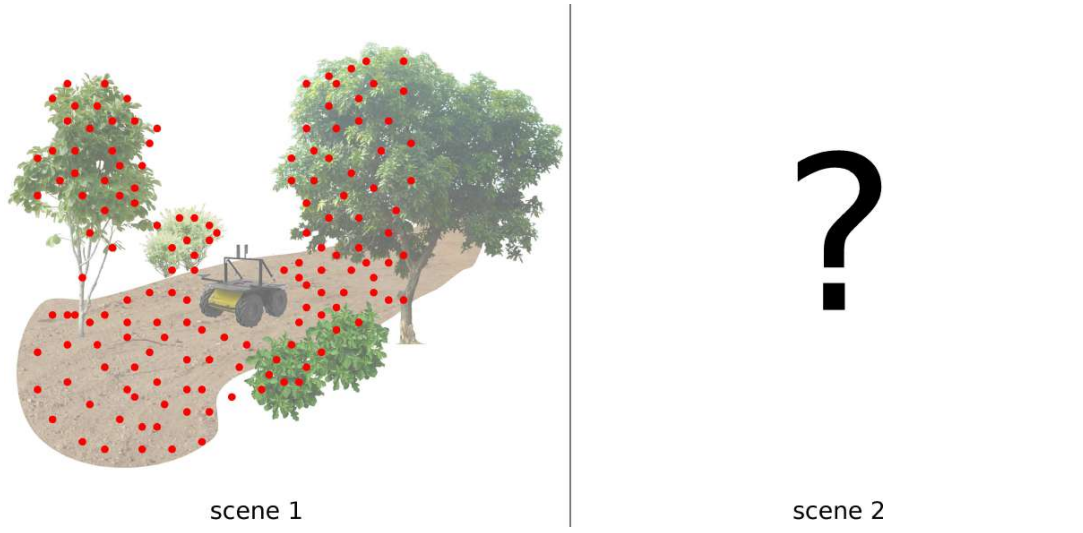


FIGURE 4.2: Data logs from the real scene have high fidelity on the training data. They typically fail to generalize to the training distribution, and have limited expressiveness.

In urban scenes, where objects with planar structure are be more common. For such objects, raycasting followed by additive noise may be a good sensor model. Off-road environments contain terrain such as uneven roads, trees, shrubs, and other vegetation. Effects such as pass-throughs and mixed pixels are seen in Lidar data. Modeling such terrain is an added challenge. A useful simulator has the qualities of being high-fidelity,

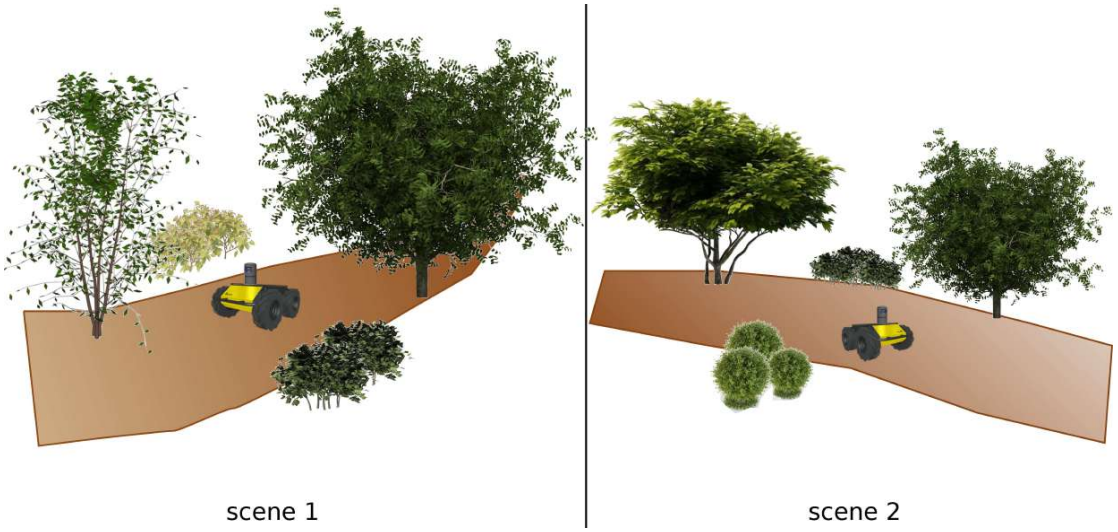


FIGURE 4.3: Open-source simulators typically are high expressive. However, the fidelity of simulation is often insufficiently evaluated.

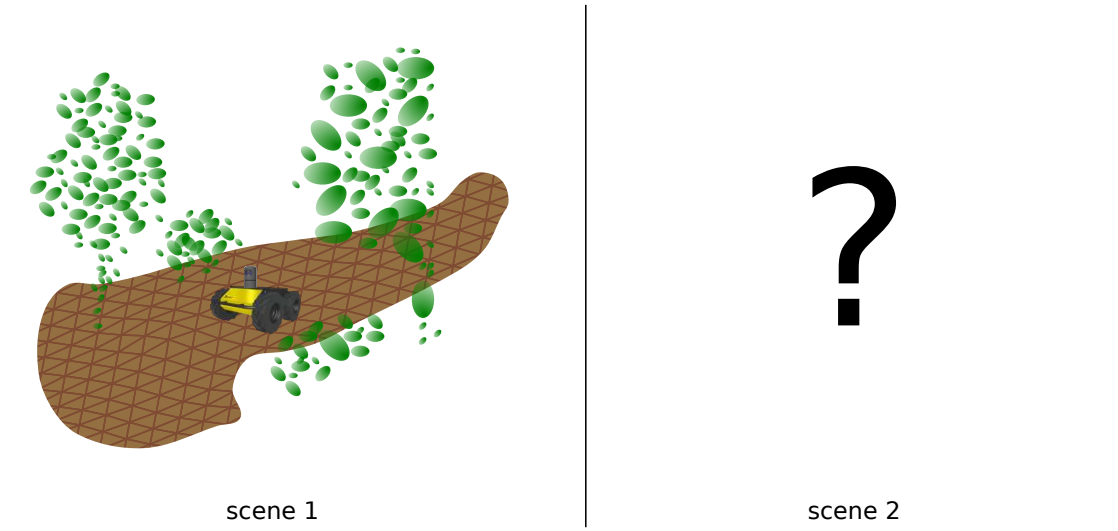


FIGURE 4.4: The hybrid geometric simulator models terrain with a combination of surface meshes and Gaussian distributions. These are fit to data. However, it is geo-specific, and new scenes cannot be simulated.

and expressive. A high-fidelity simulator is one whose output is realistic. In terms of the notation in Chapter 2, it has low simulation risk. An expressive simulator is one in which different real-world scenes of interest can be reconstructed. We discuss the available options for off-road Lidar simulation. The first option is to use data logs for simulation. Dataset in the form of sensor poses and observations, $\{q_{\text{sensor},i}, z_i\}$ can be logged in a training scene. At a query pose close to a pose in the dataset, such as $q_{\text{sensor},i}$ for some i, the real observation z_i is simply played back. In this sense, data logs have high fidelity. Of course, for poses that are far from those in the dataset, the simulation loss will be high. This is why data logs are useful only in preliminary stages

of software development. Data logs have low expressiveness, as new scenes cannot be composed. See Figure 4.2. A second option for simulation is an existing general-purpose robotics simulator, such as Gazebo [10]. The fidelity of such Lidar simulation is, at worst, low, and at best, untested. There exists little work in systematically comparing simulated data to real ranges. It in unclear how calibrated the models are to real data, at multiple levels: the sensor models, and the mesh model structures. On the other hand, such simulators are highly expressive: 3D models of vegetation available online may be obtained for use. New scenes may be constructed using interfaces designed to simplify a user's effort. See Figure 4.3. A third option is a simulator based on the hybrid geometric model presented in [18]. A sensor model which takes into account non-idealities of Lidar data was considered. Terrain was modeled using a combination of surface meshes, and permeable volumetric elements. The models were fit to real Lidar data, and high-fidelity simulation was demonstrated. However, the simulator was 'geo-specific': tied to the scene where data was collected. The models were blind to scene semantics. It was not possible to compose new scenes, and expressiveness was low. See Figure 4.4. The three options, charted on a graph of expressiveness versus fidelity, are shown in Figure 4.6. Our approach is to build a simulator that is both expressive and high-fidelity. We use the hybrid geometric models as our starting point, inheriting its realism for off-road Lidar simulation. We then propose to assign semantic meaning to the point clouds, and extract scene primitives from the training data. A primitive is associated with a class, such as tree or shrub, and a sensor-realistic geometric model. New scenes can be composed using the primitives. See Figure 4.5.

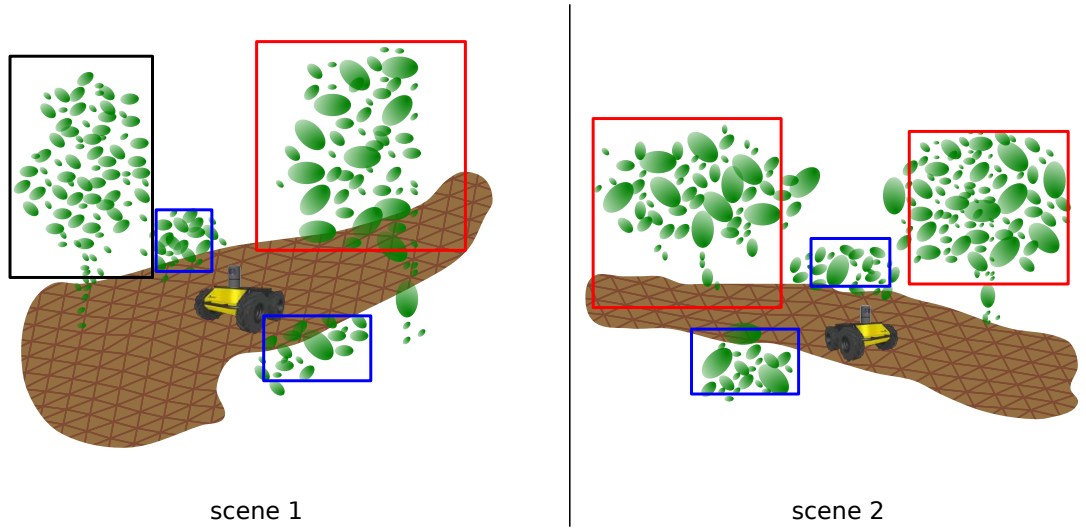


FIGURE 4.5: The proposed simulator adds semantic information to the hybrid geometric model, in order to extract primitives. The primitives can be used to compose a new scene.

The outline of the chapter is as follows. In Section 4.2 we discuss related work. We

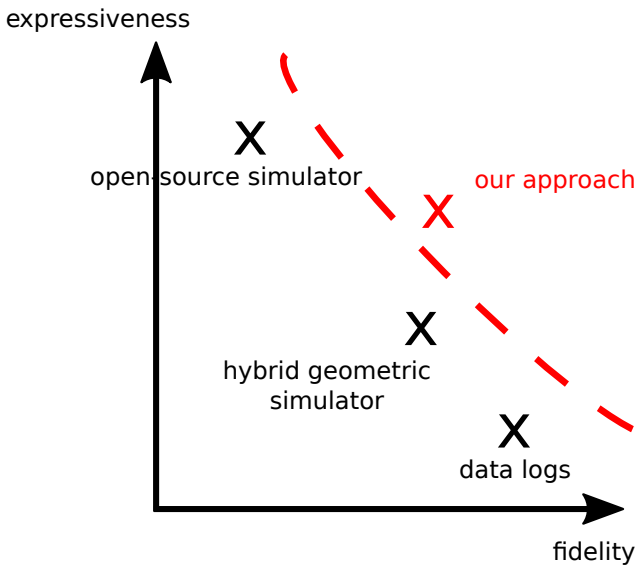


FIGURE 4.6: Existing options for off-road Lidar scene simulation are either high-fidelity, or expressive. Our proposed solution is both.

summarize work that led up to the hybrid geometric models for Lidar simulation. We make connections to recent research in robot and computer vision. In Section 4.3, we present an overview of our approach to Lidar scene simulation, followed by details. In Section 4.4, we evaluate our approach with real-world experiments.

This chapter contains work from the following.

Tallavajhula, Abhijeet, Meriçli, Çetin, and Alonzo Kelly. "Off-Road Lidar Simulation with Data-Driven Terrain Primitives." ICRA 2018.

Tallavajhula, Abhijeet, Meriçli, Çetin, and Alonzo Kelly. "Application-level Evaluation of Off-Road Lidar Simulation." Manuscript.

4.2 Related work

Interest in simulators has been growing. Gazebo [10] is an open-source simulator that was promoted, and received support for development, as the platform for the simulation track of the DARPA Robotics Challenge [4]. There have also been government projects to develop simulators, such as the VANE platform [15]. For computer vision, simulators based on game engines, such as Unity [52] and Unreal Engine 4 [53], have gained popularity for sophisticated scene rendering, and the ability to create detailed virtual worlds. Aside from realism, development of such simulators has focused on a number of factors such as software architecture, user interfaces, communication between modules, visualization, and others. In our work, we focus only on realistic models, and simulator evaluation.

A number of general-purpose robotics simulators [10, 11, 21] perform Lidar simulation. Most simulate observations by calculating the range to object models, and adding Gaussian random noise. Aside from the sensor models being simple, how to assign appropriate values to the parameters is unclear. While such a sensor model works for simulating Lidar returns from large, extended objects, off-road terrain such as vegetation present a challenge. Lidars interact with fine structure, such as leaves, resulting in effects such as pass-throughs and mixed pixels [16, 54]. The sensor model presented in [55], and the corresponding simulator [15], account for mixed pixels by raycasting multiple rays for each Lidar beam. Unfortunately, this approach is expensive when simulating observations from thin structures. In addition to a sensor model, off-road Lidar simulation requires a terrain model. The graphics community has paid attention to modeling plants, such as the generative model in [56]. However, there was no comparison to real vegetation, nor was it obvious how the model could be tuned to fit real observations. A method to reconstruct trees from real data was presented in [57]. However, the focus was on visually realistic trees. In a similar vein, [58] used domain knowledge of tree structure to guide an iterative reconstruction from data. Neither of these approaches considered modeling vegetation for the purpose of simulation.

An extensive evaluation of models for off-road Lidar simulation was carried out in [18, 59]. A hybrid geometric model was found to perform well for different types of terrain. The ground was modeled as a triangle mesh, and non-ground terrain as a set of Gaussian distributions. The sensor was modeled jointly with the terrain structure. Lidar effects such as pass-throughs were modeled by associating geometric elements with a hit probability. The models were fit to real data, and realistic simulation was demonstrated in [18]. We will refer to the resulting simulator as the hybrid geometric simulator. The hybrid geometric simulator is used as a base for our work, although, as noted in Section 4.1, it only simulates a fixed scene. Only query poses close to the training poses

would yield realistic results. No semantic information beyond ground and non-ground is attached to the models. Therefore, new scenes cannot be simulated. This is a major handicap for software development, in which we may want to test algorithms on new scenes before real-world deployment.

There has been recent work in simulation for testing robot perception algorithms. [23] presented a high-fidelity simulator for evaluation robot vision. A synthetic scene was created using Unreal Engine 4. The advantage of a simulator, in being able to test algorithms under varied simulated conditions, was demonstrated. A number of interesting trends were observed for SLAM algorithms in simulation. Testing of the same algorithms on real data, however, was limited. It was observed that performance in simulation was better than in reality, and noted that the performance trends in reality paralleled those in simulation. A simulated dataset for UAV tracking was presented in [22]. This too was on Unreal Engine 4, which was recently made open-source. The game engine is extremely flexible, allowing the creation of virtual worlds with varied characteristics. How closely the virtual worlds created in the dataset map to real worlds, however, was unaddressed. Similar to [23], the advantage of a simulator, in the performance of trackers under a number of different conditions and metrics, was demonstrated. However, how well the conclusions translate to real data was untested.

Simulated data has long been used in the computer vision community. Given the dominance of learning-based methods, gathering adequate training data can be a challenge. It may be difficult to collect large datasets, label them, or deal with different domains. The challenges are especially relevant for deep learning methods. For these algorithms, useful feedback during the entire software development path is less important. What matters is the final performance of an algorithm on some test data. The evaluation of simulated data is often in terms of the benefit to performance of a learning algorithm.

Synthetic data has been used for pedestrian detection [24, 60, 61]. In [24], for example, the task was to detect pedestrians in the field of view of a fixed camera. Not enough real data, with annotated pedestrians in the field of view, existed. The state in simulation consisted of synthetic pedestrians rendered in the real scene, using 3DS Max[1] from Autodesk. The appearance, locations, and poses of synthetic pedestrians were chosen heuristically. Simulation was evaluated at the level of algorithms. The performance (on real data) of a detector trained on purely synthetic data was compared against detectors trained on real data. In the supplementary material, it was noted that algorithm performance was critically dependent on the quality of simulation. While no quantitative results are presented for how the performance varied with simulator parameters, it did appear that some (e.g. sharpness of rendering) were selected to improve performance.

[1] https://www.autodesk.com/products/3ds-max/overview

SYNTHIA [27] is a recent synthetic dataset for semantic segmentation. Virtual urban worlds were generated using the Unity development platform. The simulation scenes were extremely expressive, consisting of diverse urban environments. How they were specified, and how well they mapped to real environments, however, was unclear. CNNs trained on a combination of real and simulated data were found to perform better, than those trained on real data alone. A similar conclusion was reached by [25], in the domain of object detection. In [25], synthetic images were appended to the real training images. Realistic simulation of texture and background was found to be unnecessary when there were enough real images. It was found to matter, however, when transferring algorithms to new domains (detecting novel categories). The importance of obtaining some amount of data from the target domain has also been demonstrated theoretically [19].

Work with similar motivations as [27], but for the task of indoor scene understanding, is presented in [28]. It used SceneNet [62], a synthetic dataset of annotated indoor scenes. OpenGL was used to obtain color and depth maps of the scenes from different viewpoints. Realistic sensor models were used, as described in [63]. Compared to other work, [62] laid emphasis on creating realistic scenes in simulation, using object co-occurrence statistics calculated from real data. For the task of semantic segmentation on real data, it was shown that training on synthetic data, followed by fine-tuning on real training data, outperformed training on real data alone. [26] also uses synthetic data, in the form of RGB-D images, for indoor scene understanding. BlenSor [12] was used as a rendering engine. Simulated scene generation was unaddressed, and scenes were generated at random. It was observed that performance on challenging real scenes could be transferred by training on simulated data, followed by a small amount of real data.

For the task of multi-object tracking, a Virtual KITTI dataset was created in [5]. To create realistic scenes, virtual seed worlds were created using annotations in the real KITTI dataset. Other aspects such as illumination and background adjusted manually. Like much of other work in creating synthetic data, [5] used a game engine (Unity) for rendering images of virtual scenes. Simulation allows testing under varied environments, and in [5], the effect of conditions such as challenging weather, on the performance of algorithms, was calculated.

4.3 Approach

4.3.1 Real and simulated scenes

We first present an outline of our approach to simulation. Various steps, including implementation details, are discussed in subsections that follow. The core of our simulator is written in C++, and we use MATLAB for labeling and scene generation.

For off-road scenes, it is unreasonable to assume that the real state x will be known. We do not have an expression for x, or an explicit way to characterize the real state space $\mathcal{X}$. Our choice of the simulated scene, and the simulated state, will necessarily be an approximation. The simulated scene will have to be inferred using information logged from a real scene. This is in contrast to the indoor scenes encountered in Chapter 3. An implication is that sensor modeling is not the only (or the main) task at hand for off-road Lidar simulation. To see why, consider building a sensor model for a Lidar interacting with vegetation using the methods in Chapter 3. To frame the problem purely as sensor modeling, we might need to accurately obtain the shape of real vegetation. This is difficult task even for the simplest of plants. Even if we could train a sensor model at the histogram level using data-driven methods, such a model might not be very useful during test. This is because the shape of vegetation during test will also have to be known. In the off-road case, therefore, we perform simultaneous mapping and sensor modeling. The description of the terrain and how the sensor interacts with the terrain are linked. For the indoor Lidar simulator, objects in simulation corresponded to objects in the real world. For off-road Lidar simulation, the scene representation may not be physically realistic. What matters is that the representation is optimized for the purpose of Lidar simulation.

The building block of the scene in simulation is a scene element $e = (\rho, \omega)$. A scene element is the output of a joint modeling of the terrain and sensor. Therefore, it consists of the element geometry ρ, and sensor model parameters ω. As in the indoor case (Section 3.4.1), the simulator state $\hat{x} = (S, q_{\text{sensor}})$ consists of a scene S and the sensor pose q_{sensor} . A scene $S = \{o_k\}$ is a set of objects. An object $o = (c, M, q_{\text{object}})$ consists of an object class c, model M, and pose q_{object}. The object class c is a categorical variable, and functions as an identifier. The object model $M = \{e_i\}$ is a set of scene elements.

In our approach, scene elements are fit to real data. Since these elements constitute the object model $M = \{e_i\}$, an object contains information about how the Lidar interacts with the real world. Our idea is to re-use this information for future simulation. We do this by storing objects from a training scene into a library. Such an objects will be called a scene primitive, $\pi = (c, M)$. A primitive is simply an object from a training scene,

transformed to identity. Since the pose of all primitives are the same, we omit writing the pose in the tuple of a primitive. The unique set of scene primitives is then the primitive set, $\Pi = \{\pi_j\}$. For Lidar simulation, we build new scenes using the primitive set. To construct a simulation scene, we also need information telling us 'which object goes where'. This is contained in the scene annotation $a = \{(c_k, q_{\text{object},k})\}$. A scene annotation is just the set of object classes and poses.

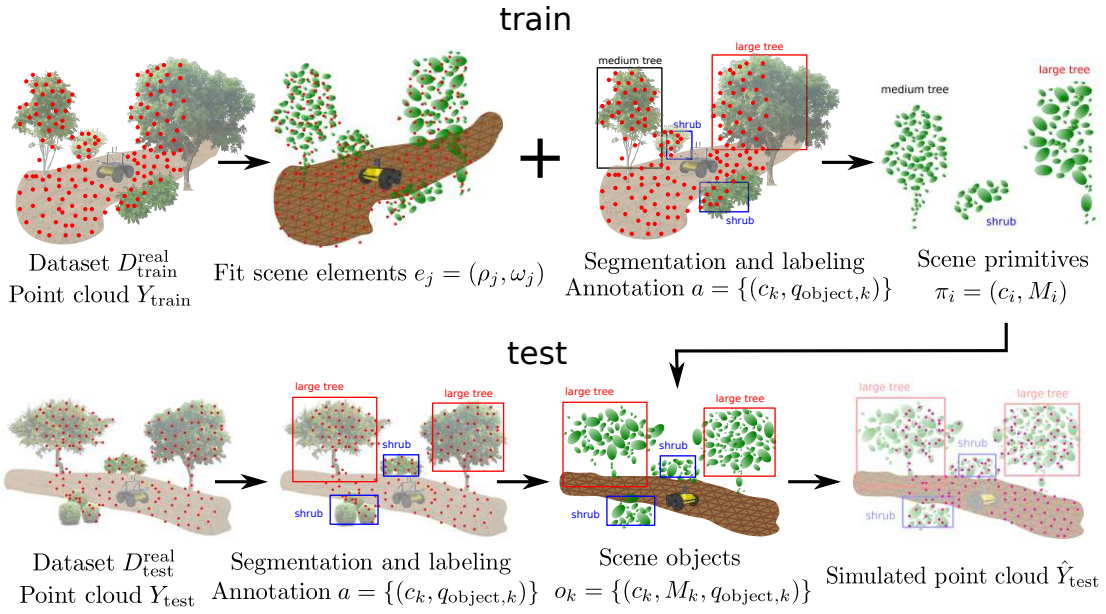


FIGURE 4.7: Outline of our approach.

We assume the availability of a training dataset $D_{\text{train}} = \{(q_{\text{sensor},i}, z_i)\}$ consisting of sensor poses and observations. This may be collected, for example, by driving a robot with a Lidar in a real scene, and logging observations (see Figure 4.7). Using the sensor poses and intrinsics, we transform the range measurements to points in the world frame, to obtain a point cloud Y_{train}. The point cloud is segmented into ground and non-ground points. We then fit scene elements $\{e_j\}$ to each segment separately. To ground points, we fit a triangle mesh; the element geometry ρ is therefore a triangle. On the non-ground points, we perform a clustering; ρ in this case is an ellipsoid. The different element geometries used are why we call this approach the hybrid-geometric simulator. Once the geometry of the point cloud has been obtained, we find the sensor model parameters ω for each element. These include hit probabilities and range variances. Having modeled the training scene, we can simulate observations by querying a Lidar pose q_{sensor}. We obtain a simulated point cloud $\hat{Y}_{\text{train}}$ by querying the sensor poses in the training data log D_{train}. We use the simulation risk to compare simulated data with reality. The simulator parameters θ are tuned to minimize this error metric.

The procedure outlined above leads to a Lidar simulator which has been optimized to simulate observations in the fixed training scene. This is exactly where the work of

[18] terminates. At this point, the scene elements $\{e_j\}$ are blind to objects semantics. Therefore, we perform object segmentation and labeling in the point cloud Y_{train}, from which the annotation $a = \{(c_k, q_{\text{object},k})\}$ is derived. The segmentation also allows us to group elements into an object model, $M_k = \{e_j \mid e_j \in \text{segment}_k\}$. These pieces of information are combined into scene objects $\{o_k\}$, $o_k = (c_k, M_k, q_{\text{object},k})$. Finally, we transform objects to identity (as a reference pose), thereby extracting a set of primitives $\Pi = \{\pi_i\}$, $\pi_i = (c_i, M_i)$ from the training scene. The approach is summarized in Figure 4.7.

For a test (or new) scene, we collect data in a similar manner as for the training scene, resulting in the dataset D_{test}. In this case, however, we do not fit scene elements to the point cloud Y_{test}, since we want to evaluate the generalization of the scene primitives obtained from the training data. The cloud Y_{test} is segmented and labeled, resulting in an annotation $a = \{(c_k, q_{\text{object},k})\}$. This annotation for the test scene is combined with the scene primitives, from the training scene, to result in objects. Having constructed a simulated test scene, we can simulate Lidar observations at query sensor poses q_{sensor}. Once again, simulator evaluation is done using the simulation risk.

4.3.1.1 Ground segmentation

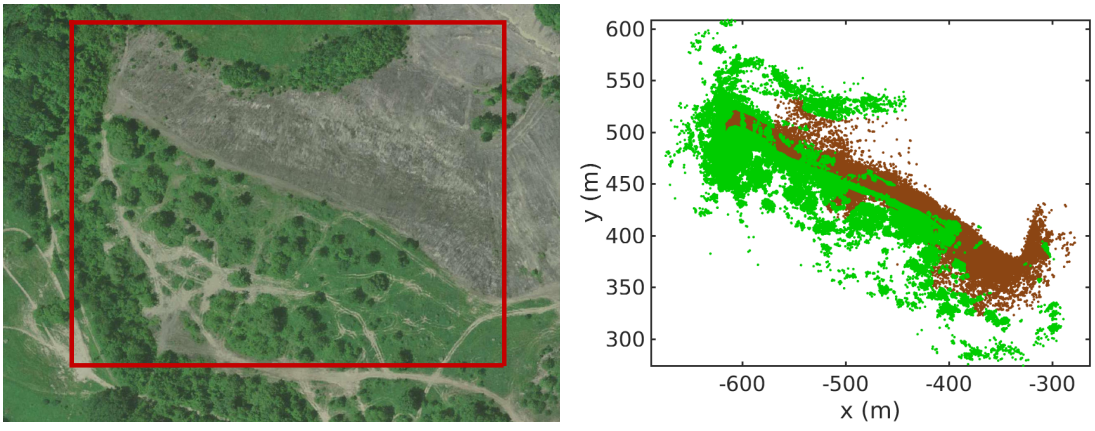


FIGURE 4.8: Example segmentation of training scene cloud into ground (brown) and non-ground (green) points. Corresponding overhead view of the off-road scene is shown on the left.

Ground segmentation is performed on the point cloud $Y = \{y_j\}$ obtained from the training data, and is a preprocessing step before scene elements are fit to Y. Ground segmentation is a routine step in point cloud processing [64, 65], and we use a simple procedure based on geometric features, as in [18]. The spherical variance $\phi(y)$ of a point y is calculated as follows. For points in a ball of radius d_ϕ, centered at y, the

covariance is computed. If the eigenvalues of the covariance matrix are $\lambda_1 \leq \lambda_2 \leq \lambda_3$, then $\phi(y) = \frac{\lambda_1}{\lambda_1+\lambda_2+\lambda_3}$. If there are less than $N_{\text{min},\phi}$ neighbors, a default value is assigned, $\phi(y) = \phi_{\text{default}}$. Optionally, the spherical variation values can be smoothed with neighbors within a radius $d_{\text{smooth},\phi}$, as

$$\phi(r_i) = \frac{\sum_j w_{ij}\phi(r_i)}{\sum_j w_{ij}}, \tag{4.1}$$

$$w_{ij} = \exp\left(-\eta_d \frac{\|y_i - y_j\|^2}{d^2_{\text{smooth},\phi}} - \eta_\phi \frac{(\phi_i - \phi_j)^2}{(1/3)^2}\right). \tag{4.2}$$

The spherical variation is a measure of the 'flatness' of the local neighborhood. It achieves the largest value of (1/3) for a sphere. Classification of a point is ground if it is less than a threshold, $\phi(y) < \phi_{\text{thresh}}$, and non-ground otherwise. The different parameters used in ground segmentation are tuned on a manually segmented point cloud.

4.3.1.2 Scene elements

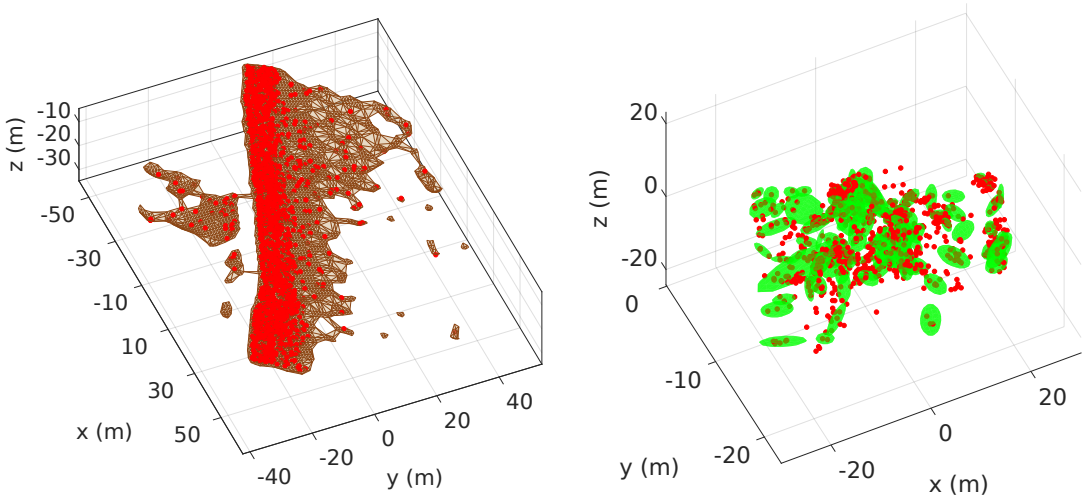


FIGURE 4.9: Hybrid geometric scene elements. Ground points are modeled by a triangle mesh. Non-ground points are clustered into Gaussian distributions. The red markers are real Lidar range data. The points are derived from the same scene shown in Figure 4.8.

Once an input cloud is segmented into ground and non-ground, we jointly model the terrain and sensor by fitting scene elements to the point cloud. A detailed study of different approaches to terrain modeling was carried out in [18]. For terrain without fine structure, surface modeling based on triangle meshes was found to perform well for simulation. The surface model is appropriate for a number of urban scenes. For terrain with fine structure, such as wires in urban environments, and vegetation in off-road

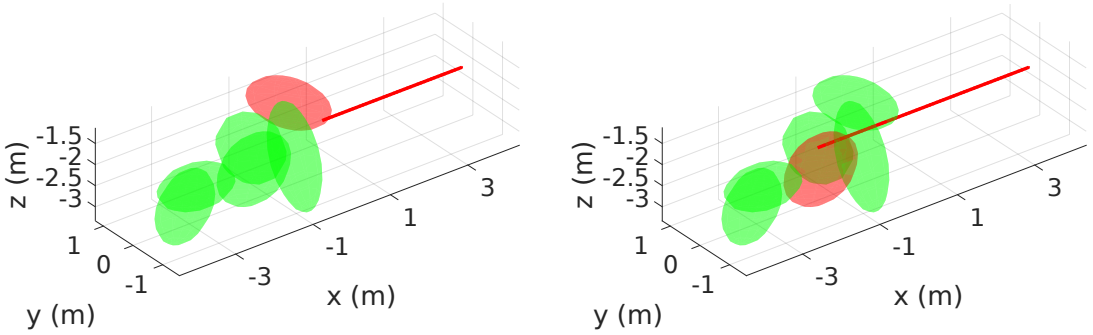


FIGURE 4.10: Elements are associated with a hit probability, according to which a ray intersecting an element may pass through. On the left, the ray first strikes the volumetric element shaded in red. It may pass through, and intersect the element shaded in red, on the right. The points are derived from the same scene shown in Figure 4.8.

environments, volumetric models were better for simulation. It was also found to be important to incorporate permeability in the models, whereby a ray striking an element (a mesh triangle or a volumetric element) can pass through with some probability. [18] considered two volumetric models. The first was a regularly sampled voxel grid, with a Gaussian distribution of observations associated with each voxel. The second was a set of Gaussian distributions, obtained from top-down clustering of the point cloud, with a Gaussian distribution of observations associated with each cluster. The voxel grid is computationally more efficient, but introduces voxelization artifacts. The top-down clusters were observed to model irregular structure in the environment better, but at the cost of increased computational cost. Compared to the surface model, the disadvantages of the volumetric model were: slower for simulation, and increased range variance when applied to planar terrain. A hybrid model was introduced to combine benefits across models. The ground points were modeled by a triangle mesh, and non-ground points by Gaussian distributions. We refer to this model as the hybrid geometric model. The hybrid geometric model was found to perform well across terrain types, from urban to off-road. The lesson for Lidar simulation, from the work of [18], was that surface elements are appropriate for terrain with planar structure, while volumetric elements are appropriate for terrain with irregular structure.

Modeling ground points. Working with the above insight, we model ground points by a triangle mesh (see Figure 4.9). We first fit a surface to the points. This is exactly regression as described in Section 3.3.2. The unknown function being fit in this case is the physical ground. We make use of the RBF interpolation tool [2] in ALGLIB [66]. ALGLIB is a C++ data analysis library. The number of points in the ground surface can be large ($\sim 1e5$), even after subsampling. The RBF interpolation tool is suitable for our use case. It uses compact Gaussian radius basis functions for regression, which

[2] http://www.alglib.net/interpolation/fastrbf.php

brings down the computational complexity to $O(N \log N)$ from $O(N^3)$ (for a method such as Gaussian Process regression [49]). The parameters of the regression tool are the radius of the basis function, and a regularization constant. A higher regularization constant results in a smoother surface fit.

After obtaining a surface fit, we triangulate it using the Delaunay triangulation tool in CGAL [67]. CGAL is a C++ library that provides a number of computational geometry algorithms. We also use the library to query intersections between a triangle mesh and a ray. We perform simple filtering steps to remove noise in the mesh generation. For example, we filter out isolated points whose nearest neighbor exceeds a threshold. Because of the uneven shape of the ground encountered in off-road scenes, the triangulation may generate large triangles as an artefact. We also filter these out using an upper threshold on the triangle side lengths.

Each ground element $e_j = (\rho_j, \omega_j)$, therefore, has a triangle for its geometry ρ_j. The sensor model parameter $\omega_j = (\sigma^2_{\text{ground}}, P_{\text{hit},j})$ consists of the range variance σ^2_{ground}, and a hit probability $P_{\text{hit},j}$. These parameters are used as follows. Suppose that a ray intersects a triangle element. We flip a coin, and with probability $P_{\text{hit},j}$, declare the intersection a hit. With probability $1 - P_{\text{hit},j}$, the ray passes through the triangle element. Suppose we sample a hit, and the nominal range is r. The simulator adds noise independently sampled from the Gaussian distribution $\mathcal{N}(0, \sigma^2_{\text{ground}})$ to r. The parameters σ^2_{ground} and $P_{\text{hit},j}$ are calculated from data. The range variance is calculated from the training data. For observed hits in the training data, after computing residuals $\xi_j = z_j - r_j$,

$$\sigma^2_{\text{ground}} = \frac{1}{N_{\text{residuals}}} \sum_{j=1}^{N_{\text{residuals}}} (z_j - r_j)^2.$$

The hit probabilities act as a filter for triangles as well. If extra triangles were generated, and no observed rays intersect with them, we expect $P_{\text{hit},j} = 0$.

Modeling non-ground points. Non-ground points for off-road terrain in our case correspond to Lidar observations from trees, shrubs, and other vegetation. To model their irregular structure, we fit Gaussian distributions to these points (see Figure 4.9). The Gaussian distributions are obtained using hierarchical clustering. We use the hierarchical k-means clustering tool part of the FLANN [68] C++ library. In top-down hierarchical clustering of points, all points initially belong to one cluster. The points are clustered using an algorithm (in this case, k-means). Each cluster is, in turn, further clustered using the same algorithm. The process continues recursively, till an edge case is met. An example edge case is a threshold on the minimum number of points in a cluster. The number of clusters at each stage is specified by a branching factor. The result of the recursive clustering is a tree of clusters, called a dendrogram. The

dendrogram is then cut at some level to minimize the cluster variance, and the result is returned. One of the parameters of the procedure is the number of clusters requested. For other internal clustering parameters in FLANN, such as the branching factor and initialization method, we used defaults.

Each cluster returned is a scene element. The geometry $\rho_j = (\mu_j, \Sigma_j)$ of the element is an ellipsoid. For M_j points in the cluster, the ellipsoid mean and covariance are calculated as

$$\mu_j = \frac{1}{M_j} \sum_{k \in \text{cluster}(j)} y_k,$$
$$\Sigma_j = \frac{1}{M_j - 1} \sum_{k \in \text{cluster}(j)} (y_k - \mu_j)(y_k - \mu_j)^T.$$

The sensor model parameters in this case are $\omega_j = (\mu_j, \Sigma_j, P_{\text{hit},j})$. The cluster mean and covariance are used for a Gaussian distribution associated with the ellipsoid. If a ray hits an volumetric element e_j, then the simulated observation is a sample from the distribution $\mathcal{N}(\mu_j, \Sigma_j)$. Ray-ellipsoid intersections can be calculated in closed-form. Sampling in this manner smears observations, taking into account some of the mixed pixel effects. The hit probability is a parameter that models pass-throughs in Lidar data. When a ray intersects an element e_j, a Bernoulli random variable is sampled with probability $P_{\text{hit},j}$. If the value is 0, the ray passes through. The ray may intersect another element e_k, in which case $P_{\text{hit},k}$ is sampled (see Figure 4.10). In an off-road environment, successive readings may not be the same due to factors such as wind which may cause vegetation to sway. Such effects also support using a stochastic sensor model. Once again, the hit probability P_{hit} works as a convenient filter. If spurious clusters have been generated which no rays intersect, we expect $P_{\text{hit}} = 0$.

For both triangle and volumetric elements, the hit probability is calculated from data as

$$p_{\text{hit}} = \frac{\#\text{hits}}{\#\text{hits} + \#\text{misses}},$$

where #hits is the number of rays which hit an element, and #misses is the number which missed the element. While counting the number of missed rays, we condition on the fact that the rays intersected (i.e. passed through) the element. The various parameters in modeling the ground and non-ground points are part of the overall simulator parameters, θ.

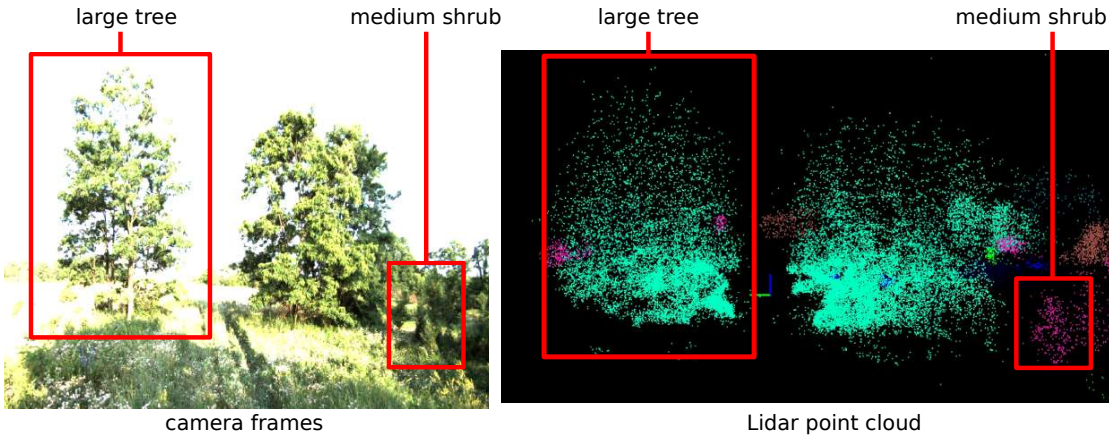


FIGURE 4.11: Example instances of the vegetation classes in off-road scenes.

4.3.1.3 Scene segmentation and labeling

In this work, segmentation and labeling is manual. We first segment the Lidar data corresponding to a scene using tools in CloudCompare [69]. The segments are then labeled using a custom MATLAB tool, visualized in Figure 4.16. We chose MATLAB for its support for 3D point visualization. In our custom tool, we could load a subset of segments, subsample them, select segments easily, and load partial labelings. We consider the following classes for vegetation objects:

- small shrub,
- medium shrub,
- large shrub,
- small tree,
- medium tree,
- large tree.

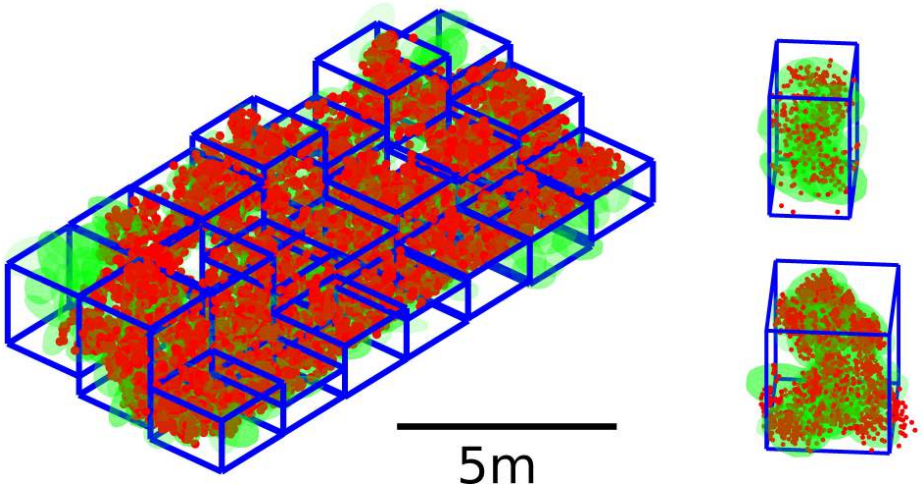


FIGURE 4.12: In off-road terrain, shrubs often occur in extended sections. We deal with such patches by dividing them into cells, as shown on the left. Like trees, shrubs also occur in isolated instances, with examples on the right.

We found that, while trees are often distinct and spacially localized, shrubs may be spread over large, irregular regions. We term such regions patches. We deal with patches by dividing them into cells of uniform size, see Figure 4.12. During training, the models in each cell are stored as primitives. During test, a patch is populated cell-wise.

4.3.1.4 Primitives and scene construction

Having obtained scene elements and labeled segments, we combine the pieces of information to obtain scene objects. For each segment with label c containing points $\{y_j\}$, we find the pose q_{object} as follows. The translation is the centroid of the points. We assume rotation along the global z-axis only. The principal axes of the points $\{y_j\}$ are found in the global x-y plane. We pick the principal axis aligned closest to the velocity vector of the sensor during data collection, and set that to be the local x-axis of the object. The orientation of the object is obtained from its local x-axis.

To find the object model M of an object o, we fit an oriented bounding box to the object points $\{y_j\}$. The box is oriented along the local coordinate frame of the object, calculated above. The extents of the box are based on the coordinates of the points in the local coordinate frame. The object model then consists of scene elements in the box interior, $M = \{e_j \mid e_j \in \text{object bounding box}\}$. Using the pose q_{object}, we then transform the object model M to identity, and store the result as a scene primitive $\pi = (c, M)$. The primitives together constitute the primitive library Π. This data-driven primitive library is our main tool for increasing the expressiveness of the hybrid geometric simulator.

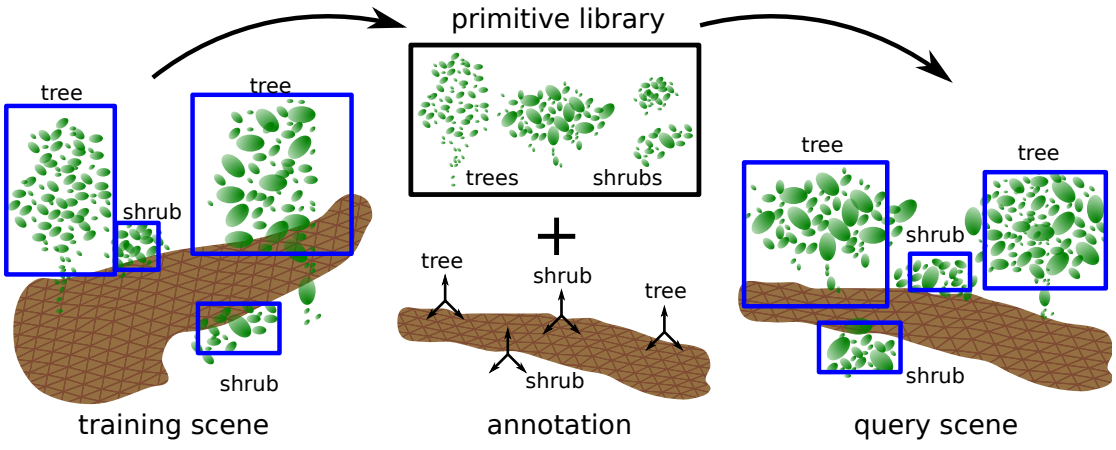


FIGURE 4.13: The scene annotation tells us which object goes where. An annotation is used along with the primitive library to construct a query scene.

For a new scene, we use an annotation $a = \{(c_k, q_{\text{object},k})\}$, consisting of object classes and poses, to populate the simulated scene. For a pair $(c_k, q_{\text{object},k})$, we pick a primitive $\pi_i = (c_i, M_i)$ such that $c_i = c_k$. In our implementation, the annotation for a new scene

is also derived from a semantic segmentation. In that case, the annotation for a scene object includes a bounding box. We use the heuristic of selecting the primitive whose bounding box most closely matches the bounding box of the scene object. For the metric between bounding boxes, we use the Euclidean norm of the box extents. The primitive model M_i is then transformed to the pose $q_{\text{object},k}$, resulting in object o_k. The annotation a can correspond to arbitrary scenes.

4.3.2 Simulator evaluation

Observation-level loss

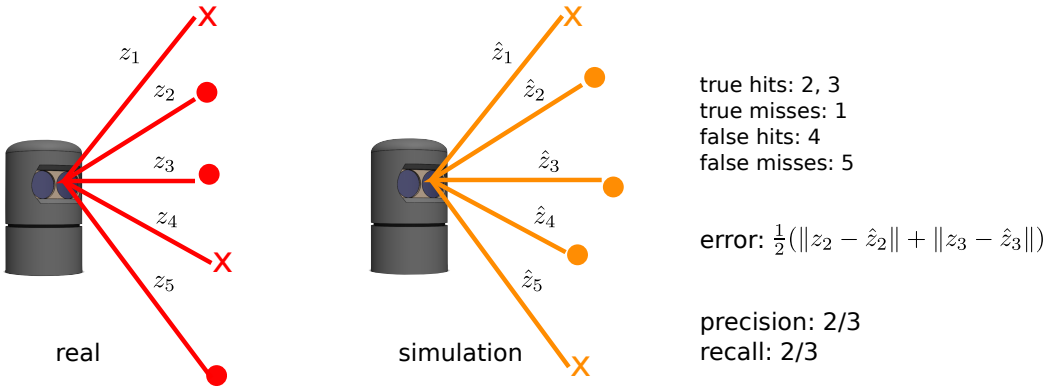


FIGURE 4.14

In our case, a sensor observation z is a packet of Lidar data. Each packet consists of returns from rays fired at different directions, from a common sensor pose q_{sensor}. Some of the rays result in hits, and others in misses. A simulated observation $\hat{z}$ will also contain hits and misses, some of which will be true, and the rest false. The observation-level loss is calculated over the true hits

$$l(z, \hat{z}) = \frac{1}{\#\text{ true hits}} \sum_{j \in \text{ true hits}} \|y_j - \hat{y}_j\|, \tag{4.3}$$

where y is the 3D point corresponding to a range return. To compare the false hits and misses, we also computed the F1 score over observations,

$$F1 = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}, \tag{4.4}$$

$$\text{precision} = \frac{\#\text{true hits}}{\#\text{true hits} + \#\text{false hits}}, \tag{4.5}$$

$$\text{recall} = \frac{\#\text{true hits}}{\#\text{true hits} + \#\text{false misses}}. \tag{4.6}$$

Apart from an observation-level error, we also compare the real point cloud Y with the simulated point cloud $\hat{Y}$, as a gross measure. The asymmetric point cloud error is defined as

$$\tilde{l}^{\text{pcd}}(Y, \hat{Y}) = \frac{1}{n'} \sum_{j=1}^{n'} \|y_{\text{nearest}(j)} - \hat{y}_j\|, \tag{4.7}$$

$$y_{\text{nearest}(j)} = \arg\min_{y \in Y} \|y - \hat{y}_j\|. \tag{4.8}$$

Using the asymmetric error above, we computed the symmetric point cloud error,

$$l^{\text{pcd}}(Y, \hat{Y}, \theta) = \frac{1}{2}(\tilde{l}^{\text{pcd}}(Y, \hat{Y}) + \tilde{l}^{\text{pcd}}(\hat{Y}, Y)). \tag{4.9}$$

Baseline simulator

Neighbor simulator. In Section 4.1, we described the use of a data log as a simulator. This is limited to simulation in the training scene. Given a training dataset $D_{\text{train}} = (q_{\text{sensor},i}, z_i)$, a query sensor pose q_{sensor} and a ray v, we calculate the distance along the ray $d_{v\parallel}$ and perpendicular to the ray $d_{v\perp}$ of Lidar hits in the dataset. The simulated observation is

$$\hat{z} = z_j,$$
$$\text{where } j = \arg\min_i d_{v\parallel,i}, \text{ for } d_{v\perp,i} < d_{v\perp,\text{thresh}}.$$

Points which are far from the query pose and ray are filtered by a threshold on the perpendicular distance to the ray. We also apply the standard filters on maximum and minimum ranges, which are intrinsic to the Lidar sensor. If no points satisfy the Boolean conditions, a null value is returned.

Mesh model simulator. As a baseline scene simulator, we implemented a baseline that was representative of what may be achieved using current general-purpose robotics simulators. In the mesh model simulator, the primitive library (and consequently, scene objects), were derived open-source 3D mesh models. This baseline was also implemented in C++, with each mesh model held in a CGAL triangle list. The sensor model consisted of raycasting with the object models, followed by adding Gaussian noise to the nominal range.

We obtained mesh models of trees and shrubs from TurboSquid[3]. The number of available models were limited due to reasons such as: scarcity of freely available mesh models of natural terrain; use of proprietary file types; large-size ($>$ 10MB) mesh models. The

[3] https://www.turbosquid.com/

raw mesh models required processing for use, as they had to be subsampled, transformed, scaled and sorted into appropriate classes. Example baseline primitives are shown in Figure 4.17. Given an annotation, scene objects are constructed from mesh model primitives for the baseline in the same way as for our hybrid geometric scene simulator.

Implementation specifics

Some steps taken to speed up the modeling and simulation were

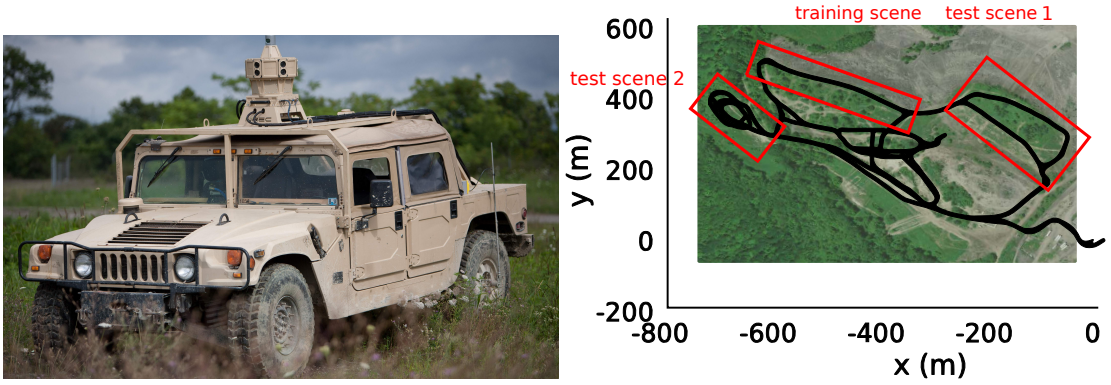
- The ground as well as non-ground segments are divided into blocks. Each block is modeled and stored independently.
- A simulated scene can be hundreds of meters long. Performing intersection checks for each simulation query is inefficient. For the non-ground volumetric elements, we create a local set of Gaussian distributions around a query ray. Similarly, for the baseline mesh model simulator, we check intersection with a local set of mesh objects. The local set is obtained using the k-d tree tools provided by the FLANN C++ library [68].
- Simulation is independent across rays, and can be parallelized. We use OpenMP [4] for a simple parallelization over the query rays.
- For the mesh model simulator, subsampling the open source meshes was found to be important to bring the simulation time down to the same order as the time taken by our approach.

Further optimizations are doubtless possible; software design and real-time simulation were not the focus of our work. The list above is a sample of parts of the implementation that can be addressed for faster simulation.

[4] http://www.openmp.org/

4.4 Experiments

4.4.1 Observation-level simulator evaluation



(A) Data collection platform. The Lidar is at the top of the sensor head.

(B) Data collection site. The vehicle path is marked in black. Test and training scenes are marked in red.

FIGURE 4.15

In our experiments, data was collected using a custom ground vehicle mounted with a Velodyne-32 Lidar[5], see Figure 4.15a. Lidar packets, each consisting of hit and miss returns, were received at 20 Hz. Each packet constituted an observation z. Vehicle pose was provided by a Novatel SPAN[6] pose system. After post-processing, pose information was available at sub-centimeter position accuracy, at a frequency of 2E2 Hz. The sensor pose q_{sensor} was calculated from the vehicle pose, and the fixed relative pose of the Lidar with respect to the vehicle. Data was collected in an off-road site nearby Pittsburgh, depicted in Figure 4.15. Lidar packets were logged as the vehicle was driven manually, at an average speed of 2.4 m/s. Modeling and evaluation steps of the simulator were conducted offline. We selected one scene for training and two for test, as marked in Figure 4.15. We ran the hybrid geometric modeling steps on the training scene, resulting in a representation of the scene as a combination of surface triangles and volumetric ellipsoids. Values for simulator parameters θ are summarized in Table 4.1. Parameters were optimized (using NLopt [70]) on a 10 sec slice of the training scene. The objective was minimization of the mean packet error+2(1 − F1 score).

For the training scene, we verified that the hybrid geometric models result in better simulation than the neighbor simulator baseline, as also found in [18], see Table 4.3. The semantic segmentation of the training scene is shown in Figure 4.16, and example primitives obtained for each class are shown in Figure 4.17. Statistics for the primitives are collected in Table 4.2. The test scene, which was longer than the training scene,

[5] http://www.velodynelidar.com/hdl-32e.html
[6] https://www.novatel.com/products/span-gnss-inertial-systems/

Parameter name	Selection method	Value
Ground segmentation		
maximum distance to neighbor	heuristic	5m
minimum neighbors to compute feature	heuristic	10
default spherical variation ϕ_{default}	optimization	0.01
threshold ϕ_{thresh}	optimization	0.09
Modeling ellipsoid elements		
minimum points per cluster	optimization	5
(hit count prior, miss count prior)	heuristic	$(0, 1)$
clusters per point	optimization	0.042
maximum Mahalanobis distance for hit	optimization	3.5
Modeling triangle elements		
regression basis radius	heuristic	5m
regression regularization constant	optimization	0.001m
maximum triangle side	optimization	10
(hit count prior, miss count prior)	heuristic	$(1, 2)$
range variance	fit to data	0.07m^2
maximum residual for hit	optimization	0.75m

TABLE 4.1: Hybrid geometric simulator parameter values

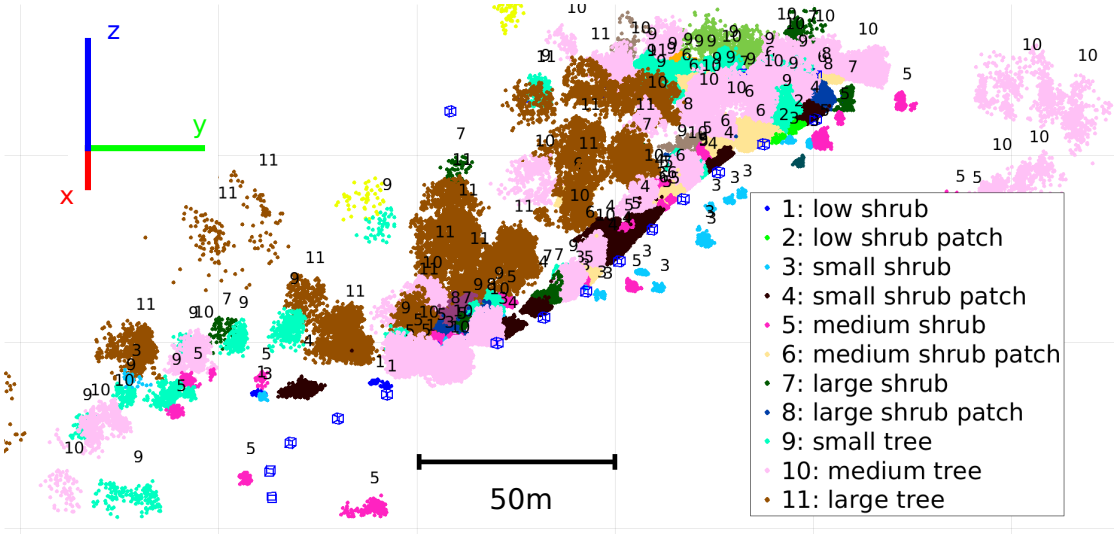


FIGURE 4.16: Segmentation and labeling for the training scene. The test scene is processed in the same way.

Primitive class	Instances	Mean ellipsoids per instance	Mean instance height (m)
low shrub	4	4	1.1
small shrub	18	12	2.1
medium shrub	27	24	3.8
large shrub	10	34	5.4
small tree	32	150	5.9
medium tree	33	301	9.4
large tree	19	161	13.4

TABLE 4.2: Training scene primitive statistics

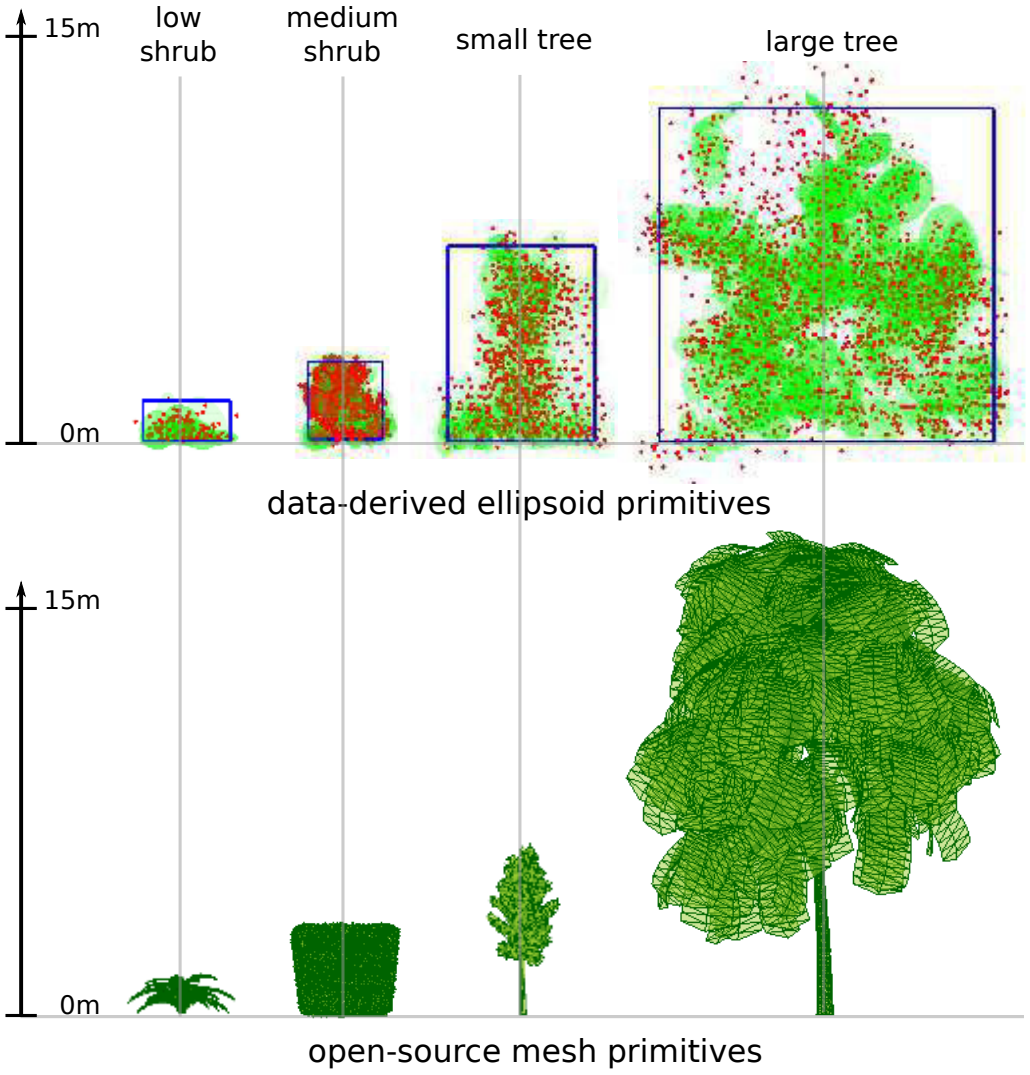


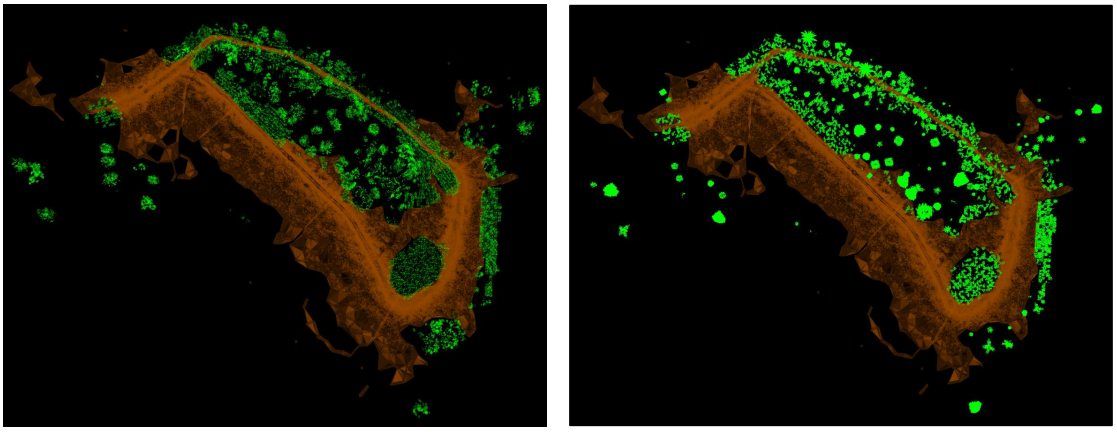
FIGURE 4.17: Examples from the primitive set. The top figure shows primitives obtained from the training data. The bottom figure shows primitives obtained from freely available mesh models.

was then constructed in simulation, see Figure 4.18a. Lidar data from the test scene was used to obtain the annotation, but the simulated objects were constructed entirely from the training scene primitives. A simulated scene was also constructed using the baseline simulator, see Figure 4.18b. From the data collection step, we obtained the test dataset of sensor poses and observations, $D_{\text{test}} = \{(q_{\text{sensor},i}, z_i)\}$. Given the dataset, we could directly model the test scene for simulation. Our aim, however, was to evaluate simulation of complex, new scenes using primitives from training scenes. By querying the simulated scenes at the sensor poses in the test dataset, we obtained the simulated sensor observations, $\{(q_{\text{sensor},i}, \hat{z}_i)\}$. Evaluation results of simulation are summarized in Table 4.3. Our simulation approach, based on data-driven terrain primitives, is quantitatively better than the baseline, based on open-source mesh primitives. Point clouds for different data sources are shown in Figure 4.19. The nearest neighbor simulator is myopic, and

cannot simulate objects at the background of a viewing direction. Compared to the mesh primitives, our approach results in simulation with more realistic object shapes, and has significantly higher recall.

Scene	Simulator	Point cloud error	Observation-level risk	Precision	Recall	F1
Training	Our approach	0.26m	3.2m	0.52	0.95	0.67
Training	Neighbor sim	0.30m	9.8m	0.55	0.61	0.57
Test 1	Our approach	0.48m	2.9m	0.57	0.85	0.68
Test 1	Mesh primitives	0.55m	3.1m	0.58	0.75	0.65
Test 2	Our approach	0.58m	3.6m	0.56	0.81	0.66
Test 2	Mesh primitives	0.72m	3.8m	0.54	0.67	0.59

TABLE 4.3



(A) Simulated objects with the hybrid geometric elements.

(B) Simulated objects with the mesh model simulator.

FIGURE 4.18: Simulated test scenes.

There are also qualitative ways in which our approach is better. The example in Figure 4.20 illustrates the benefit of the volumetric elements of our approach, over the surface triangles of the baseline. The ray origin is on the left. In our approach, the ray intersects ellipsoids, and the simulated point is close to the real point. In the baseline, the ray misses the mesh triangles close to the real point. The simulated point instead is much farther down the ray, where it intersects another mesh. This is why the length in the y-axis in the figure for the baseline (60m) is greater than that for our simulator (15m). Figure 4.21 illustrates the benefit of using permeable ellipsoids over opaque surface triangles. The ray origin is on the left, and the ray strikes a tree. The real point is in the interior of the tree. The simulated point from our approach is also in the interior, as the ray can pass through ellipsoids. The simulated point from the baseline, however, terminates at the surface of the tree. The baseline tree is visually realistic, but the hybrid geometric tree is sensor-realistic. This example in Figure 4.22 illustrates another benefit of data-driven primitives over open-source primitives. The ray origin is at the top right. A number of shrubs are in the path of the ray. The shape of data-driven

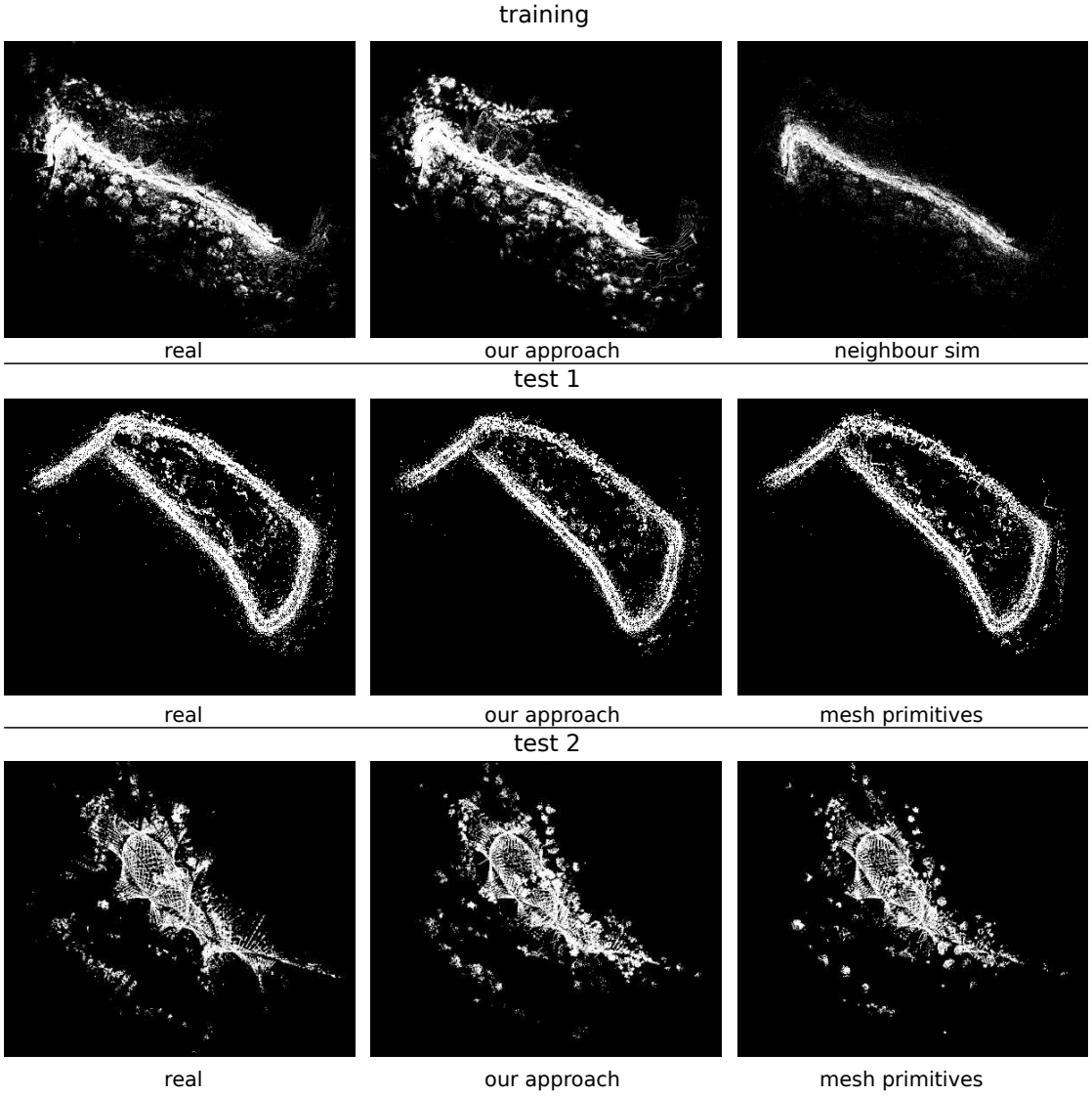


FIGURE 4.19: Real and simulated point clouds for different scenes.

primitives better match the scene objects, compared to the baseline primitives. In our approach, the simulated point is close to the real point. In the baseline, however, the simulation result is a false miss, depicted as a red cross. On the other hand, Figure 4.23 illustrates a weakness of our approach. The ray origin is at the top right. Low shrubs are in the path of the ray. The clustering in this case is coarse relative to the size of the shrubs, visible as some large ellipsoids. This causes the simulated point in our approach to have large variance. This problem is not present in the baseline, however, and the simulated point is close to the real point. The real point is behind meshes, and therefore not in view. In such cases where the clustering is not fine enough, the scene has large ellipsoids. The associated Gaussian distributions have high variance. Intersection with these ellipsoids, in turn, resulted in false hits. This is why the precision of our approach is lower than that of the baseline, see Table 4.3. This failure mode suggests

that we include the FLANN clustering parameters in the simulator optimization, a step not currently performed.

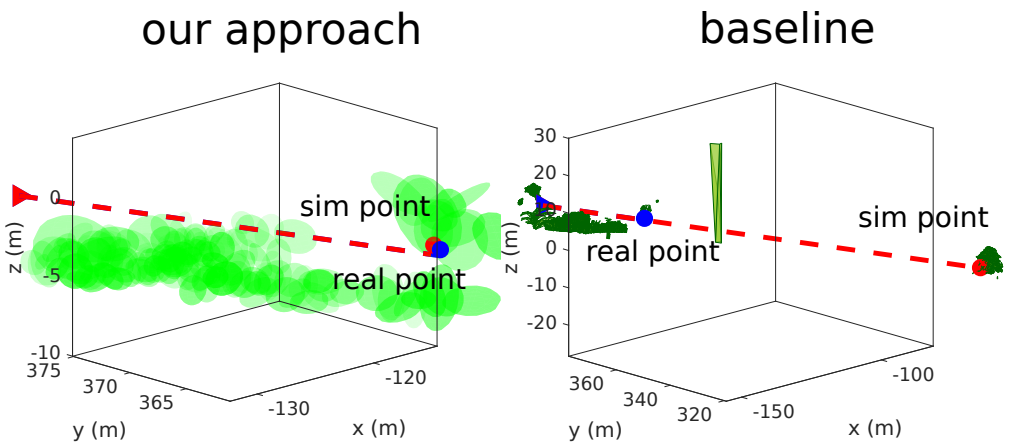


FIGURE 4.20: Example illustrating the benefit of volumetric elements over mesh models.

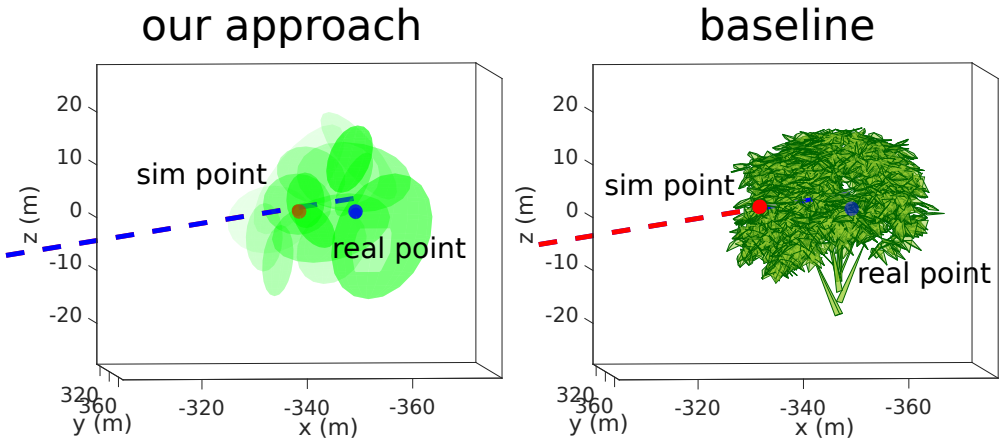


FIGURE 4.21: Example illustrating the benefit of using permeable elements.

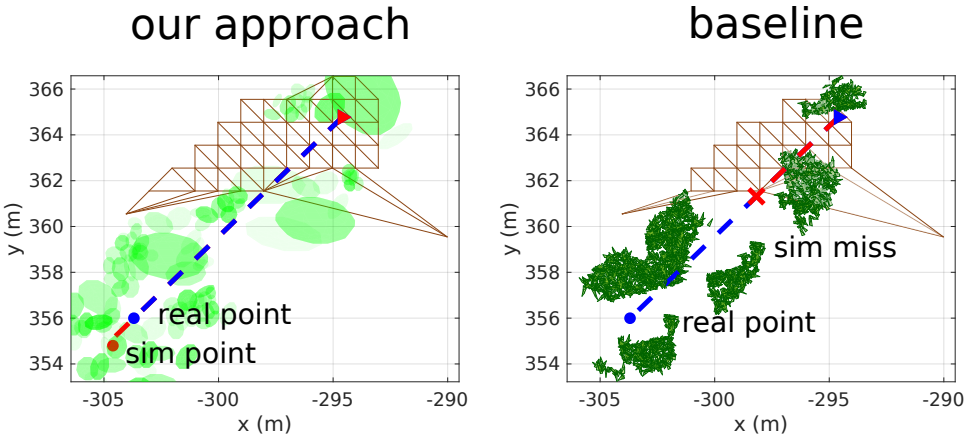


FIGURE 4.22: Example illustrating the benefit of data-driven primitives.

Another application of our work is in mapping off-road sites. Lidar mapping is the task of acquiring detailed point clouds of a site. Suppose that high-resolution point clouds were

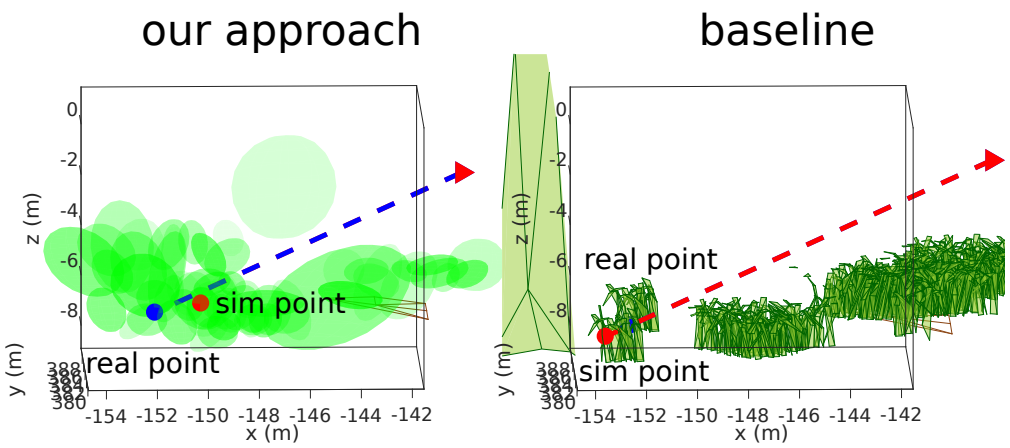


FIGURE 4.23: Example illustrating a weakness of the hybrid geometric approach, large ellipsoids.

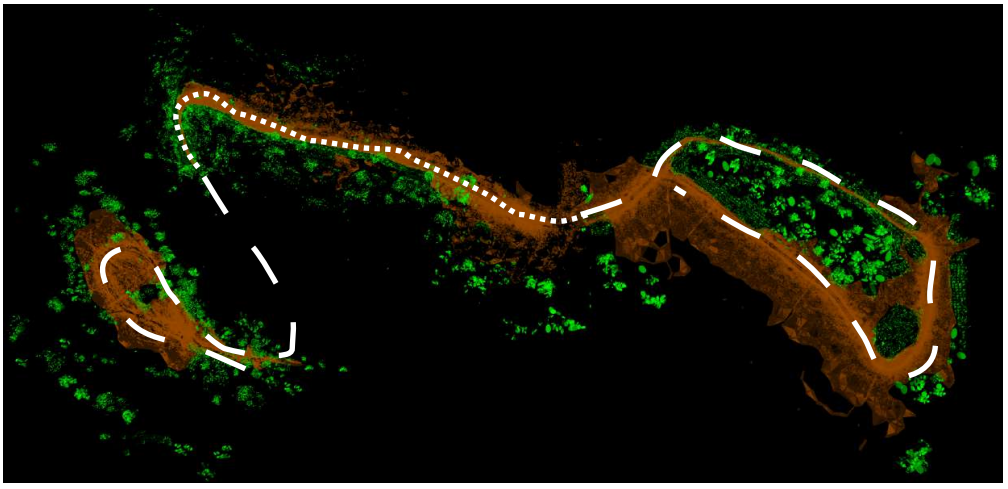


FIGURE 4.24: Application to efficient mapping an off-road site. A detailed Lidar data log from a representative section (finely dotted line) can be extrapolated to other sections (sparsely dotted lines).

required. If data were being logged by a ground vehicle, this would imply driving the vehicle at a sufficiently low speed. If the extents of the site were large, then such mapping would take a long time. It might further be slowed down or interrupted by hardware issues, or weather changes. Mapping can be addressed using our simulation approach as follows. A section of the off-road site can be marked as a training scene. A detailed point cloud can be gathered from the training scene, such that scene elements can be fit, and primitives extracted. For the rest of the site, we can construct a simulated scene, using a scene annotation. As in our approach, the scene annotation can be obtained from real Lidar point clouds. The idea to exploit is that segmentation and labeling can be performed on a point cloud with resolution lower than that required for building models. The vehicle collecting data can thus drive slowly in the section marked for training, and quickly in the rest of the off-road site. In our experiments, for example, the average vehicle speed in the training scene was about 5 miles per hour (mph). The segmentation

and labeling was performed on a subsampled point cloud that could have been collected with a vehicle speed of about 17mph. This suggests a 3-4× speedup in mapping times. If the scene annotation was obtained using satellite imagery, the mapping time may be further reduced. Once the off-road site has been constructed in simulation, a point cloud of any desired resolution may be queried from it. From this viewpoint, our approach can be seen as expensive data gathering in a representative site (the training scene), followed by extrapolation to a larger site (the full scene) in an inexpensive manner (using scene primitives).

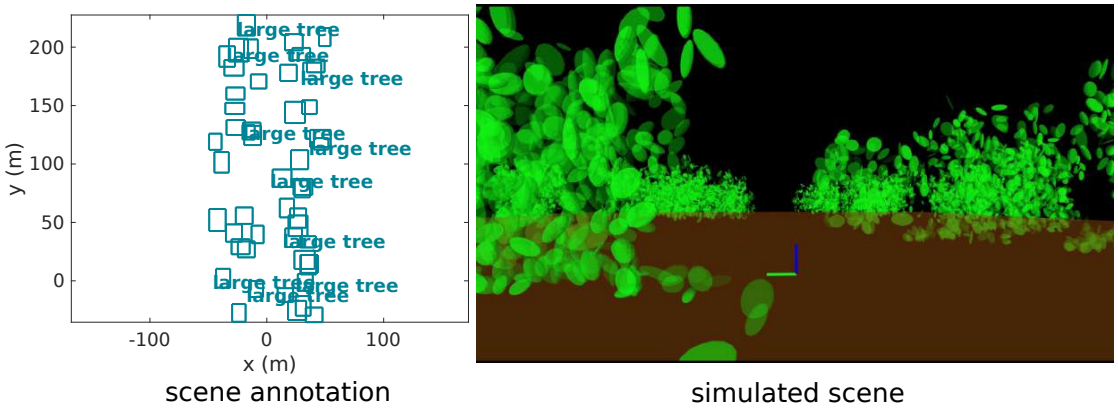


FIGURE 4.25: A synthetic scene, densely populated with large trees.

The simulated scenes in our experiments corresponded to real scenes. However, another common use case of simulators is to construct arbitrary scenes. In fact, one of the advantages of simulation is that scenes which are of interest, but difficult to reach in the real world, can be studied. Working with arbitrary scenes is also possible in our off-road Lidar simulator. All that is needed is to supply the simulator with an appropriate scene annotation (Section 4.3). For example, we may want to test perception applications in a densely forested environment. A possible annotation, and the constructed simulated scene, are depicted in Figure 4.25. Objects from the category of large trees were placed on either side of a straight path. The dimensions of the bounding boxes in the annotations were based on those encountered in the training data. Performing evaluation in real off-road scenes is advantageous for such use of simulators as well. Having shown that the data-driven terrain primitives generalize provides support for simulation in unseen, but similar, artificial scenes.

4.4.2 Application-level simulator evaluation

Scan matching

For the scan matching application, we considered scans derived from Lidar data from the first test section (Figure 4.26 (a)). Since each Lidar packet consisted of rays spanning

about 1.8deg, we aggregated 300 packets to result in a scan spanning 360deg, for a total of $N = 300$ scans. Simulated data corresponding to real data was generated using our approach and the baseline mesh model simulator.

The algorithm A used was the scan matcher in the `odoscan`[7] ROS package. It is a wrapper written around PCL [71] tools for registration, with additional features. For robustness, a RANSAC procedure is run as an outer loop over the inner scan matching iterations. Each inner scan matching is run for a maximum number of iterations, or till one of two thresholds is met. The thresholds are: a minimum change in the pose estimate, and a minimum change in the objective value, calculated between successive scan matching iterations. For fast computation, there is the option of filtering the input clouds. The filter is an approximate voxel grid, in which the the voxels are not uniform in space, but the leaves of a k-d tree. The algorithm parameter vector α was in $\mathbb{R}^7$. The parameters were: the maximum iterations for scan matching, the maximum point correspondence distance, the number of RANSAC iterations, the RANSAC threshold, the log-minimum transformation change threshold, the log-minimum objective change threshold, and the log-voxel size of the filter. Let $\{\tilde{q}_{i,i+1}\}$ be the relative poses estimated by scan matching, between time instances i and $i+1$. Let $\{q_{i,i+1}\}$ be the set of ground truth relative poses, then the algorithm objective is the pose error norm,

$$J(\alpha, D) = \frac{1}{N} \sum_{i=1}^{N} \|\tilde{q}_{i,i+1} - q_{i,i+1}\|.$$

For the difference between two poses in $SE(3)$, we report two quantities. The first is the norm between the two positions in Euclidean space. The second is the absolute value of the angle in the angle-axis representation of the rotation between the poses.

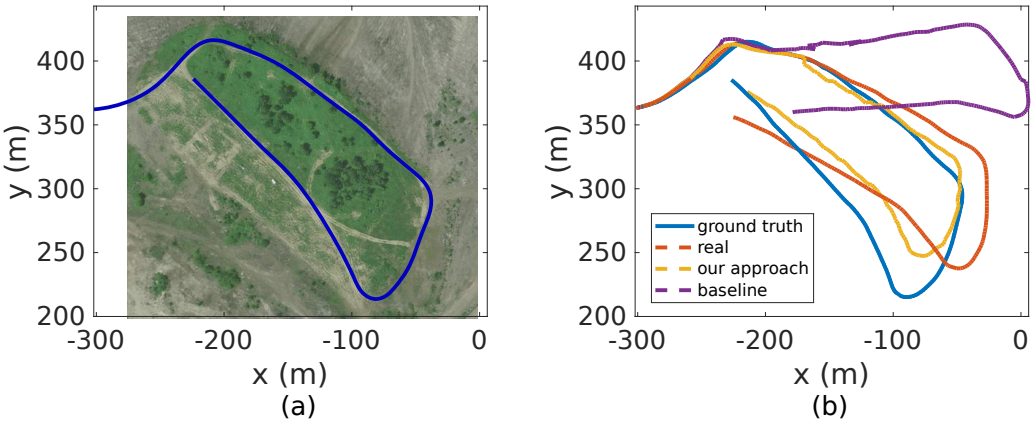


FIGURE 4.26: (a) Off-road scene, and vehicle path. (b) Estimated paths, for particular application parameters, for different data sources.

[7] https://github.com/Humhu/odoscan

To calculate the application-level risk, we sampled $M = 10^3$ algorithm parameters from a uniform distribution. The results are in Table 4.4, and our simulator has lower application-level risk than the baseline. For 100 application parameters, bar plots of the losses are in Figure 4.27. For more insight into performance, we discuss a qualitative difference between the simulators and reality. We observed that two important algorithm parameters were the log-minimum objective change threshold δ_J, and the log-voxel size of the filter, δ_{vox}. The dependence on δ_J was step-like: at a certain value, there was a sharp improvement of performance, with little further improvement. For δ_{vox}, there was an optimal value: at small values, the filter had no effect, and at large values, too much information was filtered out. These trends are difficult to ascertain beforehand, and are revealed on exploring the algorithm parameter space on real data. For particular application parameters, estimated trajectories are shown in Figure 4.26(b). The scan match performance on data from our simulator is closer to reality, than on data from the baseline simulator. Scans from the baseline were sparse compared to reality, so at the particular value of δ_{vox}, useful information is filtered out the baseline simulator. If development occurred on the baseline, we would pick a lower value of δ_{vox}, which would degrade performance in reality (which has a step dependence on the parameter). Note that obtaining the best-performing algorithm in reality, which is $\min_\alpha J(\alpha, D)$, is distinct from the calculation of simulation risk. We are interested in application performance being similar in simulation and reality.

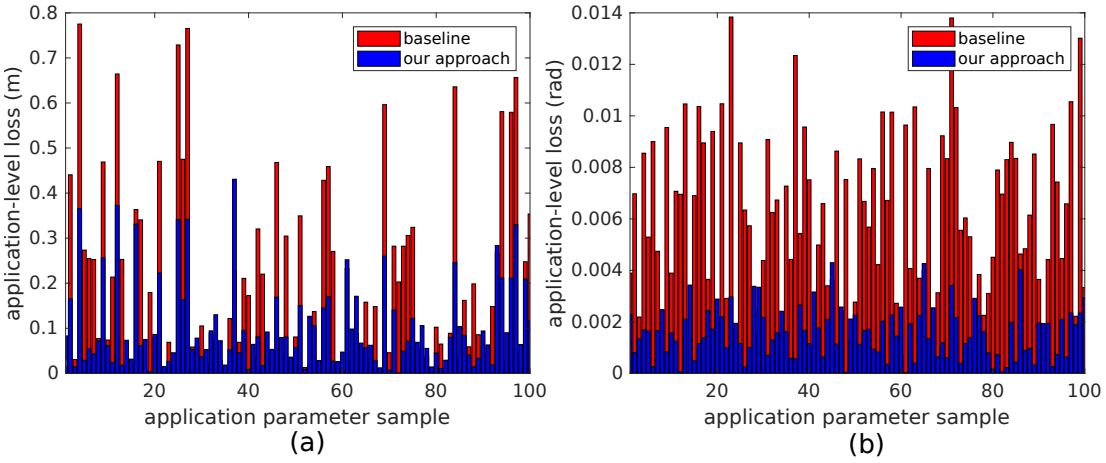


FIGURE 4.27: Simulation losses for 100 α samples with the simulation objective as (a) position error, and (b) rotation error, between real and estimated and relative poses.

Simulator	Risk for J = position error (m)	Risk for J = angle error (rad)
Our approach	0.070	1.6×10^{-3}
Baseline	0.096	6.1×10^{-3}

TABLE 4.4: Risks for off-road scan matching application

Scene	Length (m)	Log duration (s)	Number of Lidar packets
training	356.3	70.0	119260
test 1	650.0	116.5	210764
test 2	294.2	65.0	117426

TABLE 4.5: Scene statistics.

Class	Low shrub (patch)	Small shrub (patch)	Medium shrub (patch)	Large shrub (patch)
Instances	4 (2)	39 (59)	73 (55)	44 (20)

Class	Small tree	Medium tree	Large tree
Instances	111	86	51

TABLE 4.6: Number of instances per class.

4.4.3 Datasets generated

We have generated the following datasets in the course of our work, which may be of independent interest.

- **Lidar simulation**. Data logs from real scenes consist of sensor poses, along with Lidar packets. Each Lidar packet consists of returns from rays fired in $32 \times 32 = 1024$ directions. There are 32 yaw angles, with 32 pitch angles at each yaw angle. The data can be analyzed at multiple scales, from a high-level picture of point clouds, to a low-level view of individual rays resulting in hits or misses. The statistics are in Table 4.5.

- **Off-road segmentation and labeling**. The manual segmentation and labeling performed for the scenes serves as a dataset which can be used to train classifiers for automatic segmentation and labeling. The number of instances across scenes are in Table 4.6. The segments are associated with bounding boxes, so that labeled points of varying resolution can be obtained from the data logs.

Chapter 5

Conclusion

5.1 Summary of contributions

We have presented a framework for data-driven Lidar sensor simulation. In our framework, a simulator consists of three components: sensor modeling, scene generation, and simulator evaluation. A good sensor model captures the complex interaction of a Lidar with objects. Good scene generation recreates real scenes in simulation. Good simulator evaluation is thorough and takes into account the use case of a simulator for robot application development. Each component is important for useful simulation. Our framework is general, and applicable beyond Lidar simulation. For much past work in sensor simulation, we can identify and separate efforts in each of the components. Existing work on evaluation can be re-framed as specific instances of our broader concept of application-level simulator evaluation. We instantiated our approach for two domains, indoor planar Lidar simulation, and off-road Lidar simulation.

- We identified sensor modeling as a special case of distribution regression, and used this insight to construct a high-fidelity simulator for a planar Lidar sensor in indoor scenes. We showed that an appropriately chosen parametric sensor model outperforms the models used in current robot simulators. We demonstrated this in a meaningful manner, through utility of our simulator for development of applications such as detection and registration.

  We took the connection between regression and sensor modeling further by applying nonparametric distribution regression, developed in the learning community, as a modeling procedure. We showed that it is an appropriate procedure when faced with complex observation distributions, e.g. due to unmodeled state. We believe it can play a useful role in any modeler's toolbox, as a procedure that

adapts to data to capture inherent noise, without requiring the designer to make specific assumptions.

- We constructed an off-road Lidar simulator that was both high-fidelity and expressive, where past simulators were only either. We demonstrated that object models derived data generalize to new scenes better than open-source mesh models. We compared simulators on complex, real-world scenes of interest. Such evaluation justifies treating simulated data from hallucinated worlds as realistic.

  We showed how our simulator can also benefit efforts to efficiently create point cloud maps of off-road geographical sites. Our work generated off-road labeled point cloud datasets, which may be of independent interest.

We have shown that simulation tools are likely to improve the efficiency of the application development process. They expose the flaws in naive approaches earlier in the development cycle, and in a manner that mimics what would eventually happen when testing on the hardware. This is significant because it is possible to scrutinize a simulator by slowing down time, inserting breakpoints, and other methods. Our results suggest that we do not necessarily have to choose between realistic data and convenience in debugging.

5.2 Future work

5.2.1 Nonparametric sensor modeling

There are a number of conditions, other than those explored in this thesis, in which sensors behave in inherently noisy ways. Examples are range sensors in smoky environments [17], and sonar sensors under water [72]. The following extensions to the nonparametric sensor modeling procedure as described in Chapter 3 will increase its applicability.

- **Relaxed assumptions on data collection**. In Chapter 3, we worked with datasets of the form $\{x_i, z_{ij}\}$. That is, at each state, we sampled multiple observations. This step was assumed so that we could estimate observation densities $\hat{p}(z|x_i)$ at each state x_i. This form of dataset was simple to collect with a Lidar sensor in a static scene. On the other hand, there are cases where this form of data is hard to collect. Consider a tactile sensor on a robot hand. Each observation z consists of the sensor reading when the hand approaches and contacts some object. Even if the experiment were automated, it will be difficult to ensure the same state x each time an observation is sampled, due to object uncertainty

(Figure 5.1). This may be further complicated, because the velocity of the hand when contacting the object may be part of the state. Another example is sensor readings logged from a quadrotor as it hovers. The platform may be stable, but is not stationary.

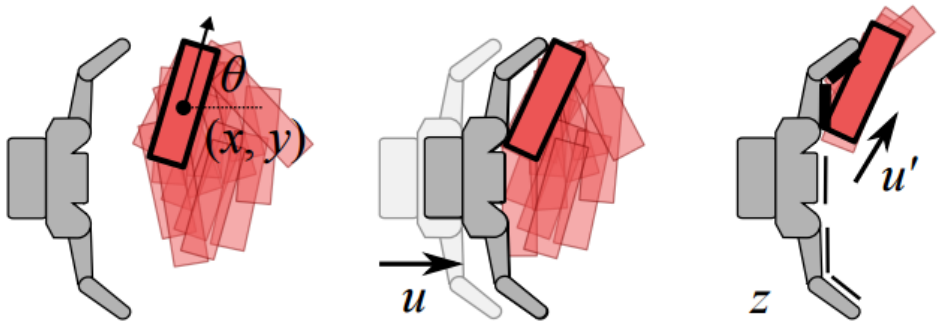


FIGURE 5.1: Collecting tactile sensor data when contacting an object. It is difficult to collect multiple observations at the same state. Figure from Koval et al. [73].

It is reasonable, however, to assume that the states at which multiple observations are logged are related. They may be part of a slowly varying trajectory, or within a bounded region, and so on. This information may be exploited in the regression. On the theoretical front, this extension calls for convergence guarantees on distribution regression, under an appropriate model of changing state.

- **Distribution regression in higher dimensions**. In our experiments, we performed distribution regression for scalar-valued output distributions. Multi-dimension sensor observations were dealt with by modeling each dimension independently. Modeling higher dimensions jointly is challenging from a computational standpoint, since representations such as histograms become memory intensive. Estimation in higher dimensions is also known to be challenging statistically. One approach to work with higher dimensions is to use copula methods [74]. As an example, consider a random variable in two dimensions. In the Gaussian copula method, the marginal cumulative distribution function (cdf) for each dimension is modeled independently. The joint distribution is then specified using a parametric, Gaussian distribution (Figure 5.2).

5.2.2 Off-road Lidar simulation

- **Further optimizations**. Our simulator opens a number of avenues for further improvements. More parameters of the simulator can be optimized. An example

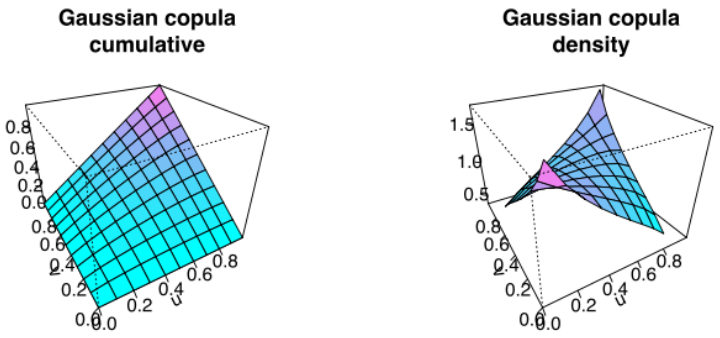


FIGURE 5.2: Gaussian copula. Marginal cdfs are modeled using nonparametric distribution regression. The joint is modeled using a Gaussian distribution. Figure from https://en.wikipedia.org/wiki/Copula_(probability_theory).

is the set of modeling parameters used for the hierarchical clustering of point cloud data. This was the clustering used to generate the ellipsoid scene elements.

Some of our implementation details (such as basic parallel processing) are in Section 4.3, but computational efficiency was not the focus of our work. In our opinion the largest gains will be obtained by optimization of ray-element intersection queries. These are the most basic type of operation in Lidar simulation. They can be sped up performing queries on tools such as the NVIDIA OptiX ray tracing engine [1].

- **Automated scene generation**. Building an off-road scene makes use of a scene annotation. In this work, we generated the annotation for a real scene manually. Automating scene generation is a useful step that will make it easier to evaluate a simulator. Conceptually, this requires off-road vegetation classification and segmentation using Lidar data. We believe this step is a matter of implementation, and not fundamental research. This view is based on extensive prior work on point cloud processing, which we briefly review. Our work has already generated labeled datasets (Section 4.4.3) which can be used for training.

  Semantic segmentation of 3D data is a well-studied problem, with applications in urban scene understanding [64, 75], off-road traversability analysis [76], and agricultural robotics [77–79]. Successful solutions have been developed, as evidenced by a number of instances of reported classification accuracies in excess of 90% [77–80]. A common approach is to classify a point based on local shape features, introduced in [81], and subsequently used in [79, 82, 83]. Additional features that have been investigated are based on laser remission values [76, 84, 85], and color [77]. While some work performs classification, followed by segmentation [81], others first segment points, and then classify the segments [64, 65, 78]. In the latter

[1] https://developer.nvidia.com/optix

case, segment-level shape descriptors may be used for classification [64, 75]. By taking into account knowledge that neighboring points share class labels, results can be improved by using classifiers based on conditional random fields (CRFs) [80], or Markov random fields (MRFs) [83]. While vegetation is often considered a single semantic class, we work with multiple vegetation classes, similar to [77, 78]. In our approach, Lidar data was processed offline, although real-time algorithms have been developed in [65, 76, 83, 84]. Finally, the source of 3D data in our work was a Lidar mounted on a ground vehicle. This is not a limitation, as prior work contains precedents for classifying scene data obtained from other sources, such as aerial robots [77, 84], and satellite images [82].

In Section 2.2 we referred to the expressiveness of a simulator. We discussed how scene generation is like a projection from real scenes to the space of simulated scenes. When automating scene generation, we are approximating the ideal projection. Errors in automated scene generation can thus also lead to errors in simulation. At the same time, automated scene generation complements expressiveness: a simulator may be able to represent reality, but if simulated scenes are too tedious to construct, the simulator's value diminishes.

- **Lidar generative models**. Instead of using a hierarchical clustering to fit ellipsoids, we could use recent work on compact generative models for point cloud data by Eckart et al. [86]. In [86], a mixture of Gaussian distributions was used as a generative model for a given 3D point cloud. This is common with our use of Gaussian distributions for non-ground objects. The key difference is that a top-down hierarchy of mixture distributions was proposed in [86]. A coarse model could be obtained by sampling distributions at the top level. Similarly, a fine model corresponded to sampling distributions at the bottom levels. The result was that model fidelity could be traded off with sampling time. Figure 5.3(a) shows 3D models for the Stanford bunny at different levels of fidelity. Using the models of [86] in our work would introduce a similar knob, which is currently missing in our simulator. Note that the models in [86] worked only with point cloud data. Integrating them in a Lidar simulator will additionally require considering the geometry of Lidar rays, including hit probabilities in the mixture distributions, and so on.

  The other generative models with recent success in generating 3D objects are deep learning methods, as in [87]. Figure 5.3(b) shows interpolation between 3D objects from the training dataset in [87]. Training a Lidar simulator could involve training a separate neural net (NN) to simulate Lidar readings for each object. Each primitive in this approach would be an optimized NN, instead of optimized ellipsoids. NNs from the library could be planted in a new simulated scene, which would be represented as a mixture of NNs. Given a simulation query, we would

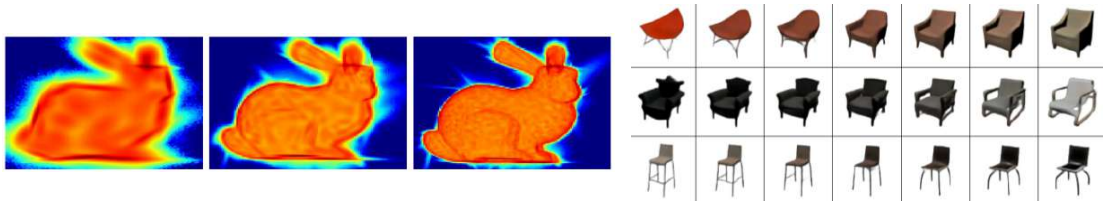


FIGURE 5.3: Compact generative models (a) from [86], and deep learning to generate 3D objects (b) from [87].

have to decide which NN to generate observations from. Simulation would also have to take into account interaction between the primitive NNs. This is because a ray may pass through an object, only to hit one behind it.

5.2.3 Simulator evaluation

- **Training with application-level simulation risk**. The application-level simulation loss measures the value of a simulator for the purpose of application development. Given a loss, it is natural to ask if it can be used for training the simulator. Basic experiments with the indoor planar Lidar suggested that a simulator trained with the application-level risk was not significantly better than one trained with an observation-level risk.

  In our experience, training with the application-level risk is more cumbersome, and takes longer. A valid objection is that, while we want a simulator to be useful for application development, it should not be tied to a single application. A workaround may be to minimize the application-level risk simultaneously for many application algorithms. This gives rise to further questions, such as which combination of applications. Training to minimize the observation-level risk, by contrast, was found to be faster and simpler. Disconnecting the simulator from an application introduces a gap between the two: the simulator writes data, which is then fed into the application, to return a risk. This discontinuity implies that analytical gradients are not available to optimize the simulator parameters with respect to the application-level risk. In this work where speed was not the focus, each calculation of the risk proved to be expensive.

  Having shown the utility of the application-level risk, however, we believe this is an interesting direction. It will require speeding up the Lidar simulator, the application, and using intelligent optimization methods. These will allow the optimization of the simulator parameters to make sufficient progress. We also recommend seeding the training with a simulator already optimized on the observation-level loss.

- **Closed-loop applications**. We assumed in Section 2.4 that applications are open-loop. This allowed us to collect a dataset $\{x_i, z_i\}$, and calculate the application-level risk offline. For closed-loop applications, by contrast, the dataset depends on the output of the application algorithms. For example, consider closed-loop navigation. Pose estimates $\hat{q}_i$ would depend on all past observations, $\{z_j\}\, j = 1 : i$, and influence the next state x_{i+1} visited. Running the application algorithm with the same parameters α in simulation would generate a dataset with different states.

  Development of open-loop applications is a rich domain with its own challenges, as explored in this thesis. Development of closed-loop applications, a certain use of simulators, brings us into the territory of reinforcement learning.

5.3 General sensor simulation outline

FIGURE 5.4: (a) In urban scenes, Lidar reflectivity can be used to detect road markings [88]. (b) Urban reconstruction [89] can be used to generate simulated scenes. (c) Relevant applications for development are detection and tracking.

Based on our framework, contributions, and experience, we present general guidelines for the construction of a sensor simulator. As an example, we consider steps for Lidar simulation in urban scenes.

What kind of real data can be gathered? Data-driven sensor simulation relies on comparing simulated and real data. It is essential that real data be gathered in conditions similar to those intended for simulation. For other robot applications, simulation may be stated as an answer to the problem of gathering adequate real data. The same answer cannot apply to simulation itself. We insist on having real data, and this makes constructing a simulator seem as difficult as developing an application on a real robot. This is exactly the view we promote. A simulator is a predictor that models the complex behavior of sensors in realistic scenes. There is no reason to expect that the construction of a simulator should be trivial compared to developing an application in reality.

We understand that setting up a system for gathering real data takes serious engineering effort. Our off-road Lidar simulation work was only possible because researchers at NREC had spent valuable time maintaining the platforms used. The payoff on the data

investment, however, is that future application development can be confidently carried out in simulation. If real data for training is not available, simulators will remain immature, specified by hand, untested, and viewed as little more than toy checks for real robot applications.

What is a suitable sensor model? The sensor model describes how the sensor interacts with the environment. Is a parametric model sufficient? Simple logs of sensor observations, when visualized as histograms, can reveal if Gaussians are good approximations. Physics-based models can be helpful in deciding what features to choose, or what input space to work in. If observation distributions are complex, then the alternative to painstaking parametrization is the use of nonparametric sensor models. These can adapt to trends in raw histograms, at the cost of requiring more training data.

A helpful exercise is to assume the environment is completely known, and then think about the sensor model. For Lidars in urban scenes, for example, this means considering that the state of all objects are known. Lidar returns from flat roads, pavements and walls may be well-approximated by Gaussian distributions. Modeling reflected intensity might be important, a problem not considered in this thesis, but to which our methods readily extend.

How are simulated scenes generated? Simulation using the best sensor models will be unrealistic if simulated scenes are not realistic. An exercise is to think about the real scenes an application will be developed for. It should be possible to reconstruct those kinds of scenes in simulation. Like real data collection, it may take effort to set up scene generation. Our suggestion is that scene generation also be data-driven. Specifying scenes by hand becomes unfeasible for complex scenes. Instead, an approach based on using statistics from real scenes is principled.

For urban scenes, approaches for scene reconstruction from points or images has been studied in work such as [89, 90]. An additional complexity is the presence of dynamic objects, such as other vehicles, and pedestrians. Scene modeling was more of a challenge than sensor modeling for the off-road scenes we considered. This is even more the case for urban scenes. Simulated urban scene construction was discussed briefly in [27], and can be explored more thoroughly in our opinion. Data-driven computer vision tools for scene understanding, visual tracking, and others, can be utilized.

What robot applications will be developed using simulation? Comparing performance of application algorithms in reality versus simulation is a measure that can be interpreted by developers working on applications. It can influence their decision to develop on simulation instead of reality, and lead to savings of time and effort. There

may be many routes to constructing a simulator, but the application-level risk places them on a comparable footing.

No matter how good simulators become, we believe applications will not (and should not) be deployed before final testing in reality. This process generates real data that can then be used for further simulator training and evaluation. Consider a Lidar simulator for urban scenes which has low application-level risk for state estimation in regular traffic conditions. For testing the state estimation in a new, but standard, city block, using simulated data is justified. If we then wanted to test estimation under reduced sensing, we might have to go out and collect real data. The simulator can then be evaluated and trained to model reduced sensing as well. In this manner, sensor simulation and robot application development can benefit from each other. The data-driven methods of this thesis can be used to assess and improve performance, and bring Lidar simulation for robotics closer to reality.

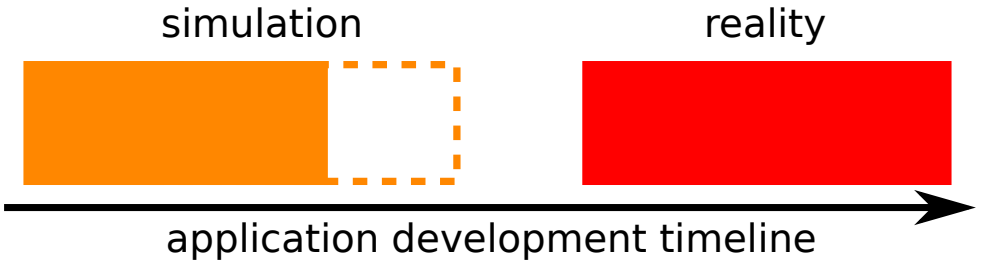


FIGURE 5.5: Our work is a step in the direction of reducing the gap between simulation and reality for robot application development.

Bibliography

[1] Carl Wellington, Aaron Courville, and Anthony Stentz. A generative model of terrain for autonomous navigation in vegetation. *The International Journal of Robotics Research*, 25(12):1287–1304, 2006.

[2] Alonzo Kelly, Nicholas Chan, Herman Herman, Daniel Huber, Robert Meyers, Pete Rander, Randy Warner, Jason Ziglar, and Erin Capstick. Real-time photorealistic virtualized reality interface for remote mobile robot control. *The International Journal of Robotics Research*, 30(3):384–404, 2011.

[3] Daniel Maturana and Sebastian Scherer. 3d convolutional neural networks for landing zone detection from lidar. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 3471–3478. IEEE, 2015.

[4] Carlos E Agüero, Nate Koenig, Ian Chen, Hugo Boyer, Steven Peters, John Hsu, Brian Gerkey, Steffi Paepcke, Jose L Rivero, Justin Manzo, et al. Inside the virtual robotics challenge: Simulating real-time robotic disaster response. *IEEE Transactions on Automation Science and Engineering*, 12(2):494–506, 2015.

[5] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtual worlds as proxy for multi-object tracking analysis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2016.

[6] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, pages 621–635. Springer, 2018.

[7] Martin Engelcke, Dushyant Rao, Dominic Zeng Wang, Chi Hay Tong, and Ingmar Posner. Vote3deep: Fast object detection in 3d point clouds using efficient convolutional neural networks. *arXiv preprint arXiv:1609.06666*, 2016.

[8] Ji Zhang and Sanjiv Singh. Loam: Lidar odometry and mapping in real-time. In *Robotics: Science and Systems*, volume 2, 2014.

[9] Tibshirani Hastie and R Tibshirani. & friedman, j.(2008). the elements of statistical learning; data mining, inference and prediction.

[10] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004.

[11] Brian Gerkey, Richard T Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, volume 1, pages 317–323, 2003.

[12] Michael Gschwandtner, Roland Kwitt, Andreas Uhl, and Wolfgang Pree. Blensor: blender sensor simulation toolbox. In *Advances in Visual Computing*, pages 199–208. Springer, 2011.

[13] Jeff Craighead, Robin Murphy, Jenny Burke, and Brian Goldiez. A survey of commercial & open source unmanned vehicle simulators. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 852–857. IEEE, 2007.

[14] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 4397–4404. IEEE, 2015.

[15] Chris Goodin, Phillip J Durst, Burhman Gates, Chris Cummins, and Jody Priddy. High fidelity sensor simulations for the virtual autonomous navigation environment. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 75–86. Springer, 2010.

[16] Dean Anderson, Herman Herman, and Alonzo Kelly. Experimental characterization of commercial flash ladar devices. In *International Conference of Sensing and Technology*, volume 2, 2005.

[17] Okke Formsma, Nick Dijkshoorn, Sander van Noort, and Arnoud Visser. Realistic simulation of laser range finder behavior in a smoky environment. In *RoboCup 2010: Robot Soccer World Cup XIV*, pages 336–349. Springer, 2011.

[18] Brett Browning, Jean-Emmanuel Deschaud, David Prasser, and Peter Rander. 3d mapping for high-fidelity unmanned ground vehicle lidar simulation. *The International Journal of Robotics Research*, 31(12):1349–1376, 2012.

[19] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine learning*, 79(1-2):151–175, 2010.

[20] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 23–30. IEEE, 2017.

[21] Stefano Carpin, Mike Lewis, Jijun Wang, Stephen Balakirsky, and Chris Scrapper. Usarsim: a robot simulator for research and education. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 1400–1405. IEEE, 2007.

[22] Matthias Mueller, Neil Smith, and Bernard Ghanem. A benchmark and simulator for uav tracking. In *European Conference on Computer Vision*, pages 445–461. Springer, 2016.

[23] John Skinner, Sourav Garg, Niko Sünderhauf, Peter Corke, Ben Upcroft, and Michael Milford. High-fidelity simulation for evaluating robotic vision performance. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 2737–2744. IEEE, 2016.

[24] Hironori Hattori, Vishnu Naresh Boddeti, Kris M Kitani, and Takeo Kanade. Learning scene-specific pedestrian detectors without real data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3819–3827, 2015.

[25] Xingchao Peng, Baochen Sun, Karim Ali, and Kate Saenko. Learning deep object detectors from 3d models. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1278–1286, 2015.

[26] Jeremie Papon and Markus Schoeler. Semantic pose using deep networks trained on synthetic rgb-d. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 774–782, 2015.

[27] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3234–3243, 2016.

[28] Ankur Handa, Viorica Patraucean, Vijay Badrinarayanan, Simon Stent, and Roberto Cipolla. Understanding real world indoor scenes with synthetic data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4077–4085, 2016.

[29] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.

[30] Sheldon M Ross. *A course in simulation*. Prentice Hall PTR, 1990.

[31] Kurt Konolige, Joseph Augenbraun, Nick Donaldson, Charles Fiebig, and Pankaj Shah. A low-cost laser distance sensor. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3002–3008. IEEE, 2008.

[32] Cang Ye and Johann Borenstein. Characterization of a 2-d laser scanner for mobile robot obstacle negotiation. In *ICRA*, pages 2512–2518, 2002.

[33] Lindsay Kleeman and Roman Kuc. Sonar sensing. In *Springer Handbook of Robotics*, pages 491–519. Springer, 2008.

[34] Tinne De Laet, Joris De Schutter, and Herman Bruyninckx. A rigorously bayesian beam model and an adaptive full scan model for range finders in dynamic environments. *J. Artif. Intell. Res.(JAIR)*, 33:179–222, 2008.

[35] Jörg Müller, Axel Rottmann, Leonhard M Reindl, and Wolfram Burgard. A probabilistic sonar sensor model for robust localization of a small-size blimp in indoor environments using a particle filter. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3589–3594. IEEE, 2009.

[36] Jonathan Ko and Dieter Fox. Gp-bayesfilters: Bayesian filtering using gaussian process prediction and observation models. *Autonomous Robots*, 27(1):75–90, 2009.

[37] Christian Plagemann, Kristian Kersting, Patrick Pfaff, and Wolfram Burgard. Gaussian beam processes: A nonparametric bayesian measurement model for range finders. In *Robotics: Science and Systems*, 2007.

[38] William Vega-Brown and Nicholas Roy. Cello-em: Adaptive sensor models without ground truth. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1907–1914. IEEE, 2013.

[39] Barnabás Póczos, Alessandro Rinaldo, Aarti Singh, and Larry Wasserman. Distribution-free distribution regression. *arXiv preprint arXiv:1302.0082*, 2013.

[40] Junier Oliva, Barnabás Póczos, and Jeff Schneider. Distribution to distribution regression. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1049–1057, 2013.

[41] Ankur Handa, Thomas Whelan, John McDonald, and Andrew J Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam.

[42] Larry Wasserman. *All of statistics: a concise course in statistical inference*. Springer Science & Business Media, 2013.

[43] Daniel Sack and Wolfram Burgard. A comparison of methods for line extraction from range data. *IFAC Proceedings Volumes*, 37(8):728–733, 2004.

[44] Viet Nguyen, Stefan Gächter, Agostino Martinelli, Nicola Tomatis, and Roland Siegwart. A comparison of line extraction algorithms using 2d range data for indoor mobile robotics. *Autonomous Robots*, 23(2):97–111, 2007.

[45] Susanne M Schennach. Nonparametric regression in the presence of measurement error. *Econometric Theory*, 20(06):1046–1093, 2004.

[46] Raymond J Carroll, Aurore Delaigle, and Peter Hall. Nonparametric prediction in measurement error models. *Journal of the American Statistical Association*, 104 (487), 2009.

[47] Raymond J Carroll, David Ruppert, Leonard A Stefanski, and Ciprian M Crainiceanu. *Measurement error in nonlinear models: a modern perspective*. CRC press, 2012.

[48] Sam Efromovich. Orthogonal series density estimation. *Wiley Interdisciplinary Reviews: Computational statistics*, 2(4):467–476, 2010.

[49] Carl Edward Rasmussen. Gaussian processes for machine learning. 2006.

[50] David Nitzan, Alfred E Brain, and Richard O Duda. The measurement and use of registered reflectance and range data in scene analysis. *Proceedings of the IEEE*, 65(2):206–220, 1977.

[51] Martial Hebert and Eric Krotkov. 3d measurements from imaging laser radars: how good are they? *Image and Vision Computing*, 10(3):170–178, 1992.

[52] Unity game engine. URL https://unity3d.com/.

[53] Unreal engine. URL https://www.unrealengine.com/.

[54] John Tuley, Nicolas Vandapel, and Martial Hebert. Analysis and removal of artifacts in 3-d ladar data. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2203–2210. IEEE, 2005.

[55] Chris Goodin, Raju Kala, Alex Carrrillo, and Linda Y Liu. Sensor modeling for the virtual autonomous navigation environment. In *Sensors, 2009 IEEE*, pages 1588–1592. IEEE, 2009.

[56] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286. ACM, 1998.

[57] Yotam Livny, Feilong Yan, Matt Olson, Baoquan Chen, Hao Zhang, and Jihad El-Sana. Automatic reconstruction of tree skeletal structures from point clouds. *ACM Transactions on Graphics (TOG)*, 29(6):151, 2010.

[58] Jonathan Binney and Gaurav S Sukhatme. 3d tree reconstruction from laser range data. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 1321–1326. IEEE, 2009.

[59] Jean-Emmanuel Deschaud, David Prasser, M Freddie Dias, Brett Browning, and Peter Rander. Automatic data driven vegetation modeling for lidar simulation. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 5030–5036. IEEE, 2012.

[60] Leonid Pishchulin, Arjun Jain, Christian Wojek, Mykhaylo Andriluka, Thorsten Thormählen, and Bernt Schiele. Learning people detection models from few training samples. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1473–1480. IEEE, 2011.

[61] David Vázquez, Antonio M Lopez, Javier Marin, Daniel Ponsa, and David Geronimo. Virtual and real world adaptation for pedestrian detection. *IEEE transactions on pattern analysis and machine intelligence*, 36(4):797–809, 2014.

[62] Ankur Handa, Viorica Pătrăucean, Simon Stent, and Roberto Cipolla. Scenenet: An annotated model generator for indoor scene understanding. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 5737–5743. IEEE, 2016.

[63] Ankur Handa, Richard A Newcombe, Adrien Angeli, and Andrew J Davison. Real-time camera tracking: When is high frame-rate best? In *European Conference on Computer Vision*, pages 222–235. Springer, 2012.

[64] Bertrand Douillard, J Underwood, Vsevolod Vlaskine, A Quadros, and S Singh. A pipeline for the segmentation and classification of 3d point clouds. In *Experimental Robotics*, pages 585–600. Springer, 2014.

[65] Michael Himmelsbach, Felix V Hundelshausen, and H-J Wuensche. Fast segmentation of 3d point clouds for ground vehicles. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 560–565. IEEE, 2010.

[66] Alglib (www.alglib.net), sergey bochkanov. URL www.alglib.net.

[67] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.9 edition, 2016. URL http://doc.cgal.org/4.9/Manual/packages.html.

[68] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[69] CloudCompare (version 2.9.alpha) [GPL software]. (2017). Retrieved from http://www.cloudcompare.org/.

[70] Steven G. Johnson, The NLopt nonlinear-optimization package. URL http://ab-initio.mit.edu/nlopt.

[71] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[72] Franz S Hover, Ryan M Eustice, Ayoung Kim, Brendan Englot, Hordur Johannsson, Michael Kaess, and John J Leonard. Advanced perception, navigation and planning for autonomous in-water ship hull inspection. *The International Journal of Robotics Research*, 31(12):1445–1464, 2012.

[73] Michael C Koval, Mehmet R Dogar, Nancy S Pollard, and Siddhartha S Srinivasa. Pose estimation for contact manipulation with manifold particle filters. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 4541–4548. IEEE, 2013.

[74] Umberto Cherubini, Elisa Luciano, and Walter Vecchiato. *Copula methods in finance*. John Wiley & Sons, 2004.

[75] Tongtong Chen, Bin Dai, Daxue Liu, and Jinze Song. Performance of global descriptors for velodyne-based urban object recognition. In *Intelligent Vehicles Symposium Proceedings, 2014 IEEE*, pages 667–673. IEEE, 2014.

[76] David M Bradley, Ranjith Unnikrishnan, and James Bagnell. Vegetation detection for driving in complex environments. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 503–508. IEEE, 2007.

[77] Alistair Reid, Fabio Ramos, and Salah Sukkarieh. Multi-class classification of vegetation in natural environments using an unmanned aerial system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2953–2959. IEEE, 2011.

[78] Ulrich Weiss, Peter Biber, Stefan Laible, Karsten Bohlmann, and Andreas Zell. Plant species classification using a 3D LIDAR sensor and machine learning. In *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pages 339–345. IEEE, 2010.

[79] Debadeepta Dey, Lily Mummert, and Rahul Sukthankar. Classification of plant structures from uncalibrated image sequences. In *Applications of Computer Vision (WACV), 2012 IEEE Workshop on*, pages 329–336. IEEE, 2012.

[80] Joachim Niemeyer, C Mallet, Franz Rottensteiner, and Uwe Sörgel. Conditional random fields for the classification of LiDAR point clouds. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences:[ISPRS Hannover Workshop 2011: High-Resolution Earth Imaging For Geospatial Information] 38-4 (2011), Nr. W19*, volume 38, pages 209–214. Göttingen: Copernicus GmbH, 2011.

[81] Jean-François Lalonde, Nicolas Vandapel, Daniel F Huber, and Martial Hebert. Natural terrain classification using three-dimensional ladar data for ground robot mobility. *Journal of field robotics*, 23(10):839–861, 2006.

[82] Boris Sofman, J Andrew Bagnell, Anthony Stentz, and Nicolas Vandapel. Terrain classification from aerial data to support ground vehicle navigation. 2006.

[83] Daniel Munoz, Nicolas Vandapel, and Martial Hebert. Onboard contextual classification of 3-d point clouds with learned high-order markov random fields. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE, 2009.

[84] Caterina Massidda, Heinrich H Bülthoff, and Paolo Stegagno. Autonomous vegetation identification for outdoor aerial navigation. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 3105–3110. IEEE, 2015.

[85] Kai M Wurm, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. Improving robot navigation in structured outdoor environments by identifying vegetation from laser data. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1217–1222. IEEE, 2009.

[86] Benjamin Eckart, Kihwan Kim, Alejandro Troccoli, Alonzo Kelly, and Jan Kautz. Accelerated generative models for 3d point cloud data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5497–5505, 2016.

[87] Alexey Dosovitskiy, Jost Tobias Springenberg, Maxim Tatarchenko, and Thomas Brox. Learning to generate chairs, tables and cars with convolutional networks. *IEEE transactions on pattern analysis and machine intelligence*, 39(4):692–705, 2017.

[88] Alberto Hata and Denis Wolf. Road marking detection using lidar reflective intensity data and its application to vehicle localization. In *Intelligent Transportation Systems (ITSC), 2014 IEEE 17th International Conference on*, pages 584–589. IEEE, 2014.

[89] Charalambos Poullis and Suya You. Photorealistic large-scale urban city model reconstruction. *IEEE transactions on visualization and computer graphics*, 15(4): 654–669, 2009.

[90] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.