VISION-BASED STAIR DETECTION AND TERRAIN CLASSIFICATION ALGORITHMS FOR MULTI-TERRAIN MOBILE ROBOTS

A Thesis

by

VISHNU PRASHANTH KALYANRAM

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Kiju Lee
Committee Members,	Sivakumar Rathinam
	Xingyong Song
Head of Department,	Guillermo Aguilar

May 2022

Major Subject: Mechanical Engineering

ABSTRACT

This work presents vision-based stair detection and environment classification algorithms for mobile robots capable of traversing staircases and different types of terrains. These algorithms are developed for a specific hardware platform, called α-WaLTR, which is equipped with wheel-and-leg transformable mechanism enabling multi-terrain locomotion. The design of the hardware platform is optimized to allow for climbing over irregular terrains and continuous obstacles, such as staircases. It is equipped with Jetson TX2 as the main processing board, an Inertial Measurement Unit (IMU) and a Global Positioning System (GPS) for odometry, and a Light Detection And Ranging (LiDAR) device and an RGB-Depth (RGB-D) camera. The stair detection algorithm takes the color and depth image feed from the RGB-D camera and uses it to identify straight line patterns that could constitute a stairway. To further embed the robot with the terrain classification capability, the color images are segmented into traversable and non-traversable regions, thereby making urban environments more accessible. Taking the computational limitations into account, it is explored how these schemes can be integrated into the robot navigation stack using Robot Operating System (ROS).

DEDICATION

To my family.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Kiju Lee, for her support, untiring enthusiasm, patience, and encouragement, over the course of this research. I would like to convey my gratitude for the opportunity to work in a collaborative and constructive environment. I would also like to thank the committee members, Dr. Song and Dr. Rathinam, for their patience, and feedback.

I also thank the research team of the Adaptive Robotics and Technology (ART) Lab for making me a part of their team, and assisting me and enriching my stay at Texas A&M.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a committee consisting of Dr. Kiju Lee of the Department of Engineering Technology & Industrial Distribution and the Department of Mechanical Engineering, Dr. Xingyong Song of the Department of Engineering Technology & Industrial Distribution, and Dr. Sivakumar Rathinam of the Department of Mechanical Engineering.

The final prototype discussed in Chapter III is a collaborative effort by Chuanqi Zheng, Siddharth Sane, Kangneoung Lee, and Yuan Wei in the Department of Mechanical Engineering and Jenna Horn in the Department of Engineering Technology & Industrial Distribution, led by Dr. Kiju Lee. The datasets employed in Chapter IV was collected and compiled in collaboration with Kangneoung Lee.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This work was supported and made possible by Defense Advanced Research Projects Agency (DARPA), under Grant Number HR00112020037, in the OFFSET Sprint-5 program. Its contents are solely the responsibility of the author and do not necessarily represent the official views of DARPA.

NOMENCLATURE

UGV	Unmanned Ground Vehicle
RGB-D	Combination of RGB (Red-Green-Blue) and Depth channels image or camera
LiDAR	Light Detection and Ranging - the acronym used for a type of range sensor
GPU	Graphics Processing Unit
SVM	Support Vector Machine
GSCNN	Gated-shape Convolutional Neural Network
CNN	Convolutional Neural Network
HPRC	Texas A&M High Performance Research Computing Facility
ROS	Robot Operating System
IMU	Inertial Measurement Unit
GNSS	Global Navigation Satellite System
GPS	Global Positioning System - a subset of GNSS
RADAR	Radio Detection and Ranging
WLAN	Wireless Local Area Network

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This chapter first provides an overview of the research area – vision-based algorithms for autonomous navigation in unmanned ground vehicles (UGVs) and mobile robotic platforms – and summarizes the existing work. The following sections detail the technical scope of work and research objectives.

1.1 Background

Research in the field of robotics, in particular ground mobile robots, has seen a marked rise over the past two decades. This has ushered in an inclination to adopt robotic systems to assist humans in numerous tasks. Simultaneously, the degree of autonomy (or the complexity of tasks that a robot can be trusted with) has also progressively increased. The subclass of UGVs has been following a trend analogous to Moore's Law – becoming cheaper to build and more compact, as their capabilities expand – and are being employed in a variety of applications. Such vehicles make use of a few different modes of locomotion, common among which are wheels [1] [2] [3], legs [4] [5], and tracks [6] [7]. With the increased need for versatility, hybrid locomotion modes have also been proposed, with both active and passive transformable wheel-leg-track designs being popular [8] [9] [10] [11].

The development of control algorithms for such robots is subject to the mode of locomotion. For instance, wheeled robots are generally limited to level terrains with a small degree of ruggedness. Robots equipped with legs can not only traverse more rugged terrains but can also negotiate complex obstacles present in their path like staircases. However, this comes at a cost. The more advanced the capabilities, the higher the required degree of intelligence. The problem to be solved in such cases is two-fold: 1) creating a scheme that enables efficient locomotion [12] (i.e. synchronization of actuators and the like); and 2) developing algorithms that sense the vehicle's surroundings and act based on the presence and type of environmental features. While the overall objectives of the project this thesis research is associated with encompass both aspects, the subject of this thesis focuses on the second type of perception algorithms that can be employed in intelligent decision-making schemes.

1.2 Scope of Work

This work is targeted towards developing vision algorithms for stair detection and terrain classification that can be implemented to a new small-size UGV platform, named α-WaLTR. It is a scaled-up and optimized design of the Wheel-and-Leg Reconfigurable mechanism (WheeLeR) prototype presented in [8]. This UGV has a modular design that can be equipped with four n-leg transformable wheels ($n = 3, 4, 6$), and therefore, combines the simplicity of the wheeled locomotion control with the enhanced locomotion capabilities enabled by wheel-leg transformation. In addition, it retains the passive-actuation capability of the WheeLeR design, wherein the transformation between the two modes of locomotion does not require another active actuator. α-WaLTR's embedded sensor suite make it suitable for search and rescue, exploration, and reconnaissance applications in urban (and suburban) environments. The design and capabilities of this platform are further detailed in Chapter 3.

To supplement the enhanced physical capability of climbing over discrete and continuous obstacles, a robust scheme for employing the feed from the sensor suite to ascertain the presence of such environmental features is necessary. One such feature, omnipresent in indoor and outdoor urban environments, is staircases that connect multiple levels in buildings and other structures. These also serve the purpose of being a contingency plan for evacuation and exploration. Therefore, the first defined challenge is to develop an algorithm that is capable of detecting staircases and thereby enable traversal of the same.

For a platform equipped with hybrid modes of locomotion, there is also a need for optimal path selection. The most popular open-source vision-based navigation algorithm for wheeled robots makes use of binary obstacle classification [13]. Such schemes may not always account for enhanced locomotion capabilities of multi-terrain UGVs. For optimal operation, it is desired that the robot's mode of operation is adaptive with respect to the terrain conditions. Therefore, another defined challenge is to classify the robot's operating environment to correspond to the improved capabilities.

α-WaLTR's software architecture is based on Robot Operating System (ROS) [14], employing open-source (e.g. the *move-base* Navigation stack [13]) and custom-designed libraries for autonomous operation. Therefore, there is a need to integrate such algorithms within the existing navigation stack while being modular and allowing for improvements in the future.

1.3 Research Objectives

The primary technical objectives of this research include the following:

1. Develop a scheme to detect ascending stair-like structures in the environment using RGB-depth (RGB-D) data.

2. Develop an algorithm for identifying (and adapting to) varying environmental features as traversable or otherwise using RGB data.

3. Test and benchmark the performance of proposed algorithms.

2. STAIR DETECTION AND TERRAIN CLASSIFICATION ALGORITHMS

This chapter details the proposed vision algorithms for staircase detection and terrain classification, alongside summarizing the related works of literature.

2.1 Staircase Detection

Staircases or stairways are one of the most commonly present features of urban and suburban environments. The presence of these structures has been rising with the number of multi-story buildings. Present both outdoors and indoors, they form an essential part of our infrastructure, owing to their use as a means of access and emergency egress. Therein lies their importance in search-and-rescue and related missions. In [15], a framework for autonomous detection and navigation of stairs was presented, involving three major phases: detecting the presence of a staircase in the frame and approaching it; aligning the body; and traversing. The focus here is on the first phase, which has been accomplished in a few different ways.

2.1.1 Related work

Because of the ubiquitous nature of these structures, several research domains have focused on attempting to solve this problem efficiently. One avenue of work in which this features heavily is in the development of human-assistive technologies. To alleviate the challenges individuals face with visual impairment, sensory augmentation is a popular route. Some existing works involve a wearable device with an inertial measurement unit (IMU) and a depth sensor capable of stair detection [16] and assistive canes equipped with Ultrasonic sensors [17] [18]. Monocular cameras have also been employed to detect staircase patterns using simultaneous spatial and frequency domain analysis [19] [20] and morphological spatial operations [21]. In [22], a novel sensor-fusion scheme for RGB-D image data was put forward that labeled input frames as containing a staircase (either ascending or descending) or otherwise using a Support Vector Machine (SVM) classifier.

When it comes to ground robots, such approaches for detecting complex stair-like features have made use of laser range finders, where the robots were tele-operable and full autonomy in navigation was not the goal [23] [24]. However, as the robots' effectors evolved and became capable of navigating complex scenarios, more robust sensing capabilities were deemed necessary. This was also accompanied by importance of on-board processing capabilities, meaning computationally intensive autonomous decision-making algorithms could be developed. The use of inputs from multiple sensors in conjunction has been profitably employed in stairway detection. Stereo-vision has been used to detect stair edges and fit planes to the stairway [25] [26] [27]. Detecting surfaces and analyzing the distance between parallel planes corresponding to stairs has also been accomplished with the use of RGB-D cameras [28] [29]. Effective techniques for stair detection from 2D image features have also been proposed. Some of these include simple image filtering and edge detection based on *a priori* inputs [30], frequency domain Gabor Filters [31], employing the Viola-Jones object detection framework [32], and using a YOLO deep-learning approach [33].

2.1.2 Machine learning based stair detection

In this thesis, a stairway detection scheme is based on and extended from the existing work presented in [22]. α-WaLTR is equipped with two RGB-D cameras on its front and back, and therefore it can make use of the two synchronous RGB-D image streams to make conclusive decisions about the presence or absence of stairways in the frames of view of the robot. The proposed method consists of 1) a pre-processing phase - where the RGB image is converted to grayscale and filtered to eliminate unnecessary features/artifacts; 2) a detection phase - where the edges and lines corresponding to the edges are detected and grouped based on location; and 3) a conclusion phase - where the depth values corresponding to the groupings are extracted and passed through a pre-trained classifier to determine if a staircase is present or not. Fig. 2.1 shows a schematic of this process in a flow chart. All image processing operations are performed using built-in OpenCV functions and libraries [34].

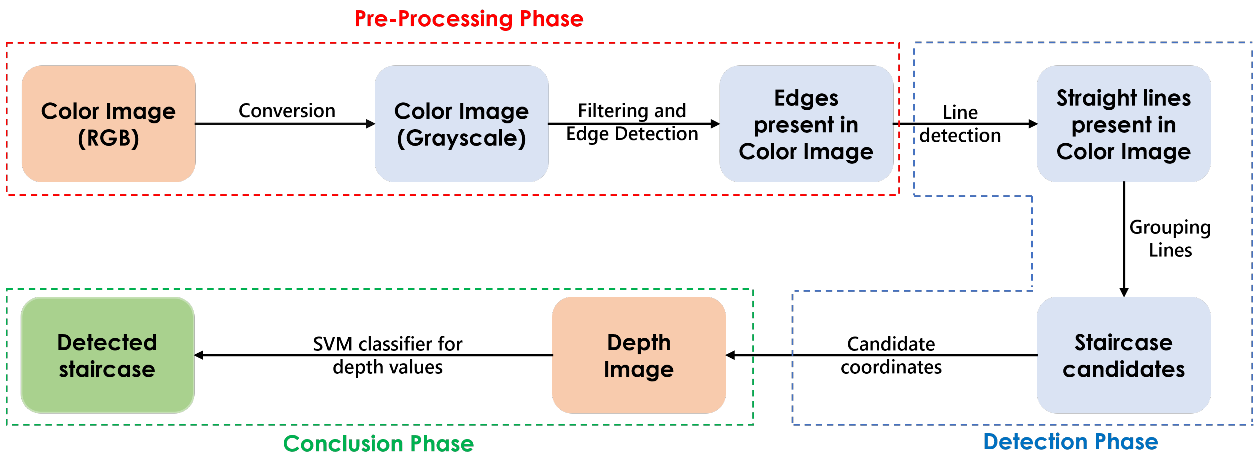

Figure 2.1: A schematic of the Staircase Detection approach.

Pre-processing phase

The color image frame is obtained as a 3-channel RGB image. Processing multi-channel data simultaneously is cumbersome and computationally inefficient. Therefore, the input image is converted to single channel grayscale image for ease of further processing. This is accomplished using OpenCV's *cv::cvtColor()* function. This color space conversion is achieved by calculating the weighted average of the three disparate bands. The individual channel weights are based on their wavelengths, and correct the image to account or human perception. At a pixel location (x, y), the grayscale intensity value $I_g(x, y)$ is computed using the following relationship [34]:

$$I_g(x,y) = 0.299 * I_R(x,y) + 0.587 * I_G(x,y) + 0.114 * I_B(x,y) \tag{2.1}$$

where, I_R, I_G, and I_G are the intensities of the Red, Blue, and Green channels respectively.

In order to further optimize computation and to eliminate unnecessary artifacts, the image is subject to morphological dilation using the *cv::dilate()* function. This process makes use of a structuring element (or Kernel) to expand shapes or elements present in the image. An elliptical kernel with a major axis of 20 pixels and eccentricity of $\sqrt{3}/2$ was employed for the application in question. Centering the Kernel at each pixel location, the intensity of all superimposed pixels of the base image are set to the maximum value of the set. At pixel location (x, y), this is mathematically expressed as:

$$dst(x,y) = \max_{(x',y'):element(x',y')\neq 0} src(x+x', y+y') \tag{2.2}$$

where src and dst are the Input and Output images respectively, and $element$ is the structuring element used for dilation.

This result of this is an image with strong edges enhanced and weak edges diminished, or in other words, it makes the lines pertaining to staircases stand out. The edges are then extracted by a process called the Canny Edge Detector (*cv::Canny()*). The Canny Detector functions as a multi-stage process. The first step is to reduce noise by filtering the image using a 5x5 Gaussian kernel. Following this, the intensity gradient of the image is computed using the Sobel operator. This is a 2D differentiation operation that produces the first derivative in the horizontal (x) and vertical (y) directions. An example of a typical 3x3 Sobel kernel is as follows:

$$S_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}; \quad S_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}; \tag{2.3}$$

By convolving the Sobel operators with the filtered image, the horizontal (G_x) and vertical (G_y) gradients are obtained. Using these, the edge gradient (G) and the gradient angle (θ) are calculated as:

$$G = \sqrt{G_x^2 + G_y^2}; \quad \theta = atan\left(\frac{G_y}{G_x}\right) \tag{2.4}$$

Depending on the gradient magnitude and direction, only the pixels that conform to a local maxima are retained. This ensures that pixels that do not constitute the edge are suppressed. The final step in edge detection is hysteresis thresholding, which makes use of an upper ('sure-edge') and a lower ('not-edge') intensity limit values to label and discard edges respectively. The pixels lying between the two are subjectively labeled based on their connectivity to other edges. An optimal choice of threshold values results in long and connected edges, and discards short and disjoint lines. The results of this phase are indicated by the red bounded box in Fig. 2.1. An example is presented in Fig. 2.2, with lines representing the stairway and other features in the frame.

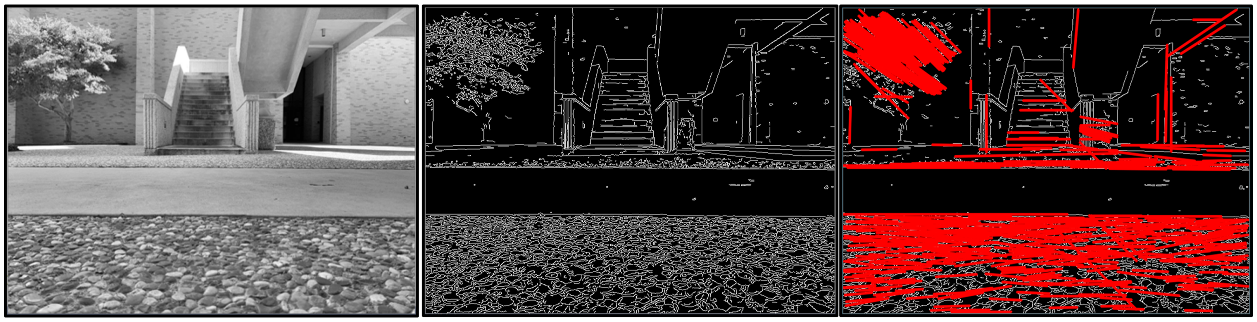

Figure 2.2: Pre-processing the RGB image ([**L**] to [**R**] sequence).

Detection phase

The derived edges are transformed to a 2-parameter space termed the Hough space, where straight lines are extracted by a voting procedure [34]. For a point represented in Cartesian space as (x, y), every line passing through it can be expressed in Polar co-ordinates as: $r_\theta = xcos\theta + ysin\theta$, or in other words, the plot the family of lines passing through a point yields a Sinusoid in the $r - \theta$ space. Extending this operation to more than one point, an intersection in the $r - \theta$ space implies that both points lie on a line that can be represented by the parameters r and θ of the point of intersection. This procedure of analyzing the number of intersections between curves is termed the Hough Line Transform. By this method, lines are extracted from the input image by computing the features that have above a set threshold of intersections in the Hough space. An optimized approach to this is the Probabilistic Hough Line Transform (PHLT) that only considers a subset of edges which are more likely to constitute a line.

In the chosen application, the PHLT is applied to the output of the Canny detector. The result of this is a vector of lines that are converted back to Cartesian space, i.e. represented by end points (x_1, y_1) and (x_2, y_2). The extracted lines are then grouped for further analysis. This grouping is based on the assumption that staircase edges would consist of parallel lines stacked vertically. The heuristic employed for this grouping made use of the slope (m) and Cartesian locations of their midpoints ($\bar{X}$):

$$m = \frac{y_2 - y_1}{x_2 - x_1}, \;\; \bar{X} = \frac{x_1 + x_2}{2} \tag{2.5}$$

Rejecting steep vertical and short lines and considering groupings of more than 2, candidate groupings are generated. The result of the PHLT being grouped into vectors of parallel lines is illustrated in Fig. 2.3. The criteria for the lines to be accepted is that their midpoints should lie within a set rectangular region and they must be roughly parallel. Here, sets 3 and 6 (without the final line) are considered an acceptable grouping. 1 is rejected for being a set of fewer than 3, sets 2 and 5 are rejected for having high slope values, and set 4 is rejected for its constituents not having similar slope despite their proximity. The blue bounded box in Fig. 2.1 represents this intermediate phase of processing. In Fig. 2.4, the midpoints extracted from the candidate are marked on the corresponding grayscale image.

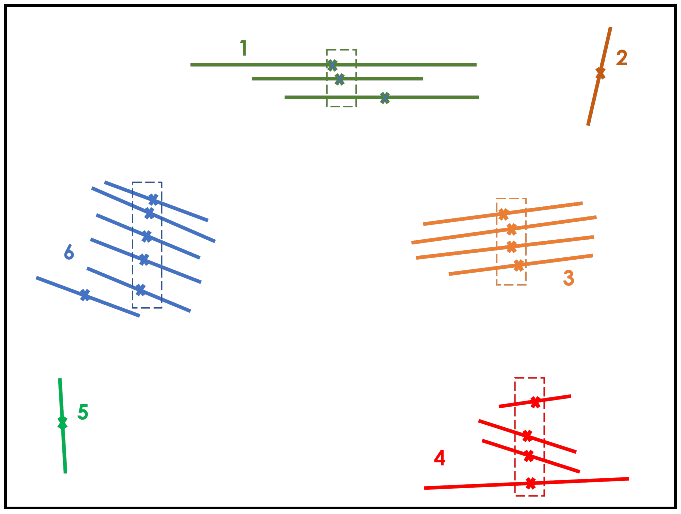

Figure 2.3: Illustration of Straight Line grouping.

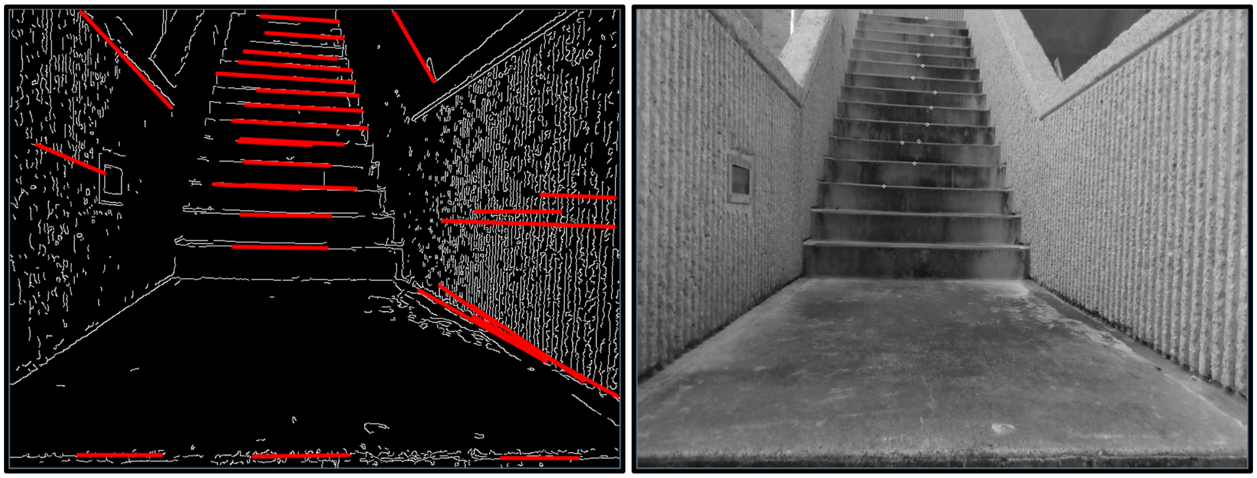

Figure 2.4: Detection of Candidates.

Conclusion phase

To detect a staircase among other structures within the image that contain a parallel grouping of lines, the contextual depth information is used to classify structures as stair-like or otherwise. This classification (as seen in the green bounded box in Fig. 2.1) adopts a Support Vector Machine (SVM) classifier that is trained in advance and can be employed in real-time. This supervised learning technique involves dividing classes of labeled data using hyperplanes (a plane in x-dimensional space), i.e. a mathematical grouping of the labeled data is accomplished. The nature of such a mathematical problem is one of optimization. The goal is to fit the best hyperplane for the given data such that maximum accuracy of classification is achieved. An illustration of this problem is seen in Fig. 2.5. In this 2-dimensional example, there are two classes of data, represented by knots and crosses. Multiple hyperplanes can be generated for a given set of training data (represented by the blue lines). However, the optimal hyperplane (shown in red) is the one that maximizes the width of the separation (termed Margin) between the two classes.

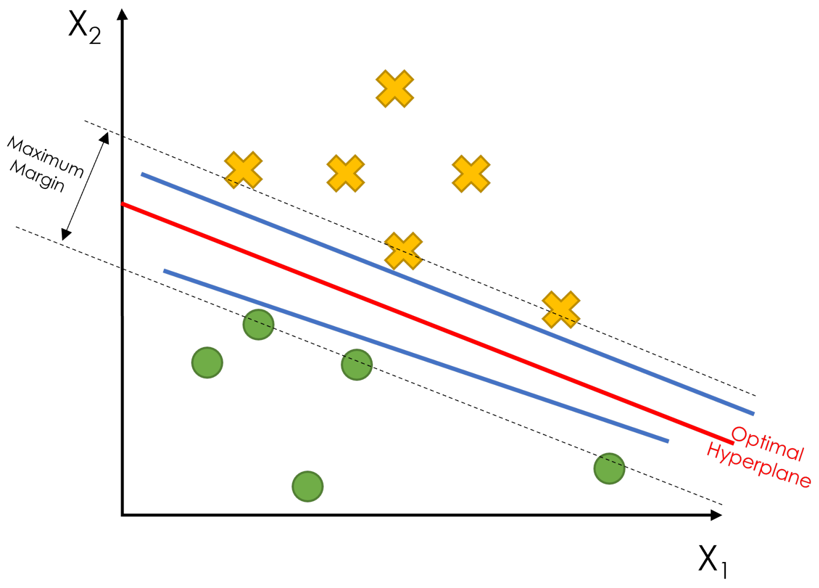

Figure 2.5: Illustration of Hyperplanes [Reproduced from [34]].

The mathematical formulation of the SVM problem is as follows [34]. In an n-dimensional space, a hyperplane is represented as:

$$f(x) = \beta_0 + \beta^T x \tag{2.6}$$

where β_0 is called the Bias, β is the Weight Vector, and x represents the training data or support vectors. These values can be scaled to generate an infinite number of representations of the same hyperplane. But the conventionally employed representation, known as the Canonical Form is:

$$|\beta_0 + \beta^T x| = 1 \tag{2.7}$$

The distance (d) between a point x and the hyperplane (β, β_0) can be expressed using the Euclidean relationship. Factoring in the canonical form of the hyperplane:

$$d = \frac{|\beta_0 + \beta^T x|}{||\beta||} = \frac{1}{||\beta||} \tag{2.8}$$

For the optimal hyperplane, the gap or the margin (M) is twice the distance between the point x and the hyperplane. Therefore,

$$M = 2 * d = \frac{2}{||\beta||} \tag{2.9}$$

In other words, the problem of obtaining the optimal hyperplane which is equivalent to maximizing M, can be expressed as a problem of minimizing a function $L(\beta)$:

$$\min_{\beta,\beta_0} L(\beta) = \frac{1}{2}||\beta||^2 \;\; s.t. \; y_i(\beta^T x_i + \beta_0) \geq 1 \; \forall i \tag{2.10}$$

where y_i represents the label of the i^{th} class of the training data.

The pre-trained SVM classifier is based on the work presented by Munoz et al. [22] while the implementation has been handled differently. The researchers created a dataset constituted by images of staircases, escalators, and non-stair scenes is used to create a multi-class classifier. A vertical line is drawn through the middle of the image captured by the camera and the values of depth are extracted at every pixel, resulting in an x-dimensional classification space (where x is the height of the image employed). The assumption here is that the camera (that is handheld) would be located in front of the staircase, and the angle formed by the steps and the vertical line would be close to 0 degree.

For the mobile robots and small-size UGVs, however, it is necessary to recognize the presence of a stairway from acute-angled points of view and from low vantage points. It is anticipated that many other structures would be present in the image frame. Therefore, for the training and testing of the classifier, the location from which the depth information is extracted becomes crucial. From each grouping of the previous phase, the raw depth value is extracted at the midpoint of the line - this is in essence the distance of that point from the camera in the field of view of the robot. This is scaled with respect to the maximum value (a threshold set arbitrarily, beyond which reliable depth information cannot be obtained) and stored as a vector of depth data. In this context, a vector refers to an array of scaled values that has passed the criteria for being grouped as a potential staircase candidate. The upper limit on the size of this array was set arbitrarily at eight, with the reasoning that this would contain sufficient information to validate the presence of a stairway; any addition would only increase the dimensionality of the SVM space and not add much value to the classification. A number of depth vectors were extracted from training images constituting stairways and other indoor and outdoor structures, and were used to train two binary SVM classifiers employing the LIBSVM library [35]. The classifiers were based in 8-dimensional space, and the dividing hyperplanes were generated based on the pattern of variation in the depth data.

The first model was trained using a labeled dataset of positives, i.e. depth vectors extracted from image of staircases, and the second using a dataset of negatives, i.e. depth vectors not pertaining to staircases. From the output of the Detection phase, the locations of the midpoints of the lines constituting each grouping is used to extract the raw depth value from the depth image. This was possible owing to the fact that the images employed were of identical dimensions and aspect ratio. The raw values of depth are sorted by location of the midpoints and scaled, before being fed as an input to the pre-trained models. If the output of the classification from both are in agreement, the candidate grouping is labeled a staircase. This can be seen in Fig. 2.6, where only the grouping representing the stairway is present (shown in black). The algorithm was implemented using the *roscpp* wrapper which is available under Appendix B.1.

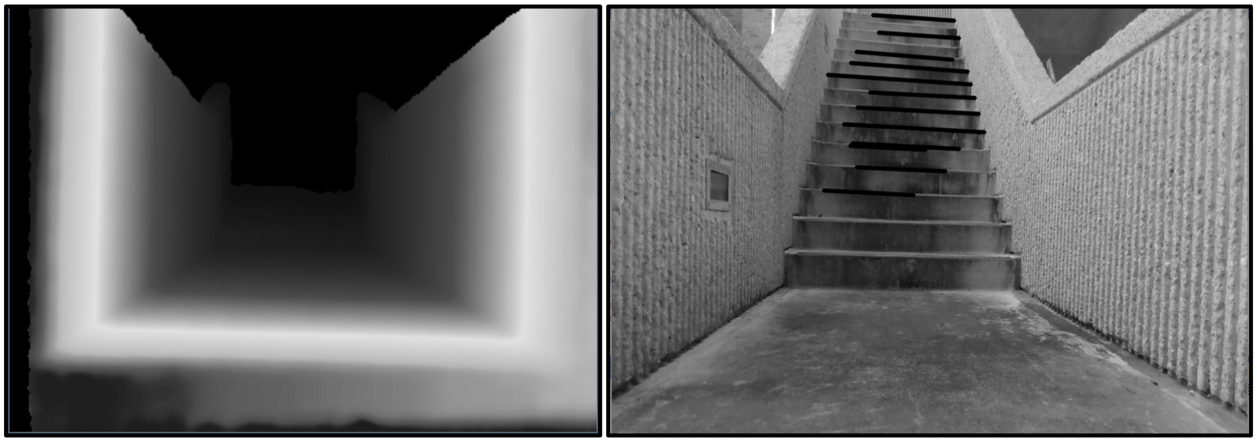
Figure 2.6: [**L**] Depth Image, [**R**] labeled result.

2.2 Semantic Segmentation and Scene Parsing for Terrain Classification

In addition to the stair detection capability, extended capability of subjective understanding of the robot's surroundings is desired to fully supplement α-WaLTR's hybrid locomotion capabilities. In other words, there is a need to classify the robot's operating terrain into more complex groupings that will account for the nature of the terrain and the mode of locomotion of the robot. The objective of this part of the research is to employ a scheme that produces a segmented output of the environment seen by the robot. It is noted that the algorithm should aim to classify a minimal, but sufficient, number of terrain classes for optimal embedded applications. This can then be applied for path planning and obstacle avoidance and traversal routines.

2.2.1 Related work

Work in the domains of navigation and terrain identification has been a facet of autonomous vehicle vision system research. Many varied techniques have been proposed, including employing visual odometry in the absence of GNSS localization [36], detecting the vanishing point to identify the roadway [37], and employing morphological processing to identify unstructured pathways from 2D images [38]. However, with the advancement in sensor hardware and processing techniques, a much more popular route in both autonomous vehicles and ground robot research has been to employ computer vision to semantically segment the operating environment. This is an avenue of research that is both established and expanding, as it aims to group features in the environment in known clusters, thereby making it viable for navigation planning.

A simpler problem was explored to segment the operating environment into lanes employing the edges and vanishing points [39]. A Deep Learning technique was also proposed to detect pedestrian lanes especially when unmarked, by segment the environment into lanes and backgrounds [40]. Urban semantic segmentation has also been explored using only data from a Monocular camera to identify drivable lanes [41]. With the goal of reducing vehicular casualties, a CNN-based architecture was developed to assist drivers on the roads [42]. A novel approach to the problem of semantic mapping in mobile robots attached visual cues to a ConvNet model, thereby enabling the robot to make subjective decisions [43]. Another popular CNN framework is PSPNet that has been been employed in off-road and unstructured environments [44]. A promising new framework GANav for unstructured outdoor environments was presented and validated in [45]. As with most technological advances, this has also been used in accessibility enhancement. Semantic segmentation has been employed in developing a device to assist the visually-impaired in walking outdoors [46]. Semantic mapping has been used with non-image data as well. The use of the Mask-RCNN architecture has been applied to segment point clouds and connect the results with a real-time SLAM algorithm for indoor mobile robots [47]. In a similar vein, the RFNet architecture has been adopted to segment multimodal RGB-D data in an outdoor environment [48].

2.2.2 Gated-shape CNN

The RGB data from the RELLIS-3D dataset [49] was chosen as the benchmark as it presents a balance between structured and unstructured environments that α-WaLTR can be expected to operate in. Although it is presented as targeting off-road autonomous driving applications, the focus on multi-modal data collection makes it of interest for our work as well. The RGB image data consists of five sequences recorded on the RELLIS campus of Texas A&M University, College Station, Texas. The corresponding ground truth images are pixel-wise annotated, and the labels include 20 different visual classes.

To benchmark the data, two state-of-the-art deep learning architectures were adopted: a) High-Resolution Network with Object Contextual Representation (HRNet [50] + OCR [51]) and b) Gated-shape Convolutional Neural Networks (GSCNN) [52]. For this thesis the latter is chosen as the model to further improve upon. This choice is based on qualitative measures of accuracy in preliminary tests involving the two. As the performance of the GSCNN model was observed to be better with images outside of the presented dataset, it was selected for further work. The GSCNN architecture consists of a two stream network [52]. The conventional approach to image segmentation is to process all the data from the image together in one neural network. The GSCNN architecture processes the information about the shape extracted from the raw image in a separate stream, whilst simultaneously making use of a feed-forward CNN as a classical stream. The outputs of the two (dense pixel features and semantic boundaries) are fused so that the contextual information is preserved. The final output merges the boundary information with the region features, so that a refined semantically segmented image is obtained. This is represented in Fig. 2.7. An illustration of how the GSCNN model trained on the RELLIS-3D dataset works is shown in Fig. 2.8.

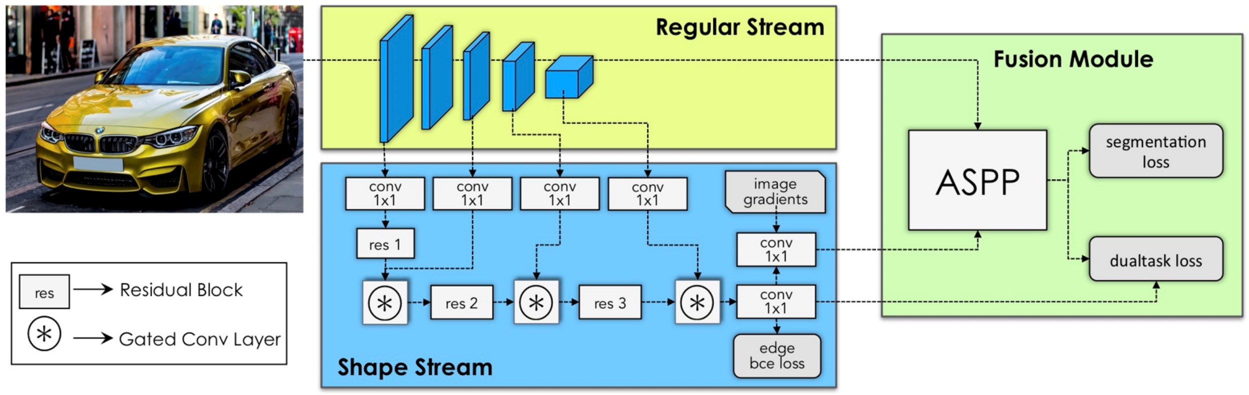

Figure 2.7: GSCNN Architecture. [Reprinted from [52]]

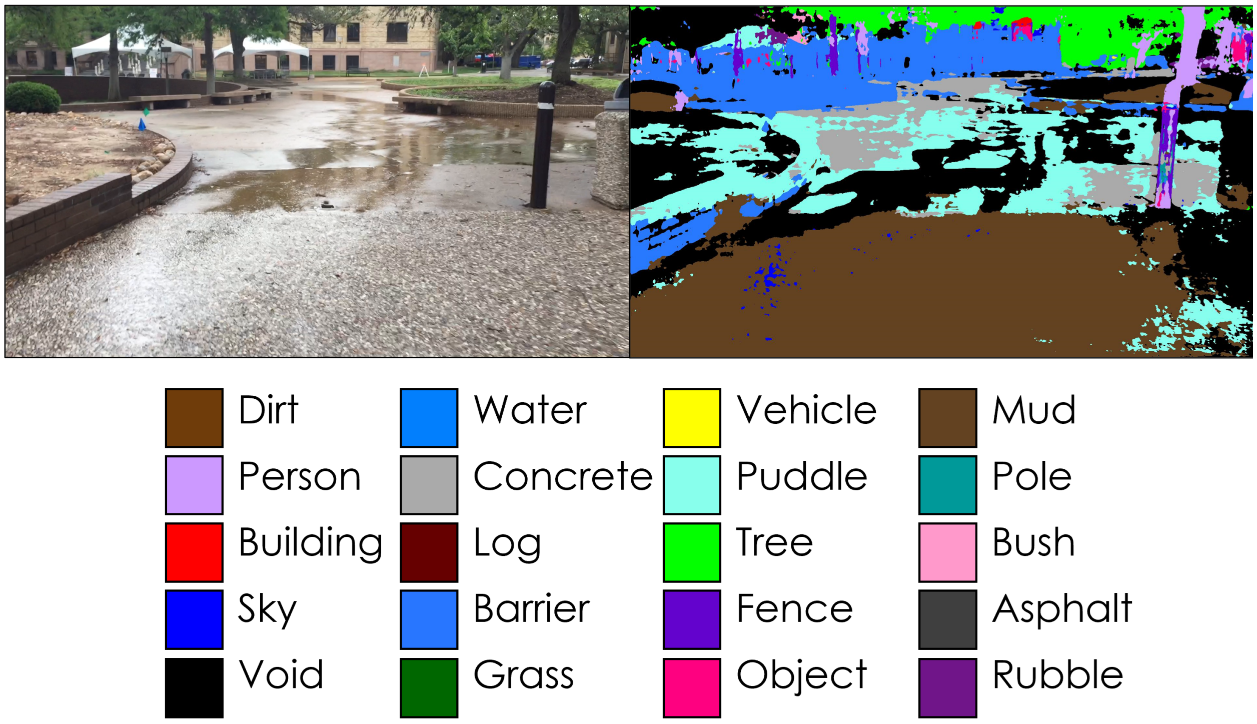

Figure 2.8: Illustration of scene segmentation: [**L**] Input frame, [**R**] Segmented image. [Color scheme adapted with permission from [49]]

2.2.3 Class merging approach

While semantic segmentation and scene parsing for visual images may involve many classes for general understanding of the environment and surroundings for various applications, this level of detailed classification is neither necessary nor computationally suitable for many embedded robotics applications. In particular, if the purpose of the classification is the autonomous navigation, the robot needs to classify traversable and non-traversable terrains while moving towards the target location.

Considering the capabilities of α-WaLTR, a scheme to merge the classes into a smaller number of classes is proposed. For instance, the concrete and asphalt classes both represent traversable terrains, and the pole and fence classes both represent obstacles for the robot. In addition, it is expected that class merging would make it possible to have a more computationally efficient approach. With these factors in mind, the following merged classes are proposed:

(1) Level terrain that can be traversed by the platform: Dirt, Grass, Asphalt, Mud, Concrete

(2) Obstacles that should be re-routed around - Vehicle, Object, Pole, Log, Person, Barrier, Puddle, Bush

(3) Larger obstructions that cannot be traversed or re-routed around - Building, Rubble, Fence, Water, Tree

(4) Sky - cannot be grouped under any other category

(5) Void - if a region cannot be labeled as any of the above

3. CONFIGURATION OF THE MOBILE ROBOT PLATFORM

This chapter delves into detail about the α-WaLTR platform's hardware and software architecture.

3.1 Description of Hardware Platform

α-WaLTR is the hardware testbed platform, capable of multi-terrain navigation. It is equipped with four wheel-leg transformable mechanisms to support enhanced locomotion on rough surfaces, low obstacles, as well as staircases. This locomotion mechanism design is based on the early WheeLeR proof-of-concept proposed in [8]. This prototype, pictured in Fig. 3.1, was approximately $17 \times 12 \times 3cm^3$ in size, and weighing about 500g.

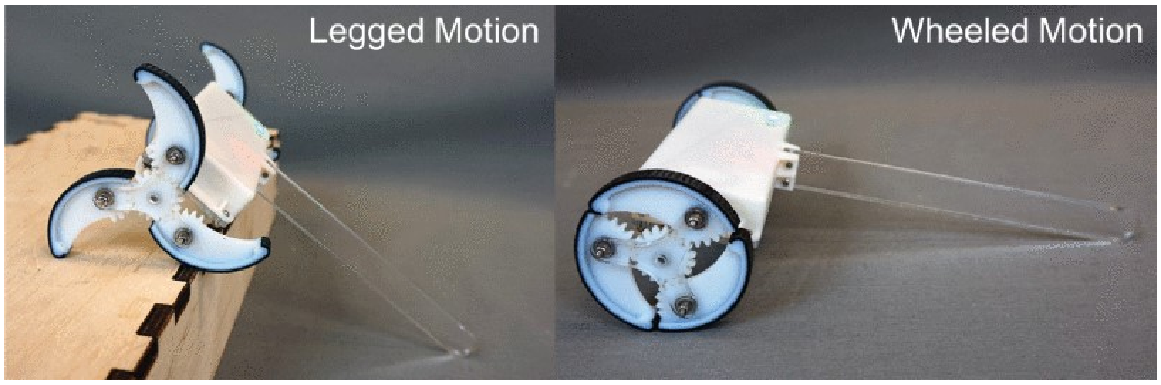

Figure 3.1: WheeLeR prototype [Reprinted from [8]].

3.1.1 Hardware overview

In order to employ this mechanism in a significantly larger ground robot, with dimensions on par with commercially available platforms like the AION-R1 [53], many improvements were necessary. The hardware platform was redesigned by a team of researchers in the ART lab at Texas A&M University under the support of DARPA Contract No. HR00112020037. This work involved multiple phases - a) scaling up the wheel-leg transformable mechanism and optimizing the design variables to account for environmental factors like friction; b) modifying the chassis dimensions to be capable of carrying a significant payload; c) optimizing the layout of the constituent components thereby enabling the platform to undergo changes in orientation in a stable manner. Further discussion of the aforementioned phase of work is beyond the scope of this thesis. Following this, work on constructing and testing the α-WaLTR platform for locomotion capabilities began.

The scaled-up UGV platform (shown in Fig. 3.2) is constructed using a carbon fiber base plate, with 3D-printed (PLA) walls. The assembly is completed by a lightweight acrylic cover. It can be equipped with n-legged wheels, thereby exhibiting a modular design ($n = 3, 4, 6$). The wheels themselves are furnished with shock reducing suspension systems. Compared to the initial prototype, α-WaLTR weighs around 12 kg, and measures $63 \times 47 \times 20 cm^3$.

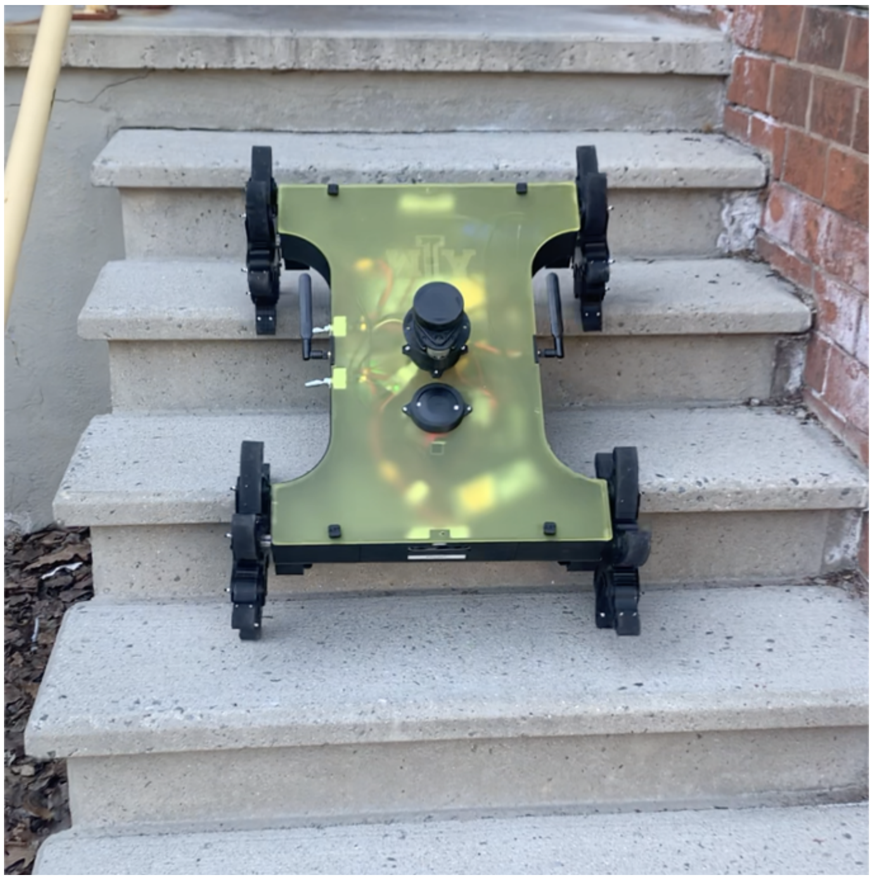

Figure 3.2: α-WaLTR platform.

α-WaLTR is capable of navigating across varied classes of challenging terrains. Equipped with wheels that can transform into legs, it retains the simplicity of wheeled navigation with fewer restrictions. The potential applications of this platform lie in urban military operations, such as search and reconnaissance. It is expected that platforms like these can be used to gain unmanned access to possibly hostile and/or inaccessible multi-storey environments, thereby enabling remote surveillance and rescue efforts.

3.1.2 Sensors employed

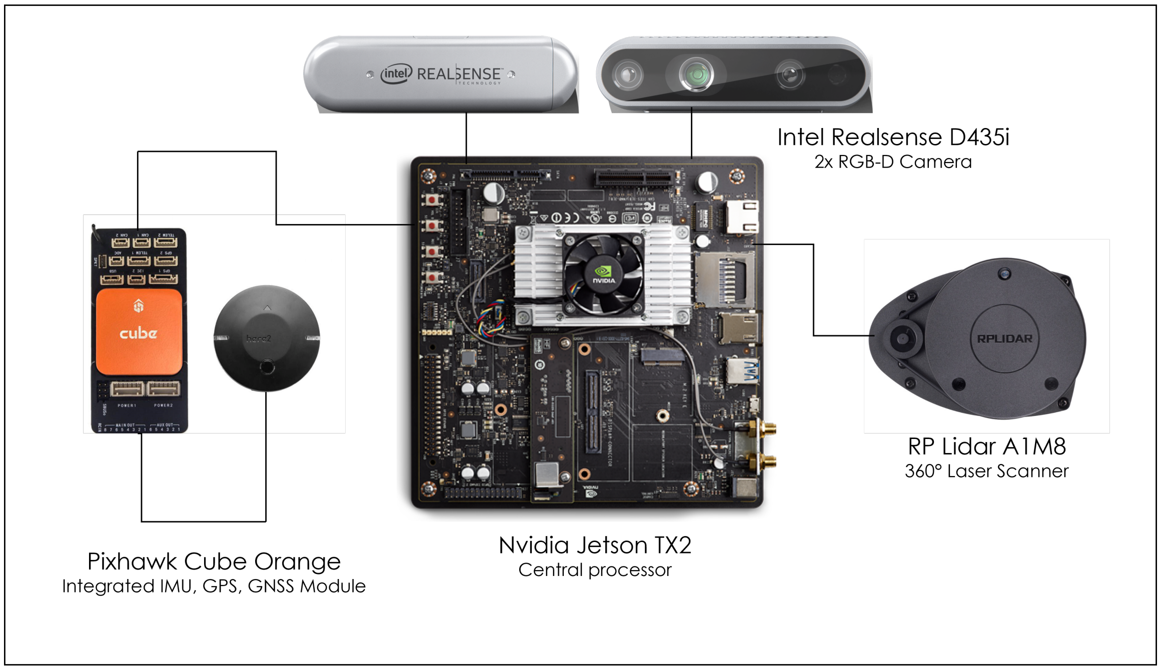

Figure 3.3: Layout of Sensors on α-WaLTR.

The α-WaLTR platform is designed to operate with a level of partial to complete autonomy. With the only user input being a global instruction, the robot is expected to negotiate obstacles and other environmental variations, by generating local instructions. This necessitates the need for sophisticated robotic perception. The platform is therefore equipped with a variety of sensors and high performance processing units to enable this. An Nvidia Jetson TX2 module is used as the main processor for all on-board computation. It is an embedded system equipped with a multi-core CPU and GPU which are capable of multi-threading. With 8GB of memory and 32GB of storage, this processor is central to autonomous operation. The embedded processor and sensor components are shown in Fig. 3.3.

The platform's vision system consists of two disparate sensors. The Intel Realsense RGB-D cameras are capable of streaming two channels (RGB/color and depth) of images in a variety of resolutions. They are stereo-depth cameras - consisting of two sensors that are placed a small distance apart. By comparing the inputs from the two, the camera's in-built software provides the user with depth information reliable upto 10m. They also contain an infrared projector, thereby not being dependent on external light sources [54]. These cameras also contain an inbuilt IMU, that can enable sensor fusion to eliminate external sources of noise. With one mounted on the front, and one on the rear end of the platform, these facilitate operation while moving in either direction (simultaneously or otherwise). The other vision sensor is a 360° lidar. This can be employed to obtain planar laser scans upto a reliable range of 12m. It allows for customization of the sampling rate and the resolution. In conjunction, the two sensors can be optimally employed to build a map of the operating environment and visualize the same in point cloud form.

The final component involved is an integrated IMU-GPS-GNSS flight controller termed the Orange Cube. Consisting of 3 sets of IMUs and an internal data filtering firmware, this can be employed reliably for inertial measurements. The GPS/GNSS Here3 module consists of a compass and magnetometer allowing for efficient localization. Compatible with various communication protocols (e.g. I2C, UART, CAN), it can be employed to communicate directly with the motion controllers for ground and aerial vehicles [55].

3.2 Software architecture

The software on the α-WaLTR platform is based completely on ROS [14] or ROS 1 to be precise. Although termed so, it is not an OS rather an open-source software framework that functions like one. It supports multi-node networking, has packages specifically written for it, and employs specific messages and communication methods. While it is designed to run on Unix-based OS, a limited number of operations (clients) can run on Windows.

The ROS architecture typically consists of two major components - the Master, and Nodes. Nodes are nothing but different processes running on the same network, tied together by the Master that references all the other nodes and enables communication between them. Typically, the Master is run on the Robot, but this can be changed at startup to a different computer on the same network. Nodes communicate by means of ROS Messages - these are data structures with certain defined fields comprising of commonly used datatypes. Each node sends out messages with a unique name termed as a Topic. A topic can be subscribed to (received) or published (sent out) by any node on the same network. These topics are not directed at any node in particular, rather they are sent into the metaphorical ether of the ROS network. Nodes can be user-defined or system-inherent, but they are in general modular, and responsible for a low-level task. A ROS network can be setup over any conventionally employed protocol, including wired or wireless Ethernet.

Every ROS node is setup or written using what is called a Client Library. These wrappers enable programmers to write scripts capable of creating new nodes and working with other nodes on the network. The two most common ones are the C++ and Python wrappers for ROS: *roscpp* and *rospy*. Scripts written employing these client libraries follow an object-oriented structure. The execution of a ROS node consists of two functions: *init()* which initializes the node and topics written by and to it; and *spin()*, which loops between function calls for continuous operation.

The ROS nodes employed on the α-WaLTR platform are centered around the navigation stack. As the primary goal is to achieve autonomous navigation based on a high-level user input, the *move_base* package was employed and it further explored in this thesis. This package, suitable for mobile ground robots, makes use of a global goal and attempts to reach it. It ties in various inputs from different layers and sends a movement command to the lower-level component in the system. An overview of the ROS Navigation Stack using *move_base* is illustrated in Fig. 3.4 [13].

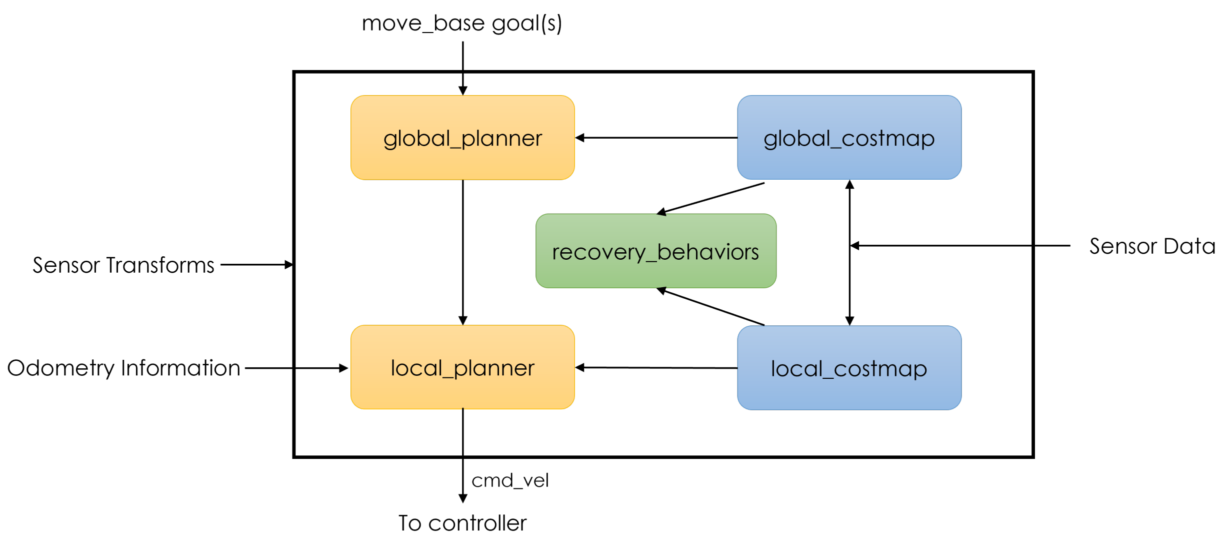

Figure 3.4: Illustration of ROS Navigation Stack architecture [Reproduced from [13]].

3.2.1 Planners

The Global and Local Planners work based on costmaps, which are occupancy grids representing the operating environment. For the α-WaLTR, the *costmap_2d::Costmap2DROS* object is used to maintain information about the state of the robot in real-time. The occupancy values are mapped from the inputs of the vision system which update as changes occur. The cost for each cell in the grid can take an integer value from 0 to 255. An occupied cell is assigned values above 128. The exact value is calculated based on how the obstacles present would interfere with the robot's footprint. Moving further from this cell, the occupancy value is determined using a discretized exponential decay. Based on an "inflation radius", a user-modifiable value for a buffer zone around the robot, the rate of this decay is determined. This process of propagating cost values based on distance from an obstacle is termed Inflation. In essence, what happens here is that the cells further away from the obstacle get assigned lower values. For cells beyond the inflation radius, a value of 0 (Freespace) is assigned. However, for cells about which there is no information (i.e. beyond the range of the sensors), an "Unknown" cost is assigned.

Fig. 3.5 shows an example of a scenario where three obstacles are placed around the robot growing increasingly distant. RViz is a visualization tool used to both subscribe to and publish topics pertaining to sensors. Here, the white box is the Local Costmap, where it is noted that there's an obstacle ahead of the robot and to the left. The other obstacles are out of the range for this local costmap. In addition, the inflation radius for the local costmap is significantly low that the cells away from the obstacle are quickly labeled as unoccupied. The green layer on which the local costmap is superimposed is the global costmap. Here we can see that the space occupied by the first and second closest obstacles are marked to indicate the same, however the third object is significantly far away to not be spotted by the sensors.

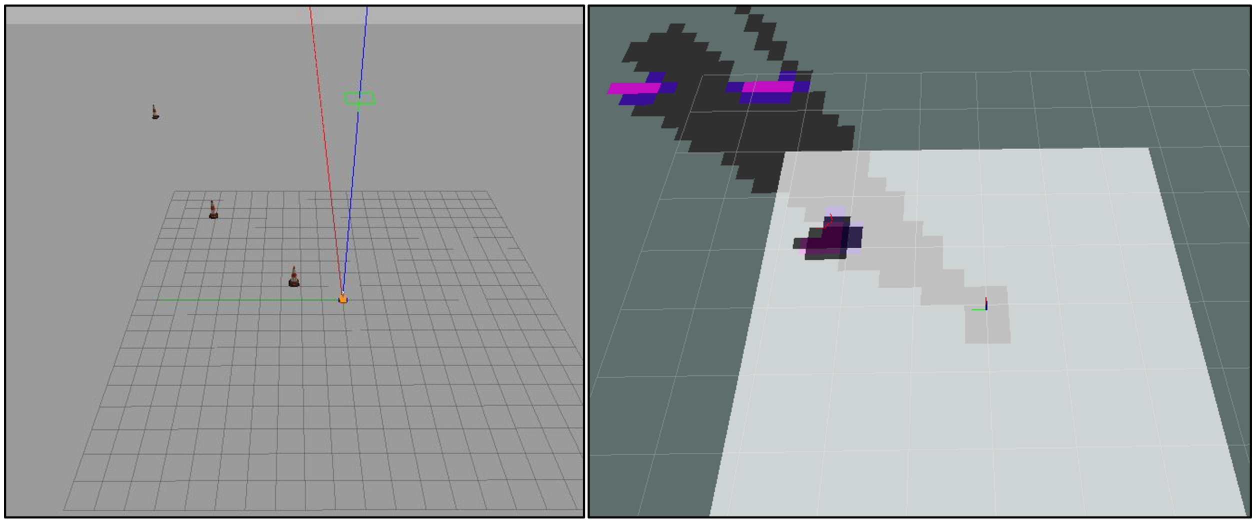

Figure 3.5: Costmap assignment vs. obstacle position. [**L**]: Gazebo simulation, [**R**]: RViz Costmap.

The global planner employed for the α-WaLTR is *navfn/NavfnROS*, that adheres to the interface settings under *nav_core::BaseGlobalPlanner*. Given a global position input, Navfn uses Dijkstra's Algorithm to search through the operating area to figure out the most optimal path. This starts from the Robot's origin (or position where the robot is located at the time when the goal position is input. The algorithm itself works by assigning a cost/distance value to the path between two cells.

By radiating outwards, each possible path is given a total cost (calculated using a cumulative sum of a quadratic approximation of the distance), with the one with the smallest cost labeled the actual path.

The output of the global plan is a path that the local planner can now employ. *base_local_planner::TrajectoryPlannerROS* used here serves to connect the path generated using Dijkstra's algorithm to the robot's kinematics. It makes use of a technique called Dynamic Window Approach (DWA) to determine a value for all the cells around the robot. The algorithm simulates a trajectory by sampling discrete values of $(d\dot{x}, d\dot{y}, d\dot{\theta})$ in the vicinity. It determines the robot's state if this velocity is applied for the simulation time dt. The result of this is a number of trajectories equal to the number of simulations. Each trajectory is evaluated based on its proximity to the global path, how far away from obstacles it is, and how close to the goal it would be. The one with the highest score (a weighted metric based on the above-mentioned parameters) is chosen as the "local path", and sent to the lower-level controller. The cost is negative if it that trajectory would result in a collision with an obstacle. This Local Trajectory Planner then sends the velocity values $(d\dot{x}, d\dot{y}, d\dot{\theta})$ to the base-controller, thereby advancing to the next state. This iterative process continues until the robot reaches the desired global goal, or if a trajectory cannot be found owing to special circumstances.

In Fig. 3.6, an example can be seen on RViz of how the paths generated change depending on the actual position. These were captured 1.5s (in terms of simulation time) apart. As illustrated, both the global and local paths are updated as the map updates. Another setting employed is to make the local costmap non-static i.e. it moves with the robot centered at its origin.

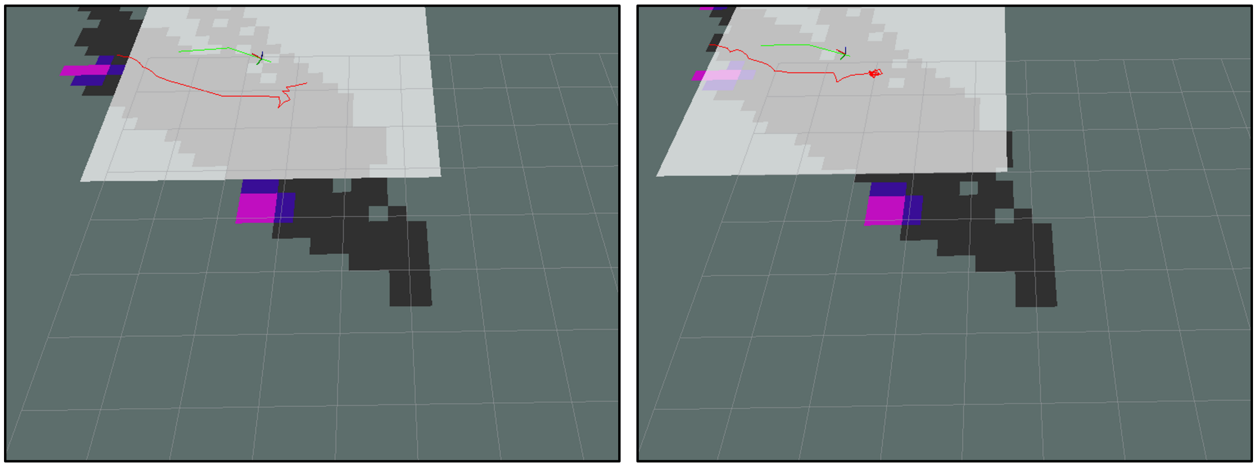

Figure 3.6: Global (Red) and Local (Green) trajectories at two different times.

3.2.2 Recovery Behaviors

One condition where the move_base fails to generate a local trajectory is when the robot is unable to extricate itself when it is perceived to be stuck between obstacles. These manuevers are termed Recovery Behaviors, and when enabled, the sequence adopted is as follows.

1. The robot is unable to proceed with generating a local trajectory - labeled stuck. Step 2.
2. Obstacles outside of a set zone are cleared from costmap. Still stuck? Step 3.
3. The robot uses the rotate_recovery::RotateRecovery object and rotates to clear out obstacles from the costmap. Still stuck? Step 4.
4. All obstacles outside of rotation footprint are cleared, and rotate recovery is re-attempted. Still stuck? Goal is infeasible.

These cases usually occur when there are unusually challenging dynamic obstacles present in the operating environment, or if the set tolerance to achieve the goal is too tight.

3.2.3 Miscellaneous

Another requirement for the move_base package to work are the sensor transforms and odometry (denoted by the topic */tf*). The /tf topic is vital to transform the sensor inputs from their coordinate systems of reference to map information. This is the topic published by the *tf* package that keeps track of changing coordinate frames and makes the link between trajectories and the output command velocities. Usually, the velocity to the base controller is sent in the form of a */cmd_vel* topic, which is a data structure containing the X, Y, and rotational velocities for the robot's base. The *tf* package can also be used to publish the odometry information for the robot, this is employed by the Extended Kalman Filter (EKF) in localization and dead-reckoning. Other packages employed are specific to the sensors present on the robot. Some of them are created by the manufacturer and provide algorithms to filter the raw data, to modify sensor configuration, etc.

4. ALGORITHM EVALUATION AND IMPLEMENTATION

This chapter discusses the results of evaluating the algorithms and proposes schemes for integrating them on the α-WaLTR platform.

4.1 Evaluation of Staircase Detection

4.1.1 Methodology and results

The proposed staircase detection algorithm was trained on two separate datasets to generate two distinct SVM classifiers. The first one was trained using images employed by Munoz et al. [22], and other RGB-D datasets. This training set consisted exclusively of image frames obtained from a human eye point of view. i.e. shot with a camera held at chest height. The second classifier was trained employing images shot on-campus by ART (Adaptive Robotics and Technology) Lab members and obtained from publicly available datasets (as detailed in Appendix A). This included images from a robot's eye view. The difference between the two is illustrated in Fig. 4.1.

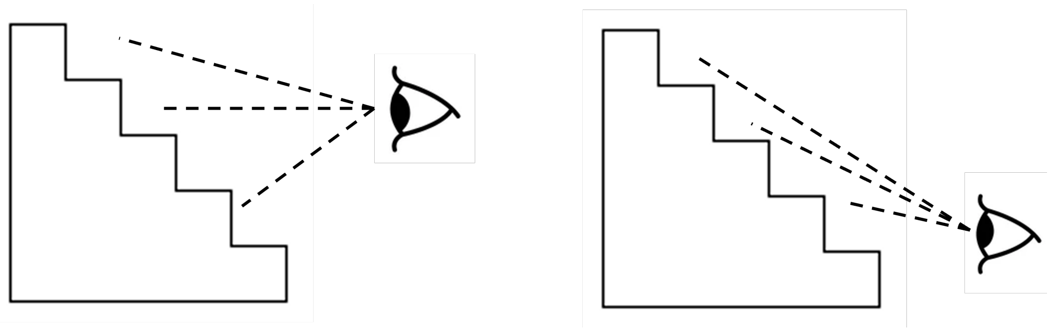

Figure 4.1: Illustration of difference between models. [**L**]: Model 1, [**R**]: Model 2

An example of the two different vantage points employed is shown in Fig. 4.2. The major difference observed is in the pattern of variation of the depth values extracted from the images. For the likes of the image on the left, the pattern follows a parabolic trend, whereas for the image on the right, it is linear in nature. A total of around 250 support vectors were generated in both cases.

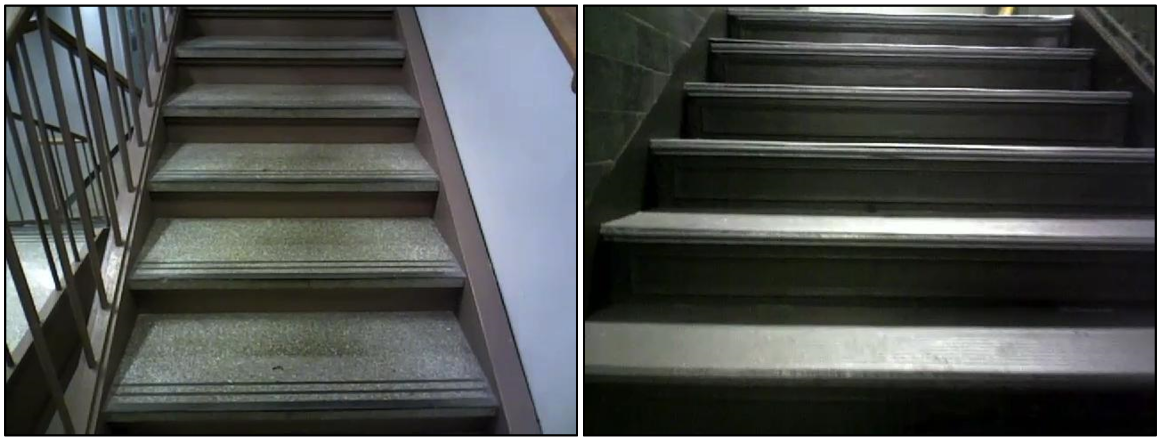

Figure 4.2: Images shot from [**L**]: Human eye view, [**R**]: Robot eye view

The test set was compiled from pictures shot on-campus and images available online, which were not included in the training. The results of this are summed up in Table 4.1. The results are reported using the specificity (proportion of images labeled true negatives among the total negative images) and sensitivity (proportion of images labeled true positives amongst the positive frames) metrics. These are preferred to the base accuracy metric (ratio of positive classifications to the total number of frames), as it paints a better picture of how effective the classification is.

	Frames Present	**Sensitivity (%)**		**Specificity (%)**	
		Model 1	**Model 2**	**Model 1**	**Model 2**
Test set 1 [22]	177	92.05	98.68	68.42	47.37
Test set 2 [56]	133	93.94	98.11	53.13	41.67
Test set 3 (TAMU Campus)	50	100	100	63.64	33.33

Table 4.1: Accuracy of staircase detection algorithm.

The model trained on the second training set (i.e., from the robot's eye view) showed a higher sensitivity than the other model while its specificity is lower. The trend was consistent among all three datasets. Model 2 could detect staircases better than Model 1 but was not as good at labeling non-stair images. In practice, this can be improved by having a balance of positive and negative images in training. As the non-stair class of images are quite similar in appearance, there might be a tendency to be biased in favor of the positives (staircases having a lot of variability in appearance). In addition, the data quality can be varied by changing the lighting conditions or image resolution.

The processing speed of this algorithm was about 4-5 frames per second (fps) on average when running on-board the α-WaLTR platform (i.e., on the Jetson TX2). Some outliers in the processing time were observed – the first few frames seemingly taking up initialization time. The majority of the computational load was down to the two feature-detection operations – the Canny Detector and the Hough line transform. Based on the observed performance, the input frame rate was set in order not to miss frames in real-time. The ROS package was configured to store incoming images in a buffer to enable FiFo frame processing. In comparison, the algorithm proposed by Munoz et al. [22] was capable of running at 2 fps, albeit on a 2.9GHz processor without GPU acceleration.

The work presented in [33] employed YOLO and reported a faster processing rate, at around 20-25 fps on an Nvidia Jetson TX1. When tested on an i7 dual-core computer, a disparity-based depth processing algorithm [27] was able to process at around 15 fps. The interpretation is that better performance is observed from higher capacity processors (as expected). However, all the tests run on the α-WaLTR platform were done in conjunction with other ROS nodes running. In other words, the performance observed here can be expected in real-time operation and not just in laboratory conditions (tested separately). In terms of accuracy, Munoz et al. [22] reported a 96% sensitivity (on par with the proposed algorithm) and a 93% specificity (much higher than observed here). In contrast, the deep-learning model [33] performed much poorer, with a reported 78% sensitivity. This is likely related to the relatively small dataset, being insufficient for training a reliable model using a deep-learning based method.

4.1.2 Potential improvements

The supervised learning technique of SVM is a popular Machine Learning tool that can be applied to a variety of vision-based applications. It supports large dimensional spaces and can be trained and employed in classifying multiple data categories. Another application it supports is regression analysis, i.e., to find trends in data. However, one downside of this is that the increase in dimensionality makes the classifier more abstract and hard to visualize. Therefore the objective in improving this particular application is to identify a method of processing the depth data such that it can be represented in n-dimensional space ($n \leq 3$). This would mean that the model is trained not on actual depth extracted from the image, but rather on inferred data. Other supervised learning techniques involving Deep Learning can also be explored. This approach may result in higher accuracy, while the resulting models would be computationally heavy to deploy with an even lower ability to visualize. SVMs avoid the “black box" nature of other supervised learning techniques and can be employed profitably if the data is processed appropriately. The choice of employing SVM was down mainly to two reasons: a) it can be implemented in real-time on board processing (as evidenced by the comparable performance with other algorithms); b) retraining is fairly easy and can be implemented reasonably quickly.

4.2 Evaluation of Semantic Scene Segmentation

4.2.1 Network tests

For efficient real-time implementation, two strategies were considered: 1) processing the images on-board, and 2) transmitting the images to an off-board hub and sending the processed information back to the robot. A set of extensive experiments were conducted to evaluate these two approaches. The first was to compare the processing speed on-board and off-board. The time to process the image frames was the metric used for comparison. The images were input at different resolutions to identify the optimal quality of images that could be employed.

The results of the experiment are summarized in Table 4.2, with four time measurements reported for each size range. When a series of frames is input to the model, the behavior observed was a steady decrease in processing time from the first frame resulting in a stabilization around the 30-40% mark of the sample size. This is labeled as ‘Stable’ in Table 4.2. This higher time at the start is accounted for by the initialization of the model and the variables associated with it. The subsequent frames exhibit lower processing times as re-initialization of certain variables is unnecessary. Three size ranges were employed for this experiment - this labeling and selection of sizes is based on the maximum frame size that can be processed by the on-board processor (Jetson TX2). It was observed that an image frame greater than 1500 $\times$ 1500 pixels cannot be processed. So this was set as the maximum possible frame size. The first range (high resolution) consisted of images ranging between frames sizes of 1000-1500 pixels along both dimensions; the second range (medium resolution) - 500-1000 pixels; and the third (low resolution) - below 500 pixels. In summary, the semantic segmentation is more efficient on the higher performance computing device as expected.

	High Resolution				**Medium Resolution**				**Low Resolution**			
Frame	1	2	5	Stable	1	2	5	Stable	1	2	5	Stable
Off-board PC	44.9	34	17.3	8.4	16.0	11.8	5.5	2.2	12.0	8.6	3.4	1.5
On-board Jetson TX2	75.7	47.4	29.9	41.3	29.8	17.9	10.5	7.0	13.9	8.0	4.5	2.8

Table 4.2: Average time (in seconds) to process a frame.

As off-board processing was observed to be more efficient, the next experiment focused on quantifying how expensive the image transmission over a network would be. The ROS wrapper of the Realsense camera API supports image compression for this very purpose. Rather than sending and receiving a large image file (which could dominate the communication over a weak network), a compressed image topic consists of the relatively lightly encoded matrix. This can be decoded at the receiving node, as the standard JPEG compression protocol is followed. Therefore, raw and compressed images were sent to another node (a PC) on the same ROS network. The configuration employed to send the real-time image stream over the ROS network, consisted of three chief elements: the robot (moving), and a network router, and the PC for receiving images (both fixed with respect to the robot). The commercially available TP-Link AX6000 router was chosen for this purpose. It enables high-speed wireless communication over both 2.4GHz and 5 GHz frequency bands. Employing the router, a wireless Local Area Network (WLAN) was setup. With the robot (assigned as the ROS Master) and the PC connected to it, this also served as the local ROS network. Measurements were made at regular intervals of distance. To the best possible extent, environmental disturbances were maintained as they would be during field exercises.

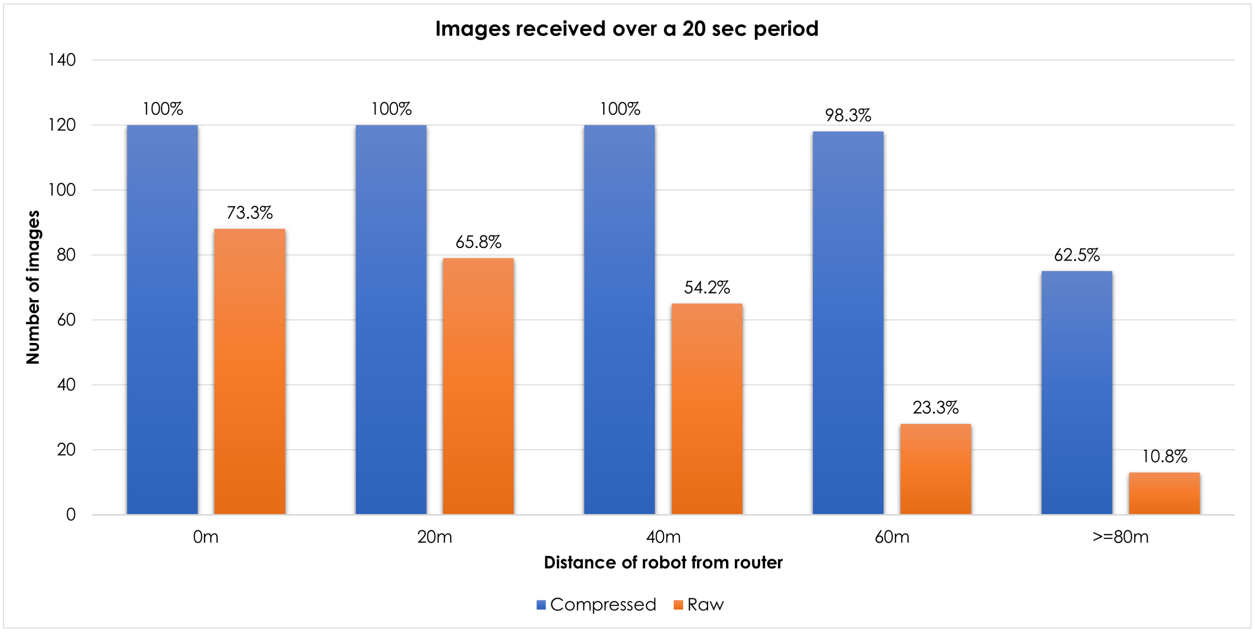

Figure 4.3: Data reception vs. Distance.

Fig. 4.3 presents the trend in data reception as the distance of the robot (x-axis in meter) from the router varies. For a 20s period, 120 images were 'published' (or sent) from the robot's camera. Almost all compressed images were received successfully until the 60m mark. After this point, a significant number of frames were lost in transmission. The raw image frames peak at a maximum of around 74% and steadily decrease with distance, and also show a significant drop after 60m.

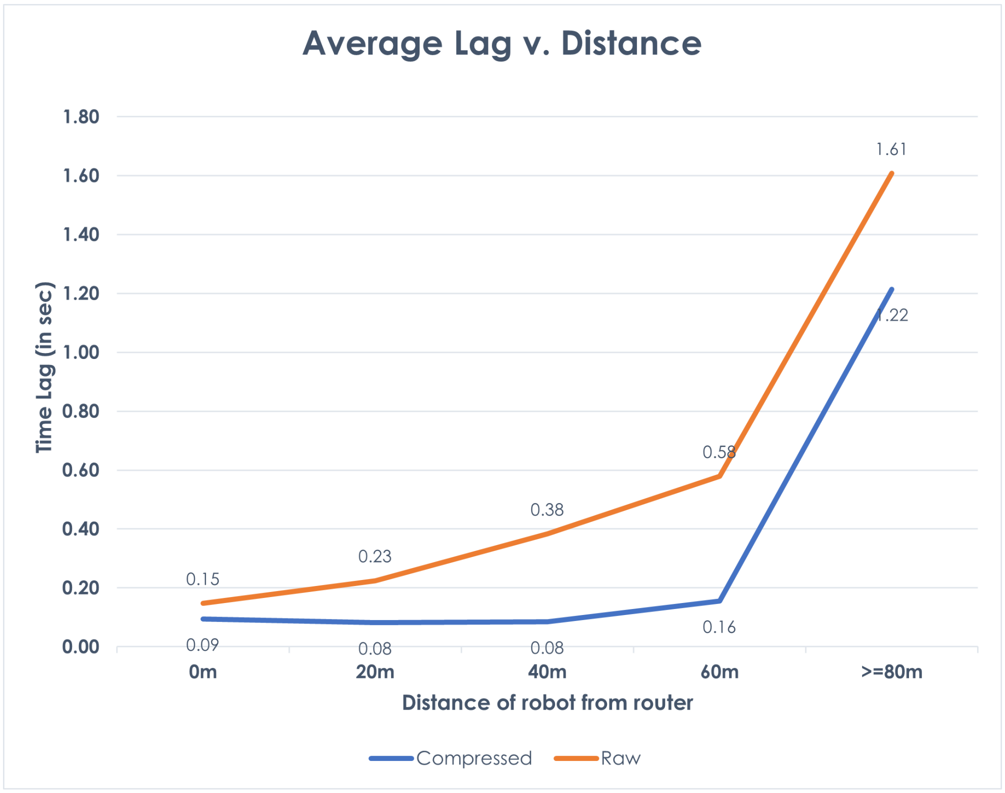

Figure 4.4: The trend observed in time lag for data reception.

Fig. 4.4 shows that the average lag in the reception of the compressed or raw images ranges between 0.1 and 1.5s. This is relatively low compared to the difference in processing time between the on-board and off-board options. Therefore, from these experiments, it was concluded that the off-board processing scheme is a feasible alternative to running the algorithm on-board. In addition, this would result in an effective results without interfering with other processes on the robot.

As a final check on quality of the images (as JPEG is a lossy compression), the compressed and raw images frames obtained from the robot at the same instant were processed with the segmentation model. The resulting images (Fig. 4.5) were observed to be similar. Significant portions of this section of research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing. The ROS client employed for this section is available for reference under Appendix B.2.

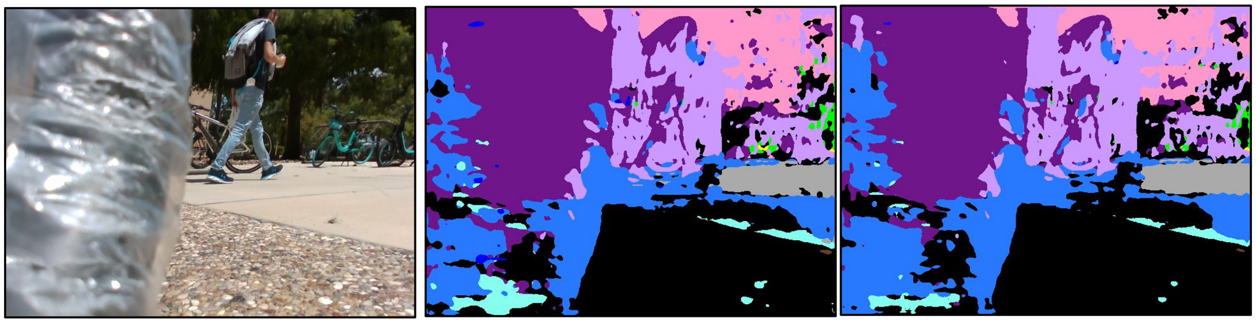
Figure 4.5: Comparison of segmentation in raw and compressed images: [**L**] Input RGB image, [**C**] Segmentation of Raw image, [**R**] Segmentation of compressed image.

4.2.2 Results of class merging

The retraining of the GSCNN model was done employing the training set of RGB images in the RELLIS-3D dataset [49]. As mentioned in Chapter 2, the class merging involved going from 20 terrain classes to 5 that were applicable to the α-WaLTR platform. This is illustrated in Fig. 4.6.

🟫 Dirt	🟦 Water	🟨 Vehicle	🟫 Mud
🟪 Person	⬜ Concrete	🟦 Puddle	🟦 Pole
🟥 Building	🟥 Log	🟩 Tree	🟪 Bush
🟦 Sky	🟦 Barrier	🟪 Fence	⬛ Asphalt
⬛ Void	🟩 Grass	🟥 Object	🟪 Rubble

(a) Terrain classes in RELLIS-3D [Adapted with permission from [49]].

🟫 Traversable Terrain
🟦 Obstacles
🟪 Large Obstruction
🟦 Sky
⬛ Void

(b) Merged classes.

Figure 4.6: Semantic Segmentation Color Scheme.

Following this, various images were used to compile a test set to validate the re-training of the GSCNN. Sample results of the merged class test are as shown in Figs. 4.7, 4.8, 4.9.

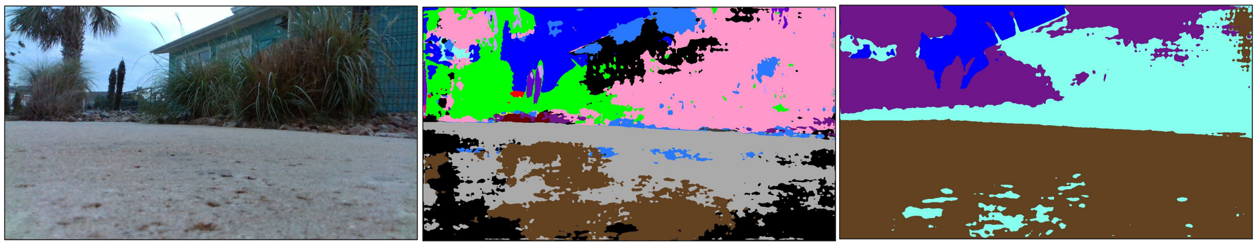

Figure 4.7: [**L**] Input RGB image, [**C**] Original Segmentation, [**R**] Merged Segmentation.

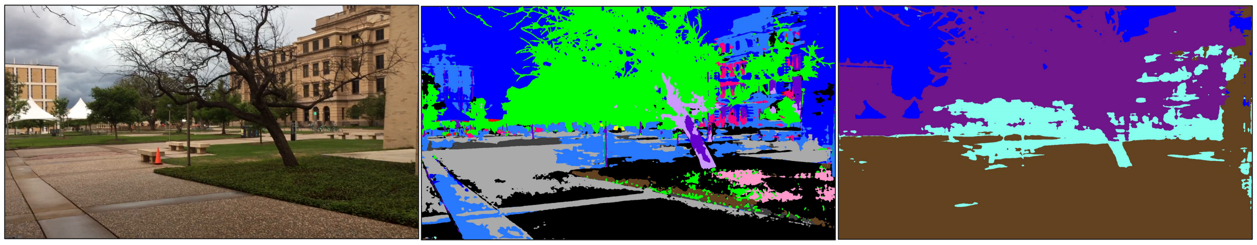

Figure 4.8: [**L**] Input RGB image, [**C**] Original Segmentation, [**R**] Merged Segmentation.

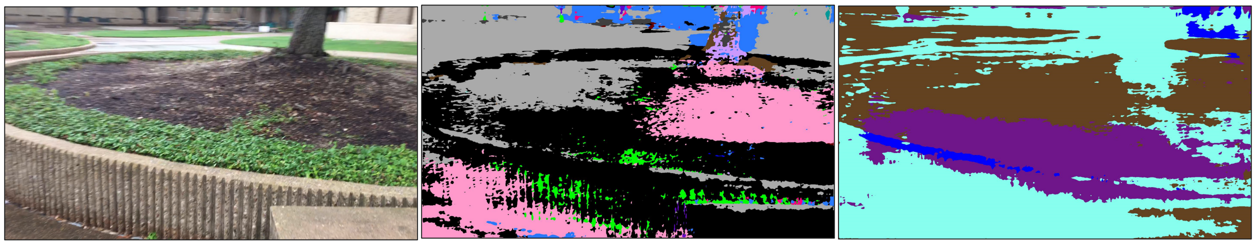

Figure 4.9: [**L**] Input RGB image, [**C**] Original Segmentation, [**R**] Merged Segmentation.

These test images were evaluated subjectively, as comparing them to the original ground truths provided with the dataset would not be beneficial. In Fig. 4.7, the lack of contextual depth information in the model is evident. Obstacles closer are labeled correctly, however, some further away become part of background large obstruction class. Similarly, in Fig. 4.8, the trees in the foreground and background are labeled different. In Fig. 4.9, the labeling is more accurate than the original segmentation. The raised area around the tree is labeled as an obstruction in the latter, but as a mix of classes in the former. One other thing that stands out from all the three is how level terrain is detected much more clearly than before and labeled so. The primary objective of this exercise was to identify negotiable terrain the robot's environment. This has been accomplished to a considerable extent. Further work on this can involve adding contextual depth to improve the accuracy of classification and conversion from a 2D image to an input to the navigation stack.

4.3 Adaptive Path Planning

To integrate the image segmentation and the navigation stack of the robot, a decision-making scheme is presented to supplement α-WaLTR's multi-modal capabilities. As illustrated in Fig. 3.4, the navigation stack consists of a global and a local trajectory planner that generates paths for the robot to follow. By default, the behavior of these planners is not subject to terrain type and conditions. This is ideal for mobile robots that are expected to operate in environments without much variability. However, to take advantage of α-WaLTR's enhanced locomotion abilities, it would be beneficial if the robot were to adapt not just mechanically, but also from a software side, to its operational environment. The goal of this phase of work is to modify the existing path planning framework to make it sensitive to changes in the terrain.

4.3.1 Reconfigurable parameter setting

The navigation stack operates on a set of modifiable configuration settings for each layer, with parameters input in the form of YAML (.yaml) files. The structure of these files supports the nesting and communication of all the common data types. The configuration is imported by the ROS node at start-up and is generally associated with a launch file. Because of the nature of this, the configuration is generally static and not modifiable in operation. For this purpose, a separate ROS package *dynamic_reconfigure* is employed. As with most other ROS packages, it is compatible with *roscpp* and *rospy* clients, and supports external reconfiguration of a subset of the node's parameters at runtime without necessitating a restart. It can be executed from the command line using the following syntax:

```
$ rosrun dynamic_reconfigure dynparam <COMMAND>
```

The user can generate a list of configurable nodes, pull up the current configuration, modify parameters, save/load to file, etc. For this work, a Python script (Appendix B.4) was employed to switch between a pre-determined set of parameters, depending on sensor input.

To validate the hypothesis that an adaptive path planning scheme would be advantageous, experiments were run in simulation. The parameters of interest here pertain to the smoothness (or vice versa, Granularity) of the local path planning. As explained in Chapter 3, the number of trajectories simulated by the local path planner is determined by a set of discrete increments. By increasing the granularity settings, the size of each discrete step is increased. Therefore, the number of steps required to completely simulate the robot's vicinity is decreased. In other words, higher the granularity, less the time and computational effort in navigation. A coarser path (as explained in Fig. 4.10) would be ideal for negotiating terrains that are relatively obstacle-free, and make use of less computational resources. When the presence of challenging terrain is detected (say, from the segmented images), the path planning would be modified in real-time to account for this.

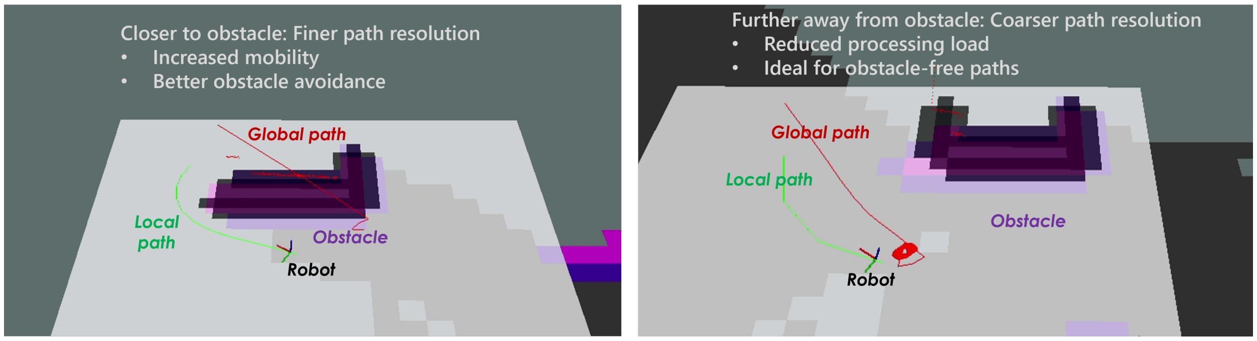
Closer to obstacle: Finer path resolution
• Increased mobility
• Better obstacle avoidance
Global path
Local path
Obstacle
Robot
Further away from obstacle: Coarser path resolution
• Reduced processing load
• Ideal for obstacle-free paths
Global path
Local path
Obstacle
Robot

Figure 4.10: Illustration of proposed Adaptive path planning scheme.

Experiments were performed on Gazebo employing a simulated robot that mimicked the software architecture of α-WaLTR. Although the simulated model does not replicate the physics of the platform, this is seen as being representative of the real robot. In addition, exercises in simulations are less expensive and low-risk compared to the real world. It also allows for a variety of scenarios that cannot be replicated. Three sets of environments were used for these experiments: A) with obstacles placed randomly around the robot (Fig. 4.11), B) with obstacles around the robot sparsely distributed following a circular pattern (Fig. 4.12), and C) with a densely crowded set of obstacles around the robot following a circular pattern (Fig. 4.13).

Each environment had three levels of granularity: 1) Coarse (0.85 linear, 0.75 angular), 2) Fine (0.25 linear, 0.2 angular), 3) Hybrid (switching between Coarse and Fine). The criteria to switch between High and Low levels was based on a simulated sensor input. To retain simplicity, the LaserScan data from the Lidar was employed for this. Whenever an obstacle was within an arbitrary 1.5m radius of the robot, it would switch to Low, and once the proximity increased, it would switch back to High. The values of granularity were not chosen arbitrarily, rather initial trials were made to check how varying the linear granularity affected the angular value. In each environment, 8 trials were performed by varying the robot's heading angle by 45° from 0° to 360°. The robot was started off at $(0, 0)$ and set a target of $(15, 15)$ each time. In the respective figures, the robot is located at the starting position, and the target is denoted by a red star in each case.

4.3.2 Results of simulation

Two metrics were used to evaluate the performance of the navigation modes. The first one was the time it took to move from start to finish. The second metric was the length of the path traversed by the robot. The local trajectory consists of straight-line segments generated as the robot moves toward each local goal. Therefore, measuring the path length consisted of making a cumulative sum of the length of each line segment (Appendix B.3).

The results of the simulation are shown in Fig. 4.14 and Fig. 4.15. Both graphs are skewed to start at a threshold above zero to highlight the differences between them. The performance in

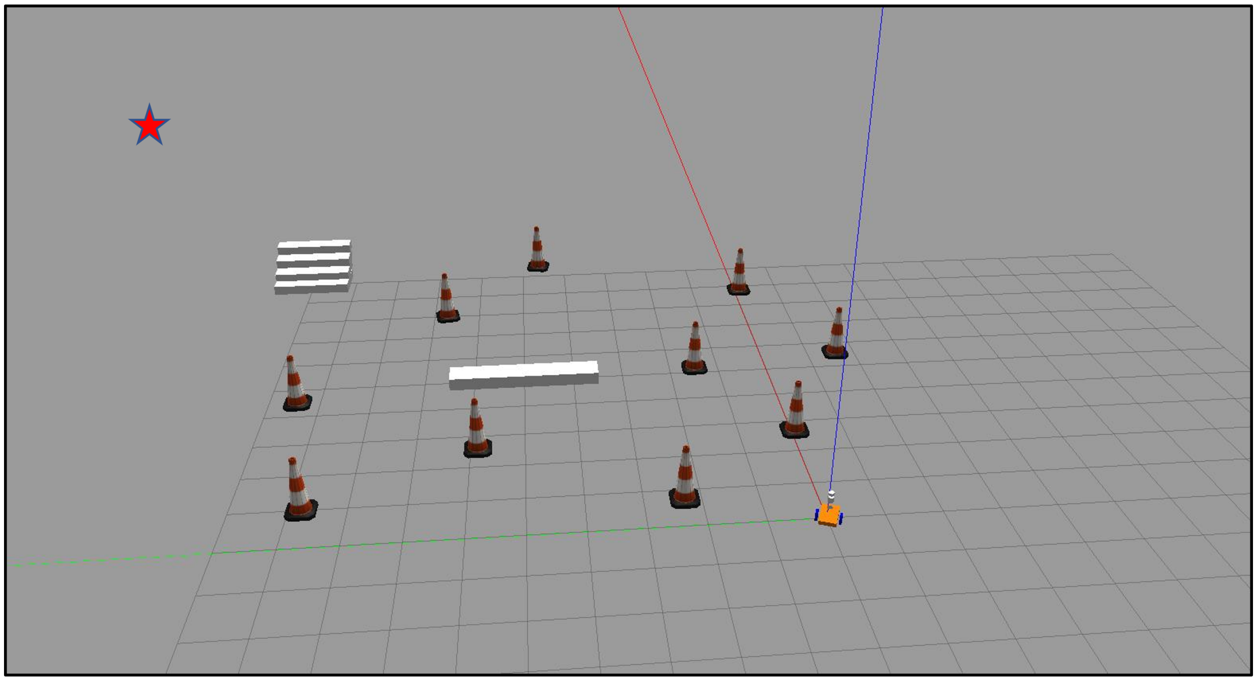
Figure 4.11: **Environment A**: Random pattern of obstacles.

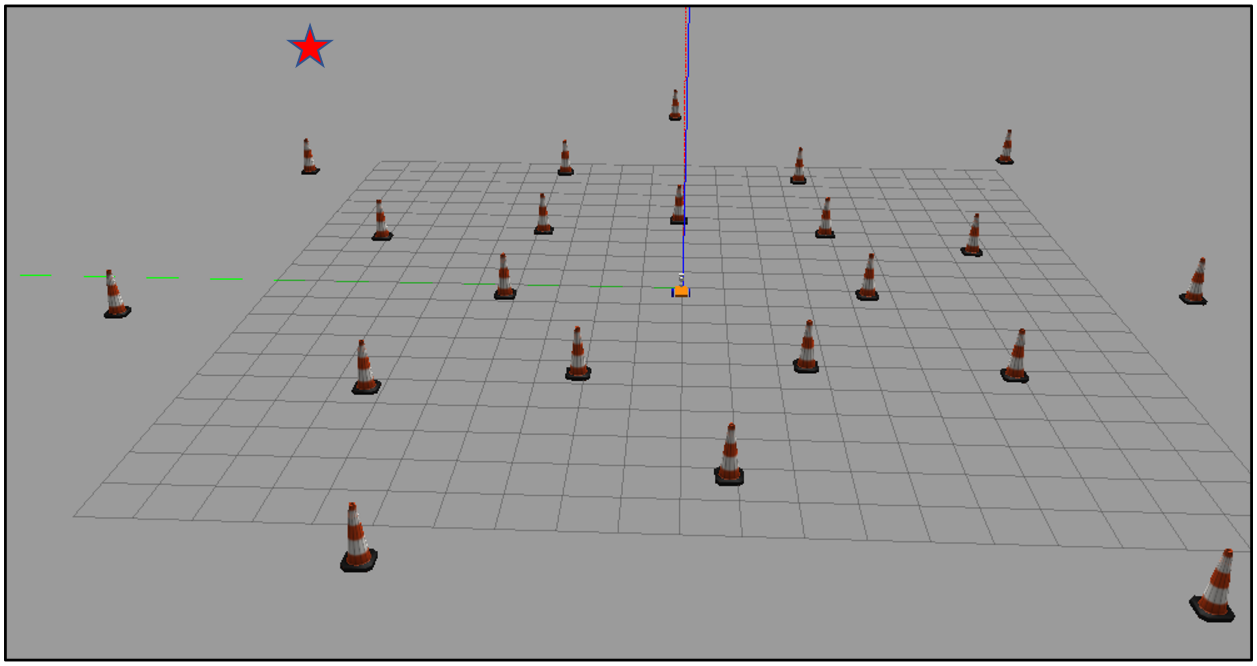
Figure 4.12: **Environment B**: Sparse circular pattern of obstacles.

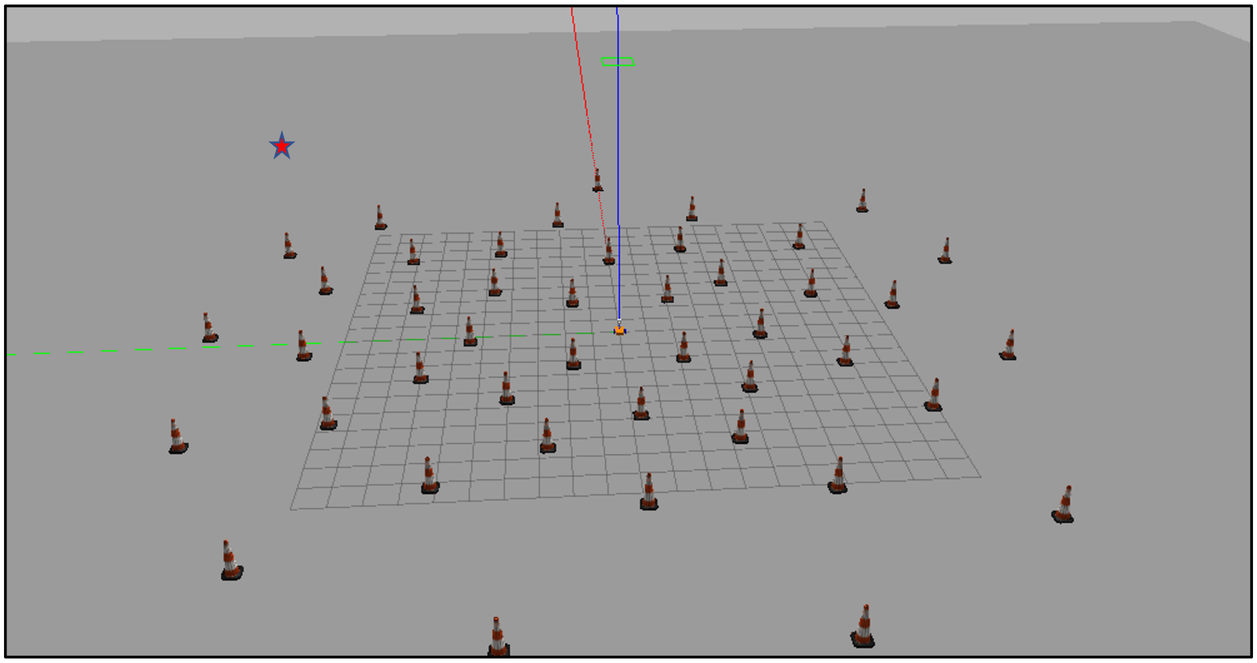

Figure 4.13: **Environment C**: Dense circular pattern of obstacles.

terms of both metrics in environment C (with densely populated obstacles) was similar for all three modes. This is supported by the fact that in the presence of densely located obstacles, the Hybrid mode performs in a similar manner to the Fine mode. The Coarse mode also exhibits similar behavior because the robot is forced to follow a roughly similar path irrespective of the mode to the goal. Therefore, in this case, it can be inferred that the additional computational effort does not translate to direct advantages. However in the cases of A and C, the Fine setting for Granularity showed a significant impact on the path length and the time to reach the goal. The path length and time in the environment A were significantly higher than the other two because of the presence of a different type of obstacle in the robot's operating zone. Negotiating it adds to the path length and time. In this case, the Hybrid mode resulted in the shortest path while the Coarse mode resulted in the shorted time to reach the target. However, the results for the Coarse and Hybrid settings were similar to each other. In both the environments, the difference in the time taken between the two modes is around 1s, and the path length difference is around 0.15m.

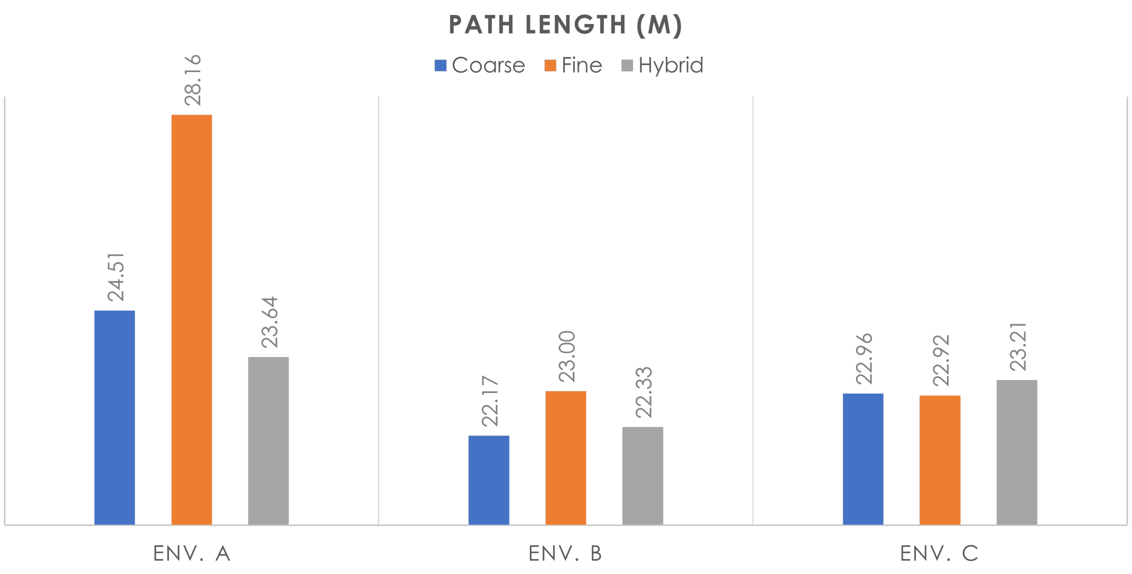

Figure 4.14: Simulation Result: Path length.

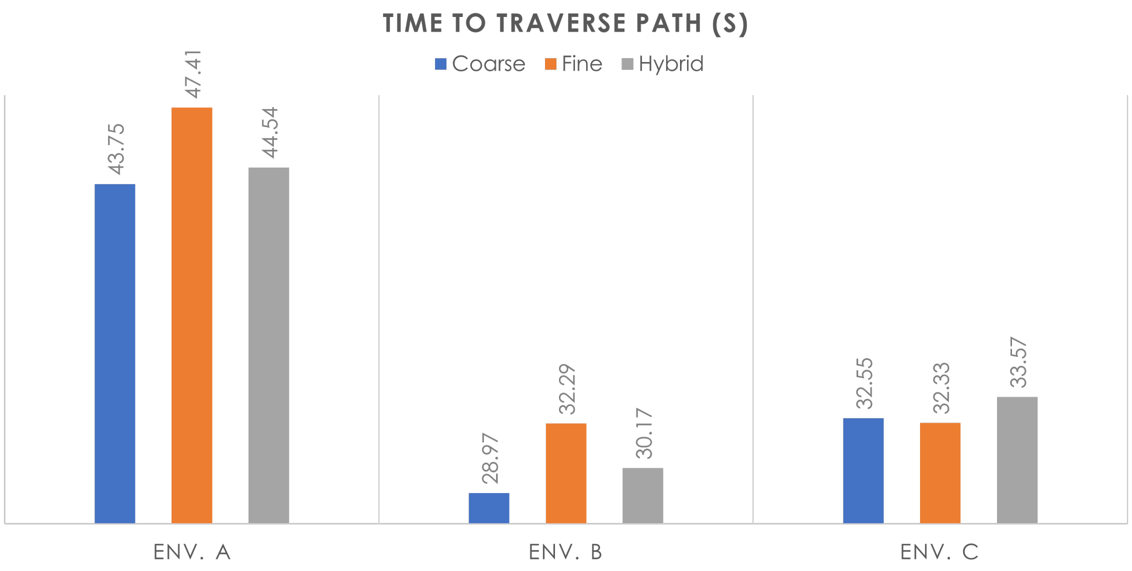

Figure 4.15: Simulation Result: Time for traversal.

The experiments and results here are preliminary. In these limited experiments, the potential benefits of the Hybrid mode over the Coarse mode are not clearly demonstrated. However, if the environment involves both crowded and open areas with diverse obstacles, the hybrid mode is expected to outperform the other two in terms of travel time and distance. Further simulations may be performed to confirm this. By having a preset number of modes the robot could switch between them based on external stimuli. In a real-world application, this would involve testing and bench-marking the performance of a robot platform in different terrains or around different classes of obstacles to determine the optimal set of parameters for each terrain. The switching between them can be accomplished by any sensory input. Computationally, this would not affect the performance of the other processes, as it runs as a separate node on the same network.

5. SUMMARY AND CONCLUSIONS

With the continuing advances in vision systems, ground robots will continue to become more sophisticated and autonomous. Making vision-based decision making algorithms more deterministic (as opposed to the current machine learning and deep learning schemes) is a possible avenue of research. This would make them more reproducible and adaptable to varied applications.

The use of supervised learning tools as part of this research was seen as a step toward making a more complex objective decision framework. While at present, unmanned robots cannot be imbued with complex subjective decision-making capabilities, a system of interlocking AI algorithms goes a long way towards achieving this. Advances in electro-mechanical interfaces are soon to follow, thereby increasing the complexity of navigation instructions.

Work in the future would involve integrating the varied software packages in one – with algorithms for stairway detection and semantic scene parsing being part of a single yet modular 'vision' algorithm. In theory, this would be an optimal consumption of computational resources compared to disconnected ones operating in parallel. In addition, while on-board processing and decision-making is desired of all autonomous robots, off-board computation is an efficient alternative, as evidenced in this work. Robots in the field, especially ones involved in reconnaissance and search & rescue operations, cannot be expected to communicate effectively over conventionally used LAN protocols, making efficient computational resource consumption higher on the list of priorities.

REFERENCES

[1] P. F. Muir, C. P. Neuman, I. J. Cox, and G. T. Wilfong, *Kinematic Modeling for Feedback Control of an Omnidirectional Wheeled Mobile Robot*, pp. 25–31. New York, NY: Springer New York, 1990.

[2] T. Lauwers, G. Kantor, and R. Hollis, "A dynamically stable single-wheeled mobile robot with inverse mouse-ball drive," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2884–2889, 2006.

[3] J.-X. Xu, Z.-Q. Guo, and T. H. Lee, "Design and implementation of integral sliding-mode control on an underactuated two-wheeled mobile robot," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 7, pp. 3671–3681, 2014.

[4] B. U. Rehman *et al.*, "Towards a multi-legged mobile manipulator," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3618–3624, 2016.

[5] D. Messuri and C. Klein, "Automatic body regulation for maintaining stability of a legged vehicle during rough-terrain locomotion," *IEEE Journal on Robotics and Automation*, vol. 1, no. 3, pp. 132–141, 1985.

[6] J. Morales *et al.*, "Power consumption modeling of skid-steer tracked mobile robots on rigid terrain," *IEEE Transactions on Robotics*, vol. 25, no. 5, pp. 1098–1108, 2009.

[7] Y. Liu and G. Liu, "Mobile manipulation using tracks of a tracked mobile robot," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 948–953, 2009.

[8] C. Zheng and K. Lee, "Wheeler: Wheel-leg reconfigurable mechanism with passive gears for mobile robot applications," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9292–9298, 2019.

[9] T. Sun, X. Xiang, W. Su, H. Wu, and Y. Song, "A transformable wheel-legged mobile robot: Design, analysis and experiment," *Robotics and Autonomous Systems*, vol. 98, pp. 30–41,

2017.

[10] X. Duan *et al.*, "Kinematic modeling of a small mobile robot with multi-locomotion modes," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5582–5587, 2006.

[11] Z. Luo *et al.*, "A reconfigurable hybrid wheel-track mobile robot based on watt II six-bar linkage," *Mechanism and Machine Theory*, vol. 128, pp. 16–32, 2018.

[12] J. Wright and I. Jordanov, "Intelligent approaches in locomotion - a review," *Journal of Intelligent & Robotic Systems*, vol. 80, no. 2, pp. 255–277, 2015.

[13] "ROS navigation stack." [Online]. Available at: https://wiki.ros.org/move_base (Accessed: 04 March 2022).

[14] Stanford Artificial Intelligence Laboratory, "Robotic operating system." [Online]. Available at: https://www.ros.org (Accessed: 04 March 2022).

[15] J. A. Hesch, G. L. Mariottini, and S. I. Roumeliotis, "Descending-stair detection, approach, and traversal with an autonomous tracked vehicle," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[16] A. Ciobanu *et al.*, "Real-time indoor staircase detection on mobile devices," in *International Conference on Control Systems and Computer Science*, 2017.

[17] S. A. Bouhamed, I. K. Kallel, and D. S. Masmoudi, "Stair case detection and recognition using ultrasonic signal," in *International Conference on Telecommunications and Signal Processing*, 2013.

[18] S. Ponnada *et al.*, "A hybrid approach for identification of manhole and staircase to assist visually challenged," *IEEE Access*, 2018.

[19] S. Se and M. Brady, "Vision-based detection of staircases," in *Asian Conference on Computer Vision (ACCV)*, 2000.

[20] S. Carbonara and C. Guaragnella, “Efficient stairs detection algorithm assisted navigation for vision impaired people,” in *IEEE International Symposium on Innovations in Intelligent Systems and Applications (INISTA)*, 2014.

[21] K. Romić, I. Galić, and T. Galba, “Technology assisting the blind — video processing based staircase detection,” in *International Symposium on Electronics in Marine*, 2015.

[22] R. Munoz, X. Rong, and Y. Tian, “Depth-aware indoor staircase detection and recognition for the visually impaired,” in *IEEE International Conference on Multimedia & Expo Workshops*, 2016.

[23] H. Mano *et al.*, “Treaded control system for rescue robots in indoor environment,” in *IEEE International Conference on Robotics and Biomimetics*, 2009.

[24] E. Mihankhah *et al.*, “Autonomous staircase detection and stair climbing for a tracked mobile robot using fuzzy controller,” in *IEEE International Conference on Robotics and Biomimetics*, 2009.

[25] X. Lu and R. Manduchi, “Detection and localization of curbs and stairways using stereo vision,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2005.

[26] S. Fazli, H. M. Dehnavi, and P. Moallem, “A robust obstacle detection method in highly textured environments using stereo vision,” in *International Conference on Machine Vision*, 2009.

[27] T. Schwarze and Z. Zhong, “Stair detection and tracking from egocentric stereo vision,” in *IEEE International Conference on Image Processing (ICIP)*, 2015.

[28] A. L. F. Castro *et al.*, “A novel approach for natural landmarks identification using RGB-D sensors,” in *International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2016.

[29] D. S. Chan *et al.*, “Efficient stairway detection and modeling for autonomous robot climbing,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[30] M. U. Masood *et al.*, “Design and development of a semi-autonomous stair climbing robotic platform for rough terrains,” in *International Conference on Control, Automation and Systems (ICCAS)*, 2017.

[31] Y. Cong *et al.*, “A stairway detection algorithm based on vision for UGV stair climbing,” in *IEEE International Conference on Networking, Sensing and Control*, 2008.

[32] S. Wang and H. Wang, “2D staircase detection using real AdaBoost,” in *International Conference on Information, Communications and Signal Processing (ICICS)*, 2009.

[33] U. Patil *et al.*, “Deep learning based stair detection and statistical image filtering for autonomous stair climbing,” in *IEEE International Conference on Robotic Computing (IRC)*, 2019.

[34] G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000.

[35] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, 2011.

[36] Y. Lu and D. Song, “Robust rgb-d odometry using point and line features,” in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

[37] H. Kong *et al.*, “Vanishing point detection for road detection,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[38] H. Jiang *et al.*, “Curve path detection of unstructured roads for the outdoor robot navigation,” *Mathematical and Computer Modelling*, 2013.

[39] M. C. Le, S. L. Phung, and A. Bouzerdoum, “Pedestrian lane detection in unstructured environments for assistive navigation,” in *International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pp. 1–8, 2014.

[40] T. N. A. Nguyen, S. L. Phung, and A. Bouzerdoum, “Hybrid deep learning-gaussian process network for pedestrian lane detection in unstructured scenes,” *IEEE Trans. Neural Netw. Learn. Syst.*, 2020.

[41] W. Zhou, S. Worrall, A. Zyner, and E. Nebot, “Automated process for incorporating drivable path into real-time semantic segmentation,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[42] B. Baheti *et al.*, “Eff-unet: A novel architecture for semantic segmentation in unstructured environment,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[43] A. Mousavian *et al.*, “Visual representations for semantic target driven navigation,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[44] L. Dabbiru *et al.*, “Traversability mapping in off-road environment using semantic segmentation,” in *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure*, 2021.

[45] T. Guan *et al.*, “GANav: Efficient terrain segmentation for robot navigation in unstructured outdoor environments,” *Computing Research Repository (CoRR)*, vol. abs/2103.04233, 2021.

[46] I. H. Hsieh *et al.*, “Outdoor walking guide for the visually-impaired people based on semantic segmentation and depth map,” in *International Conference on Pervasive Artificial Intelligence (ICPAI)*, 2020.

[47] S. Kowalewski, A. L. Maurin, and J. C. Andersen, “Semantic mapping and object detection for indoor mobile robots,” *IOP Conference Series: Materials Science and Engineering*, 2019.

[48] J. Hu *et al.*, “Evaluation of multimodal semantic segmentation using RGB-D data,” in *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III*, 2021.

[49] P. Jiang *et al.*, “RELLIS-3D dataset: Data, benchmarks and analysis,” *Computing Research Repository (CoRR)*, vol. abs/2011.12954, 2020.

[50] J. Wang *et al.*, “Deep high-resolution representation learning for visual recognition,” *Computing Research Repository (CoRR)*, vol. abs/1908.07919, 2019.

[51] Y. Yuan, X. Chen, and J. Wang, "Object-contextual representations for semantic segmentation," *European Conference on Computer Vision (ECCV)*, 2020.

[52] T. Takikawa, D. Acuna, V. Jampani, and S. Fidler, "Gated-scnn: Gated shape cnns for semantic segmentation," *International Conference on Computer Vision (ICCV)*, 2019.

[53] AION Robotics, "R1 ArduPilot autonomous rover UGV." [Online]. Available at: https://www.aionrobotics.com/r1-ardupilot-edition (Accessed: 04 March 2022).

[54] L. Keselman *et al.*, "Intel RealSense stereoscopic depth cameras," *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2017.

[55] L. Meier *et al.*, "Pixhawk: A system for autonomous flight using onboard computer vision," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[56] Y. Lu and D. Song, "Robust RGB-D odometry using point and line features," in *International Conference on Computer Vision (ICCV)*, pp. 3934–3942, 2015.

[57] K. Lai *et al.*, "A large-scale hierarchical multi-view RGB-D object dataset," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1817–1824, 2011.

[58] J. Sturm *et al.*, "A benchmark for the evaluation of RGB-D SLAM systems," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 573–580, 2012.

[59] N. Silberman *et al.*, "Indoor segmentation and support inference from RGBD images," in *European Conference on Computer Vision (ECCV)*, pp. 746–760, 2012.

[60] T. Hackel *et al.*, "SEMANTIC3D.NET: A new large-scale point cloud classification benchmark," in *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. IV-1-W1, pp. 91–98, 2017.

[61] J. Jeong *et al.*, "Complex urban dataset with multi-level sensors from highly diverse urban environments," *International Journal of Robotics Research (IJRR)*, vol. 38, no. 6, pp. 642–657, 2019.

[62] A. Geiger *et al.*, "Vision meets robotics: The KITTI dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[63] P. Sun *et al.*, "Scalability in perception for autonomous driving: Waymo open dataset," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (CVPR)*, pp. 2446–2454, 2020.

[64] M. Wigness *et al.*, "A RUGD dataset for autonomous navigation and visual perception in unstructured outdoor environments," in *International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[65] M. Cordts *et al.*, "The Cityscapes dataset for semantic urban scene understanding," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

APPENDIX A

DATASETS

Sensor Type / Type of Data	Frames	Recorded Sequences	Point clouds
RGB-D Camera	[57], [22], [56], [58]	[59]	[60]
Stereo Camera	[61]	[62]	
Lidar		[63]	[61], [49], [62]
Monocular Camera	[49], [64]	[65], [63]	
RADAR		[63]	

Table A.1: A summarized list of datasets comprising urban and unstructured environments for computer vision.

APPENDIX B

IMPLEMENTED CODE

B.1 ROS Package for staircase detection

```
#include <iostream>
#include <ros/ros.h>
#include <ros/package.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/opencv.hpp>
#include "opencv2/imgproc.hpp"
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui.hpp"
#include <fstream>
#include "svm.h"
#include <stdlib.h>
#include <time.h>

static const std::string OPENCV_WINDOW_1 = "Image window Color";
static const std::string OPENCV_WINDOW_2 = "Image window Depth";

float lLength(cv::Vec4i line);
float slope(cv::Vec4i line);
void SortStair(std::vector<cv::Vec4i> Lines, std::vector<cv::Vec4i> *Sorted);

class StairDetector
{
  private:
   ros::NodeHandle nh_;
```

```cpp
   image_transport::ImageTransport it_;
   image_transport::Subscriber color_image_sub;
   image_transport::Subscriber depth_image_sub;
   image_transport::Publisher stair_pub, cand_pub, all_pub;

   cv::Mat image_RGB, imageD;
   int frame, positive;

   std::string rgb_topic = "/camera/color/image_raw";
   std::string depth_topic = "/camera/depth/image_rect_raw";
   int update_rate;

   ros::Rate* _loop_rate;

  public:
   void run();
   void rgb_callback(const sensor_msgs::ImageConstPtr& msg);
   void depth_callback(const sensor_msgs::ImageConstPtr& msg);
   void depth_process(cv::Mat *imageRGB, cv::Mat *candidates, cv::Mat imageD,
    std::vector<std::vector<cv::Vec4i>> *StairLines, int *f, int *f2);
   void rgb_pre_process(cv::Mat imageRGB, cv::Mat *all, std::vector<std::
    vector<cv::Vec4i>> *StairLines, int x);

   StairDetector(ros::NodeHandle _nh);
   ~StairDetector();
};

StairDetector::StairDetector(ros::NodeHandle _nh):nh_(_nh),it_(_nh)
   {
     this->update_rate = 5;
     this->frame = 0;
     this->positive = 0;
     std::cout << "\nColor Sub";
     this->color_image_sub = this->it_.subscribe(this->rgb_topic, 15, & StairDetector::rgb_callback, this);
     std::cout << "\nDepth Sub";
     this->depth_image_sub = this->it_.subscribe(this->depth_topic, 15, & StairDetector::depth_callback, this);
     this->_loop_rate = new ros::Rate(this->update_rate);
     this->stair_pub = this->it_.advertise("/camera/color/stair_case_roi", 1);
     this->cand_pub = this->it_.advertise("/camera/color/stair_case_candidates", 1);
     this->all_pub = this->it_.advertise("/camera/color/stair_case_all", 1);
     std::cout << "\nConstructor Done";
   }

StairDetector::~StairDetector()
   {
     delete this->_loop_rate;
   }

void StairDetector::rgb_callback(const sensor_msgs::ImageConstPtr& msg) {
     std::cout << "\nStarting RGB write";
     cv::Mat cv_ptr;
     cv_ptr = cv_bridge::toCvShare(msg, sensor_msgs::image_encodings::BGR8)->image;

     this->image_RGB = cv_ptr.clone();
     if (this->image_RGB.empty())
  std::cout << "\nEmpty RGB";
}

void StairDetector::depth_callback(const sensor_msgs::ImageConstPtr& msg) {
     std::cout << "\nStarting Depth write";
     cv_bridge::CvImagePtr cv_ptr;
     cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::TYPE_8UC1
   );
     if (cv_ptr->image.empty())
  std::cout << "\nEmpty Depth";
     this->imageD = cv_ptr->image;
}

void StairDetector::rgb_pre_process(cv::Mat imageRGB, cv::Mat *imshow1, std::
   vector<std::vector<cv::Vec4i>> *StairLines, int frameC)
{

        std::cout << "\nStarting RGB Processing:";
        int cannyLT = 25, cannyUT = 40, houghTh = 75;
        cv::Mat RGBGray, RGBflt, RGBflt2, dltdRGB;
  *imshow1 = imageRGB.clone();

        cv::cvtColor(imageRGB, RGBGray, cv::COLOR_RGB2GRAY);
  cv::Mat element = getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(20, 10),
    cv::Point(-1, -1));
        cv::dilate(RGBGray, dltdRGB, element);
  cv::Canny(dltdRGB, RGBflt2, cannyLT, cannyUT, 3);
        std::vector<cv::Vec4i> linesP, Stairs;
        cv::HoughLinesP(RGBflt2, linesP, 1, 3 * CV_PI / 180, houghTh, 40, 5);

  /*Loop 1 - to reject short and steep lines*/
        for (size_t k = 0; k < linesP.size(); k++)
        {
            cv::Vec4i l = linesP[k];
            float len = lLength(l);
            if (slope(l) < 80){
                Stairs.push_back(l);
                cv::line(*imshow1, cv::Point(l[0], l[1]), cv::Point(l[2], l
   [3]), cv::Scalar(255, 0, 0), 2, cv::LINE_AA);
            }
        }
std::cout << "\nStairs extracted:";
cv::imshow("All lines", *imshow1);

/*Loop 2 - to reject overlapping lines*/
        for (size_t k = 0; k < Stairs.size(); k++)
        {
            for (size_t j = k + 1; j < Stairs.size(); j++)
            {
                cv::Vec4i l = Stairs[k];
                float x = (l[0] + l[2]) / 2;
                float y = (l[1] + l[3]) / 2;

                cv::Vec4i l2 = Stairs[j];
                float x2 = (l2[0] + l2[2]) / 2;
                float y2 = (l2[1] + l2[3]) / 2;

                if (((abs(x2 - x) < 5) && (abs(y2 - y) < 20)) || ((abs(x2 - x)
  < 20) && (abs(y2 - y) < 5))) {
                    if (lLength(l) > lLength(l2))
                        Stairs.erase(Stairs.begin() + j);
                    else
                        Stairs.erase(Stairs.begin() + k);
                }
            }
        }

std::cout << "\nSorting Stairs:";
/*Loop 3 - nested - to group lines by slope and position*/
        for (size_t k = 0; k < Stairs.size(); k++)
        {
            std::vector<cv::Vec4i> SimSlope, SortedSimSlope;
            for (size_t j = k + 1; j < Stairs.size(); j++)
            {
                cv::Vec4i l = Stairs[k];
                float x = (l[0] + l[2]) / 2;
                float y = (l[1] + l[3]) / 2;

                cv::Vec4i l2 = Stairs[j];
                float x2 = (l2[0] + l2[2]) / 2;
                float y2 = (l2[1] + l2[3]) / 2;

                if ((abs(y2 - y) < 85) && (abs(slope(l) - slope(l2)) < 1.5)) {
                    SimSlope.push_back(l2);
                }
            }

            if (SimSlope.size() > 0) {
                SimSlope.push_back(Stairs[k]);
                SortStair(SimSlope, &SortedSimSlope);
                StairLines->push_back(SortedSimSlope);
    std::cout << "Sorted";
            }
        }

  std::cout << "\nRGB Processing done";
  cv::waitKey(1);
}

void StairDetector::depth_process(cv::Mat *imageRGB, cv::Mat *candidates, cv::
    Mat imageD, std::vector<std::vector<cv::Vec4i>> *StairLines, int *
    frameCount, int *framePositive) {
  *candidates = imageRGB->clone();
  std::ofstream file;
  file.open("Train_Depth.txt", std::ofstream::app);
  file << "\n\nFrame: "<< *frameCount << "\n";
```

```

std::cout << "\nStarting Depth Processing:";

std::string ppath = ros::package::getPath("staircase_det");
std::cout << ppath;

//Trained models located in src under package
std::string mfile1 = ppath + '/' + "src/StairModel";
std::string mfile2 = ppath + '/' + "src/nonStairModel";
const char* MODEL_FILE = mfile1.c_str();
  const char* MODEL_FILE2 = mfile2.c_str();
  struct svm_model* SVMModel;
  struct svm_model* SVMModel2;
  if (((SVMModel = svm_load_model(MODEL_FILE)) == 0) || ((SVMModel2 =
 svm_load_model(MODEL_FILE2)) == 0)) {
      fprintf(stderr, "Can't load SVM model %s", MODEL_FILE);
  }
  struct svm_node* svmVec = new svm_node();
std::cout << "\nSVM Model loaded";
  std::cout << "\nInitialization";

double *predictions = new double;
double *predictions2 = new double;
  std::cout << "Extracting Depth values from image:";
  std::vector<std::vector<cv::Vec4i>> Stairs = *StairLines;

for (int i = 0; i < StairLines->size(); i++) {
      std::vector<cv::Vec4i> Stair = Stairs[i];
  std::vector<float> depths;
  std::vector<float> xl, yl;
    bool depth_flag = true;
  for (int j = 0; j < Stair.size(); j++) {
    cv::Vec4i l = Stair[j];
```

```
      xl.push_back((l[0] + l[2]) / 2);
      yl.push_back((l[1] + l[3]) / 2);
    }
    for (int j = 0; j < yl.size(); j++) {
      float Dist = 0.001*imageD.at<u_int16_t>(yl[j], xl[j]);

      /*Scale Depth here as needed*/
      //Dist = -0.012*Dist + 3.5; //TAMU Grayscale set
      //Dist = -0.01875*Dist + 4.375; //TAMU Redscale set
      depths.push_back(Dist / 10.0);
      if (Dist > 10.0) //Ignoring depths further than reliable range of 10m
        depth_flag = false;
      }

    svmVec = (struct svm_node*)malloc(8 * sizeof(struct svm_node));
    int v;
    int rgb[3];
    for(size_t i = 0; i<3; i++){
      rgb[i] = rand() % 256; //To randomly generate a color for one set of candidates
      }
    if ((depths.size() <= 8) && (depth_flag)) {//Using only candidate sets of size 8
      for (size_t x = 0; x < Stair.size(); x++){
        cv::Vec4i l = Stair[x];
        cv::line(*candidates, cv::Point(l[0], l[1]), cv::Point(l[2], l[3]), cv::Scalar(rgb[0], rgb[1], rgb[2]), 2, cv::LINE_AA);
      }
      for (v = 0; v < 8; v++) {
        if (v < depths.size()) {
          svmVec[v].index = v + 1;//SV Index
          svmVec[v].value = depths[v];//SV Value
        }
        else {//If candidate is of size < 8, 0.0 is assigned to remaining index positions
          svmVec[v].index = v + 1;
          svmVec[v].value = 0.0;
        }
      }
      svmVec[v].index = -1; //To denote end of depth vector

      file << "\t Candidate: "<< i << "\t [" << rgb[0] << "," << rgb[1] << "," << rgb[2] << "] \t";
      for (size_t u = 0; u < depths.size(); u++){
        file << u+1 << ":" << depths[u] << "\t";
      }
      file << "\n";

      predictions[0] = svm_predict(SVMModel, svmVec);//LIBSVM function svm_predict()
      predictions2[0] = svm_predict(SVMModel2, svmVec);

      std::cout << "\nStarting Classification:";
      //Either Model1 - Positive (Presence of Stair)
      //or Model2 - Negative (Absence of non-Stair)
      if ((*predictions == 1.0) || (*predictions2 == -1.0)){
        std::cout << "\n Positive classification";
        if (Stair.size() > 2) {
          ++(*framePositive);
          for (size_t i = 0; i < Stair.size(); i++) {
            cv::Vec4i l = Stair[i];
            cv::line(*imageRGB, cv::Point(l[0], l[1]), cv::Point(l[2], l[3]), cv::Scalar(0, 0, 255), 2, cv::LINE_AA);
          }
        break;//break here to stop testing once the first positive candidate is found
```

```
        }
      }
    }
  }
  char txt3[100];
  snprintf(txt3, 100, "Frame : %d; Positive: %d", *frameCount, *framePositive)
   ;
  int w = imageRGB->rows, h = imageRGB->cols;
  std::cout << "Showing final RGB Image";
    cv::putText(*imageRGB, txt3, cv::Point(w/8, h/8), cv::FONT_HERSHEY_SIMPLEX
   , 0.6, cv::Scalar(255, 255, 255), 2);
    cv::Mat imshow3 = imageRGB->clone();
  cv::imshow("Final Stair", imshow3);
    cv::imshow(OPENCV_WINDOW_2, imageD);
  cv::waitKey(1);
  std::cout << "\nDepth Processing Done";
  file.close();

}

void StairDetector::run(){
  std::cout << "\nBegin RunFn";

  while (ros::ok()){
    srand(time(NULL));
    ros::Time Stamp = ros::Time::now();
    std::cout << "\nInitialize variables";
    cv::Mat *RGBImage = &(this->image_RGB), RGBFin;
    cv::Mat *cand = new cv::Mat;
    cv::Mat *all = new cv::Mat;
    std::vector<std::vector<cv::Vec4i>> *StairLines = new std::vector<std::
   vector<cv::Vec4i>>;
    int *frameC = &(this->frame);
    int *frameP = &(this->positive);
    std::cout << "\n Empty?: " << this->image_RGB.empty();
    if (this->image_RGB.empty() || this->imageD.empty()) {
      std::cout << "\nNo Image written";
    }
    else {
      ++(this->frame);
      std::cout << "\nImage processing:";
      this->rgb_pre_process(this->image_RGB, all, StairLines, this->frame);
      std::cout << "\nFrom run(): RGB Processing done";
      this->depth_process(RGBImage, cand, this->imageD, StairLines, frameC,
   frameP);
      std::cout << "\nFrom run(): Depth Processing done:";
      std::string det = std::string("Det") + std::to_string(this->frame) + std
   ::string(".jpg");
      std::string can = std::string("Cand") + std::to_string(this->frame) +
   std::string(".jpg");
      cv::imwrite(det, *RGBImage);
      cv::imwrite(can, *cand);
    }
    std::cout << "\n Proceeding to Spin";
    ros::spinOnce();
    this->_loop_rate->sleep();
  }
}

float lLength(cv::Vec4i line)
{
    return sqrt((line[2] - line[0]) ^ 2 + (line[3] - line[1]) ^ 2);
}

float slope(cv::Vec4i line)
{
    float slope = abs(atan2((line[3] - line[1]), (line[2] - line[0])) * 180 /
   CV_PI);
    if (slope > 90)
    {
        if (slope > 180)
        {
            if (slope > 270)
                slope = slope - 180.00;
            else
                slope = 360.00 - slope;
        }
        else
            slope = 180.00 - slope;
    }
    return slope;
}

void SortStair(std::vector<cv::Vec4i> Lines, std::vector<cv::Vec4i> *Sorted){
    std::vector<float> MidX;
    for (int i = 0; i < Lines.size(); i++) {
        cv::Vec4i l = Lines[i];
        float mX = (l[1] + l[3]) / 2;
        MidX.push_back(mX);
    }
    while (MidX.size() > 0) {
        int p = distance(MidX.begin(), min_element(MidX.begin(), MidX.end()));
        Sorted->push_back(Lines[p]);
        MidX.erase(MidX.begin() + p);
        Lines.erase(Lines.begin() + p);
    }
}

int main(int argc, char** argv)
```

```
{
    ros::init(argc, argv, "staircase_detection");
    ros::NodeHandle nh;
    StairDetector ic(nh);
    std::cout << "\nBegin Run";
    ic.run();
    return 0;
}
```

B.2 ROS Client for logging images

```
#!/usr/bin/env python

import base64
import logging
import time
import numpy as np
from PIL import Image
import cv2
from io import BytesIO

from cv_bridge import CvBridge

import roslibpy

class imgCompressTest:
    def __init__(self):
         # Configure logging
         fmt = '%(asctime)s %(levelname)8s: %(message)s'
         logging.basicConfig(format=fmt, level=logging.INFO)
         self.log = logging.getLogger(__name__)
         self.client = roslibpy.Ros(host='localhost', port=9090)
         #Raw Image ROS Topic
         self.subscriber2 = roslibpy.Topic(self.client, '/camera/color/
    image_raw', 'sensor_msgs/Image')
         #Compressed Image ROS Topic
         self.subscriber = roslibpy.Topic(self.client, '/camera/color/
    image_raw/compressed', 'sensor_msgs/CompressedImage')

    def receive_image(self, msg):
         self.log.info('Received image seq=%d', msg['header']['seq'])
         base64_bytes = msg['data'].encode('ascii')
```

```
        image_bytes = base64.b64decode(base64_bytes)
        time_pas = msg['header']['stamp']['secs']
        msg_time = time.strftime("%H:%M:%S", time.localtime(time_pas))
        format = msg['format'].split()
        with open('CompF_{}.{}'.format(str(msg_time)+':'+str(msg['header']['
    stamp']['nsecs']), format[1]) , 'wb') as image_file:
            image_file.write(image_bytes)


def main():
    imgC = imgCompressTest()
    imgC.subscriber.subscribe(imgC.receive_image)
    imgC.client.run_forever() #Until stopped

if __name__ == '__main__':
    main()
```

B.3 ROS Client for path length and time calculation

```
#!/usr/bin/env python

import math
import rospy
from nav_msgs.msg import Path

class path_length(object):
  def path(self, msg):
    self.data.append(msg.poses[0])
    i = len(self.data) - 1
    self.sum_len += math.sqrt(pow((self.data[i].pose.position.x - self.data[i
   -1].pose.position.x), 2) + pow((self.data[i].pose.position.y - self.data[i
   -1].pose.position.y), 2))
        rospy.loginfo(self.sum_len)

  def __init__(self):
    self.sub = rospy.Subscriber('/move_base/TrajectoryPlannerROS/local_plan',
   Path, self.path)
    self.sum_len = 0.0
    self.data = []
    self.r = rospy.Rate(0.5)

  def start_pthlen(self):
    while not rospy.is_shutdown():
      self.r.sleep()

if __name__ == "__main__":
  rospy.init_node("path_length")
  myObj = path_length()
  myObj.start_pthlen()
```

B.4 ROS Client for parameter reconfiguration in runtime

```python
#!/usr/bin/env python

import rospy
import dynamic_reconfigure.client
from sensor_msgs.msg import LaserScan


class dyn_rec_mb(object):
  def laser_val(self, msg):
    initB = self.b
    if (min(msg.ranges) < 1.5):
      self.b = True
    else:
      self.b = False

    if (self.b == initB):
      self.StateChange = False
    else:
      self.StateChange = True
      rospy.loginfo("State Change - reconfig now!")

    if self.StateChange:
        rospy.loginfo("Changing params!")
        if self.b:
          self.client.update_configuration({'sim_granularity':0.25, '
    angular_sim_granularity':0.2})
        else:
          self.client.update_configuration({'sim_granularity':0.85, '
    angular_sim_granularity':0.75})

  def callback(self, config):
    rospy.loginfo("Config set to {sim_granularity}".format(**config))

  def __init__(self):
    self.client = dynamic_reconfigure.client.Client("move_base/
TrajectoryPlannerROS", config_callback=self.callback)
    self.sub = rospy.Subscriber('/scan', LaserScan, self.laser_val)
    self.b = False
    self.StateChange = False
    self.r = rospy.Rate(0.5)

  def start_dyn(self):
    while not rospy.is_shutdown():
      self.r.sleep()

if __name__ == "__main__":
  rospy.init_node("dynamic_client")
  myObj = dyn_rec_mb()
  myObj.start_dyn()
```