Data-driven traversability estimation for mobile robot navigation

by

Mengze Li

M.Eng. Nanjing University of Post and Telecommunication, 2014
B.Sc. Nanjing University of Post and Telecommunication , 2011

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE FACULTY OF GRADUATE STUDIES
OF LAKEHEAD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Data-driven traversability estimation for mobile robot navigation

by

Mengze Li

Supervisory Committee

Dr. Thiago Eustaquio Alves de Oliveira,
Supervisor
(*Department of Computer Science, Lakehead University, Thunder Bay, Ontario, Canada*)

Dr. Yimin Yang,
Internal Examiner
(*Department of Computer Science, Lakehead University, Thunder Bay, Ontario, Canada*)

Dr. Thangarajah Akilan,
External Examiner
(*Department of Software Engineering, Lakehead University, Thunder Bay, Ontario, Canada*)

ABSTRACT

Mobile robots have a promising application prospect as they can assist or replace humans to perform laborious, repetitive or dangerous tasks in various scenarios. There has been a large number of studies for mobile robot navigation since 1980s, while terrain traversability estimation is an important topic in this field — estimating if an area is traversable and how long will it take to drive through is necessary for navigating the robot and planning paths.

However, most existing terrain traversability estimation methods are designed based on simple fixed rules and manually tuned parameters, suffering low accuracy and poor generalization due to their simplicity of structure and biases to the environment where they are tuned. To address this problem, we proposed a set of data-driven traversability estimation methods based on Convolutional Neural Networks (CNN), which are trained and tested them in different simulation environments.

There are 3 main goals for our methods:

1. High accuracy. Accuracy of the result is the core of an traversability estimation method.

2. Low computational cost. Since most mobile robots are equiped with very limited computing power and energy, a practical traversability estimation method should be able to work with a low computational cost.

3. Good generalization. A good traversability estimation method should generalize to different type of environments or provide a function to automatically fit to a new environment instead of manual tuning.

In this thesis, we first reviewed some representative conventional terrain traversability estimation methods and introduced several related fields including mobile robot path planning, localization and map building. Then we proposed our CNN-based methods, demonstrated how to build the simulation framework and collect terrain samples with driving data. Finally we compared the performance of our work with benchmark methods in both classification and regression traversability estimation tasks on the collected datasets and proved the improvements made by our methods.

ACKNOWLEDGEMENTS

I would like to give thanks to the following funding sources, for their financial support to my research:

- **Lakehead University Faculty of Graduate Studies**;
- **Lakehead University Faculty of Science and Environmental Studies**;
- **Dr. Thiago Alves de Oliveira (Discovery Grant)**;

I would like to thank NASA-SRC for sharing the simulation environment used in the experiment. I gratefully acknowledge Dr. Oliveira and Dr. Choudhury's helpful comments about this thesis and my previous work. Thank Yan Peizhi for sharing the thesis template with me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobile robot navigation has been a research topic in high prominence since the 1980s [1–4]. The main goal of it is to guide a robot in its working environment in an efficient way to reach a destination, follow a path and/or perform any other tasks given by the user.

The application of mobile robot navigation ranges from indoor services [5], search and rescue missions [6] to extraterrestrial exploration [7]. Deploying mobile robots with autonomous navigation is an solution for handling tasks in those environments that are too dangerous or far away for deploying manned vehicles.

1.1 Mobile robot navigation

There are four main steps in mobile robot navigation: localizing the robot, building a map, planning the path and finally following the path towards the destination.

Knowing its location is essential for a robot to perform all types of tasks. Popular robot localizing methods include Simultaneous Localization And Mapping (SLAM) and Wireless Localization based on Received Signal Strength Indicator (WL-RSSI).

SLAM methods estimate the robot location by fusing the motion model of robot and observations from the environment in a probability estimator. There are two key solutions for SLAM problem: Extended Kalman Filter SLAM (EKF-SLAM) assumes the noise in motion estimation is Gaussian additive. While Rao-Blackwellised based Filter SLAM (FastSLAM) directly represents the non-linear process model and non-Gaussian pose distribution through recursive Monte Carlo sampling or particle filtering. By applying Rao-Blackwellised particle filtering, the sample space is reduced

and thus having a relatively lower computational complexity.

Being able to localize itself without requiring external information and build a map simultaneously is the main advantage of SLAM. However, it usually performs poorly when applied to environments where same landmarks can not be easily observed again or over long distance driving in large areas due to its high computation cost and long term accuracy loss, and not all SLAM methods generate a map can be used for robot path planning (e.g. Visual Fast SLAM).

On the other hand, WL-RSSI methods provide a more accurate way for localizing the robot at a cost of deploying wireless emitters and pre-calibration of their signal strength pattern — before performing tasks, robot travels in the target area and collect wireless signal strength of each emitter coupled with ground truth coordinates provided by high-precision localizing methods such as triangulation. Then these collected data points will help to build a function (usually by training a machine learning model, e.g. a neural network) from RSSI to robot pose.

The next step is building a map of the environment in a structure upon which path planning can be done efficiently. Nowadays the most widely used map formats for mobile robot navigation planar are grid maps and maps based on point clouds — a set of points with coordinates and other supplementary information (e.g. color) representing the surface of ground and objects of the environment, and grid/elevation maps derived from it. Individual local point clouds are captured by a depth camera mounted on the robot. Then, by virtue of the robot pose information obtained in the previous localizing step, these local point clouds can be merged into a global map, outlier and redundant points can be filtered out while fitting methods are applied to align them to minimize map error. Depending on the specific application scenario, such point cloud maps may be transformed into octrees or elevation maps for further use.

Finally, in the path planning step — to perform tasks given by the user, a mobile robot usually needs to arrive at a series of designated locations with its limited capability of overcoming obstacles and battery capacity, therefore planning an optimal path to the destination is necessary. Various methods are used to find that optimal path, heuristic searching methods such as A* are common for searching in grid maps while Rapidly-exploring Random Tree (RRT) and its heuristic version RRT* are widely used in non-grid maps.

As the time, energy and other costs for driving on different type of terrain varies greatly, especially in unstructured outdoor environments, estimating the traversability

of terrain becomes essential for robot navigation, especially for path planning — the estimated traversability for terrains plays the role of the "distance" between grids/nodes in those aforementioned planning methods which is the key for finding an optimal path with lowest cost.

1.2 Traversability estimation

An intuitive way to estimate traversability is building a map for the target terrain and analysing geographic factors within it. Popular map types include elevation maps [8], 2.5D grid maps [9] and maps based on point clouds [27].

In this type of methods, traversability is usually computed as a weighted sum of map features like roughness and slope of the area where the mobile robot drives on. Such estimation methods are highly interpretable since the data and functions used in the computation are given explicitly, and thus parameters must be tuned manually when they are applied to a new environment.

Another option for traversability estimation is directly generating an index for traversability — in either a classification or regression way — from images of the target terrain. In [24], geographic features including rocks, slopes and horizon are extracted from the images captured by a robot and used to compute the traversability through a set of fuzzy rules, without the need to build any map. Kim *et al.* [25] proposed a way to learn the image-traversability mapping through unsupervised on-line learning to reduce the work of collecting training data before deployment.

There is also a set of researches like [7] focusing on estimating the motion and trajectory of robot on given terrains rather than a simple class or index of traversability. These methods can provide much more detailed estimation for better control of mobile robot driving at a cost of being highly dependent on training environment and robot kinematic information.

However, there are flaws in these methods: Manually selected features and functions combining them will inevitably introduce bias into the estimation and prone to be too simple to fit complex traversability pattern on terrains; Using raw map data (e.g. raw point clouds in [27]) leads to high computation cost; Estimation directly made on highly detailed information (e.g. photos in [25] and the robot trajectories in [7]) usually pay with inferiority on accuracy because the training data may be biased to their own experiment environments thus hard to generalize to other scenarios, and the estimating rule is also interpretable.

To address these issues, this thesis investigates several elements of mobile robot navigation, such as robot localization, map building and path planning with the objective of developing a data-driven framework for estimating terrain traversability using Convolutional Neural Network (CNN) models, as well as a training dataset of terrain samples and traversability indicators.

There are 3 main goals for our traversability estimation framework:

1. High accuracy. The traversability estimation accuracy of most conventional methods are fundamentally limited by their simplicity in structure. On the contrary, by virtue of their universal approximating ability, methods based on neural networks can achieve high level accuracy through selecting appropriate networks and enough training no matter what the underlying pattern of traversability is.

2. Low computation cost. Mobile robots usually have very limited energy, thus reducing the computation cost for traversability estimation can help to extend their working time between each recharge. This improvement can also increase the robot task performing efficiency as it shortens the time used in path planning.

3. Low bias and better generalization. Manually designed hyperparameters are the key to conventional traversability estimation methods. However, they will be inevitably biased to the environments where they are originally tuned and tested. Tuning such parameters whenever the method is applied to a new type of environment is laborious. We plan to design a data-driven traversability estimation framework based on CNN which can automatically fit unseen environments by training on new data, therefore improving its generalization while saving human labor.

More discussions of existing traversability estimating methods will be made in chapter 2, following by reviews of robot localizing, map building and path planning in chapter 3-5. Chapter 6 is our approach and experiment deign, while chapter 7 shows experiment results which justify the improvements made by our method. At the end, a conclusion is given in chapter 8.

Chapter 2

Path planning

To perform tasks at user-specified locations, an autonomous mobile robot should be equiped with path planning algorithms to find a valid path from its current location to the goal point. Furthermore, the driving cost, such as time and energy consumption, should also be taken into consideration to find an optimal path, where traversability estimation plays an essential role.

In this chapter, we will introduce several path planning methods for mobile robots and show how they optimize the resulting path based on the length of straight edges, which must be obtained by traversability estimation in real-world applications.

2.1 Dynamic Visibility Graph (DVG)

In vanilla visibility graph algorithm, a shortest path is obtained by connecting tangent lines between vertices of obstacles that cross the S-G line — the straight line connecting start point and goal point, as shown in Fig 2.1:

To reduce the number of vertices involved in computation, an active region is defined here by vertices with maximum distance from the S-G line, denoted as M_P in Fig 2.1, which belong to the obstacles crossed by the S-G line. Only vertices within this region will be considered in the shortest path computation.

However, there may exist a shorter path outside the active region as Fig 2.2a shows. To address this problem, Huang and Chung [10] proposed Dynamic Visibility Graph (DVG), which ensures the shortest path is located in the active region by iteratively expanding it if shorter outer path is found.

When terrain traversability are taken in to consideration, the actual robot driving

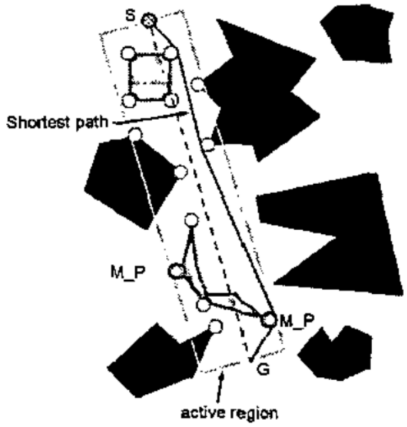
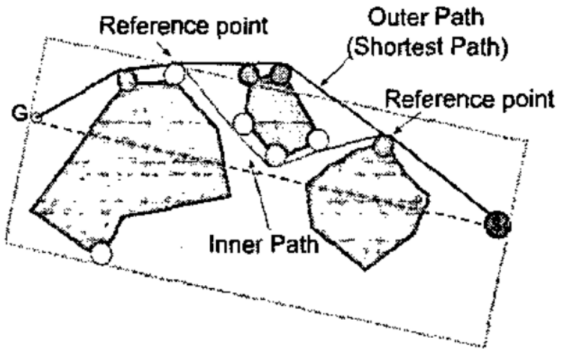
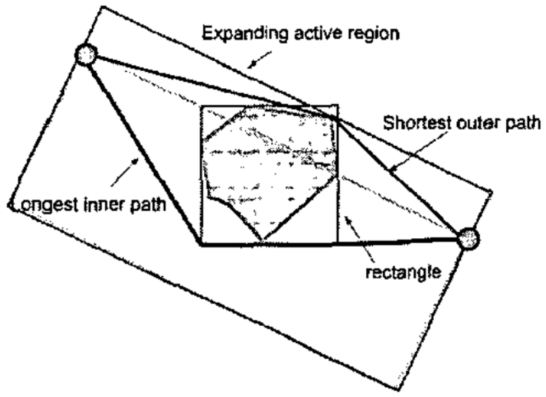


Figure 2.1: Active region in visibility graph (from [10])



(a) A case that shortest path is out of the active region



(b) Expanding active region

Figure 2.2: Active region expansion (from [10])

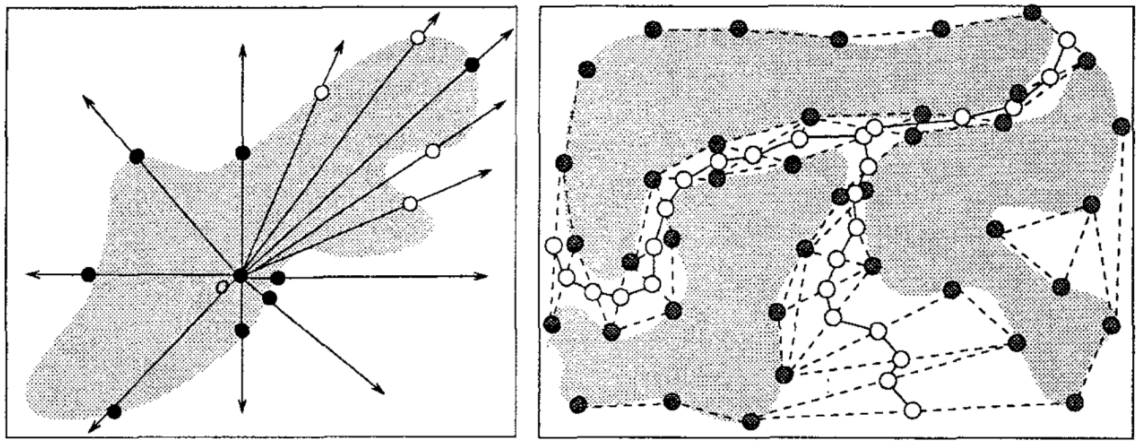
cost for each path may no longer be proportional to its Euclidean length. Therefore, when computing the shortest path and determining whether to expand the active region, the traversability indicator of straight edges will be used as their length.

2.2 Randomized Roadmap

A computationally efficient way to find a path between arbitrary start and goal point in an area is building a roadmap — a graph with valid robot configurations as vertices and straight path without obstacles as edges. Amato and Wu [11] proposed a randomized roadmap method which can obtain high quality roadmaps even in a configuration space (C-space) full of obstacles.

The first step of this method is evenly generating nodes at the surface of every

obstacle. As shown in Fig 2.3a, multiple uniformly distributed rays are draw from the center of an obstacle (e.g. an average of the coordinates of its vertices), then use binary search to locate points where the obstacle boundary intersects with these rays. The resulting point distribution may not close to uniform when the shape of obstacle is irregular. To remedy this problem, new rays and points (such as transparent nodes in Fig 2.3a) will be added when the Euclidean distance between two neighbor points or the difference between their surface normal orientation is too high.



(a) generating nodes (b) connecting nodes

Figure 2.3: Randomized roadmap building process (from [11])

The second step is attempting to connect these surface nodes to their k-nearest neighbors as Fig 2.3b shows, thus a roadmap is built. When planning a new path, this method will first try to connect the start and goal point to their k-nearest neighbor nodes in the roadmap. If failed, it will make a random walk and try to connect the end node. This procedure will be repeated until both the start point and the goal point are connected to the roadmap or a threshold of attempt is reached. Finally, a valid path could be found through a breadth-first search or shortest path computation (e.g. Dijkstra algorithm).

Similar to the situation in the aforementioned DVG method, traversability indicators can be computed for each edge in the roadmap and used for searching the shortest path. This can help to find path with lower robot driving cost in real-world applications rather than paths with shorter Euclidean length.

2.3 Windowed Hierarchical Cooperative A* (WHCA*)

In some scenarios, only the local map of the robot's vicinity and the orientation of the goal point are provided instead of a prior global map. Silver [13] proposed a heuristic method based on grid map — Windowed Hierarchical Cooperative A* (WHCA*) for path planning in such scenarios.

This algorithm generates the distance to every goal for all grids in a grid map as the heuristic for path searching. Then it finds the path for the robot through a Reverse Resumable A* (RRA*) search based on a reservation table. In each time window, all available paths are searched in order. At every time step, visited grids of previously determined paths will be "reserved" in a table, no longer available for following paths in the current time window. This hierarchical search makes it possible to plan paths for multiple robots at the same time. Windowing the search also limits the searching depth, which improves the computational efficiency by avoiding calculating long-term contingencies which may not actually occur.

When applied to an unknown environment, the heuristic can no longer be calculated based on the complete map information such as RRA* did in WHCA*. Sartoretti *et al.* [14] designed a heuristic function computed distributively by the robot which only depends on the information of its field of vision (FOV), as shown in Fig 2.4. As the positions of all goals are known, the robot will be provided with a unit vector pointing to its goal and a magnitude proportional to its Euclidean distance to the goal at all times, to help them navigate in the unknown map. This vector and magnitude, plus with FOV sub-maps of obstacles, robot's and neighbors' positions, neighbors' goals (mapped to a grid on the edge if outside of the FOV) and its own goal, would be inputted to a neural network to get a navigation model through training.

The robot driving cost for crossing a grid can be estimated based on the terrain, and the length of a path will be computed as the sum of the cost of all grids that make up it instead of just counting the number of grids. By virtue of this, the heuristic search can find better paths with lower actual driving cost.

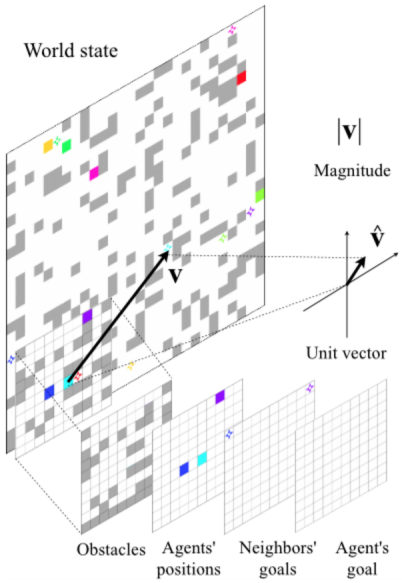


Figure 2.4: FOV of each robot (here for the one in the light blue grid), neighbour robots are marked by other color, goals of each robot are marked by star with the same color (from [14])

2.4 Rapidly exploring Random Trees (RRT)

Rapidly exploring Random Trees (RRT) [15–17] is a simple and efficient randomized algorithm for solving single-query path planning problems in high-dimensional configuration spaces.

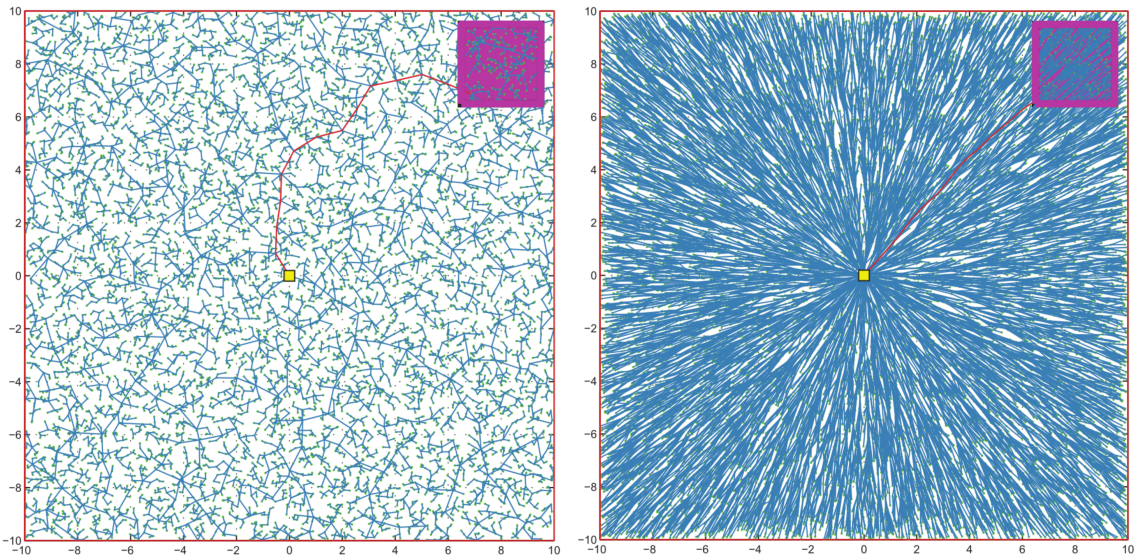
A tree rooted from the start point is incrementally built as the RRT algorithm shown in Algorithm 1 — in each iteration, a new point x_{rand} is randomly sampled from the free C-space, then its nearest vertex in the current tree $x_{nearest}$ will be found to execute a "steer" operation, that is connecting these two points if the distance between them is no greater than a given threshold. Otherwise an edge from $x_{nearest}$ to x_{rand} with maximum step length will be created instead, where the end point is denoted as x_{new}. Finally, vertex x_{new} and edge ($x_{nearest}$, x_{new}) will be added to the tree.

The execution of the RRT algorithm will end when the goal point is added into the tree. By setting the probability of sampling the goal point as x_{rand}, the user can decide how greedily the tree grows towards the goal.

Algorithm 1: RRT

$V \leftarrow x_{init}$;
$E \leftarrow \emptyset$;
for $i = 1 : n$ **do**
    $x_{rand} \leftarrow SampleFree_i$;
    $x_{nearest} \leftarrow Nearest(G = (V, E), x_{rand})$;
    $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$;
    if $ObstacleFree(x_{nearest, x_{new}})$ **then**
        $V \leftarrow V \cup x_{new}$;
        $E \leftarrow E \cup (x_{nearest}, x_{new})$;
return $G = (V, E)$;

The resulting path obtained through vanilla RRT is very likely to be jagged as shown in Fig 2.5a.



(a) RRT (b) optimal RRT (RRT*)

Figure 2.5: RRT vs. RRT* resulting tree and path (from [18])

One way to address this problem is rewiring x_{new} and its nearby tree nodes every time a new point is added — among the set X_{near} consisting of tree nodes within a specific radius of x_{new}, the node x_{min} with the lowest cost to connect to x_{new} will be selected to be the parent of x_{new}. Afterwards, all the other $x_{near} \in X_{near}$ will be rewired as x_{new}'s child node if the path cost is reduced in this way. Rewiring operation can help the resulting path of RRT converge to the optimal, therefore, this variant of RRT is named as optimal RRT (RRT*). Paths obtained by RRT* are much

less jagged as shown in Fig 2.5b. It is worth noting that standard RRT and RRT* only consider the Euclidean length of paths. However, in real-world scenarios there is likely to be paths with longer length but lower driving cost. To find actual optimal paths in practice, the traversability of terrain must be taken in to consideration in path planning.

As discussed in this chapter, terrain traversability estimation plays an essential role in real-world applications of path planning methods — this is because the actual driving cost of a path may not simply be proportional to its length but highly dependent on features of the terrains it passes. There are three main requirements for traversability estimation: high accuracy, low computation cost and good generalization to different types of terrain. These requirements are the criteria for evaluating existing traversability estimation methods and also the goals for designing our own method. More discussions about traversability estimation will be made in the next chapter.

Chapter 3

Terrain traversability estimation

Terrain traversability estimation is an essential part of mobile robot navigation since an accurate estimation can help a robot to know where is non-traversable to avoid getting stuck there, and it is also helpful in robot motion control, path-planning and exploration.

Many studies have been made in this field. There are two common types of input data for terrain traversability estimation methods — raw images/videos and point clouds captured by depth cameras, laser range finders or other Lidar devices. On the other hand, based on the final estimation result, these methods can be divided into 3 categories:

1. Classification of terrain traversability — terrains are classified into traversable and non-traversable classes or several other traversability classes. Such methods focus on identifying and avoiding non-traversable terrains. Fuzzy logic is often used to combine multiple fuzzy rules for the estimation.

2. One or several magnitudes for traversability — these magnitudes usually correspond to some robot driving costs, for example: they can be proportional to the time cost, energy cost or the probability for the robot to pass through. Properties of the terrain, such as height difference, slope inclination could be used to calculate the magnitude.

3. Indicators for robot motion control and/or path planning — in this category, terrain traversability usually is not estimated directly and explicitly. Instead, some traversability related indicators are calculated and used for predicting/-controlling the motion of the mobile robot or planning a path for exploring the

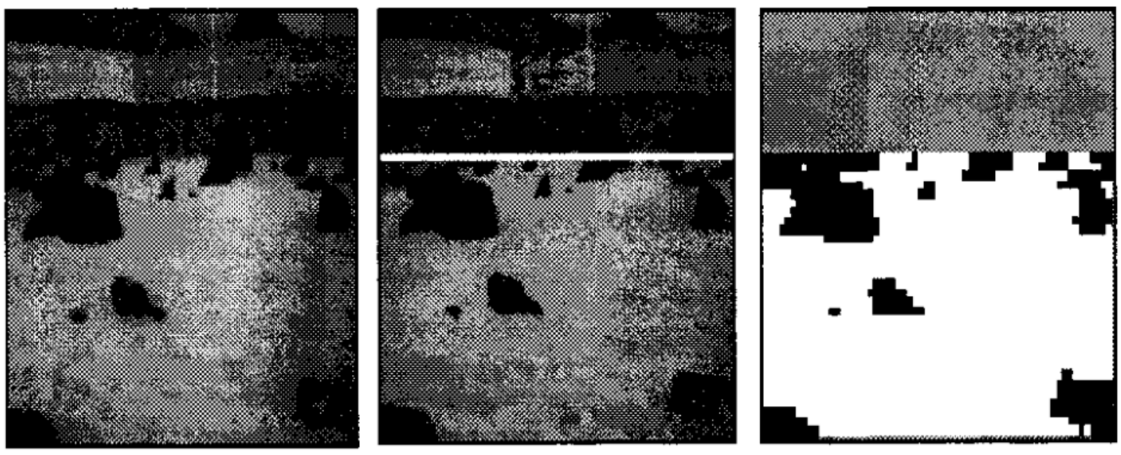
given area in an efficient way.

Several representative traversability estimation methods of these categories will be introduced in this chapter, in addition with an analysis of their drawbacks to show how can we make improvements on the estimation performance.

3.1 Terrain traversability classification

A research proposed by Howard and Seraji [24] in 2000 is a representative example for ordinary terrain traversability classification methods. It focuses on planetary environments for autonomous rovers, a set of fuzzy rules are applied to assess a terrain based on roughness — a measurement calculated from the number and size of rocks and slopes within it.

As shown in Fig 3.1, all terrain features are extracted from raw input images provided by a pair of cameras mounted on the rover. There cameras are calibrated in advance to determine the relationship between their input images and therefore locate real-world object position. A horizon line is extracted to discriminate the background and the ground with rocks. Inclinations of slopes on the ground are also calculated accordingly.



1. Original Image 2. Horizon Line Extraction 3. Rock Detection

Figure 3.1: Roughness calculation (from [24])

After these rock and slope features are calculated, fuzzy rules are applied as showed in Fig 3.2 to generate a roughness level of this terrain. Besides, when the rover is

turning, roughness of the terrain in front of it, on its left and on its right will all be taken into consideration to give an overall traversability estimation.

ROUGHNESS \ SLOPE	FLAT	SLOPED	STEEP
SMOOTH	HIGH	MED	LOW
ROUGH	MED	LOW	LOW
ROCKY	LOW	LOW	LOW

Figure 3.2: Fuzzy rules for the traversability index (from [24])

In 2006, Kim *et al.* [25] conducted a more comprehensive study and proposed an unsupervised traversability classification learning framework with autonomous data collection for robot navigation in more complex outdoor environments.

As shown in Fig 3.3, they deployed a wheeled robot in outdoor environments with lush vegetation. A local grid map is built for the vicinity of the robot where grids are divided and corresponded to visual features by a standard stereo rig.

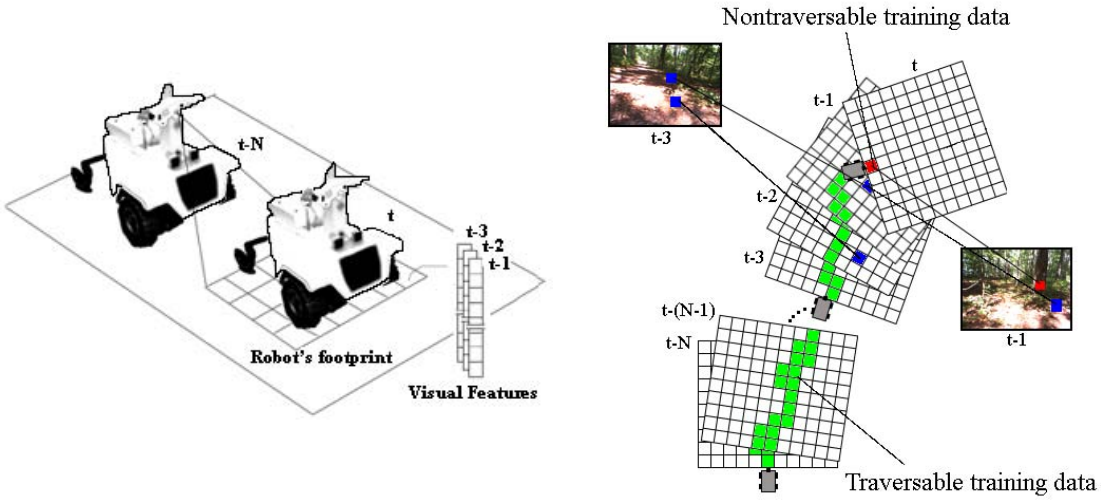


Figure 3.3: Autonomous data collection and labeling (from [25])

The robot keeps imaging the terrain region in front of it, extracting a set of features from the image and assign them to the corresponding map grid. After that, when the robot tries to drive on a previously captured grid, the grid is labeled by the driving

result — traversable or non-traversable, and this label is then paired with the visual features extracted from the image of this grid before as a training sample.

For the autonomous data collection, coordinates of the goal and the robot itself are given by GPS, the best cost path avoiding colliding with any obstacle is computed by a standard path planning algorithm which can work in two modes: conventional and learning-based. In the conventional mode, the cost map is built only based on the elevation of terrain computed through stereo camera. While in the learning-based mode, traversability of terrain in the cost map is given by an on-line classifier, there are 3 possible outcomes — traversable, non-traversable and unknown.

The traversability estimation method is tested in two test sites with different type of unstructured outdoor terrain. The trajectories of robot in these tests with different path planning mode are shown in Fig 3.4 and 3.5:

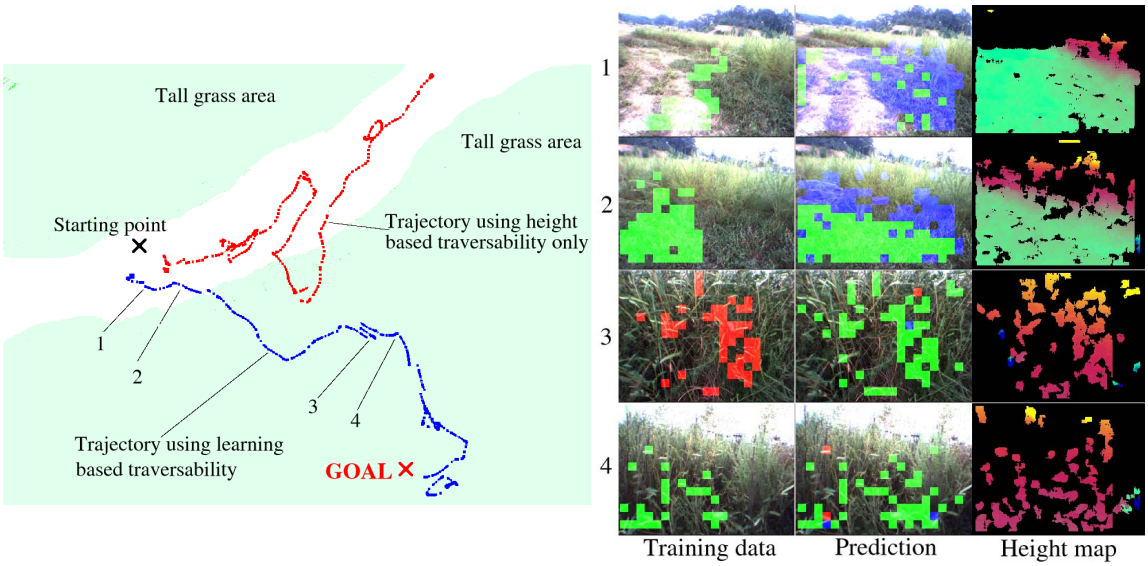


Figure 3.4: Paths and images of the first test site (from [25])

Here, red trajectories correspond to the conventional path planning mode, while blue trajectories are planned and executed in learning-based mode. In the images on the right side, green, red and blue grids represent traversable, non-traversable and unknown patches respectively. Images on the rightmost column are elevation map got through the stereo camera mounted on the robot, the color of points ranging from blue to yellow represents the elevation from lowest to highest.

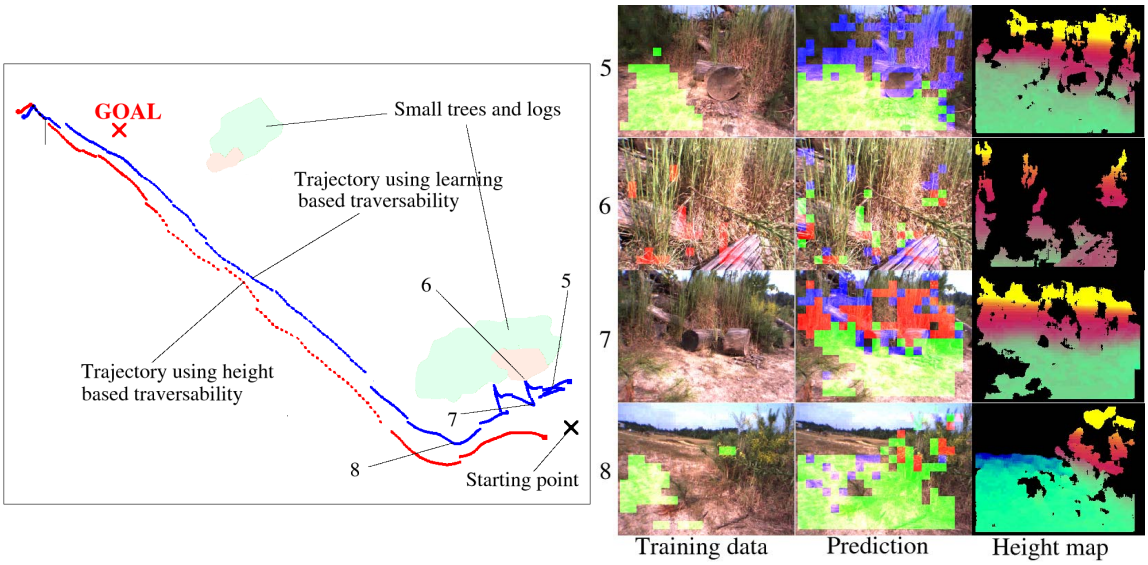


Figure 3.5: Paths and images of the second test site (from [25])

Experiment in the first test site demonstrates how the learning-based path planning can lead the robot to the goal located in tall grass area where is considered non-traversable and inaccessible by the conventional elevation based planner. This is because the online learner can find the tall grass area is traversable by trying to drive on it. On the other hand, the second test shows how the robot learn to avoid obstacles which is well-suited to the conventional method — by trying to drive on the obstacle several times from different directions, the the learning-based planner finally learns how to recognize and avoid it.

The online learner keeps clustering samples based on their coordinate in the hyper space of 13 extracted features. When a new sample feature vector is inputted, all existing clusters whose center is within a given radius from it will be updated accordingly, if there is no such cluster, a new one will be created. Clusters will be labeled by labeled samples within it. At the same time, new input feature vectors(which have not been labeled yet) will be classified depending on the ratio of the distance to its closest non-traversable cluster and the distance to its closest traversable cluster — if the ratio exceeds a given threshold, the terrain will be predicted as traversable, otherwise non-traversable.

3.2 Terrain traversability magnitude estimation

Nowadays, RGB depth cameras are widely used in autonomous mobile robots since they can generate accurate 3D maps of the vicinity in the form of point clouds. To more efficiently utilize the map to navigate mobile robot, many researches have been done for building traversability magnitude or descriptor on the point cloud.

3.2.1 Unevenness Point Descriptor (UPD)

In [26], Bellone *et al.* proposed Unevenness Point Descriptor (UPD) as a novel indicator for estimating the traversability of a terrain through its 3D point cloud representation. UPD is derived from Principal Component Analysis(PCA), for a point p_q in point cloud I, its neighborhood P_q^k is defined as:

$$P_q^k = \{p_i^k \in I : |p_i^k - p_q| \leq d_m \ \ \forall i = 1, 2, \ldots, k\} \tag{3.1}$$

where d_m is a given searching radius for neighbor points, k is the number of neighbors of p_q, and $|\cdot|$ is the symbol for Euclidean distance calculation.

Given $\vec{n_i}$ is the normal vector of point p_i computed through PCA on its neighborhood, define the sum of all n_i of neighbors of point p_q as:

$$\vec{r}_k^{\,q} = \vec{n}_1 + \vec{n}_2 + \cdots + \vec{n}_k = \sum_{i=1}^{k} \vec{n}_i \tag{3.2}$$

Then, compute a local inverse "unevenness index" ζ_k^q:

$$\zeta_k^q = \frac{||\vec{r}_k^{\,q}||}{k} \tag{3.3}$$

Finally, the UPD of point p_q is defined as:

$$UPD(p_q) = \{r_x^q, r_y^q, r_z^q, \zeta_k^q\} \tag{3.4}$$

here r_x^q, r_y^q, r_z^q are the scalar components of $\vec{r}_k^{\,q}$, they provide information about the orientation of the local surface in the global map frame. On the other hand, ζ_k^q represents the degree of local roughness, which is determined by the distribution of the direction of normal vectors in the neighborhood as shown in Fig 3.6.

Besides roughness, orientation is another factor should be taken into consideration

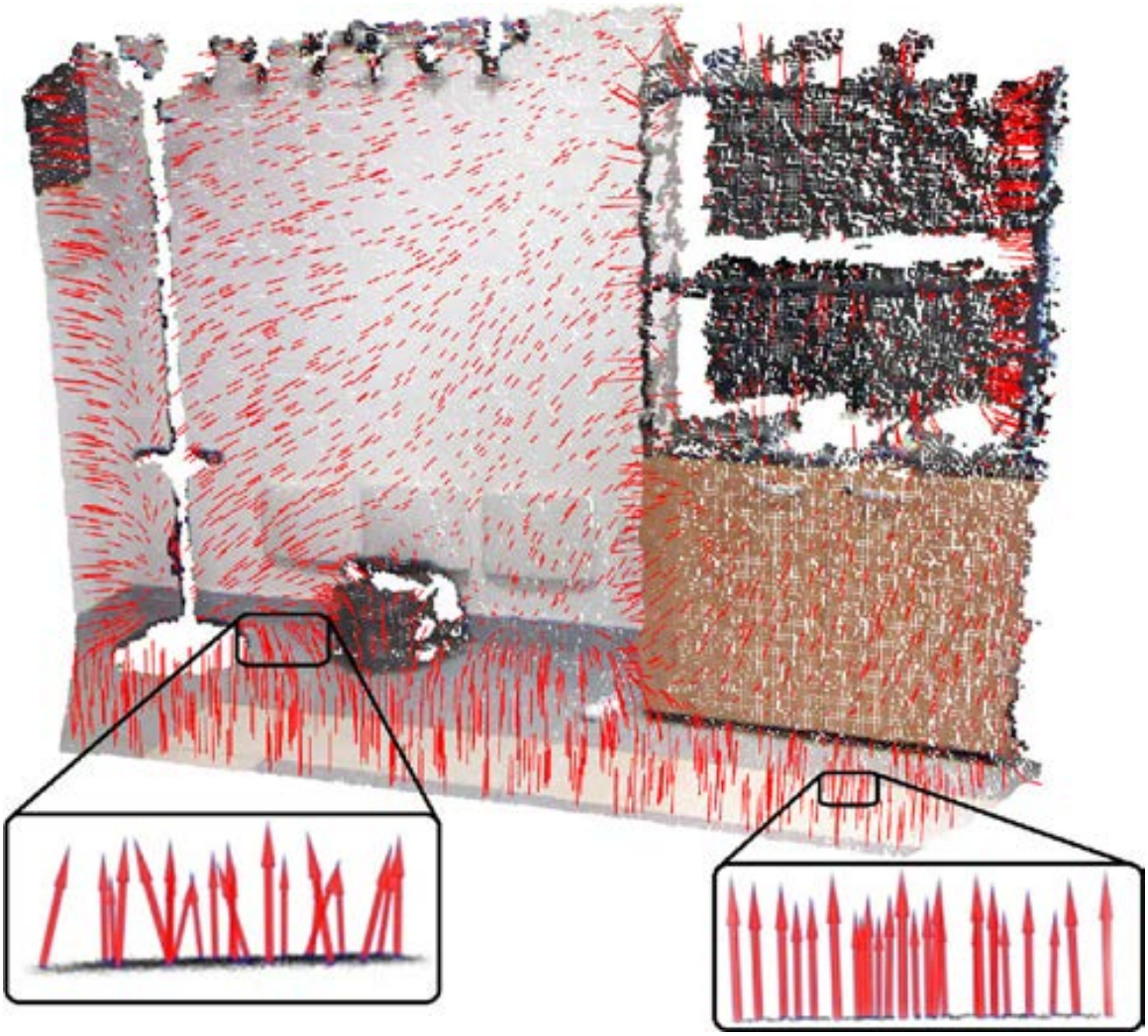


Figure 3.6: left: Typical distribution of normal vectors found in an irregular region; right: typical distribution of normal vectors in a regular region (from [26]).

for traversability estimation. A ground reference can be obtained from GPS or IMU and used to calculate the angle between the r vector of a given point p_q and the z-axis:

$$\alpha_q = \arccos\left(\frac{r_z^q}{||r_q||}\right) \tag{3.5}$$

A threshold α_{max} can be set according to the mobile robot capability, points with $\alpha_q > \alpha_{max}$ will be considered as non-traversable. From Fig 3.7 it can be seen that UPD is able to mark the ramp with an angle less than the threshold as traversable whereas ordinary roughness index method would misclassify it as non-traversable

because the roughness index is very low as the ramp is an even and smooth surface.

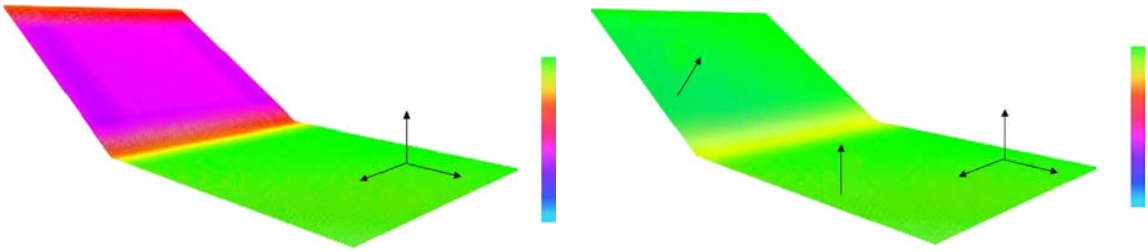


Figure 3.7: Traversability estimation result for a ramp by ordinary roughness index(left) and UPD(right), estimated traversable area are marked with green color.

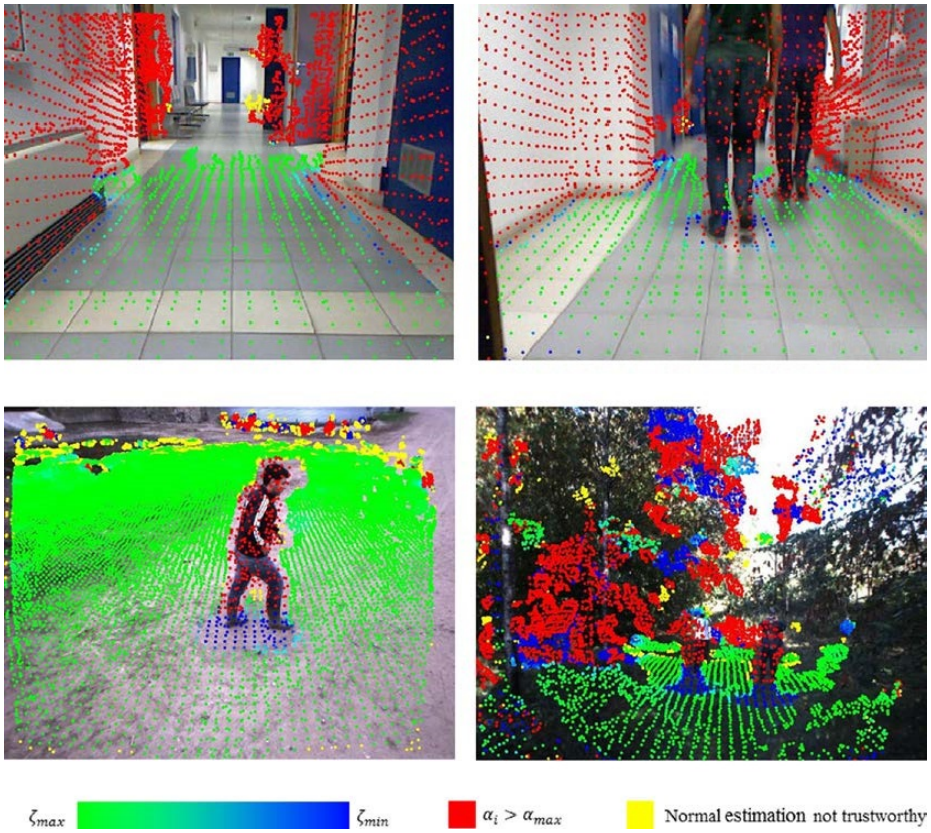


Figure 3.8: UPD-based traversability estimation results in indoor(top) and outdoor(bottom) scenarios (from [26]).

Tests have been done in both indoor and outdoor environments to evaluate the accuracy of UPD-based estimation. Experiment results prove that UPD makes a good traversability classification in both scenarios as shown in Fig 3.8.

3.2.2 Terrain roughness based method

Philipp *et al.* proposed a more comprehensive pointcloud-based traversability estimation method in their study about mobile robot navigation [27]. Like the UPD, result of this method is also composed of roughness and orientation factors.

In this method, the neighborhood of a point $\mathbf{p}_i$ in the point cloud is defined as the set including all points within a given radius r_{plane} from $\mathbf{p}_i$. To access the roughness of this neighborhood, a plane is determined by applying PCA on these points, its normal is denoted as $\mathbf{n}_i$ while the gravity center is $\overline{\mathbf{p}}_i$.

Then the signed distance between points along $\mathbf{n}_i$ can be computed as:

$$d_{ik} = (\overline{\mathbf{p}}_i - \mathbf{p}_k) \cdot \mathbf{n}_i,\ k \in \mathcal{B}_{res}(\mathbf{p}_i) \tag{3.6}$$

where $\overline{\mathbf{p}}_i$ is the gravity center of the neighborhood of point $\mathbf{p}_i$, $\mathcal{B}_{res}(\mathbf{p}_i)$ is a set of indices of all points within a sphere of radius r_{res} around $\mathbf{p}_i$.

The terrain roughness at $\mathbf{p}_i$ is defined as the absolute value of the difference between the maximum and minimum d_{ik} in $(d_{ik})_{k\in\mathcal{B}_{res}(\mathbf{p}_i)}$.

However, the result may be polluted by outlier points. To attenuate the influence of noise, a fixed fraction $f_\eta < 1$ of points will be filtered out as outliers:

$$N_i = ceil\left(\frac{f_\eta |\mathcal{B}_{res}(\mathbf{p}_i)|}{2}\right) \tag{3.7}$$

here $|\mathcal{B}_{res}(\mathbf{p}_i)|$ is the number of points within r_{res} from $\mathbf{p}_i$. N_i points with highest d_{ik} and N_i points with lowest d_{ik} are defined as outliers, the set of them is noted as $\mathcal{O}_{res}(\mathbf{p}_i)$.

Then the terrain roughness ρ_i at point $\mathbf{p}_i$ can be computed as:

$$\rho_i = |\max_k(d_{ik}) - \min_k(d_{ik})|, k \in \mathcal{B}_{res}(\mathbf{p}_i) \setminus \mathcal{O}_{res}(\mathbf{p}_i) \tag{3.8}$$

The relationship of r_{plane}, r_{res}, $\mathbf{n}_i$, $\mathbf{p}_i$ and ρ_i are shown in Fig 3.9.

The overall terrain roughness of the ground under the robot is computed based on the set of points within a corresponding cuboid of the robot's size. Given the pose (including position and orientation) of the robot as $\mathbf{T}$, the overall roughness is a normalized result from all points:

$$\rho_{cub} = \frac{1}{|\mathcal{C}_{rob}(\mathbf{T})|} \sum_{m\in\mathcal{C}_{rob}(\mathbf{T})} \rho_m \in [0, \rho_{max}] \tag{3.9}$$

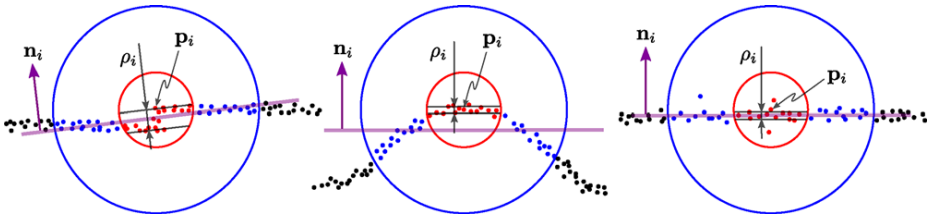


Figure 3.9: Computation of the per-point terrain roughness value for a map point $\mathbf{p}_i$. Spheres with radius of r_{plane}, r_{res} are in blue and red color respectively (from [27]).

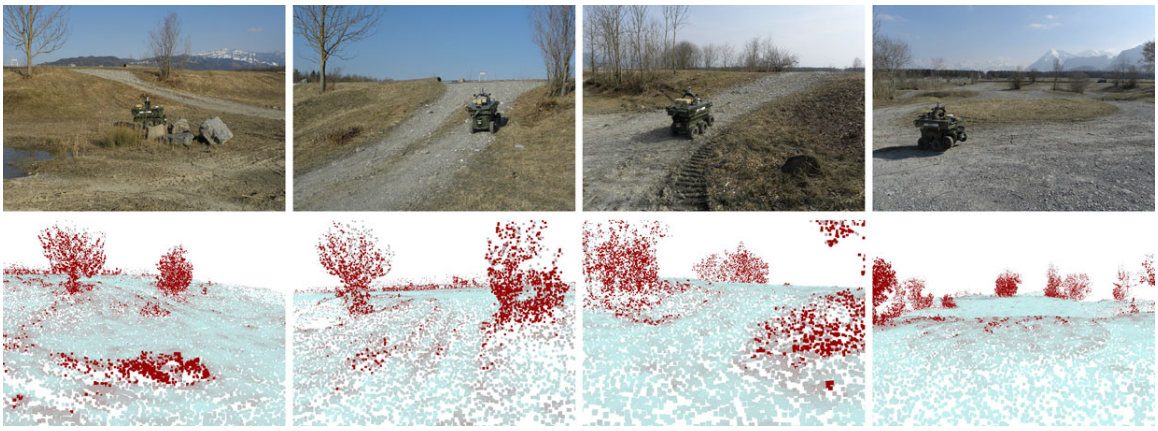
where ρ_{cub} is the overall terrain roughness of the cuboid, $|\mathcal{C}_{rob}(\mathbf{T})|$ is the number of points within the cuboid, ρ_{max} is a fixed threshold of roughness given by the user — any terrain including a point with roughness higher than ρ_{max} will be directly classified as non-traversable.

Besides roughness, the roll and pitch angles of the pose of the mobile robot are also be used to compute the terrain traversability:

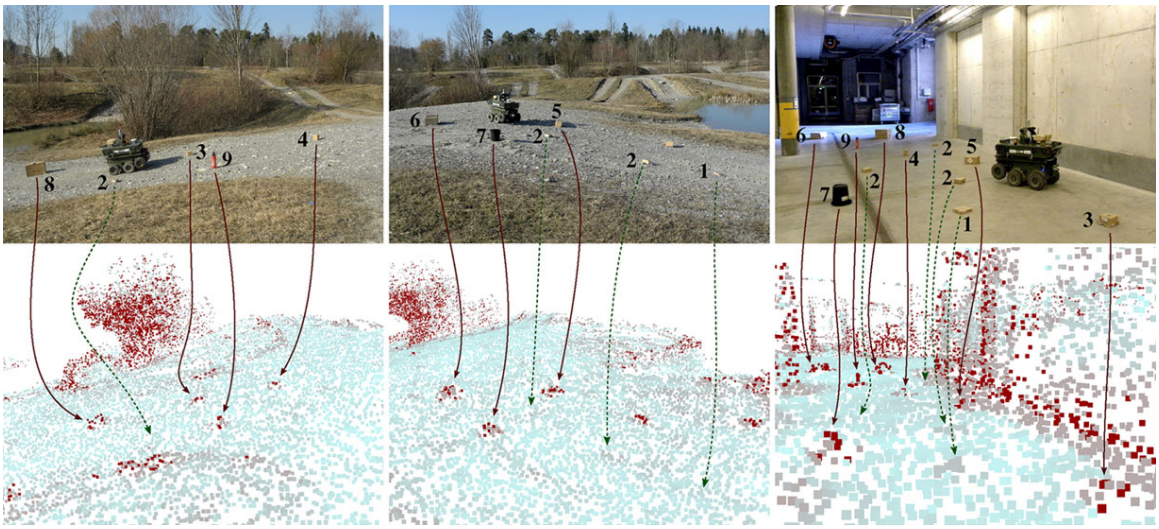
$$\begin{aligned}\tau = \; & 1 - w_{rough}\frac{\rho_{cub}}{\rho_{max}} + w_{roll}\frac{|\psi|}{\psi_{max}} \\ & + w_{pitch}\max\left(\frac{\theta}{\theta_{min}}, \frac{\theta}{\theta_{max}}\right) \in [0,1]\end{aligned} \tag{3.10}$$

here ψ, θ are the roll and pitch angle of robot pose; ψ_{max} is a user-defined bound of roll angle, and since the robot's capability of driving on slopes may different for ascending and descending, there are independent lower bound θ_{min} and upper bound θ_{max} of pitch angle; w_{rough}, w_{roll} and w_{pitch} are weights for balancing these three factors about traversability. τ is the final traversability index, ranging from 0 to 1.

This terrain traversability assessment method is applied on a wheeled robot and tested in various environments as shown in Fig 3.10 — the first row of each sub figure is photographs of different scenes, the second row is corresponding point clouds colored according to the resulting per-point terrain roughness, where non-traversable areas such as trees, walls and obstacles are marked as dark red, while traversable areas such as roads and flat grounds are marked as light blue to gray based on their roughness values.



(a) Terrain traversability assessment results in an unstructured, non-planar environment



(b) Obstacle detection

Figure 3.10: Terrain traversability assessment results, non-traversable areas are marked by red points (from [27])

3.2.3 Disadvantages of parameterized traversability estimation methods

However, hyperparameters that have a significant impact on the traversability estimation result, such as neighbor radius d_m, inclination threshold α_{max} in UPD [26] and plane radius r_{plane}, roughness threshold ρ_{max} in the terrain roughness based method [27] have to be tuned manually. This means it requires a lot of manual labor to find a parameter configuration with accurate results, and this manual tuning work needs to be done repeatedly when the method is applied in different type of

environments. Furthermore, there is even no guarantee that such a configuration can be found within acceptable time cost.

Another main disadvantage of parameterized methods is that they don't have enough approximating ability to depict the rules and features determining the traversability, and thus give inaccurate results because of underfitting and ignoring important details in the environment.

A pair of simplified UPD and terrain roughness based traversability estimation method are tested in our experiment as benchmarks. Experiment results show that there can be a clear difference between hyperparameters learned from the training dataset and the optimal settings on the testing dataset. Moreover, these benchmark methods suffer a relatively low accuracy of results even when they are directly optimized on the testing dataset. More details and discussions about it will be given in the experiment chapter.

3.3 Vehicle configuration prediction

Another way to estimate terrain traversability is directly predicting the pose of mobile robot — including altitude and configuration angles(roll, pitch, yaw) — during its driving.

Ho *et al.* [28] used Gaussian Process (GP) as a continuous representation of the vehicle pose over terrain and proposed a method to estimate traversability over terrain by learning vehicle response by experience. GP regression is performed on the input data which is a combination of elevation map, vehicle kinematics and experience(Kin-GP-VE) as Fig 3.11 shows.

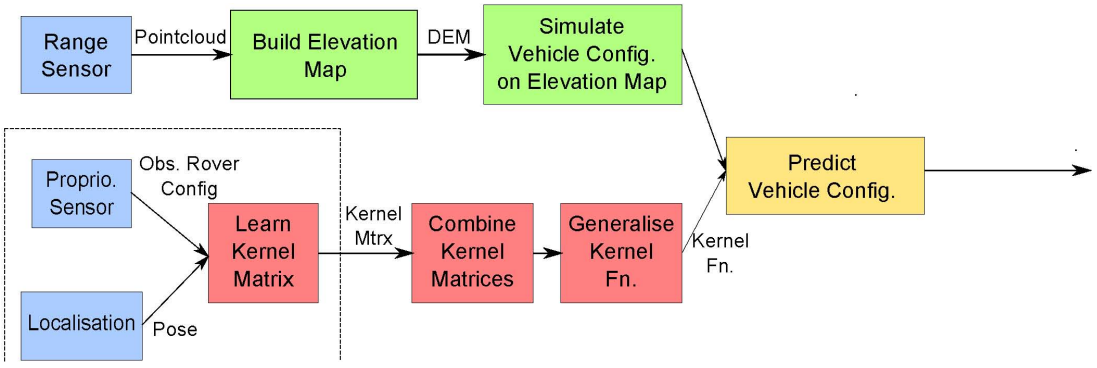


Figure 3.11: System architecture of Kin-GP-VE (from [28])

As the kernel function defines the covariance between each pair of variables in GP, the choice of kernel function will inevitably introduce bias in to the estimation. To alleviate this problem, Ho et al. choose to learn the kernel function based on vehicle experience in training.

The training is done by driving a planetary rover in a Mars analogue terrain called Marsyard as shown in Fig 3.12. The rover makes traversals in a grid pattern to cover all the three experiment areas while recording vehicle configurations.

Point clouds are also obtained via the RGB-D camera mounted on the rover and further transformed into an elevation map. By virtue of exploiting the explicit correlation between vehicle configurations in GP, more accurate traversability estimations can be made over unknown terrains in an incomplete map as the one in Fig 3.13.

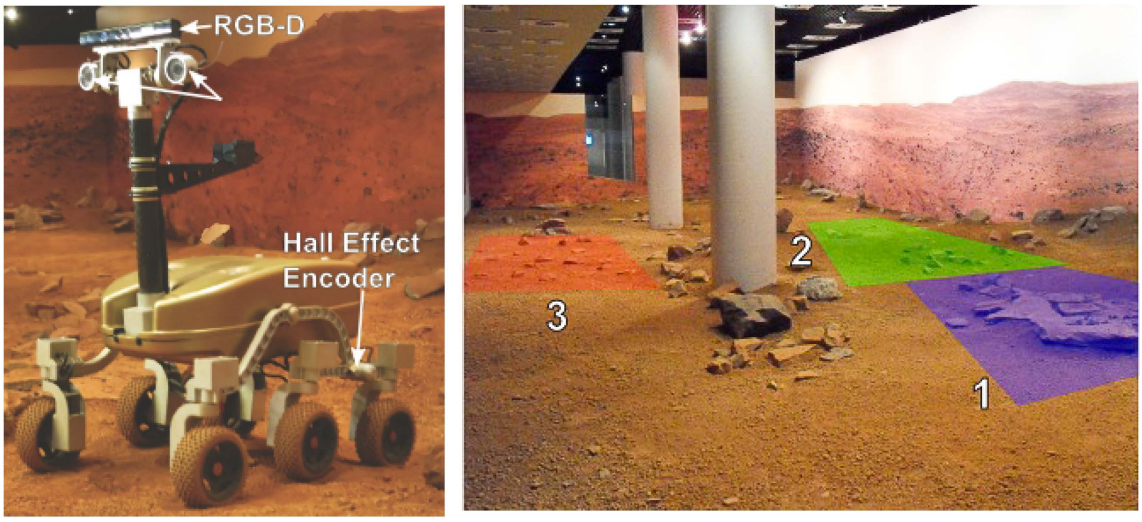
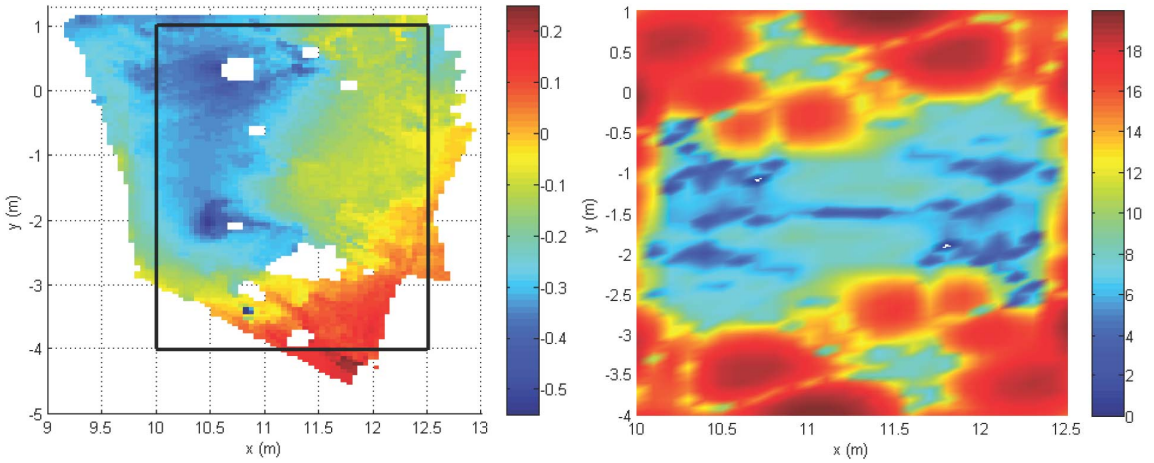


Figure 3.12: Rover & experiment environment (from [28])



(a) elevation map (b) roll estimation

Figure 3.13: Elevation map & roll estimation in area 2. In the left figure darker color represents lower elevation, while in the right figure darker color means a lower roll angle of the robot when it drives on the corresponding area. (from [28])

In this chapter, we reviewed several types of traversability estimation methods and discussed their drawbacks caused by manually set parameters and rules. To improve the accuracy and generalization of the traversability estimation, we propose a set of data-driven methods based on pre-trained CNN models and a simulation framework for collecting training data. Conventional methods introduced in this chapter are also re-implemented and tested on the same dataset as benchmarks.

Before explaining our work, it is necessary to investigate what is the format of the environment map on which the traversability is estimated and how the robot is localized while scanning the environment and collecting driving data.

Chapter 4

Mobile robot localization

To enable the implementation of the traversability estimation methods proposed in this thesis and the experiments to validate them, an autonomous robot would need to employ localization and mapping methods. This is due to the fact the traversability methods label terrain samples depending on whether a robot is able to traverse them or how hard/long it was to go from source to destination while traversing the area. Localization is essential for the labeling process, mapping is crucial for the sample collection.

In this chapter, we will investigate two widely used categories of localization methods in mobile robot navigation — Simultaneous Localization And Mapping (SLAM) and Wireless Localization based on Received signal strength Indicator (WL-RSSI). A wireless localization method proposed in our previous study [54] is introduced in Section 4.2.3. Then, discussions about map building and the map format we used in our experiments are presented in Section 4.3.

4.1 Simultaneous Localization And Mapping (SLAM)

In many application scenarios of mobile robots, it's unlikely to get a prior map of the environment before the navigation task starts. One way to address this problem is localizing the robot and building the map at the same time. This research topic is named as Simultaneous Localization and Mapping (SLAM).

4.1.1 Probabilistic formulations of SLAM

Studies treating SLAM as a probabilistic problem originated in the 1986 IEEE Robotics and Automation Conference, while the acronym 'SLAM' was first presented 9 years later in 1995 [29]. An early study [30] found the correlation between each estimate of landmarks, and later researches then proved the convergence of SLAM [31] [32]. These fundamental works formulated the basic form of the probabilistic model of SLAM and paved the way for further studies.

In general, the task of SLAM is to maintain an estimate for the joint probabilistic distribution of both the vehicle state and landmark locations as shown below:

$$P(\mathbf{x}_k, \mathbf{m} | \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0) \tag{4.1}$$

where $\mathbf{x}_k$ is a state vector representing the location and orientation of the mobile robot vehicle at time instant k (k=0 means the beginning); $\mathbf{m}$ is a set of location vectors of landmarks ($\mathbf{m}_i$ referring the ith landmark); $\mathbf{Z}_{0:k}$ and $\mathbf{U}_{0:k}$ are lists of observations and control inputs respectively ranging from time instant 0 to k.

Two transition models — one for the vehicle motion and the other for the observation — are used alternately to update the joint estimate [33].

The motion model and observation model:

$$P(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{u}_k) \tag{4.2a}$$

$$P(\mathbf{z}_k \mid \mathbf{x}_k, \mathbf{m}) \tag{4.2b}$$

The prediction (time-update) and correction (measurement-update):

$$\begin{gathered} P(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k}, x_0) = \\ \int P(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{u}_k)\ P(\mathbf{x}_{k-1}, \mathbf{m} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k-1}, x_0) d\mathbf{x}_{k-1} \end{gathered} \tag{4.3a}$$

$$P(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, x_0) = \quad \frac{P(\mathbf{z}_k \mid \mathbf{x}_k, \mathbf{m}) P(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k}, \mathbf{x}_0)}{P(\mathbf{z}_k \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k})} \tag{4.3b}$$

where $\mathbf{z}_k$ and $\mathbf{u}_k$ represent the observation and control input at time instant k respectively.

Model 4.2a represents the probability distribution of $\mathbf{x}_k$ inferred by $\mathbf{x}_{k-1}$ and $\mathbf{u}_k$,

then it is used in equation 4.3a to calculate the joint estimate of $\mathbf{x}_k$ and $\mathbf{m}$ based on $\mathbf{Z}_{0:k-1}$, $\mathbf{U}_{0:k}$ and x_0.

On the other hand, model 4.2b represents the probability distribution of $\mathbf{z}_k$ inferred by $\mathbf{x}_k$ and $\mathbf{m}$, then equation 4.3b uses it to calculate the joint estimate from $\mathbf{Z}_{0:k}$, $\mathbf{U}_{0:k}$ and x_0.

In such a recursive process, the joint estimate of robot location and landmarks at time instant k can be updated from their previous version at time instant k-1 with new observation and control inputs. It is also worth noting that estimates of each step are all dependent on the initial estimate of the vehicle state $\mathbf{x}_0$, which is a source of correlated estimate error between landmarks.

Based on this probabilistic frame, two classical solutions dominate the research field of SLAM problems for decades — Extended Kalman Filter based SLAM (EKF-SLAM) and Rao-Blackwellised Filter based SLAM (FastSLAM). While new alternatives with great potential are emerging, trying to improve the computational complexity, data association, and environment representation.

4.1.2 Extended Kalman Filter SLAM (EKF-SLAM)

Kalman Filter [35] is an approach used to get the optimal estimation for a linear system model that cannot be obtained directly but can only be estimated by observations with noise. To apply filtering to non-linear systems, Extended Kalman Filter (EKF) [36] [37] was proposed, which estimates the non-linear system by a linear approximation around the current state at each time step.

EKF-SLAM applies the standard EKF method [38] [39] to calculate the mean and covariance matrix of locations of landmarks in the map and the motion of the robot. In EKF-SLAM, the motion and observation transition model are defined as follows:

$$\begin{aligned} P(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{u}_k) &\Longleftrightarrow \mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k \\ P(\mathbf{z}_k \mid \mathbf{x}_k, \mathbf{m}) &\Longleftrightarrow \mathbf{z}_k = \mathbf{h}(\mathbf{x}_k, \mathbf{m}) + \mathbf{v}_k \end{aligned} \tag{4.4}$$

where $\mathbf{f}$ is the vehicle motion function; $\mathbf{h}$ is the landmark location observation function; $\mathbf{w}_k$ and $\mathbf{v}_k$ are additive, zero mean Gaussian noises, their covariance are $\mathbf{Q}_k$ and $\mathbf{R}_k$ respectively.

Through EKF, the mean and covariance of the SLAM joint estimate could be

calculated as:

$$\begin{bmatrix} \hat{\mathbf{x}}_{k|k} \\ \hat{\mathbf{m}}_k \end{bmatrix} = E\left[\begin{bmatrix} \mathbf{x}_k \\ \mathbf{m}_k \end{bmatrix} \middle| \mathbf{Z}_{0:k} \right] \tag{4.5}$$

$$\mathbf{P}_{k|k} = \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xm} \\ \mathbf{P}_{xm}^\top & \mathbf{P}_{mm} \end{bmatrix}_{k|k} = E\left[\begin{pmatrix} \mathbf{x}_k - \hat{\mathbf{x}_k} \\ \mathbf{m} - \hat{\mathbf{m}}_k \end{pmatrix} \begin{pmatrix} \mathbf{x}_k - \hat{\mathbf{x}_k} \\ \mathbf{m} - \hat{\mathbf{m}}_k \end{pmatrix}^\top \middle| \mathbf{Z}_{0:k} \right] \tag{4.6}$$

here $\hat{\mathbf{x}}_{p|q} = E[\mathbf{x}_p|\mathbf{Z}^q]$ $(p \geq q)$, where $\mathbf{Z}^q$ is the sequence of observations taken up until time q. Assuming the estimate error is denoted as $\tilde{\mathbf{x}}_{p|q} = \hat{\mathbf{x}}_{p|q} - \mathbf{x}_p$, the recursive estimate of covariance can be calculated by $\mathbf{P}_{p|q} = E[\tilde{\mathbf{x}}_{p|q}\tilde{\mathbf{x}}_{p|q}^T \mid \mathbf{Z}^q]$.

The EKF estimation proceeds recursively in two steps:

The first step is the time-update. At each time step k, the vehicle motion estimate will be made based on the vehicle motion function mentioned in equation 4.4 and the estimate of the previous time step $k-1$:

$$\begin{aligned} \hat{\mathbf{x}}_{k|k-1} &= \mathbf{f}(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k) \\ \mathbf{P}_{xx,k|k-1} &= \nabla\mathbf{f}\, \mathbf{P}_{xx,k-1|k-1}\, \nabla\mathbf{f}^\top + \mathbf{Q}_k \end{aligned} \tag{4.7}$$

where $\nabla\mathbf{f}$ means the Jacobian of motion function $\mathbf{f}$ at $\hat{\mathbf{x}}_{k-1|k-1}$ — the robot location estimate of last time step.

Then, in the second step, this motion function based only estimate will be incorporated with landmarks and observations in the observation-update as:

$$\begin{bmatrix} \hat{\mathbf{x}}_{k|k} \\ \hat{\mathbf{m}}_k \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{x}}_{k|k-1} \\ \hat{\mathbf{m}}_{k-1} \end{bmatrix} + \mathbf{W}_k[\mathbf{z}(k) - \mathbf{h}(\hat{\mathbf{x}}_{k|k-1},\ \hat{\mathbf{m}}_{k|k-1})] \tag{4.8}$$

$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} - \mathbf{W}_k\mathbf{S}_k\mathbf{W}_k^\top \tag{4.9}$$

where $\mathbf{S}_k = \nabla\mathbf{h}\mathbf{P}_{k|k-1}\nabla\mathbf{h}^\top + \mathbf{R}_k$, $\mathbf{W}_k = \mathbf{P}_{k|k-1}\nabla\mathbf{h}^\top\mathbf{S}_k^{-1}$, $\nabla\mathbf{h}$ is the Jacobian of $\mathbf{h}$ at $\hat{\mathbf{x}}_{k|k-1}$ and $\hat{\mathbf{m}}_{k-1}$.

As the number of observations made during navigation increases, the error of estimated relative locations between landmarks will monotonically approach to zero. However, for absolute locations, the error will only converge to a lower bound which is inherited from the error of the initial vehicle state estimate $\mathbf{x}_0$.

Fig 4.1 shows how the initial vehicle state estimate error affects all the following

landmark location estimates.

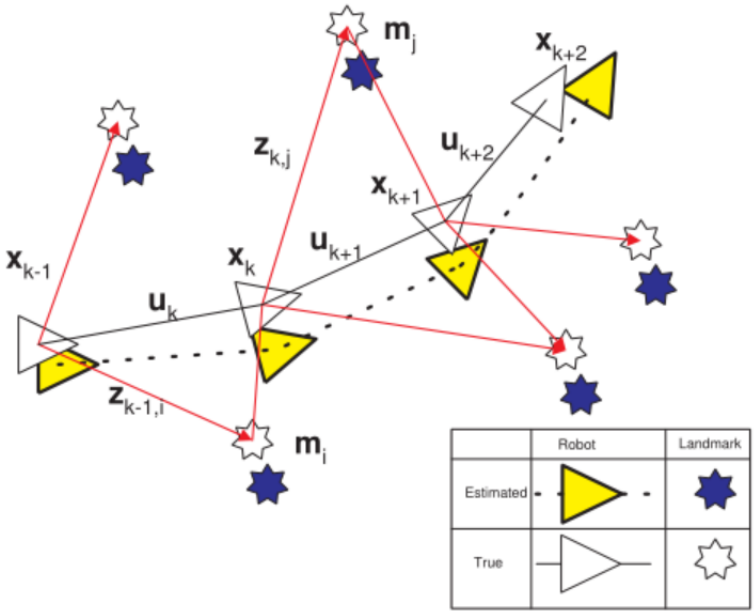


Figure 4.1: In EKF-SLAM, the initial vehicle state estimate error affects all the following landmark location estimates (from [33])

In EKF-SLAM, all the observations are combined and treated as a consecutive trajectory while matches are made between each pair of landmarks even for those not in the current view of the robot. This strategy ensures a monotonic convergence of the estimate for the correlation of landmarks, but also brings some problems: Julier and Uhlmann [40] presented the inconsistency problem of EKF-SLAM caused by implementing a model with linear Gaussian assumptions on non-linear motion and observations. On the other hand, as demonstrated in the joint estimate update formula (equation 4.6), the covariance matrix ($\mathbf{P}_{mm}$) of landmarks which are all correlated with each other needs to be calculated in every time step, which means the time complexity of EKF-SLAM is quadratic with the number of landmarks, making it very computationally costing for applying EKF-SLAM on large maps with thousands or even more landmarks. Many efforts have been made to deal with these problems, and one of the most important alternative methods devised for this purpose is FastSLAM, which will be discussed next.

4.1.3 Rao-Blackwellised based Filter SLAM (FastSLAM)

The second main class of SLAM methods is Rao-Blackwellised based Filter SLAM, which is also known as FastSLAM.

Proposed by Montemerlo *et al.* [41] in 2002, FastSLAM adopts a fundamentally different way to implement probabilistic SLAM. Instead of building the map relying on the relative location between landmarks, it computes the estimate of landmark locations depending on the vehicle position and orientation estimates from which the landmark can be observed. By using Rao-Blackwellisation filter, the time complexity for solving SLAM problems is reduced to the linear level.

The first step of FastSLAM is calculating the vehicle state estimate. A particle filter similar to the Monte Carlo localization (MCL) algorithm [42] is implemented to do this following the equations shown below [41]:

$$\begin{aligned} S_t &= \{s^{t,[m]}\}_m = s_1^{[m]}, s_2^{[m]}, ..., s_t^{[m]}{}_m \\ s_t^{[m]} &\sim p(s_t \mid u_t, s_{t-1}^{[m]}) \\ w_t^{[m]} &= \frac{target\ distribution}{proposal\ distribution} = \frac{p(s^{t,[m]} \mid z^t, u^t)}{p(s^{t,[m]} \mid z^{t-1}, u^{t-1})} \end{aligned} \tag{4.10}$$

where S_t is a particle set including m particles; Particle $s^{t,[m]}$ is the m-th random sample of the vehicle path(state history) until time t; $s_k^{[m]}$ is the vehicle's pose state at time k of this particle; u^t and z^t are the control input and observation at time t respectively; $w_t^{[m]}$ is the importance factor for weighting particles to get the final vehicle pose estimate.

According to the assumption that the vehicle and landmarks state transition process of SLAM is a Markov chain, which means the state at any time step t would only dependent on its direct predecessor at time step $t-1$. The estimate for the k-th landmark's location θ_k at time step t can be computed through Bayes theorem as:

$$\begin{aligned} & p(\theta_k \mid s^t, z^t, u^t) \\ \overset{\text{Bayes}}{\propto} \ & p(z_t \mid \theta_k, s^t, z^{t-1}, u^t)\, p(\theta_k \mid s^t, z^{t-1}, u^t) \\ \overset{\text{Markov}}{=} \ & p(z_t \mid \theta_k, s_t)\, p(\theta_k \mid s^{t-1}, z^{t-1}, u^{t-1}) \end{aligned} \tag{4.11}$$

where θ_k is the location of the k-th landmark; s is the pose of vehicle, z is the

observation and u is the control input; the subscript t means this variable corresponds to the state at time step t only, while the superscript t representing the variable refers to the whole state history until(and including) time step t.

A landmark's estimate would only be updated when a new observation of it is made, otherwise, it would be left unchanged. It can be seen that the landmark location updating process only depends on the sampled vehicle path and already known observations and control inputs. This is the main property of FastSLAM and also makes it computationally efficient — since the landmark locations become independent to each other, the time complexity of FastSLAM is only $O(MK)$ where M is the number of particles and K is the number of landmarks, whereas EKF-SLAM requires time quadratic in K.

However, the variance of the vehicle pose estimate grows with time. A resampling step is necessary to retain appropriate weights for particles. The new weights are computed as follows:

$$\begin{aligned} w_t^{[m]} &\propto \frac{p(s^{t,[m]} \mid z^t, u^t)}{p(s^{t,[m]} \mid z^{t-1}, u^t)} \\ &\overset{\text{Bayes}}{\propto} p(z^t \mid s^{t,[m]}, z^{t-1}, u^t) \\ &\overset{\text{Markov}}{=} \int p(z^t \mid \theta, s^{t,[m]})\, p(\theta \mid s^{t-1,[m],z^{t\ -1},u^{t\ -1}})\, d\theta \\ &\overset{\text{EKF}}{\approx} \int p(z_t \mid \theta^{[m]}, s_t^{[m]})\, p(\theta^{[m]})\, d\theta \end{aligned} \tag{4.12}$$

Although resampling could reinstate uniform weighting for particles, it also causes accuracy loss in the long term. This is because applying RB particle filters needs an assumption that errors in the previous state of the estimated system should "fade away" exponentially, while SLAM systems don't have such a property, leading the historical errors to accumulate with time.

While the EKF and RB based SLAM approaches present a mathematically sound solution to the localization and mapping problem, in practice the accuracy of these approaches degrade sharply when a robot is unable to observe and re-observe landmarks between time steps. For this reason, other localization methods are needed if a robot has to localize itself in environments that are poor in landmarks. One of these alternatives is the use of wireless signal strength based localization methods.

4.2 Wireless Localization based on Received signal strength Indicator (WL-RSSI)

As demonstrated in the previous section, SLAM is a computationally expensive way for localizing mobile robots and it requires expensive special devices such as laser rangefinders and depth cameras. The localization accuracy of SLAM also fluctuates because it's highly dependent on the environment it is applied and the particular choice of SLAM method.

A more affordable and stable alternative way for mobile robot localization is Wireless Localization based on Received Signal Strength Indicator (WL-RSSI), that is localizing a robot by analysing the strength of wireless signals it receives from multiple wireless emitters deployed in the area with known positions (called "anchors").

4.2.1 Conventional WL-RSSI methods

There are two conventional localization ways in WL-RSSI:

1. Estimating position by distances from anchors. The distance d_i from the i-th anchor to the robot can be inferred from the signal strength it receives. Iterating through all anchors we will have the following equation set:

$$\left.\begin{aligned} (x_1 - x)^2 + (y_1 - y)^2 + (z_1 - z)^2 &= d_1^2 \\ &\vdots \\ (x_n - x)^2 + (y_n - y)^2 + (z_n - z)^2 &= d_n^2 \end{aligned}\right\} \tag{4.13}$$

   here (x, y, z) is the coordinate of the robot and (x_i, y_i, z_i) is the coordinate of the i-th anchor. With this equation set, we can estimate the robot position by calculating a (x, y, z) with the minimum mean square error (MSE) like Sugano *et al.* did in [43].

2. Estimating position by signal fingerprints. Before performing the localizing task, sensors (called sniffers) are deployed at different locations in the target area to collect the signal strength of every anchor received in each location (a signal fingerprint), thus a road map can be built with these fingerprints. When localizing a robot, we can compare the current signal fingerprint of the robot to the prerecorded road map and apply an estimation method such as k-NN to

determine the robot position.

However, In these conventional methods RSSI is usually used as a fixed function with distance (as shown in Fig 4.2), but in real world applications RSSI will be affected by path-loss, fading and shadowing effects. The actual distribution of RSSI is highly dependent on the environment where anchors are deployed. As Heurtefeux and Valois showed in [44], RSSI of different platform working on different power level varies a lot in their distribution pattern (Fig 4.3 - 4.5).

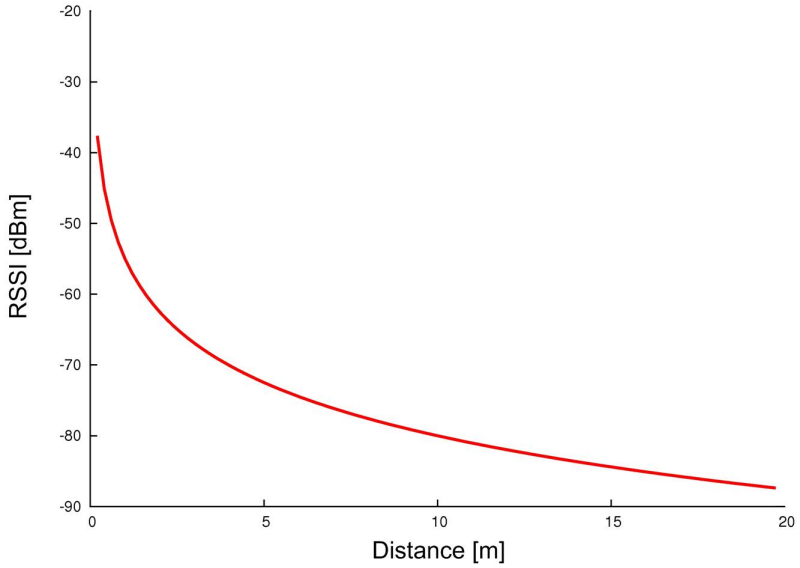


Figure 4.2: Expected relationship between RSSI and distance (K = 25, Pr1 = −55dBm) (from [33])

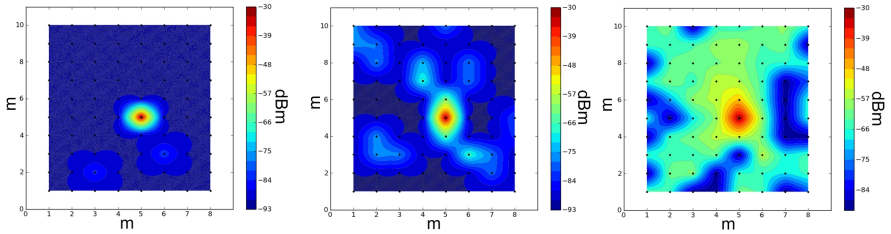


Figure 4.3: RSSI propagation on Strasbourg Platform (−30, −15 and 0 dBm) (from [44])

4.2.2 Improved WL-RSSI methods

Many improvements and new methods have been proposed to solve this problem [45–47]. One efficient way of them is training a neural network with RSSI recorded

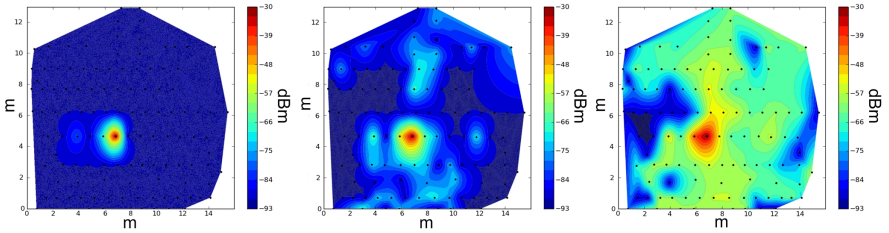


Figure 4.4: RSSI propagation on Grenoble Platform (−30, −15 and 0 dBm) (from [44])

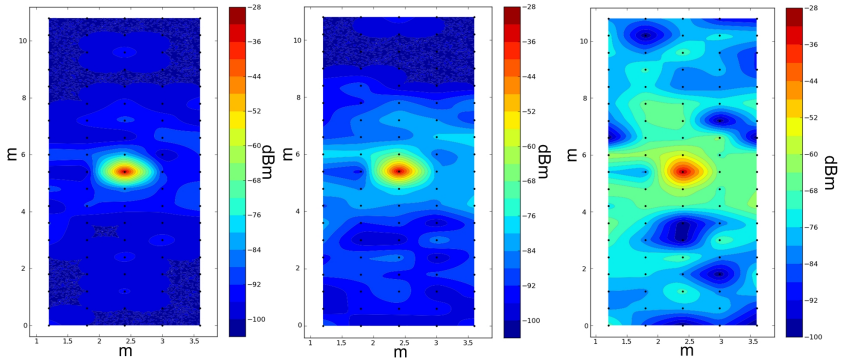


Figure 4.5: RSSI propagation on Lille Platform (−30, −15 and 0 dBm) (from [44])

by those sniffers to approximate to the actual RSSI distribution function. Based on whether the mobile robot localization result is required to be a tag of a subarea or an exact coordinate, a neural network for classification or regression task can be applied.

In [48], Rohra *et al.* studied wireless localization as a classification problem — 7 routers are placed as wireless anchors in 4 rooms in their laboratory, RSSI received by a cellphone at different places in the experimental area are recorded as training data (shown in table 4.1). Then a fuzzy neural network is trained to predict which room the phone is in currently.

Table 4.1: Wireless Indoor Localization data samples

WS1	WS2	...	WS7	Class
-64	-56	...	-81	1
-42	-53	...	-69	2
-48	-54	...	-84	3
-58	-56	...	-84	4
...	...	...	...	...

Signal strength in numerical form, lower numbers correspond to weaker signals.

On the other hand, Mohammadi and Al-Fuqaha [49] set up their experiment in a

mixed area of open and indoor areas. As shown in Fig 4.6, 13 Bluetooth Low Energy (BLE) iBeacons are deployed in a university library and on the square besides it, while an iPhone 6S is used to measure and record the RSSI from these beacons.

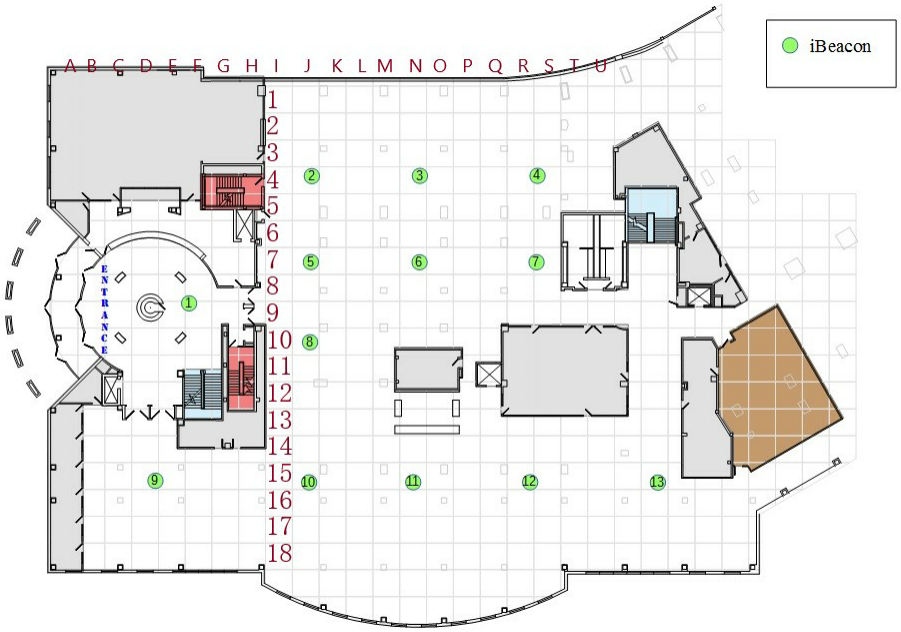


Figure 4.6: iBeacon layout (from [49])

The whole area is divided into 26 columns (marked by A-Z) and 18 rows (1-18) and the row/column index of a grid are used as its coordinate. In the experiment, actual locations (grid indices) and RSSI readings from the 13 beacons at different times and places are recorded in the form shown in table 4.2. A neural network is trained on this dataset to predict the coordinate of the signal receiver based on RSSI as a regression problem.

Table 4.2: BLE RSSI data samples

location	b1	b2	...	b13
O02	-200	-200	...	-200
P01	-200	-200	...	-200
P02	-200	-200	...	-200
K03	-200	-80	...	-200
...	...	...	...	...

4.2.3 Our wireless localization method

Further improvements can be made by adjusting network architecture and training methods. In one of our previous work [54], we applied a hybrid Particle Swarm Optimization (PSO) and Gravitational Search Algorithm (GSA) method to WL-RSSI neural network training and enhanced it with pretraining by Extreme Learning Machine (ELM).

ELM [50] is a type of Feedforward Neural Networks (FNN) and corresponding training method. The most essential advantage of ELM is that its weights don't need to be tuned iteratively. All input weights (from input layer to hidden layer) are randomly generated at the beginning, and then output weights are calculated at once.

The training rule for an ELM with a single hidden layer is:

$$\begin{aligned} H &= \delta(P \times W_I + B) \\ W_O &= H^{-1} \times T \end{aligned} \tag{4.14}$$

where P is the training sample matrix, T is the desired output matrix. W_I and W_O are matrices containing the weights of connections from input layer to hidden layer and hidden layer to output layer. B is a vector of biases for hidden layer, δ is the activation function. H^{-1} is the pseudo inverse (Moore–Penrose inverse) of matrix H.

PSO is an evolutionary optimization method proposed by Kennedy and Eberhart [51] in 1995. A set of particles which represent candidate solutions to the target optimization problem are applied, and a fitness function is defined to measure the quality of a particle.

All particles move around in the search space to find the best solution, their coordinates are updated iteratively — in each iteration, every particle will gain a new velocity determined by three factors: 1. its velocity of last step(velocity of the first iteration is randomly assigned); 2. the vector from itself pointing to the location of the particle with highest fitness value(global best) of this iteration; 3. the vector from its current location pointing to the location in its whole trajectory with highest fitness value(particle best).

The resulting new velocity is a linear combination of these three factors, and the

the location of each particle will be updated accordingly as shown below:

$$\begin{aligned} X_i(t+1) &= X_i(t) + V_i(t+1) \\ V_i(t+1) &= w \times V_i(t) + \\ &\quad c_1 \times rand \times (pbest_i - X_i(t)) + \\ &\quad c_2 \times rand \times (gbest - X_i(t)) \end{aligned} \tag{4.15}$$

where $X_i(t)$ is the coordinate of particle i at iteration t, $V_i(t)$ is its velocity; w, c_1 c_2 are weights for three factors respectively, $rand$ is a random number between 0 and 1; $pbest_i$ is the coordinate of particle i with the highest fitness value in its trajectory, while $gbest$ is the global best particle coordinate so far.

Gravitational Search Algorithm (GSA) was proposed by Rashedi *et al.* [52] as a new heuristic optimization algorithm in 2009. It is designed for finding an optimization solution in the search spaces by emulating physical gravitational rules. GSA applies a collection of particles (candidate solutions) with masses proportional to their fitness value. In each update iteration, all particles attract each other by the gravity forces between them. The greater the mass and the closer the distance, the greater the gravity. Therefore, particles with greatest masses because of being close to the a local or global best optimization solution will attract the other particles to cluster around them.

The gravitational force can be modelled as following equations:

$$\begin{aligned} F_{ij}(t) &= G(t) \times \frac{M_i(t) \times M_j(t)}{R_{ij}(t) + \epsilon} \times (X_j(t) - X_i(t)) \\ F_i^d(t) &= \sum_{j=1, j \neq i}^{N} rand_j \times F_{ij}^d(t) \end{aligned} \tag{4.16}$$

where $M_i(t)$, $M_j(t)$, $X_i(t)$, $X_j(t)$ are masses and coordinate of particle i, j at time step t respectively; $F_{ij}(t)$, $R_{ij}(t)$ are the attracting force and Euclidean distance between them; $rand_j$ is a random number in $[0, 1]$, $F_i^d(t)$ is the total force that acts on particle i in dimension d, it's a randomly weighted sum of attracting forces from all the other particles .

Hence, the acceleration of particle i at time step t is calculated by the law of

motion:

$$a_i(t) = \frac{F_i(t)}{M_i(t)} \tag{4.17}$$

Finally, we get the velocity and coordinate update formulas for particle i at time step t:

$$\begin{aligned} V_i(t) &= rand_i \times V_i(t) + a_i(t) \\ X_i(t+1) &= X_i(t) + v_i(t+1) \end{aligned} \tag{4.18}$$

here $rand_i$ is also a random number in interval $[0, 1]$.

Hybrid PSO-GSA was presented by Mirjalili and Hashim [53] in 2012. As a combination of PSO and GSA, PSO-GSA draws on advantages from both of them — the better exploration ability for particles from PSO and the ability of clustering particles to areas with higher fitness values of GSA to search these areas more thoroughly which helps to improve the possibility of finding the global optimal solution.

The particle velocity and coordinate update formulas of PSO-GSA is shown below:

$$X_i(t+1) = X_i(t) + V_i(t+1) \tag{4.19a}$$

$$\begin{aligned} V_i(t+1) = \quad & w \times V_i(t) + \\ & c\prime_1 \times rand \times ac_i(t) + \\ & c\prime_2 \times rand \times (gbest - X_i(t)) \end{aligned} \tag{4.19b}$$

where w is an inertia coefficient, which will decrease as the iteration progresses; $c\prime_1$ and $c\prime_2$ are two weighting factors, and $rand$ is a random number in $[0, 1]$.

Equation (4.19b) is a combination of the velocity update equations of PSO and GSA. It consists of three parts:

1. $w \times V_i(t)$, which is common to PSO and GSA, represents the affect of inertia on the particle. w here is the inertia factor, it will decrease as the iteration progresses.

2. $c\prime_1 \times rand \times ac_i(t)$, which comes from GSA. Here ac_i is the acceleration of particle i caused by the gravitational force from other particles;

3. $c\prime_2 \times rand \times (gbest - X_i(t))$, which comes from PSO. $gbest$ is the coordinate of the global best particle with the highest fitness value so far.

In our work [54], we apply PSO-GSA on optimizing the weights of a feedforward neural network for wireless localization tasks from [48] and [49] to get a more accurate result in both classification and regression scenarios.

Furthermore, we pretrain the network multiple times with ELM and use the resulting network weights to replace part of the randomly initialized particles in PSO-GSA as they are more likely to be located in areas with high fitness value in the searching space. Experimental results prove that our method improves both the accuracy and the convergence speed of the network compared with the ordinary Back-Propagation(BP) and PSO-GSA. The performance comparison is shown in Fig 4.7 and 4.8.

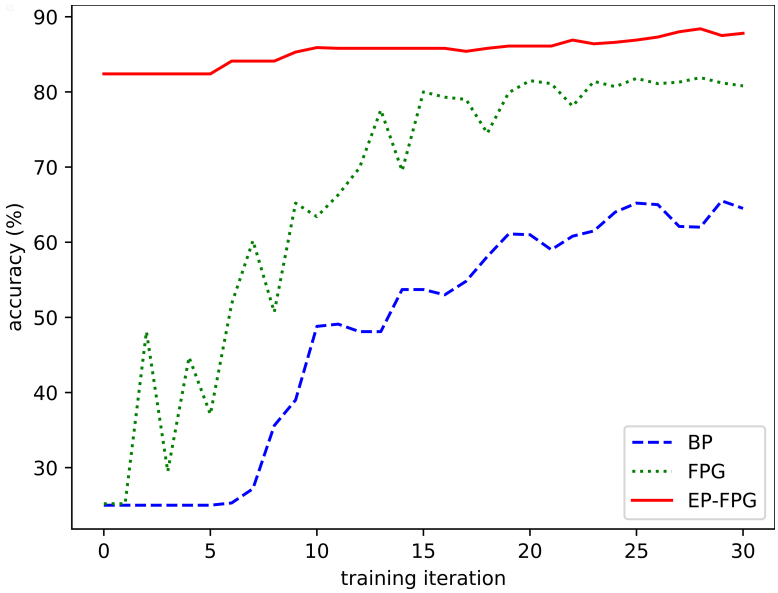


Figure 4.7: Localization classification accuracy of BP, FPG and EP-FPG on Wireless Indoor Localization dataset.

While the wireless-based localization method provides a solution for localizing the robot in the environment, it does not generate a map depicting the terrain that can be used for traversability estimation. The process of building a map and the map format employed in our approach is presented in the next section.

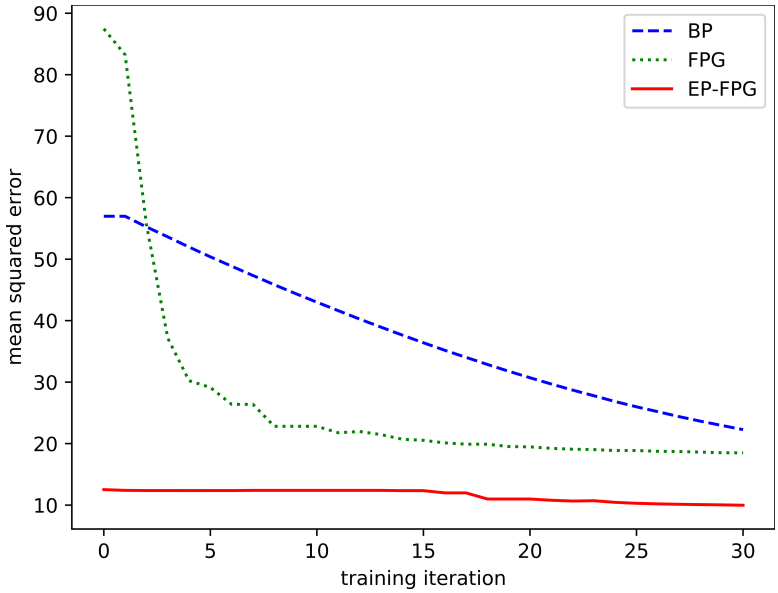


Figure 4.8: Mean squared error of the localization result of BP, FPG and EP-FPG on the Bluetooth Low energy (BLE) dataset.

4.3 Map building

Traversability estimation are directly made on map information of the environment, which means the map format and how the map is built will determine the way traversability estimation is performed.

There are a variety of map formats and map building methods with different advantages have been proposed for mobile robot navigation [55–57]. In this dissertation we focus on octree — a memory and computation efficient frame for representing a terrain map, and introduce how to construct an octree based on point clouds and finally extract an elevation map from it.

The map building process can be divided into 3 steps: point cloud transformation, octree construction and elevation map extraction.

4.3.1 Point cloud transformation

For a mobile robot equipped with a depth camera, inputting point clouds are usually represented in the coordinate frame of the camera rather than the chassis of the robot or the environment. To build a map on a fixed global frame with these point clouds, a rigid transformation must be done based on localizing and kinematics information

of the robot as shown below:

$$\vec{p}_{new} = \mathbf{R} \times \left(\vec{p}_{ori} \; - \; \vec{t} \right) \tag{4.20}$$

here $\vec{p}_{ori}$, $\vec{p}_{new}$ are the original and transformed location vector of a point in the point cloud; $\vec{t}$ is the translation vector while $\mathbf{R}$ is the rotation matrix from the original frame to the new one.

When the transformation to the global frame is not provided directly, we need to perform such transformations from the original camera frame to its parent frames recursively until reaching the main frame of robot. Then a transformation to the global frame can be done based on the location information of the robot.

4.3.2 Octree construction

Octree is a memory efficient data structure for storing 3D occupancy grid map — it merges small grids within a same vicinity into a larger grid node to save storage space, and also delays the initialization of map volumes until measurements need to be integrated thus reducing the computational complexity for building it too.

As shown in Fig 4.9, an octree partitions a 3D space by recursively dividing it into 8 octants. Starting from an empty octree with a root node only, every time a point

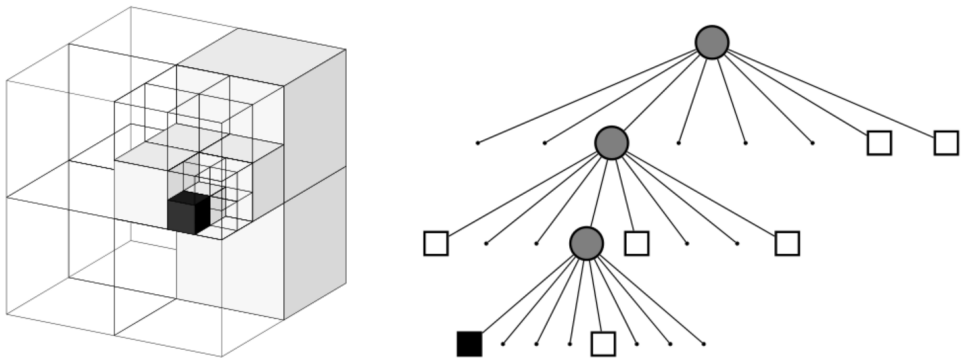


Figure 4.9: Example of an octree storing free (shaded white) and occupied (black) cells. The volumetric model is shown on the left and the corresponding tree representation on the right. (from [57]).

cloud is captured, corresponding cells for points will be inserted into it. The octree grows as existing nodes including both occupied and free sub-cells being divided into 8 sub-cells recursively. In some frameworks of octree such as OctoMap [57], the state

of cells can be represented in the form of probability, and the initial state are assumed as unknown rather than free.

4.3.3 Elevation map extraction

Assuming the environment is in single-level structure, an elevation map can be extracted from the octree by traversing all the nodes of the tree as shown in Algorithm 2:

Algorithm 2: Height map extraction

```
for i = 1 : N_row do
    for j = 1 : N_col do
        H_u[i][j] = -inf
        H_l[i][j] =  inf
for n in {OctreeNodes} do
    x, y, z = n.x, n.y, n.z
    size = n.size
    col_beg = (x - size/2)/res
    col_end = (x + size/2)/res
    row_beg = (y - size/2)/res
    row_end = (y + size/2)/res
    for i = row_beg : row_end do
        for j = col_beg : col_end do
            if n is an occupied node then
                H_u[i][j] = max((z + size/2)/res, H_u[i][j])
            else
                H_l[i][j] = min((z - size/2)/res, H_l[i][j])
for i = 1 : N_row do
    for j = 1 : N_col do
        H[i][j] = min(H_l[i][j], H_u[i][j])
```

Here N_{row}, N_{col} are the number of rows and columns of the resulting elevation map. H_l, H_u and H are 2D arrays for recording the lower bound, upper bound and final result of the height of each grid in the elevation map. n is the tree node in current iteration of the traverse, $n.x$, $n.y$, $n.z$ and $n.size$ are the x, y, z coordinate of the center of the node and its side length. res is the resolution of the elevation map. row, col with different subscripts are the row and column number for updating according elevation map grid.

In some implementations of octree there may be space overlapped by both occupied and free nodes, so it is necessary to compute the lower bound and upper bound of the grid height respectively and get the final result depending on them.

With mobile robot localization methods and a full elevation map of the experiment environment, driving samples for terrain traversability estimation training — as pairs of robot driving odometry (recorded through the localization method) and corresponding submap (extracted from the full map) — can be collected. More details about the process of data collection will be demonstrated in the following chapter.

Chapter 5

Experiment environment & methodology

In this chapter we will give a comprehensive introduction of our experiment environment and process.

The experiment process can be divided into 4 main steps:

1. Preparing the simulation environment. This thesis uses an adapted version of the sophisticated simulation case provided in the NASA Space Robotics Challenge Phase 2 (NASA-SRCP2).

2. Building a full elevation map. Exploring the simulation environment by driving the robot manually and scanning the ground by a depth camera mounted on it, then merge collected point clouds, convert the result to an octree and then a full elevation map in order.

3. Collecting training samples consisting of robot odometries by driving the robot following designated paths in different areas of the environment. Then cut corresponding map strips from the full elevation map as a training sample together with the binary and time cost traversability label.

4. Training CNN models on the collected dataset and compare their estimation performance with benchmark methods.

Fig 5.1 shows the whole process of the experiment.

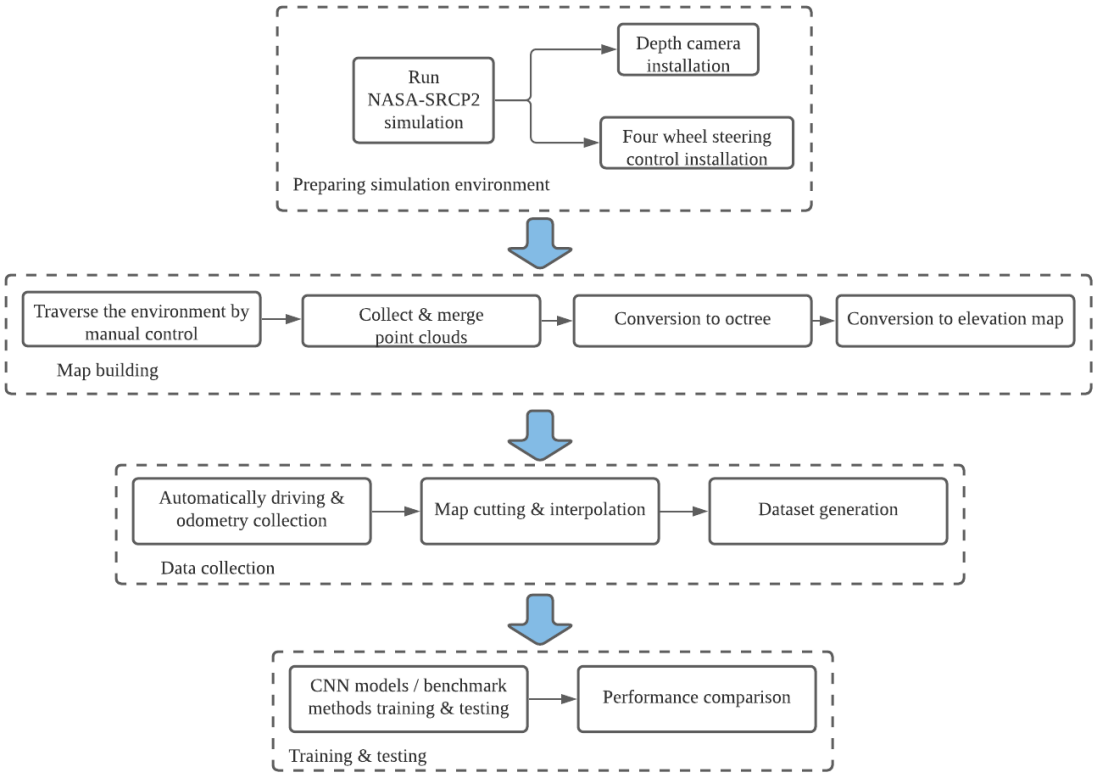


Figure 5.1: Diagram for the whole experiment process

5.1 Simulation Environment

We choose to use a sophisticated simulation environment from NASA-SRCP2 which is built for testing robot driving and exploring in an open planetary environment.

NASA-SRC is a competition held for encouraging and collecting autonomous control solutions for robots planned to be used to assist humans in extraterrestrial planet exploration and colonization. For the phase 2 of the competition, the goal is building an autonomous navigating system for mobile robots to explore and find useful resources like frozen oxygen and water. The first simulation environment we use comes from the task 1 of its qualification round.

This simulation is built based on Robot Operating System (ROS) and Gazebo. ROS is an open-source, meta-operating system for robots [61]. It works in a similar way with an standard computer operating system — its functionalities includes hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, remote procedure call (RPC) services, and package management. Besides, a large number of user developed ROS packages

for various functions are also available on the official site for downloading and using.

In our experimental setup, ROS is mainly used for controlling the robot. Among all of its functions, message handling (for communication between modules), RPC services (for modules which need request/reply interactions) and hardware driving (for implementing robot control) are most frequently used. Some user developed packages are also used for different purpose such as expanding service functions (actionlib), implementing ackerman wheel control (four_wheel_steering_controller), robot twist teleoperation via keyboard (teleop_twist_keyboard) and so on. We will explain their functions and configurations in following corresponding sections where they are used.

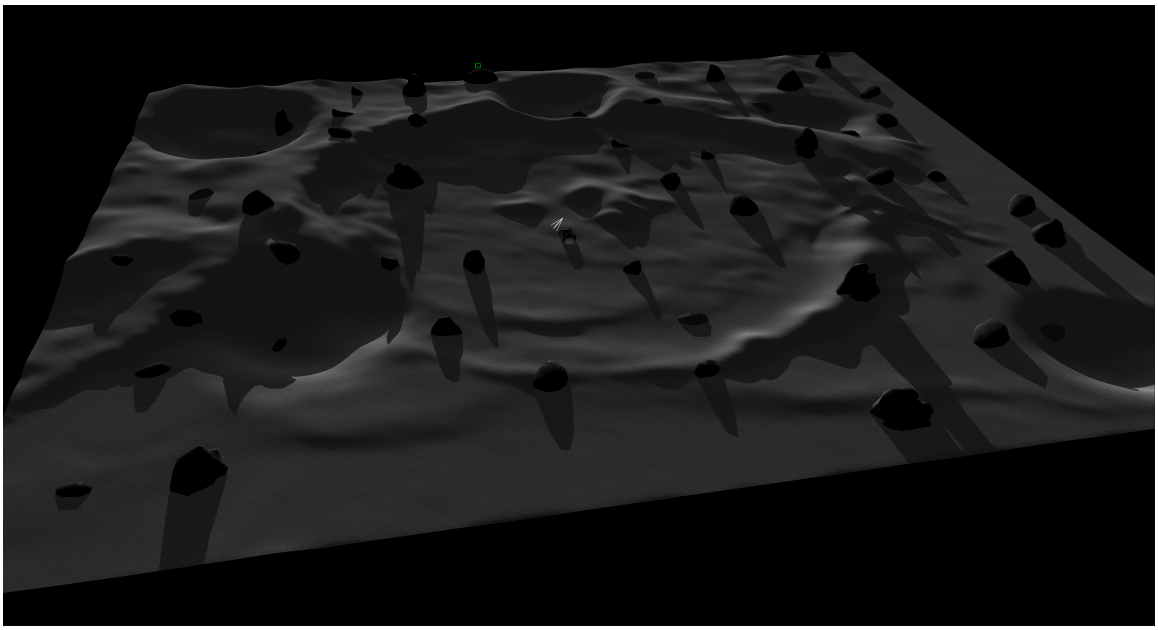
On the other hand, Gazebo is an open-source 3D robotics simulator which is mainly used for building the planetary surface environment, processing physical interactions between the robot and ground/obstacles, and tracking position/orientation/-movement properties of all objects in our experiment. The function of getting/setting object properties plays an important role in the odometry data collection step. Details about its usage will be introduced section 5.4.

Two planetary environments are tested in the experiment:

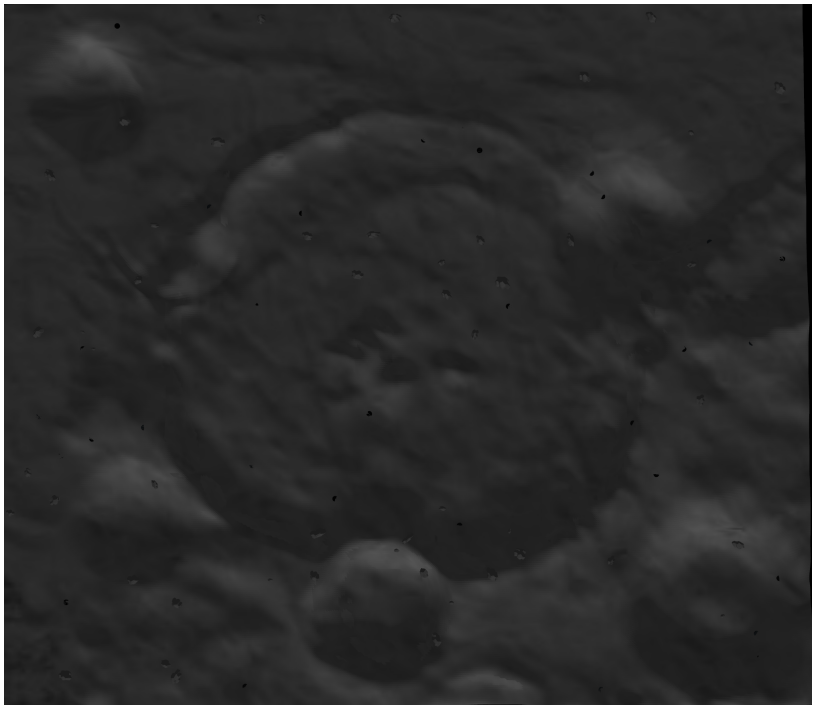
The first environment comes from NASA-SRCP2 — as shown in Fig 5.3, it is designated to replicate lunar geography and conditions, without atmosphere and with the correct Lunar gravity ($1.62m/s^2$). This area is 126 meters long from north to south and 145.5 meters wide from west to east, containing a variety of hills, craters, rocks and other common Lunar environment features. Four invisible "bounding boxes" are placed along the boundaries which are high enough to prevent the robot driving out.

It is worth noting that, in contrast with the original NASA-SRC setting that all the rocks are randomly generated every time the simulation is started, we use a fixed random seed (an integer 1 in particular) to make sure the number, shape, size and distribution of rocks are always the same. All the other objects in the original environment except aforementioned lunar surface, rocks and bounding boxes are all removed in our experiment.

The "world" frame is set at the center of this area with its x-axis pointing to the east, y-axis pointing to the north and z-axis pointing to the up, consistent with conventions in ROS REP-103 [62]. It is also used as the "map" frame in following map building and odometry recording steps. Base on the "world" frame, the altitude of the lowest place in this map is about -3.2 meters while the highest place (on top of a rock, bounding box area is not included) is about 6.0 meters high.



(a) The first experiment environment



(a) Simulated lunar terrain and rocks

Figure 5.3: An overview of the first simulation environment (145.5m × 126.0m)

Since it is infeasible to simulate lunar soil as dust particles in Gazebo, NASA-SRC team choose to construct the ground as a smooth rigid surface, which enables

the robot to climb up steep slopes in combination with the low gravity.

All important parameters of this environment related to our experiment are listed in table 5.1. More details can be found in the appendix and simulation files if needed.

Table 5.1: Simulation parameters of environment 1

Parameter	Value
Length	$126.0m$
Width	$145.5m$
Highest altitude	$6.0m$
Lowest altitude	-$3.2m$
Gravity	1.62 m/s^2
Friction factor mu	1.0
Friction factor mu2	1.0
World frame orientation	x-east, y-north, z-up

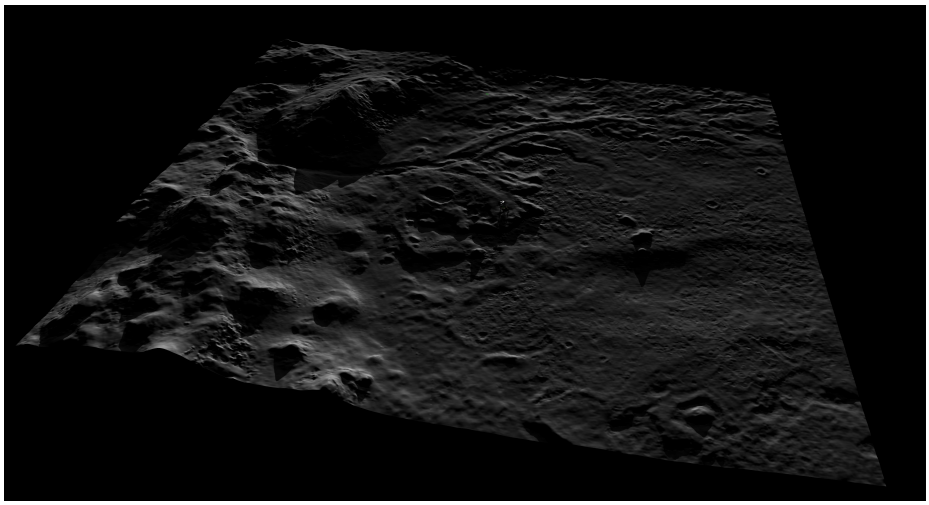
The second environment comes from [63]. As shown in Fig 5.4, it depicts the Delta outcrop of the Jezero crater on mars. The size of the terrain is rescaled to fit the size of the robot, and no extra rocks are generated here.

Parameters of this environment are listed in table 5.2, all the other settings not mentioned here are the same as in the first experiment environment.

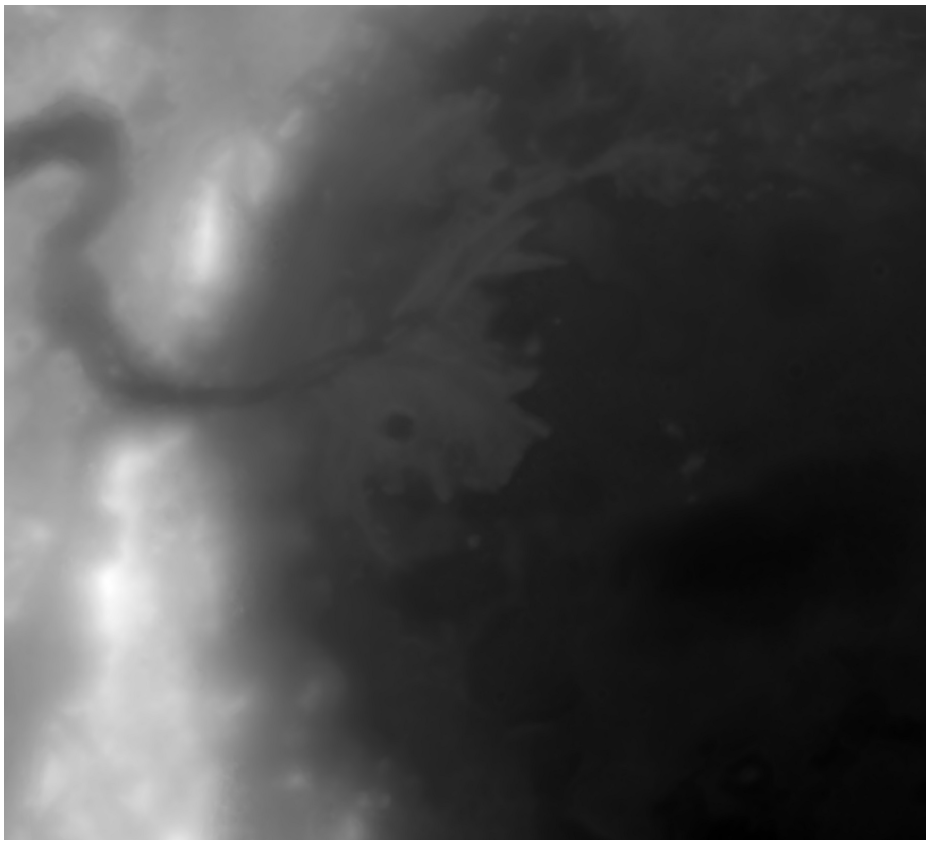
Table 5.2: Simulation parameters of environment 2

Parameter	Value
Length	$208.2m$
Width	$188.6m$
Highest altitude	$26.0m$
Lowest altitude	$5.6m$
Gravity	1.62 m/s^2

As shown in Fig 5.5, the robot used to drive in this environment is a four wheeled rover. It is composed of a main chassis, four legs, four wheels and some sensors mounted on the chassis. The small grey cuboid hovering over the chassis is a depth camera (Intel RealSense R200) looking forward and 45° downward. It is the only thing we added to the original NASA-SRC scout model. Note that the square cone expanding from it is just a visualization frame for its field of view (FOV), not a physical entity.



(a) The first experiment environment

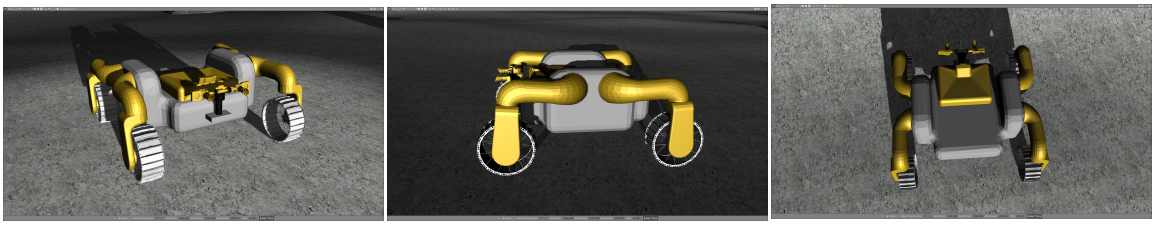


(b) Mars terrain elevation map in grayscale

Figure 5.4: An overview of the second simulation environment (208.2m × 188.6m)

The robot can move or rotate in the space with 6 Degrees of Freedom (6 DoF), while controlling is done by applying torques on the wheels, steering arms and the

brake.



Front Side Top

Figure 5.5: Robot structure

Each wheel can rotate and/or steer independently with a given velocity/steering angle command message. An effort controller for each wheel and its steering joint will calculate the actual effort should be applied on the joint with PID control. Besides the four wheel joints and four steering joints, the joints linking each leg and the chassis are also rotatable, although uncontrollable. Appropriate torques are applied to these joints to keep the legs parallel to the ground while still allow some elasticity to the legs to help the robot keep stable when driving over small rocks and bumps on the ground.

Important robot parameters are listed in the following table 5.3. More related details can be found in the official documents.

Table 5.3: Robot parameters

Parameter	Value
Total Length	2.2860m
Total Width	2.2098m
Total Height (depth camera not included)	0.9490m
Wheel Radius	0.275m
Wheel Separation Width	1.87325m
Wheel Separation Length	1.5748m

5.2 Four wheel steering manual control

The scout robot needs to be manually controlled to travel around the environment and collect point clouds from the attached depth camera. We use a ROS package "teleop_twist_keyboard" to publish a command velocity (cmd_vel) topic with message

type as geometry_msgs/Twist (consisting of a linear velocity vector (x, y, z) and an angular velocity vector $(roll, pitch, yaw)$), where only one pair of linear velocity and steering angle is included. The values of velocity and angle of this message are determined by the last pressed plus with current linear velocity and steering angle. Keys 'i' and ',' are mapped to commands ordering the robot to move forward and backward, while keys 'u', 'o', 'm', '.' are mapped to commands ordering the robot to move forward/backward and turning left or right at the same time. On the other hand, keys 'q', 'z' can increase/decrease the current velocity and keys 'e', 'c' can increase/decrease current steering angle, and all the other keys will order the robot to stop — when such a key are pressed, both linear velocity and steering angle in the cmd_vel will be set to zero, but the current velocity and angle being stored in this keyboard control node won't be affected.

After the cmd_vel message is published, the linear velocity and steering angle for each wheel will be calculated and published separately in 8 different topics with Float64 message type. The conversion of cmd_vel follows the conversion rules given in ROS package "four_wheel_steering_controller", as shown in Algorithm 3.

Here *twist* is a message from the cmd_vel topic. *vel* and *ang* variables with a suffix are resulting velocity and steering angle for each wheel (f: front, b: back, l:left, r:right). $wheel\_radius$(0.275m in our case) is the radius of wheel; $wheel\_steering\_y\_offset$(0m) is the distance between a front wheel and its steering joint; $steering\_track$(1.87325m) is equal to the left/right wheel separation minus 2 times $wheel\_steering\_y\_offset$; $wheel\_base$(1.5748m) is the front/back wheel separation.

In summary, what this conversion algorithm does is trying to move the left wheels and right wheels along a cycle respectively, and these two cycles share a same center, as shown in Fig 5.6.

Each wheel will also be assigned with an appropriate velocity based on the length of the arc it is moving along. This design can help the wheeled robot steer in a more flexible way and also reduce friction between wheels and the ground.

5.3 Building full height map

We build the map of the environment by collecting point cloud data from a simulated Intel RealSense R200 depth camera. It has a field of view (FOV) covering a pyramid-shaped space with a range of 5 meters, and generates a point cloud (in ROS

Algorithm 3: Converting cmd_vel

```
1  vel_fl = vel_fr = vel_bl = vel_br = 0
2  ang_fl = ang_fr = ang_bl = ang_br = 0
3  lin_x = twist.linear.x
4  yaw = twist.angular.z
   /* Compute wheels velocities:                                    */
5  if abs(lin_x) > 0.001 then
6      sign = copysign(1.0, lin_x)
7      vel_steering_offset = (yaw × wheel_steering_y_offset)/wheel_radius
       /* Compute wheels velocities:                                */
8      vel_fl = sign * hypot(
9      (lin_x − yaw * steering_track/2), (wheel_base * yaw/2.0)
        )/wheel_radius − vel_steering_offset
10     vel_fr = sign * hypot(
        (lin_x + yaw * steering_track/2), (wheel_base * yaw/2.0)
        )/wheel_radius + vel_steering_offset
11     vel_bl = vel_fl
12     vel_br = vel_fr
       /* Compute steering angles:                                  */
13     if abs(2.0 * lin_x) > abs(yaw * steering_track) then
14         ang_fl = atan(yaw * wheel_base/(2.0 * lin_x − yaw * steering_track))
15         ang_fr = atan(yaw * wheel_base/(2.0 * lin_x + yaw * steering_track))
16     else
17         abs(lin_x) > 0.001
18     ang_fl = copysign(pi/2, yaw)
19     ang_fr = copysign(pi/2, yaw)
20     ang_bl = −ang_fl
21     ang_br = −ang_fr
```

PointCloud2 message type) with 800 × 640 resolution at 60 Hz, which depicts the contours of the ground and objects on it as shown in Fig 5.7.

Specifically, a depth image is constructed by fusing images from a pair of horizontally separated IR cameras. And then the point cloud is synthesized by combining images taken by the RGB camera at the center of R200 and the depth images. In our experiment, this process is done by inputting R200 color and depth image messages to the point_cloud_xyz node of a ROS package called depth_image_proc.

A map frame is automatically assigned to each point cloud obtained in this way because we are publishing an odometry message topic of the robot with respect to

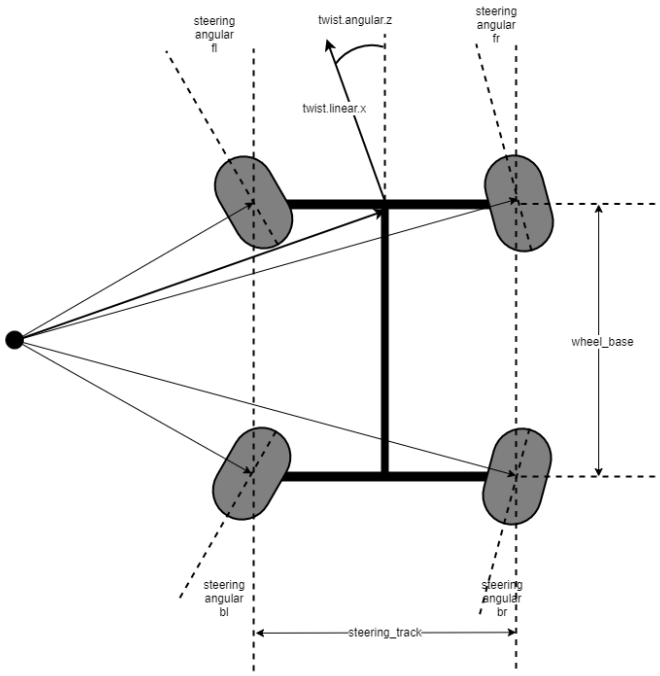


Figure 5.6: Four wheel steering schematic diagram, wheels on the same side will move along a same cycle.

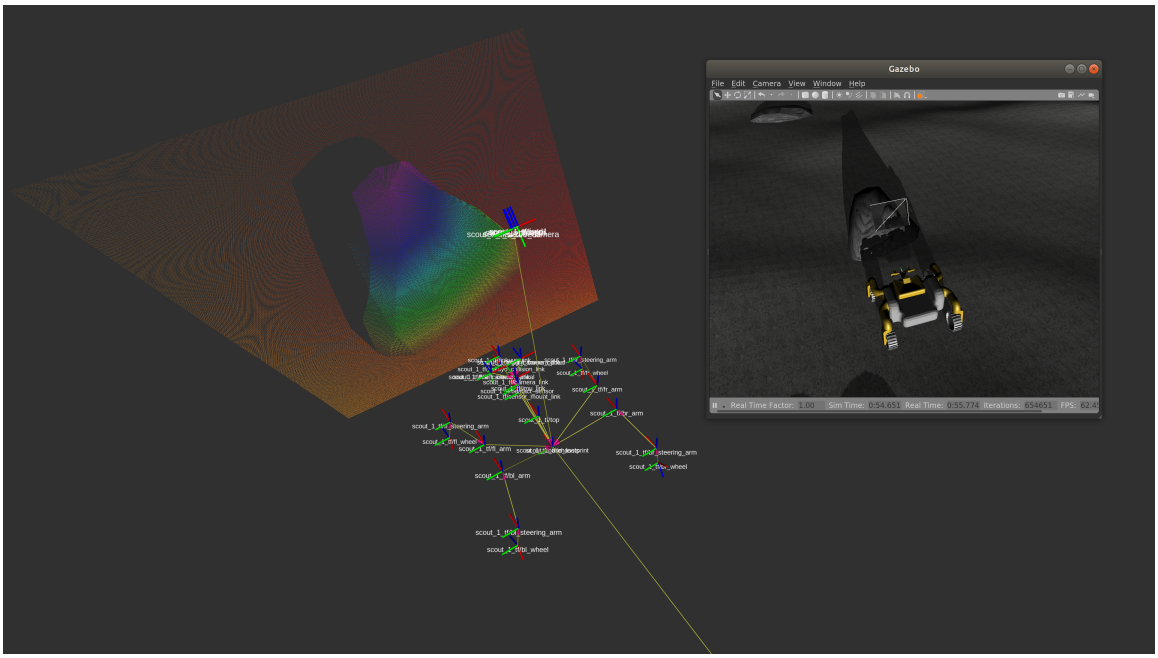


Figure 5.7: Point Cloud got by R200 with obstacles in front of the robot

the world frame of the environment. This map frame will be used to merge these

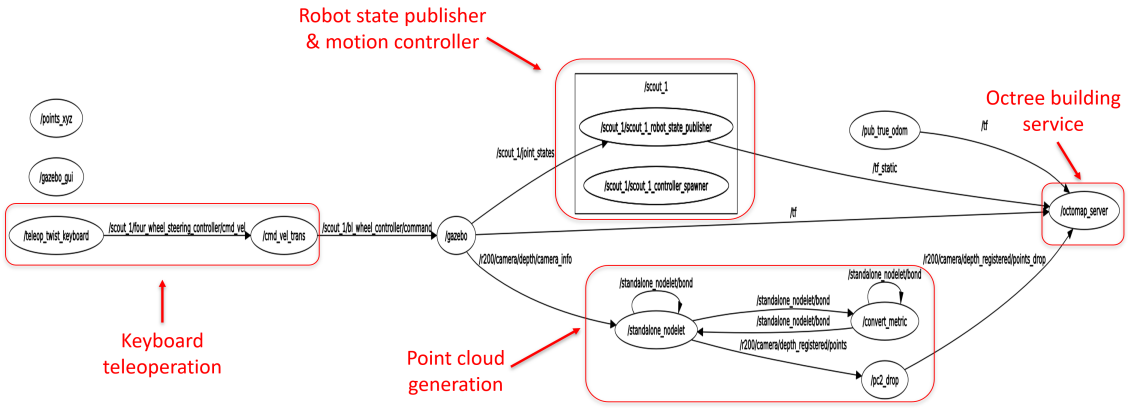


Figure 5.8: ROS node graph for map building (the node "depth_image_proc" is wrapped in the "standalone_nodelet" node)

point clouds into the octree map in the next step. Since in our experiment we are focusing on traversability estimation rather than robot localization, we choose to get the exact position and orientation of the robot directly from the simulation software Gazebo to simplify the whole process and eliminate the influence of localizing errors incurred by particular localizing methods on the final results. We have proved that wireless received signal strength indicator (WL-RSSI) based localization methods can achieve a high accuracy in real-world applications in our previous work [54], and the accuracy can be further improved by adding more wireless emitters as localization "anchors" if needed. More discussions about it can be found in the previous chapter.

After filtered to a lower frequency (node "pc2_drop" in Fig 5.8), this point cloud message is then remapped to octomap_server_node, which is the map building node of another ROS package called octomap_server based on the octomap library, to build an octree map. Octree is an efficient data structure for storing occupancy grids in 3D space. In an octree the root node represents the whole space, and each internal node has exactly eight children, which means it subdivides the space it represents into eight octants while each leaf node has a property representing if it is occupied or free.

Outlier and grid filters are integrated in octomap_server, converting input point clouds into an octree, and then it is incrementally merged into the full octree map. Here we choose to set the grid side length of the octomap as 3cm which is a little shorter than the grid side length of 5cm we plan to apply for the final grid height

map. This is because we want to assure that a grid in the sub height map of a sample we cut from the full height map with any orientation will include at least one center point of a grid in the octomap. Although we use a 2D Clough-Tocher interpolator to generate the submap for a sample where the this property is not required, it may still be necessary if other interpolation methods are applied, therefore we choose to keep this property. Fig 5.9 shows an octomap example (from experiment environment 1).

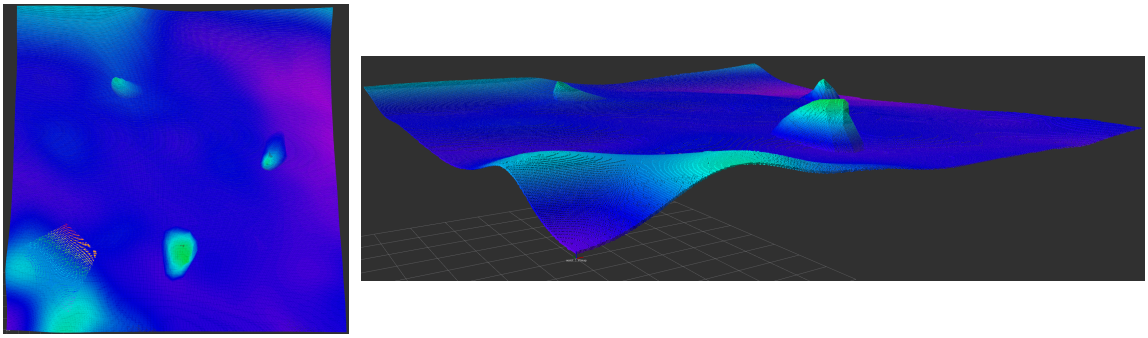


Figure 5.9: An octree map of a sub-area of the first experiment environment

We set the octomap grid of row 0 and column 0 at the place where its lower left vertex is placed on the world frame origin, and make sure the whole environment area will be covered by the octomap. As the first environment has a size of $145.5m \times 126m$, this means there are at least $4230(rows) \times 5000(columns) \approx 21$ million unit grids in the full octomap even without considering there may be more than one grid in a vertical column in the 3D space. Such a large octree exceeds the processing power of our computer (CPU: AMD Ryzen 7 3700X 8-Core, GPU: Nvidia GeForce RTX 2070 super, memory: 16GB), so we divide the full area into 20 sub areas in the x-y 2D plane.

The division of the area is shown in table 5.4. Given grid row index increases from bottom (south) to top (north) of the map, and column index increases from left (west) to right (east), the first line of each table cell is the row/column coordinate of the origin grid (at the lower left corner) of each submap, while the second line is the number of rows and columns that submap includes.

After these submaps are built, they are converted from octomap type to grid height map type. The process of conversion is:

1. Iterating through all leaf nodes of the octree of occupied grids and free grids in order.

Table 5.4: Submap division

submap row/col index	col 1	col 2	col 3	col 4	col 5
row 1	(1000, -2500) (1115, 1000)	(1000, -1500) (1115, 1000)	(1000, -500) (1115, 1000)	(1000, 500) (1115, 1000)	(1000, 1500) (1115, 1000)
row 2	(0, -2500) (1000, 1000)	(0, -1500) (1000, 1000)	(0, -500) (1000, 1000)	(0, 500) (1000, 1000)	(0, 1500) (1000, 1000)
row 3	(-1000, -2500) (1000, 1000)	(-1000, -1500) (1000, 1000)	(-1000, -500) (1000, 1000)	(-1000, 500) (1000, 1000)	(-1000, 1500) (1000, 1000)
row 4	(-2115, -2500) (1115, 1000)	(-2115, -1500) (1115, 1000)	(-2115, -500) (1115, 1000)	(-2115, 500) (1115, 1000)	(-2115, 1500) (1115, 1000)

2. For occupied grids, calculate the ceiling height, while for free grids, calculate the floor height, and then record these two types of heights in two lists respectively.

3. Iterating through the coordinate of all grids in the final height map, and taking the lower one of the occupied ceiling height and the free floor height as the final height. If corresponding heights are missing, then mark this grid with a sentinel height value.

The reason we use this approach is that the octomap library may merge multiple small grids into a larger grid and mark the space inside this new large grid but not occupied as "free" grids. In this way, octomap can reduce the size of the octree at a cost of requiring users to combine the occupied and free octree data to get the correct occupancy results.

These sub height maps are then merged into a full height map according to their location. There may be some "holes" in the height map, that is where the corresponding grid is absent in the octomap. When there is a part of an obstacle too high to be scanned by the depth camera, or points in the point cloud of that area is too sparse when it is scanned, such holes would appear. To fill these holes, we assign them with a valid height value got from their neighbor grids. The filling strategy is shown in Algorithm 4 in pseudo code.

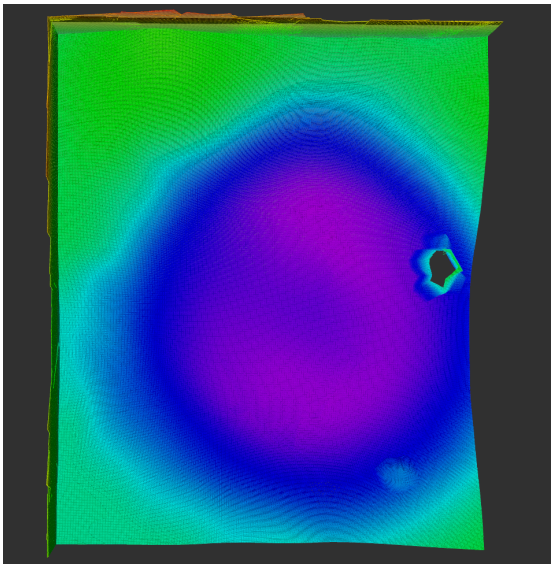
"Neighbor grids" here refers to grids with a difference of 1 in row and/or column index with the target grid, which means a grid has 8 neighbors unless it is located on the edge or corner of the height map.

Fig 5.10a, 5.10b show two types of holes, while Fig 5.11a, 5.11b show the full height map before and after filling. Note that hole grids are assigned with a sentinel height

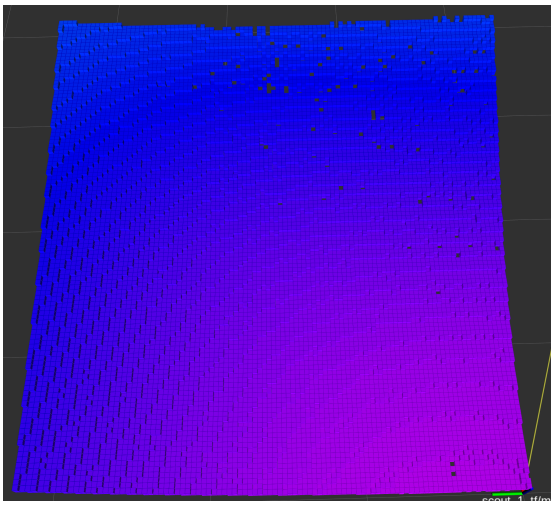
Algorithm 4: Filling holes in height map

```
Build an empty list hole_list
for row_id, col_id, height in height_map do
    if height is not valid then
        hole_list.append((row_id, col_id))
while hole_list is not empty do
    for row_id, col_id in hole_list do
        if at least one neighbor grid of (row_id, col_id) has a valid height then
            height_map[row_id][col_id] = mean(valid neighbor grid heights)
```

value (6m) which is much higher than valid height values, and all height values in the map is rescaled to integers in range of [0, 255] to be shown as a grayscale picture. Therefore, the the height map before filling looks brighter than the other one.



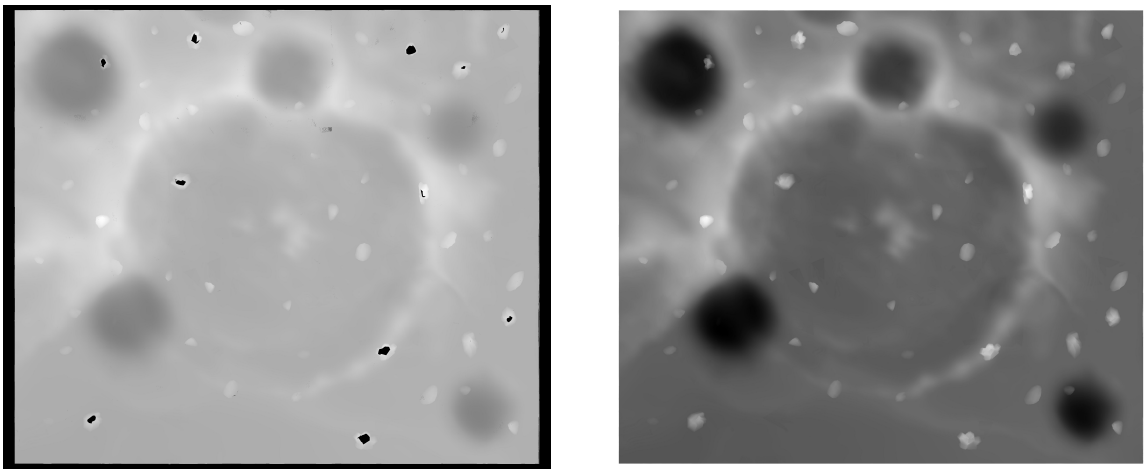
(a) a hole corresponding to the higher part of an obstacle



(b) small holes caused by sparse point cloud far from the depth camera

Figure 5.10: Examples of different types of holes in map building

For the second environment, the height map is directly obtained by converting the stl format terrain model file, where a 2D Clough-Tocher interpolation [64] is applied to get the height value of point.



(a) full height map before filling holes (b) full height map after filling holes

Figure 5.11: Full height map before/after filling holes (lighter color shows higher places in both the colored octree and height map

5.4 Data Collection

The traversability of a ground area for a robot can be measured by different indicators such the time cost or energy consumption for traversing this area. In our research we choose to use the time cost as the measurement, and to simplify the traversability estimation problem we only focus on the situation where a robot drives straight ahead rather than including possible steering.

Before evaluating any traversability estimation method, we need to collect driving data samples first. In our experiment, a driving sample consists of two parts: a list of odometry data of the robot starting from a designated point driving straight forward towards a given direction for a given distance, and a corresponding rectangular sub height map of the strip area the robot passes.

In detail, we divide the whole map into $4m \times 4m$ "sample grids", edges of which are parallel to x-axis and y-axis respectively and aligned to the origin of the world frame — for instance, the grid with four vertices (0,0), (4,0), (4,4) and (0,4) is a sample grid — and we set the center of each sample grid as the starting point of a driving sample. Thus, We have $30(rows) \times 36(columns) = 1080$ such sample grids within the first map of $126.0m \times 145.5m$, and $46(rows) \times 52(columns)$ within the second map of $188.6m \times 208.2m$.

Next, to fully utilize all the space of the simulation environment to collect as many driving samples as possible, we place the robot at each starting point with an

orientation from 8 choices in different runs, namely $(0, 1, \ldots, 7) \times \pi/4$ (only the 4 orthogonal orientations are used in the second environment) where according to right hand rule from ROS REP-103 [62] of ROS the orientation value equals 0 when the robot pointing to the east and increases in the counter-clockwise direction. In this way, we will have 8 driving samples with every starting point.

Then the robot was commanded to move forward with a given velocity (20m/s) for a given distance (4.614m) within a time limit (15s). Odometry data of the robot published in a ROS topic named "/scout_1/true_odom" will be recorded during the driving time. The result whether the robot finally reach the destination and the total time cost will also be calculated and added to the odometry data. This odometry recording process is shown in Fig 5.12.

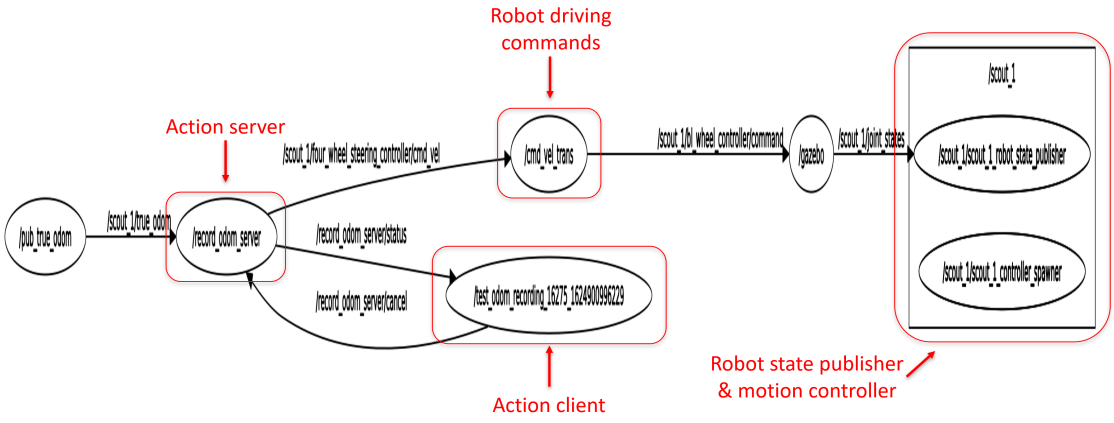


Figure 5.12: ROS node graph for odometry recording

As for the strip height map, it covers the area the robot passes, extending along the driving direction and a little wider than the width of the robot. Note that in this strip we also leave some extra spaces for both the area behind the back edge of the robot when it is located at the starting point and the area ahead the front edge of the robot when it is located at the designated destination. The reason is that the robot may starts on an uphill and slides back a little before it goes forward, and also, it may cross the destination a short distance before the last odometry message is published and recorded, and we want all the ground the robot may step on are included in the strip height map.

Parameters of the odometry recording and strip map are listed in table 5.5.

As shown in Fig 5.13, we choose an actual driving length as about $4.6m$ since it

Table 5.5: Robot parameters

Parameter	Value
Strip size	$8.5m \times 4.0m$
Robot size	$2.286m \times 2.2098m$
Robot velocity	$1.5m/s$
Actual traveling length	$4.614m$
Front reserved length	$0.6m$
Back reserved length	$1.0m$

is long enough in most cases for the robot to accelerate to the designated velocity before it arrives at the destination and it is also roughly two times of the robot length which can make it more convenient to apply some traversability estimation methods in the next section on collected driving samples.

To implement this data collection procedure, we build two ROS nodes: The first node "recording_odom_client" is an action client based on the ROS package "actionlib", which is responsible for enumerating all valid starting points and orientations, sending them to the server node and dumping the received response; The second node "recording_odom_server" is the corresponding action server, where the robot's motion commands are actually given and the odometry messages are actually recorded.

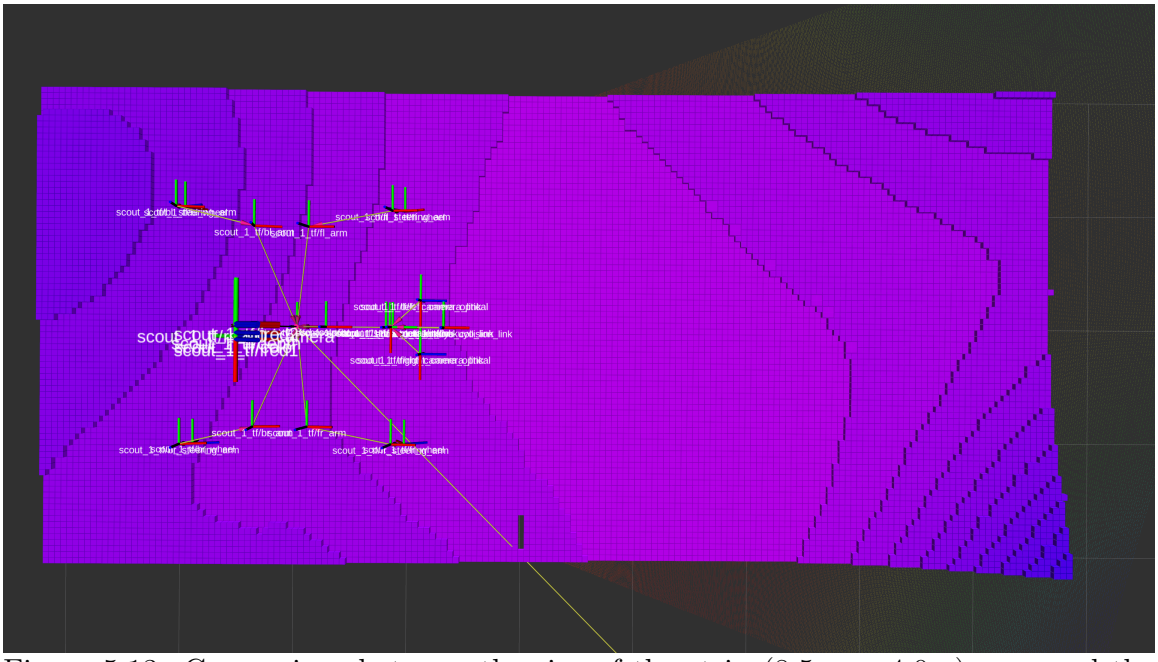


Figure 5.13: Comparison between the size of the strip (8.5m × 4.0m) map and the robot (2.286m × 2.2098m)

The whole robot driving and odometry recording process in one run can be divided into 5 steps:

Step 1: Planning. At the beginning of one run, the client node generates a pair of starting orientation and position by iterating through all possible combinations. Those grids at the end row and/or column of the currently selected orientation won't be selected as the starting position since there is not enough space for robot to drive before hitting the bounding boxes in these starting configurations. For example, when the orientation is set to east ($orientation = 0$), grids in the rightmost column will not be selected, and when the orientation is set to northeast ($orientation = \pi/4$), grids in the top row and rightmost column will not be selected.

Step 2: Teleport. When the starting orientation and position are determined, the robot will be "teleported" to there with a given height (introduced in step 3), and then the program will wait for 5 seconds for the robot to fall on the ground.

Step 3: Conditions check. In this step, a check will be done to see if the robot is ready to start driving. In particular, 5 conditions must be met:

- The linear velocity of the robot along the x, y, z axis must be lower than $0.01m/s$.
- The angular velocity of the robot around the x, y, z axis must be lower than $1°/s$.
- Roll and pitch of the robot must not be greater than 45°.
- The absolute distance between the planned starting point and the actual landing point along x and y axis must not be greater than $1m$.
- The absolute difference between the planned starting orientation and the actual orientation must not be greater than 10°.

The conditions are checked every 0.5 seconds for at most 10 seconds. If all of them are met in 5 consecutive seconds, the robot will be determined as ready for driving and the process will go to the next step. Otherwise, the robot will be determined as not in a good condition for driving. This may be because the robot is teleported to a place too high above the ground and then jumps or slides far away from the planned position. Therefore, the process will go back to step 2 and try to teleport the robot to the same place with a lower height. All available heights include $6.5m$, $4.0m$, $3.0m$,

$2.0m$, $1.0m$, $0.0m$ and $-1.0m$. If the check fails at all heights, this pair of starting orientation and position will be abandoned.

Step 4: Robot driving and odometry recording. After the robot is determined to be ready, the client node will send a request to inform the server node to drive the robot forward and start to record its odometry. Actually, the server node is always subscribing the odometry message topic of the robot which is published in 20Hz. The driving and recording will start at the next time it receives a new odometry message after it receives the request.

The server node publish a ROS geometry_msgs/Twist (consisting of a linear velocity vector and an angular velocity vector) message with a linear velocity of $20m/s$ and an angular velocity of 0 every time the it receives an odometry message to keep driving the robot forward until this run is finished.

A run will be finished when one of three terminating conditions is met: 1. the robot arrives at the destination; 2. the time limit is exceeded; 3. the robot overturned. 4. the robot deviates from the designated path.

The first two conditions are simple and self-explanatory. The third condition means the roll and pitch of the robot should never exceed the safe threshold(45°), while the last one is composed of multiple rules:

1. The orientation of the robot should not deviates from the initial orientation by more than or equal to 90°.

2. left wheels of the robot should not step out of the left boundary of the designated strip area.

3. right wheels of the robot should not step out of the right boundary of the designated strip area.

4. back wheels of the robot should not step out of the back boundary of the designated strip area.

These rules will be checked in order every time a new robot odometry message is received, and if one of them is violated then the odometry recording will be ended and this run will be marked as a negative sample in which the robot cannot reach the destination successfully. This third terminating condition is set since we want to make sure the robot never step on somewhere not included in the planned strip

area. By virtue of this, we can guarantee that all the ground the robot may step on is included in the strip area.

Step 5: Sending response and dumping data. After all the odometry information is collected, the server node will send a response back to the client node. The response message will include the actual starting orientation and position, a success flag representing if the robot arrives at the destination successfully or not (success or fail), fail type (timeout, overturned, orientation deviation, cross left/right/back boundary), beginning and end timestamp, forward velocity from topic cmd_vel, and a list of ROS odometry messages.

The client will dump this response plus with the information about the planned starting orientation and grid into a data file for extracting training data and labels later. Interactions between the server and client nodes are also shown in the ROS node graph (Fig 5.12).

The success flag and time cost of a run will be used as labels in the classification and regression training in the next section. While the input data of training is the corresponding strip height map we will cut from the full height map for each driving sample.

Since the orientation of grids in a strip height map follows its actual driving orientation, which is very likely different from the cardinal orientation of the full height map, and the grid size is also different($3cm \times 3cm$ for full map, $5cm \times 5cm$ for strip map). A 2D Clough-Tocher interpolation [64] is applied to calculate the height of each grid in the strip map.

In the following interpolation process, we regard the position and height of the center point of each grid as the position and height of that grid, and we use an integer to represent the height of a grid, where 1 means $3cm$ — the octomap grid side length.

To generate a strip map through interpolation, first we will find a minimum cardinal rectangle area composed of full map grids which can cover the whole strip area. Then we expand this area with 5 rows/columns of full map grids in all four directions, since heights of surrounding full map grids are needed to interpolate the height of strip grids near the boundary of the strip area. A bounding height of $200 = 6m \div 3cm$ will be applied for those expanded grids outside the full map, and finally we get a strip height map. Note that although the grid side length of a strip map is $5cm$ in the x-y plane, its height is still represented as an integer where 1 means $3cm$.

At the end, we collected 7797 samples in environment 1 — a sample is composed of a traversable flag, a time cost (if traversable) and a strip height map. These

samples are collected by positioning the robot to the center of each sample grid in the environment (with a gird side length of 4.0m) with one of eight driving directions (4 orthogonal directions and 4 diagonal directions) and then command it to drive forward with a designated distance (4.6m). Among the first dataset, 7091 samples are positive(traversable) accounting for 90.9%, 706 samples are negative(untraversable), accounting for 9.1%. This dataset is divided into a training set(6238 samples in total, in which 5680 samples are positive, positive rate 91.1%) including 80% samples, and a testing set(1559 samples in total, in which 1411 samples are positive, positive rate 90.5%) including the rest 20% samples.

For the second environment, 8762 samples are collected in the same way, but we only use the four orthogonal driving direction this time. Since there are only 95 samples (1.1%) are non-traversable, we just use the traversable samples to train a time cost estimation model. The dataset is divided into a training set of 7010 samples (80%) and a testing set of 1657 samples (20%).

These two datasets are then used to train and test benchmark traversability estimation methods and our CNN-based models — the strip height maps ($height = 170, width = 80, grid\ side\ length = 5cm$) is the input data, and the binary traversability flag and time cost are the output labels. Some strip height maps are shown in Fig 5.14 as examples. Note that strip maps are aligned with the designated driving direction, which means in all data samples the robot moves from the bottom of the strip area to the top of it.

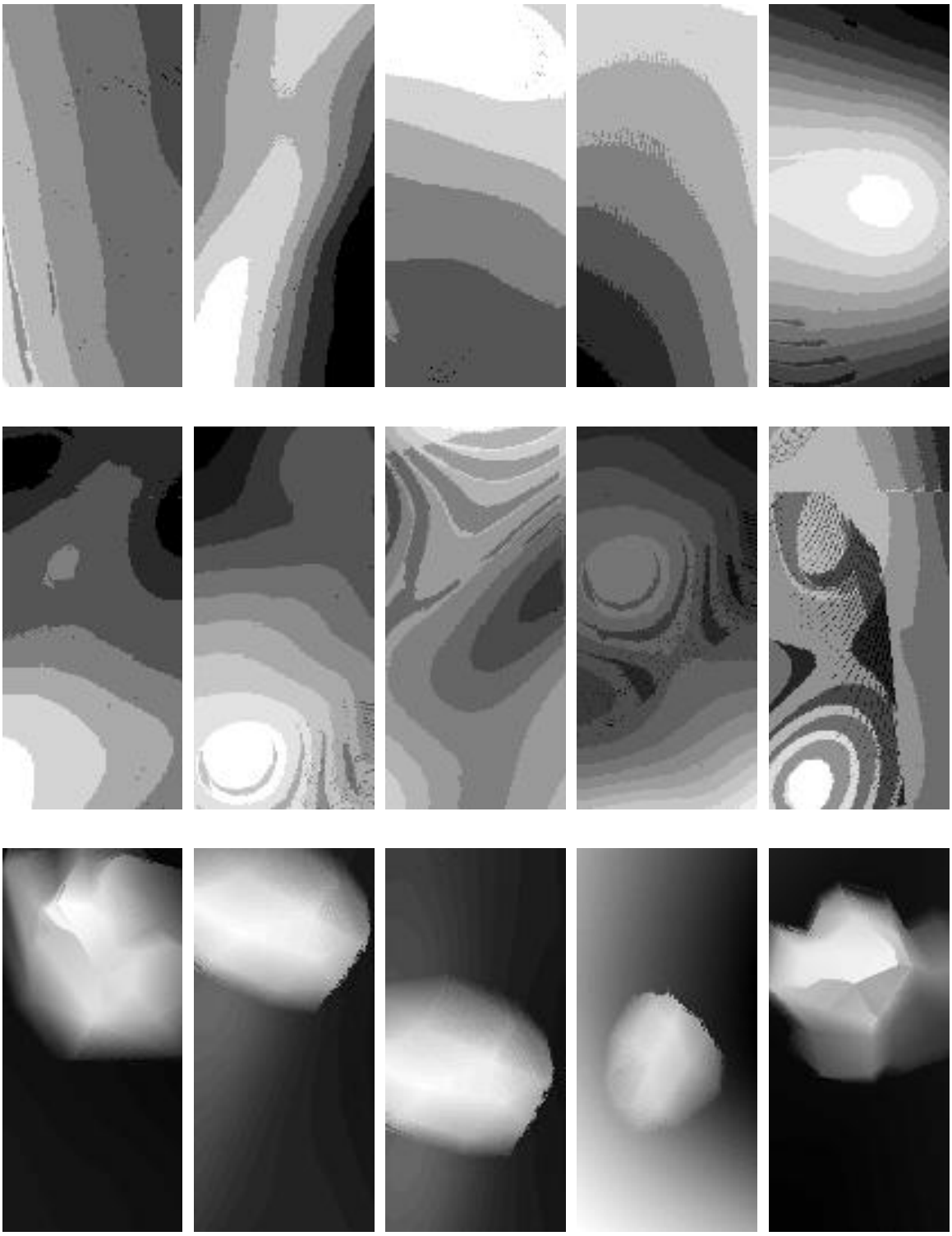


Figure 5.14: Strip height map samples (all height values are scaled to 0-255 to be presented as gray scale figures as shown here).

Chapter 6

Experiment results

This chapter we present the results of the data-driven methods in both classification and regression traversability estimation tasks and also compare the results against benchmark traversability estimation methods based on simple map features and manual rules.

This chapter divides the traversability estimation problem into two parts: a classification problem — whether an area is traversable or not, and a regression problem — how long does it take for a robot to drive through an area if it is traversable.

The main focus of this thesis is to address these problems by training CNN models on the height maps and odometry data collected. The performance of different CNN models on datasets representing different instances of traversability estimation problems is also compared with traditional non-parametric and parametric methods that consider simple ground features.

For the classification problem, four CNN models are trained (ResNet50, ResNet50V2, InceptionResNetV2, InceptionV3), using ImageNet pretrained convolution layer weights. The original top layers are replaced with 2 fully connection layers of 1000 nodes and sigmoid activation function and an output layer of 2 nodes with softmax activation function. The prediction is made in one hot format and the loss function is categorical cross entropy, which can better fit the natural imbalance of the terrain samples. The CNN model prediction process is shown in Fig 6.1.

On the other hand, several representative benchmark methods are re-implemented and tested on the same dataset:

1. Two ROS 2D grid map methods. They classify samples based on the maximum/average grid height of the strip height map — a threshold height is found

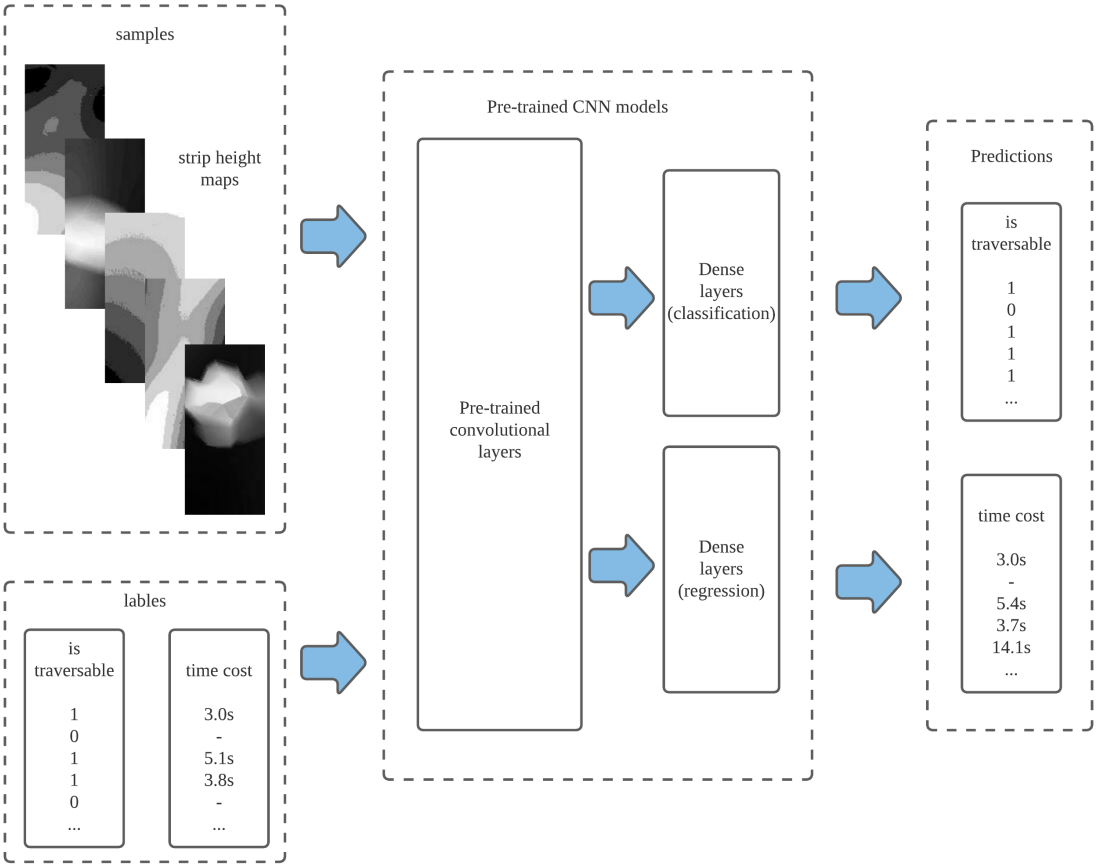


Figure 6.1: CNN-based model training process

on the training set and used to classify testing samples. This method is also directly applied on the testing dataset to show the upper boundary of the performance of such linear methods based on simple ground features.

2. A UPD-like method. A perceptron with 2 hidden dense layers of 1000 nodes, sharing the same architecture with the top layers of all the CNN models applied here. Input data is the overall roll, pitch and roughness of the strip area obtained from PCA. This benchmark method is heavily inspired by the UPD method described in section 3.2.1 of chapter 2. Comparison with this method can show the contribution of the convolutional layers in CNN models to the final performance.

3. A roughness indicator based method. A simplified version of the traversability estimation method in [27], which is discussed in section 3.2.2 of chapter 2. It calculates roll, pitch and roughness for every point in the point cloud based on

its neighbor points, and then estimate the overall traversability of an area by these features of each point within. Since this method is originally designed for sparse point cloud, the computing burden will be too high if we directly apply it on our grid height map. Therefore, we sample only some grids from the strip map to calculate these features — assuming the row index of a grid starting with 0 increases from top to bottom of the strip map while the column index increases from left to right, only grids whose row index and column index are both multiples of 10 will be sampled to calculate its roughness, but grids within $1m$ (i.e. 20 strip map grids) of strip boundary will be excluded since part of their neighbors are outside the strip area. In addition, only grids with both even row and column index will be used to determine the plane cycle. Parameters used in this method are shown in table 6.1.

Table 6.1: Parameters for the roughness indicator based method

Parameter	Value
sample step length	$0.5m$ (i.e. 10 grids)
r_{plane}	$1.0m$ (i.e. 20 grids)
r_{res}	$0.3m$ (i.e. 6 grids)
number of grids for determining the plane cycle	407
number of grids in the result cycle	113
f_η	0
roll/pitch threshold	90°
$roughness_{max}$	$1.0m$

Here we set r_{plane} and r_{res} as $1.0m$ and $0.3m$ since we can get roughly 100 grids in the result cycle which is close to the number of nearest-neighbor points in the original version of this method while keeping a ratio of 0.3 between r_{res} and r_{plane}. There are $16\ (rows) \times 5\ (columns) = 80$ pairs of plane cycle and result cycle evenly distributed with a step length of $0.5m$ (i.e. 10 grids), which can guarantee every grid in the center area of the strip is covered. $f_\eta = 0$ because noise points in point clouds are already filtered in the map building step, there is no need to exclude outliers here. $roughness_{max}$ is a parameter depicting the level of roughness within the result cycle. $roughness_{max} = 1.0m$ is the optimal value for time cost estimation we got in the regression training task which will be discussed later in this section.

This method is also tested by both the ordinary training & testing process (the ordinary version) and directly training on testing data (the optimal version). Results of the second way can show the upper boundary of the performance of this method.

In both ways, we sort the samples in the training/testing dataset by their roughness except those ones exceeding the roll/pitch threshold (which will be marked as untraversable), and then we find an optimal roughness threshold dividing the dataset into positive (traversable) and negative (untraversable) samples with the highest accuracy. Finally we will apply this threshold to the testing dataset.

The classification accuracy of all methods are shown in Fig 6.2.

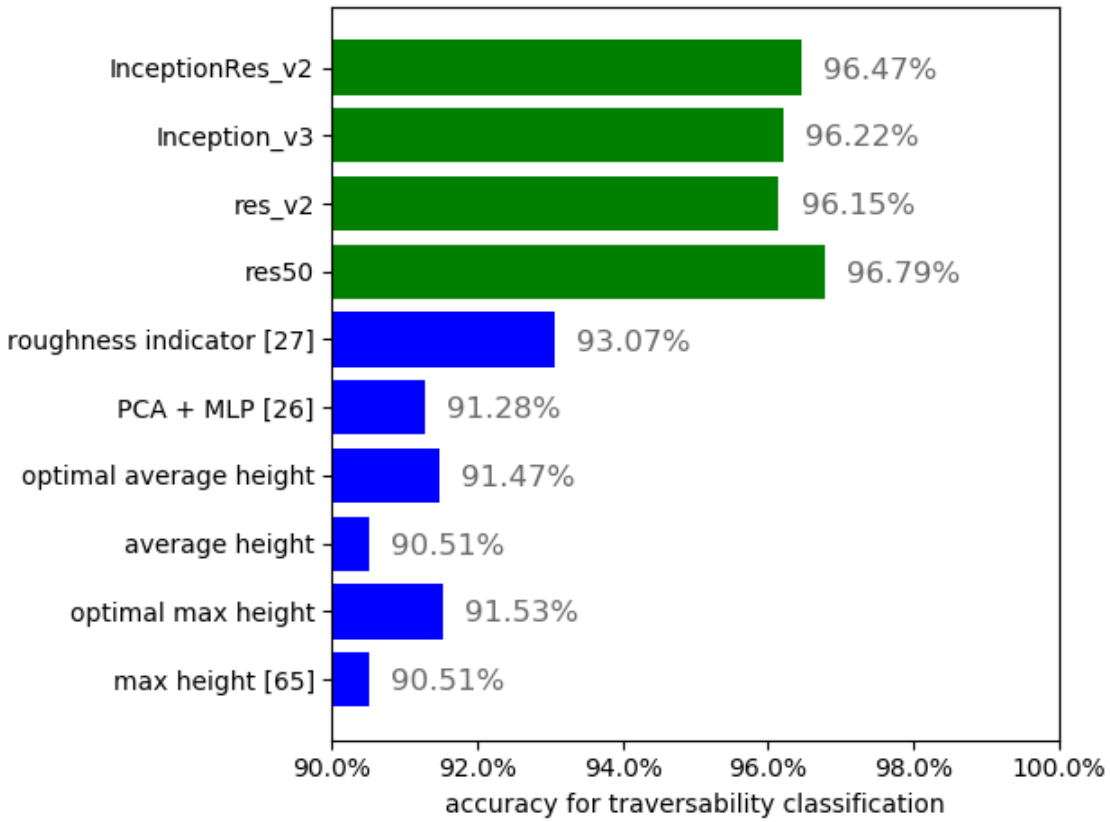


Figure 6.2: Traversability classification accuracy comparison in environment 1 (performance of our CNN-based methods are shown in green color)

We can see all CNN methods have an accuracy higher than all the benchmark methods, including their optimal version. Furthermore, considering there are 90.51% positive samples in the testing dataset, which means even a classification method always giving positive result would have a testing accuracy of 90.51%. Comparing with

this baseline, even the benchmark method with best performance can only improve the accuracy by 2.56%, leaving an inaccuracy of at least 6.93% (roughness indicator method), whereas our CNN-based methods can improve the accuracy by at least 5% and leave an inaccuracy of at most 3.85% (Res_v2), which is about 50% lower than the lowest inaccuracy of benchmark methods. This comparison shows the improvement on traversability classification accuracy made by CNN-based methods.

Then, our CNN-based methods are compared with benchmark methods on the regression problem — predicting the exact time cost of the robot for driving through a given strip. The architecture of all neuron network models are the same except their output layer is replaced with a one neuron layer with a linear activation function for outputting a real scalar value as the estimated time cost of driving in seconds. Meanwhile, results of benchmark methods being directly optimized on testing dataset (optimal version) are given in this regression performance comparison too.

Fig 6.3 and 6.4 display the time cost distribution of samples in the training and testing dataset of the two experiment environments. In these figures, the number of samples of each time span (the y-axis) is shown in log scale.

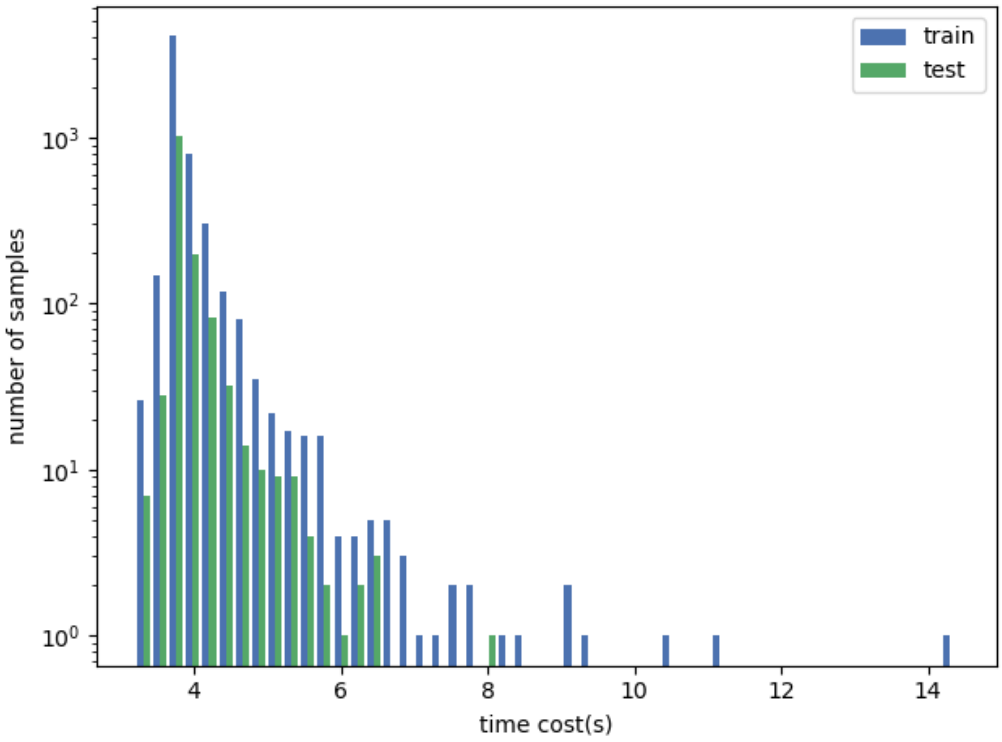


Figure 6.3: Time cost distribution of samples in environment 1 (in log scale)

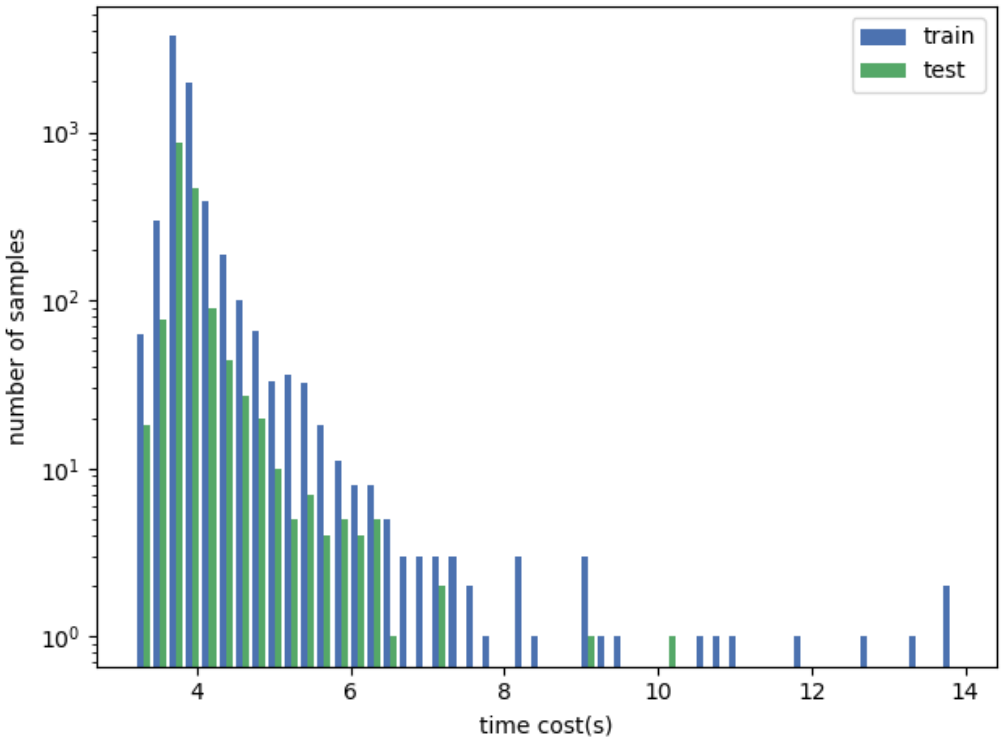


Figure 6.4: Time cost distribution of samples in environment 2 (in log scale)

It can be seen that more than 99.9% cases are in the range of $3s$ to $10s$, which proves the choice for timeout threshold — 15s, is long enough. It is worth noting that the time cost is not only dependent on the terrain but also affected by the structure and motion controller of the robot. New driving data can be collected in our simulation framework for training the models with a different robot.

Furthermore, as the original output of some benchmark methods cannot match the range of time cost in our dataset well (e.g. the original output of the roughness indicator method ranges in [0, 1]), translation and scaling are necessary to get an optimal prediction. We apply a linear transformation for the original result of all benchmark methods except the UPD (section 3.2.1) based one (scaling is done in its MLP part), optimizing their parameters on the training/testing dataset for the ordinary/optimal version method respectively to get a minimum root mean square error (RMSE).

This transformation and its optimization is shown below:

$$\hat{y_i'} = a \times \hat{y}_i + b$$

$$e_i = y_i - \hat{y_i'} = y_i - a \times \hat{y}_i - b$$

$$mse = \frac{\sum_{i=1}^{n} e_i^2}{n} = c_1 \times a^2 + c_2 \times b^2 + c_3 \times ab + c_4 \times a + c_5 \times b + c_6$$

$$set \quad \begin{cases} \frac{\partial mse}{\partial a} = 2c_1 \times a + c_3 \times b + c_4 = 0 \\ \frac{\partial mse}{\partial b} = 2c_2 \times b + c_3 \times a + c_5 = 0 \end{cases}$$

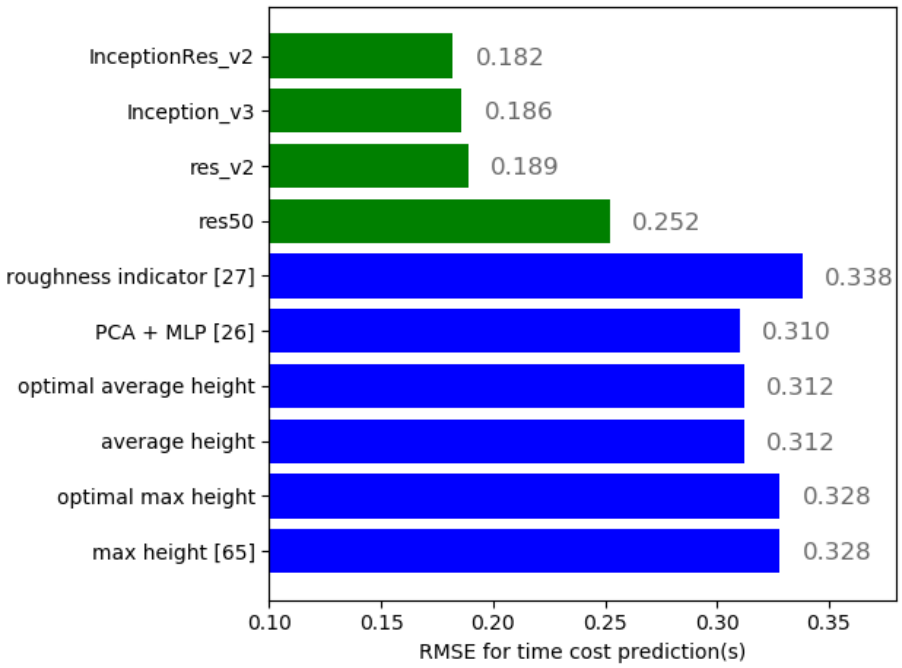
$$\Longrightarrow \quad a = \frac{c_3c_5 - 2c_2c_4}{4c_1c_2 - c_3^2}, \quad b = \frac{c_3c_4 - 2c_1c_5}{4c_1c_2 - c_3^2}$$

here a, b are parameters of the liner transformation; y_i is the true time cost of the i-th sample; $\hat{y}_i$ and $\hat{y_i'}$ are the original and transformed predicted time cost; e_i is the predicting error of the transformed result; n is the number of samples; mse is the mean square predicting error; $c_1 \ldots c_6$ are constants which can be calculated from the sample data. The MSE gets its minimum value when the partial derivatives of it with respect to a and b are both 0. Finally we get the values of a and b by solving these two equations.

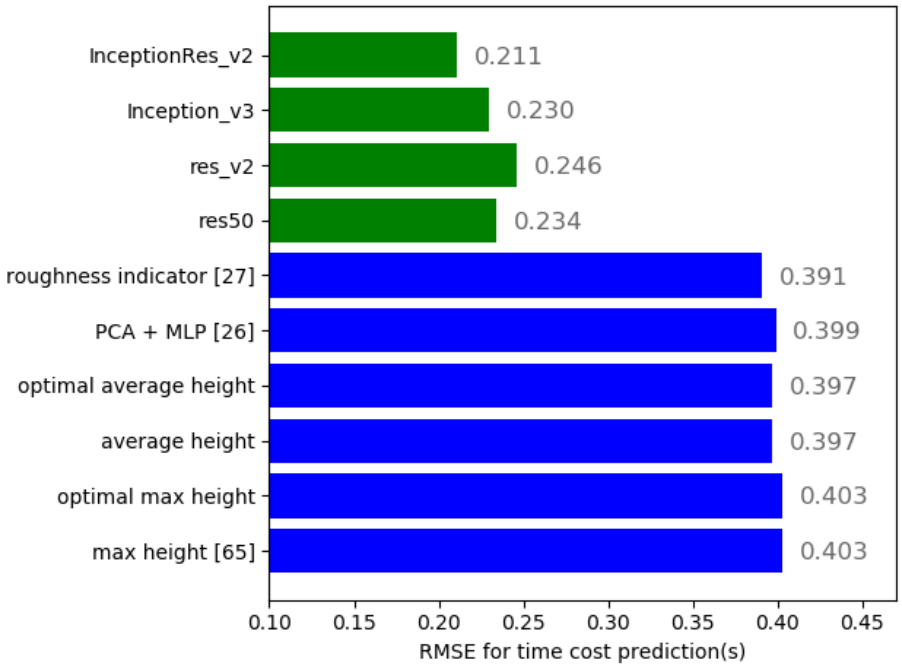
The root mean square error (RMSE) of the time cost prediction of all methods are shown in Fig 6.5.

Here RMSE of all the CNN-based methods is lower than their counterpart of benchmark methods in both environments. The best RMSE of benchmark methods is 70% longer than the best RMSE of CNN-based methods in environment 1 and 88% longer in environment 2. Moreover, compared with their ordinary version, not much improvement is made by those optimal benchmark methods, which are directly trained on the testing dataset. It implies that the estimation performance of parameterized methods is limited by its simple structure — the estimation error is high even with the optimal parameter configuration.

These two comparisons show the performance improvement made by CNN-based methods on traversability estimation as a regression problem (time cost prediction).



(a) experiment environment 1



(b) experiment environment 2

Figure 6.5: Comparison of time cost estimation RMSE

Through the experiment results, we justify that our approach as applying CNN on height map can make more accurate traversability estimations for both classification and regression tasks compared with conventional methods with manually designed features. More discussion about the reasons of this improvement are made in the next chapter.

Chapter 7

Conclusion

The goal of this thesis is proposing data-driven CNN-based terrain traversability estimation methods for mobile robots, in addition with a terrain traversability dataset. Firstly we introduced the usage of traversability estimation in mobile robot path planning and some representative conventional traversability estimation methods, then mobile robot localization and terrain map building are reviewed as they are the fundamental of traversability estimation. Finally, we proposed our data-driven CNN-based methods and showed that they perform better in both classification and regression estimation tasks in different testing environments than all the benchmark method.

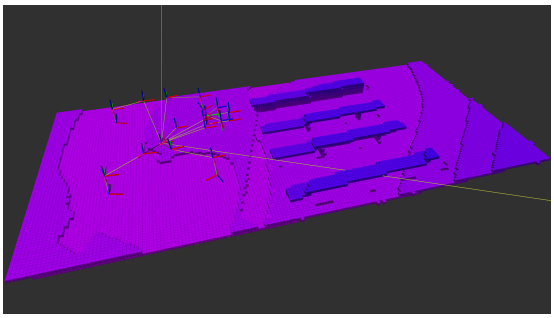
There are two sources for the performance improvement made by our methods:

First, CNN can extract more and better features from the raw map information. From experimental results, we can see all CNN models get a higher classification accuracy and a lower regression RMSE than the conventional methods based on manually designed features even when they are directly optimized on the testing dataset. This means the performance of CNNs exceeds the upper limit of the performance of those benchmark methods on their chosen features.

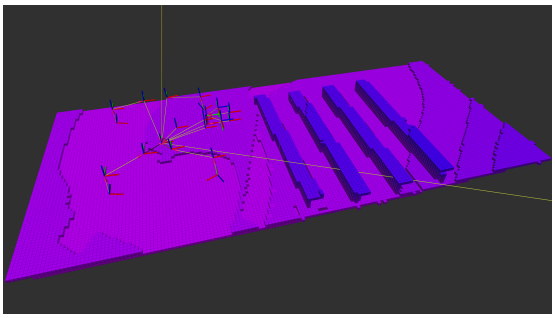
Examples in Fig 7.1 demonstrate that our CNN-based models generate better features from the map: There are 4 height maps with obstacles of the same number and size but with different angle or location to the robot. The map in Fig 7.1a is intuitively highly traversable since obstacles in it will not hinder the robot moving forward, whereas obstacles in Fig 7.1b are likely to stop the robot from moving forward and make the map less traversable or completely non-traversable. Thus, it is reasonable to infer that obstacles with different angles to the driving direction will lead to different traversability levels.

The relative location of obstacles to the robot is another factor affecting the traversability. As shown in Fig 7.1c, 7.1d, although the robot may be slowed down by an obstacle in front of one of its wheels, it can climb up and then pass the obstacle. But if the obstacle is right in front of the robot's chassis, the result varies according to the height of the obstacle — if it is lower than the gap between the ground and the robot's chassis, the robot will not be hindered. Otherwise, the robot is very likely to be completely blocked.

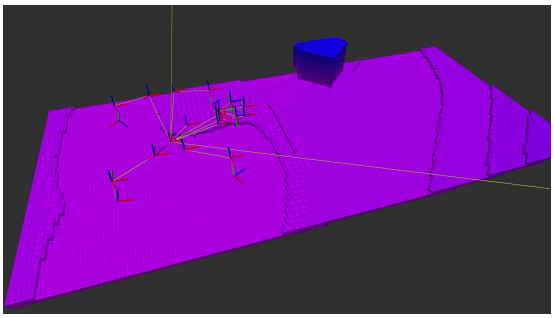
However, maps on the same row in Fig 7.1 will generate similar results if they are estimated by conventional methods with manually designed features like PCA roughness, which ignore the detailed difference between these maps.



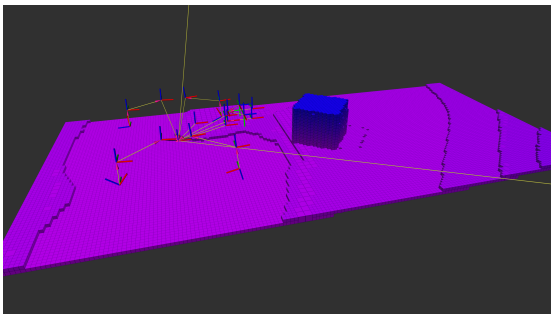
(a) a map with obstacles along the robot driving direction



(b) a map with obstacles perpendicular to the robot driving direction



(c) a map with an obstacle near the left map boundary



(d) a map with an obstacle in the middle of the path

Figure 7.1: Examples of detailed features which are essential to traversability estimation but difficult to capture with conventional methods

On the contrary, CNNs can capture such details — obstacles with various sizes, shapes, directions, locations can be captured by convolution layers and transformed to high-level features, and then fully connection layers will make the classification or regression prediction based on them.

Moreover, the pretrained CNN models we used here can help to accelerate the training. This is because convolution layers of these models are pretrained by huge image datasets such as ImageNet and therefore gain the ability to accurately extract image features like lines, angles, squares and cycles. Although those features are usually used to perform object recognition tasks, they are also helpful to traversability estimation on height grid maps as demonstrated in the experiment.

The second factor is the fully connected layers on the top of CNN models, namely a perceptron if we take them as an independent network. The better performance of this structure at fitting the latent function of traversability based on given features is due to its nonlinearity. As we demonstrated in the experiment and the discussed above, there are usually multiple essential features determining the overall traversability of an area together, but the final result may not change linearly with any one or any weighted combination of them. For example, an obstacle right in front of the robot (such as the one in Fig 7.1c) will stop the robot from moving forward if it is higher than the gap between the ground and the bottom of the robot's chassis. In opposite, it won't even slow down the robot a bit if it is not high enough. Therefore, the height change of such an obstacle matters the result most when its value is around the height of the gap. This relationship is naturally not suitable to be fitted with linear approximators. While by virtue of the universal approximation ability of neuron networks the perceptron on the top of a CNN model can fit any latent function accurately as long as it has enough width, depth and training epochs.

Besides the advantage at result accuracy, CNNs methods can also save human labor work for tuning estimation function parameters like those ones manually configured for the estimation method in [27], since all such parameters are automatically learned by training.

Future work

We proposed CNN-based methods on the height map, which are used to estimate the traversability in the outdoor environment of automatic robot driving, and showed their performance improvement in both classification and regression tasks. Starting from this result, there are various research topics can be further explored:

1. Creating a larger terrain traversability dataset. More terrain samples can be collected in various environments with different type of mobile robots. Such a

dataset can be used for fast training and testing of new traversability estimation methods without laborious data collection work.

2. Generalizing the traversability estimation to more situations. To simplify the data collection and network training task, we choose to estimate the traversability only in the condition that the robot starts from a static status and goes straight forward with a fixed velocity for a given distance. The next step is to include different driving direction, initial velocity, and traveling distance. The ultimate goal is getting an estimation model which can be applied on arbitrary robot status and paths.

3. Predicting the robot's trajectory. To get more precise control of a robot, we need to be able to predict its trajectory when it travels through an area rather than only the time and energy cost. One way to achieve this is to train a sequential model such as LSTM on the odometry of the robot with input motion commands and map information. However, as stated above, the motion commands can be highly sparse and the number of possible ground situations is enormous. It is a challenge to get a general trajectory predicting model as special simplifying and training designs are necessary.

4. Implementing CNN-based traversability estimation methods in autonomous robot driving and exploration. One of the most important application scenario of traversability estimation is autonomous robot exploration. Although many path planning algorithms such as RRT* have been applied to find an optimal or sub optimal path for the robot, in most cases these algorithms work on traversability estimations based on simple features and rules tuned manually, which are very inaccurate compared with estimations made by CNNs as we showed in the experiment. If CNN-based traversability estimation methods can be incorporated with autonomous robot driving and exploration algorithms, better and more accurate path plans can be made and therefore help robots to achieve a better performance in their real-world exploration tasks such as geographical surveying, resource finding and rescuing.

Bibliography

[1] Arkin, R.C., 1989. Motor schema—based mobile robot navigation. The International journal of robotics research, 8(4), pp.92-112.

[2] Koren, Y. and Borenstein, J., 1991, April. Potential field methods and their inherent limitations for mobile robot navigation. In ICRA (Vol. 2, pp. 1398-1404).

[3] Zelinsky, A., 1992. A mobile robot navigation exploration algorithm. IEEE Transactions of Robotics and Automation, 8(6), pp.707-717.

[4] DeSouza, G.N. and Kak, A.C., 2002. Vision for mobile robot navigation: A survey. IEEE transactions on pattern analysis and machine intelligence, 24(2), pp.237-267.

[5] Park, S. and Hashimoto, S., 2009. Autonomous mobile robot navigation using passive RFID in indoor environment. IEEE Transactions on industrial electronics, 56(7), pp.2366-2373.

[6] Pérez-Higueras, N., Jardón, A., Rodríguez, Á. and Balaguer, C., 2020. 3D Exploration and Navigation with Optimal-RRT Planners for Ground Robots in Indoor Incidents. Sensors, 20(1), p.220.

[7] Ho, K., Peynot, T. and Sukkarieh, S., 2013, May. Traversability estimation for a planetary rover via experimental kernel learning in a gaussian process framework. In 2013 IEEE International Conference on Robotics and Automation (pp. 3475-3482). IEEE.

[8] Ye, C., 2007. Navigating a mobile robot by a traversability field histogram. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 37(2), pp.361-372.

[9] Gu, J., Cao, Q. and Huang, Y., 2008. Rapid traversability assessment in 2.5 D grid-based map on rough terrain. International Journal of Advanced Robotic Systems, 5(4), p.40.

[10] Huang, H.P. and Chung, S.Y., 2004, September. Dynamic visibility graph for path planning. In 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566) (Vol. 3, pp. 2813-2818). IEEE.

[11] Amato, N.M. and Wu, Y., 1996, April. A randomized roadmap method for path and manipulation planning. In Proceedings of IEEE international conference on robotics and automation (Vol. 1, pp. 113-120). IEEE.

[12] Ma, H., Tovey, C., Sharon, G., Kumar, T.S. and Koenig, S., 2016, March. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In Thirtieth AAAI Conference on Artificial Intelligence.

[13] Silver, D., 2005. Cooperative Pathfinding. AIIDE, 1, pp.117-122.

[14] Sartoretti, G., Kerr, J., Shi, Y., Wagner, G., Kumar, T.S., Koenig, S. and Choset, H., 2019. PRIMAL: Pathfinding via reinforcement and imitation multi-Agent learning. IEEE Robotics and Automation Letters, 4(3), pp.2378-2385.

[15] Kuffner, J.J. and LaValle, S.M., 2000, April. RRT-connect: An efficient approach to single-query path planning. In Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065) (Vol. 2, pp. 995-1001). IEEE.

[16] LaValle, S.M. and Kuffner Jr, J.J., 2001. Randomized kinodynamic planning. The international journal of robotics research, 20(5), pp.378-400.

[17] LaValle, S.M., 2006. Planning algorithms. Cambridge university press.

[18] Karaman, S. and Frazzoli, E., 2011. Sampling-based algorithms for optimal motion planning. The international journal of robotics research, 30(7), pp.846-894.

[19] Asano, T., Asano, T., Guibas, L., Hershberger, J. and Imai, H., 1985, October. Visibility-polygon search and Euclidean shortest paths. In 26th Annual Symposium on Foundations of Computer Science (sfcs 1985) (pp. 155-164). IEEE.

[20] Keil, J.M. and Sack, J.R., 1985. Minimum decompositions of polygonal objects. In Machine Intelligence and Pattern Recognition (Vol. 2, pp. 197-216). North-Holland.

[21] Khatib, O., 1986. Real-time obstacle avoidance for manipulators and mobile robots. In Autonomous robot vehicles (pp. 396-404). Springer, New York, NY.

[22] Zou, A.M., Hou, Z.G., Fu, S.Y. and Tan, M., 2006, May. Neural networks for mobile robot navigation: a survey. In International Symposium on Neural Networks (pp. 1218-1226). Springer, Berlin, Heidelberg.

[23] Yang, S.X. and Meng, M., 2000. An efficient neural network approach to dynamic robot motion planning. Neural networks, 13(2), pp.143-148.

[24] Howard, A. and Seraji, H., 2000, November. Real-time assessment of terrain traversability for autonomous rover navigation. In Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113) (Vol. 1, pp. 58-63). IEEE.

[25] Kim, D., Sun, J., Oh, S.M., Rehg, J.M. and Bobick, A.F., 2006, May. Traversability classification using unsupervised on-line visual learning for outdoor robot navigation. In Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006. (pp. 518-525). IEEE.

[26] Reina, G., Bellone, M., Spedicato, L. and Giannoccaro, N.I., 2014. 3D traversability awareness for rough terrain mobile robots. Sensor Review.

[27] Krüsi, P., Furgale, P., Bosse, M. and Siegwart, R., 2017. Driving on point clouds: Motion planning, trajectory optimization, and terrain assessment in generic non-planar environments. Journal of Field Robotics, 34(5), pp.940-984.

[28] Ho, K., Peynot, T. and Sukkarieh, S., 2013, May. Traversability estimation for a planetary rover via experimental kernel learning in a gaussian process framework. In 2013 IEEE International Conference on Robotics and Automation (pp. 3475-3482). IEEE.

[29] Durrant-Whyte, H., Rye, D. and Nebot, E., 1996. Localization of autonomous guided vehicles. In Robotics Research (pp. 613-625). Springer, London.

[30] Leonard, J.J. and Durrant-Whyte, H.F., 1991, November. Simultaneous map building and localization for an autonomous mobile robot. In IROS (Vol. 3, pp. 1442-1447).

[31] Csorba, M. and Durrant-Whyte, H.F., 1997, June. New approach to map-building using relative position estimates. In Navigation and Control Technologies for Unmanned Systems II (Vol. 3087, pp. 115-125). International Society for Optics and Photonics.

[32] Csorba, M., 1997. Simultaneous localisation and map building (Doctoral dissertation, University of Oxford).

[33] Durrant-Whyte, H. and Bailey, T., 2006. Simultaneous localization and mapping: part I. IEEE robotics & automation magazine, 13(2), pp.99-110.

[34] Bailey, T. and Durrant-Whyte, H., 2006. Simultaneous localization and mapping (SLAM): Part II. IEEE robotics & automation magazine, 13(3), pp.108-117.

[35] Kalman, Rudolph Emil. "A new approach to linear filtering and prediction problems." (1960): 35-45.

[36] McElhoe, B.A., 1966. An assessment of the navigation and course corrections for a manned flyby of mars or venus. IEEE Transactions on Aerospace and Electronic Systems, (4), pp.613-623.

[37] Smith, G.L., Schmidt, S.F. and McGee, L.A., 1962. Application of statistical filter theory to the optimal estimation of position and velocity on board a circumlunar vehicle. National Aeronautics and Space Administration.

[38] Maybeck, P., Stochastic models, estimation and control, vol. 1; 1979.

[39] Dissanayake, M.G., Newman, P., Clark, S., Durrant-Whyte, H.F. and Csorba, M., 2001. A solution to the simultaneous localization and map building (SLAM) problem. IEEE Transactions on robotics and automation, 17(3), pp.229-241.

[40] Julier, S.J. and Uhlmann, J.K., 2001, May. A counter example to the theory of simultaneous localization and map building. In Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164) (Vol. 4, pp. 4238-4243). IEEE.

[41] Montemerlo, M., Thrun, S., Koller, D. and Wegbreit, B., 2002. FastSLAM: A factored solution to the simultaneous localization and mapping problem. Aaai/iaai, 593598.

[42] Dellaert, F., Fox, D., Burgard, W. and Thrun, S., 1999, May. Monte carlo localization for mobile robots. In Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C) (Vol. 2, pp. 1322-1328). IEEE.

[43] Sugano, M., Kawazoe, T., Ohta, Y. and Murata, M., 2006. Indoor Localization System using RSSI Measurement of Wireless Sensor Network based on ZigBee Standard. Wireless and Optical Communications, 538, pp.1-6.

[44] Heurtefeux, K. and Valois, F., 2012, March. Is RSSI a good choice for localization in wireless sensor network?. In 2012 IEEE 26th international conference on advanced information networking and applications (pp. 732-739). IEEE.

[45] Yiu, S., Dashti, M., Claussen, H. and Perez-Cruz, F., 2017. Wireless RSSI fingerprinting localization. Signal Processing, 131, pp.235-244.

[46] Barsocchi, P., Lenzi, S., Chessa, S. and Giunta, G., 2009, April. A novel approach to indoor RSSI localization by automatic calibration of the wireless propagation model. In VTC Spring 2009-IEEE 69th Vehicular Technology Conference (pp. 1-5). IEEE.

[47] Bekcibasi, U. and Tenruh, M., 2014. Increasing RSSI localization accuracy with distance reference anchor in wireless sensor networks. Acta Polytechnica Hungarica, 11(8), pp.103-120.

[48] Rohra, J.G., Perumal, B., Narayanan, S.J., Thakur, P. and Bhatt, R.B., 2017. User localization in an indoor environment using fuzzy hybrid of particle swarm optimization & gravitational search algorithm with neural networks. In Proceedings of Sixth International Conference on Soft Computing for Problem Solving (pp. 286-295). Springer, Singapore.

[49] Mohammadi, M. and Al-Fuqaha, A., 2018. Enabling cognitive smart cities using big data and machine learning: Approaches and challenges. IEEE Communications Magazine, 56(2), pp.94-101.

[50] Huang, G.B., Zhu, Q.Y. and Siew, C.K., 2006. Extreme learning machine: theory and applications. Neurocomputing, 70(1-3), pp.489-501.

[51] Kennedy, J. and Eberhart, R., 1995, November. Particle swarm optimization. In Proceedings of ICNN'95-international conference on neural networks (Vol. 4, pp. 1942-1948). IEEE.

[52] Rashedi, E., Nezamabadi-Pour, H. and Saryazdi, S., 2009. GSA: a gravitational search algorithm. Information sciences, 179(13), pp.2232-2248.

[53] Mirjalili, S., Hashim, S.Z.M. and Sardroudi, H.M., 2012. Training feedforward neural networks using hybrid particle swarm optimization and gravitational search algorithm. Applied Mathematics and Computation, 218(22), pp.11125-11137.

[54] Li, M. and de Oliveira, T.E.A., 2020. EP-FPG applied to RSSI-Based Wireless Indoor Localization. In 2020 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE) (pp. 1-6). IEEE.

[55] Elfes, A., 1989. Using occupancy grids for mobile robot perception and navigation. Computer, 22(6), pp.46-57.

[56] Yan, R.J., Wu, J., Lee, J.Y. and Han, C.S., 2013, October. 3D point cloud map construction based on line segments with two mutually perpendicular laser sensors. In 2013 13th International Conference on Control, Automation and Systems (ICCAS 2013) (pp. 1114-1116). IEEE.

[57] Hornung, A., Wurm, K.M., Bennewitz, M., Stachniss, C. and Burgard, W., 2013. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. Autonomous robots, 34(3), pp.189-206.

[58] Grisetti, G., Stachniss, C. and Burgard, W., 2007. Improved techniques for grid mapping with rao-blackwellized particle filters. IEEE transactions on Robotics, 23(1), pp.34-46.

[59] Doucet, A., 1998. On sequential simulation-based methods for Bayesian filtering.

[60] Doucet, A., De Freitas, N. and Gordon, N., 2001. An introduction to sequential Monte Carlo methods. In Sequential Monte Carlo methods in practice (pp. 3-14). Springer, New York, NY.

[61] Stanford Artificial Intelligence Laboratory et al., 2018. Robotic Operating System, Available at: https://www.ros.org.

[62] Foote, T., Purvis, M., (2014) Standard Units of Measure and Coordinate Conventions, Available at: https://www.ros.org/reps/rep-0103.html

[63] Goudge, Timothy, 2019, "Jezero Delta, Mars 3D Print File", Available at https://doi.org/10.18738/T8/TUANBI, Texas Data Repository, V1

[64] Alfeld, P., 1984. A trivariate clough—tocher scheme for tetrahedral data. Computer Aided Geometric Design, 1(2), pp.169-181.

[65] Marder-Eppstein, E., Lu, D., Hershberger, D. et al., "ROS costmap_2d", Available at http://wiki.ros.org/costmap_2d