Progress Report 1
for Final Project 1 (2147416)

Home Where:

An autonomous rocker-bogie delivery robot

by

6438169421 Pattaradanai Lakkananithiphan

6438188321 Yossaphat Kulvatunyou

6438146021 Pongpol Srichart

6438095621 Thana Wanavit

Under the **advice** of Asst. Prof. Surat Kwanmuang

Progress Report I

Table of contents

Timeline changes

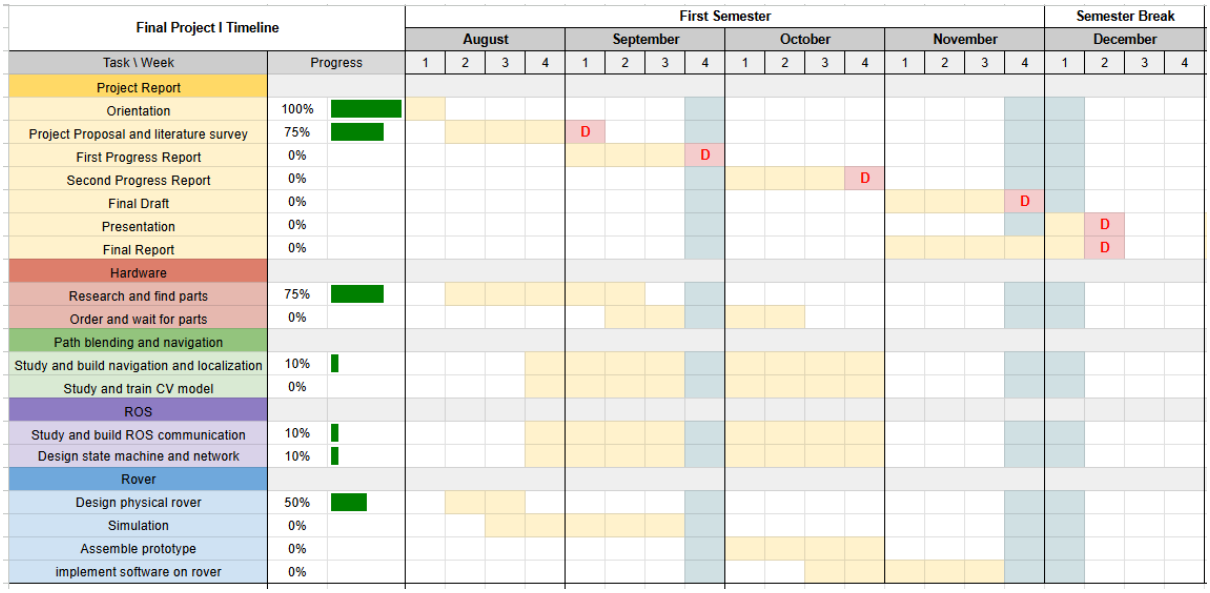

Fig 1. The detailed timeline of the project from August to December 2024

According to the original timeline given in the proposal, as can be seen above, the following should be included in this report: all the needed parts must be listed and ordered, the robot simulation must be complete, and the creation of the project's ROS package, as well as the navigation and localization system, including a terrain classifier, should be roughly halfway finished.

Unfortunately, one unforeseen problem arose after the proposal was submitted. After consulting with the project's advisor, it was made clear that the acquisition of parts has to be delayed until mid-October at a minimum as financial issues prevent the ordering and payment reimbursement before that date.

With that change, the timeline has to be altered especially in the rover assembly phase of the project. The more realistic timeline would have the assembly and implementation of software squished together between late October to late November. This means that the simulation phase as well as the development of individual software systems can be slightly extended further than was previously planned. To use this time effectively, each team member spent 1-2 weeks studying the programs and algorithms they will be using to help make the remaining development time more error-free.

Parts

The total parts that must be ordered or procured for this project are listed below along with the usage of each component and the amount required.

Parts	Usage	Amount
Intel RealSense Depth Cameras D435i	For obstacle detection and terrain segmentation as well as the included IMU	1
RPLIDAR S series	For obstacle detection and avoidance	1
ZLLG55ASM150 5.5-inch Hub motor Wheel	To make the robot mobile	6
ZLAC8015D Dual motor driver	To drive the hub motor	3
RaspberryPi Pico	Interface with external sensors	1
6s lipo battery	Power the motor	1
GPS Ublox NEO-M8N GPS Module	For Localization	1
rs485 to USB adapters	Connect Motor drivers to the Laptop	3
USB hub splitter 1 to at least 4	To allow the Laptop to connect to multiple devices	1
A Laptop computer	To control the robot	1
Frame materials	Structure and mechanism	-
3D printed parts	Structure and mechanism	-
Wires and cables	Connect electrical components	-
XT90 pdb	Connect the battery to the multiple drivers	1

Table 1. The parts required for this project

Rover design

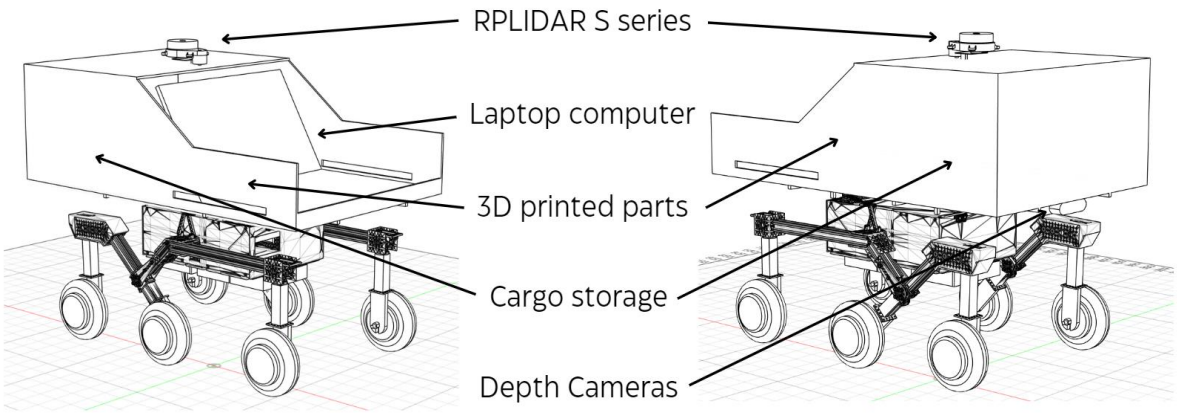

Fig 2. The 3D model

This work in this section is under the responsibility of and is worked on by Pongpol Srichart.

This design outlines the placement of all components for the robot's assembly. Our mechanical design is largely inspired by the JPL Open Source Rover Project [7], with modifications to incorporate delivery package capabilities. To streamline testing and coding adjustments, the laptop is positioned at the robot's rear, ensuring ease of access for software modifications. The LiDAR sensor is mounted at the top to ensure an unobstructed field of view, while the depth camera is placed at the front to allow for immediate object detection. Additional components not listed below are housed in the bottom compartment.

Terrain classification

This work in this section is under the responsibility of and is worked on by Pattaradanai Lakkananithiphan.

As stated in the project proposal, this section is dedicated to the making of a machine-learning model whose purpose is to take image input from the depth camera and classify the pixels in the image into classes. This information can be used by the robot operating system to aid the LIDAR navigation system in detecting obstacles. The machine learning algorithm to be used was identified as a semantic segmentation model, of which "DeepLabV3+" was selected as the prime candidate for the task with the segmentation dataset to be used in training the model being "Cityscapes" [6] combined with real-world data collected and labeled.

Semantic Segmentation

To be considered as working, the model must be able to do the following. First, the model must be able to do semantic segmentation on the images fed through the depth camera. Second, the model's performance must reach a certain level. Furthermore, the model's output image must be convertible to a form that can be used by the operating system. Lastly, the model must be able to operate in real time.

The first requirement can be satisfied by selecting an RGBD depth camera that provides a way to convert the point cloud into an RGB image, which can then be easily used in making predictions using an image-to-image model. The other requirements will require time and effort, which has two months allocated to them as stated in the timeline in the proposal. The second and fourth requirements can be individually developed and tested right away, however, the degree to which the vision system can work with the other parts of the robot may not be clear right away, therefore the third requirement could be subjected to change, and would be worked on later once the other system is online. Therefore the tasks that will be focused on during the two-month window, and will be presented in this and the next project proposal, are the second and fourth requirements.

Before starting the project, the following benchmark had been made to measure the progress for requirements two and four. The second requirement will assess the performance of the segmentation mask using a metric known as Intersection over Union (IoU), which is the

ratio between the overlapping space of the true mask and the predicted mask and the total space covered by the two masks. Since the robot will use the segmentation to roughly gauge its surroundings, the IoU need not be so high, therefore the benchmark will be set at IoU = 0.8. For the fourth benchmark, the frame rate in FPS of greater than 2 FPS will be used as the minimum requirement, however, if time allows, this project aims to get the frame rate number closer to 10 FPS. Note that these benchmarks are created with no prior experience with the model and the robot system, therefore it could be too conservative or too ambitious for the use case. If such benchmarks are not sufficient or are too much for the use case, they will be adjusted such that the robot still works according to the objective, but remains as efficient as possible.

Model

Typically, the library **torch** in Python is used to build an image-to-image model, which includes DeepLabV3+. A gigantic dataset will then be downloaded into the computer to be fed to the model. The model would then be trained on those data for some time on a decently strong machine, then it would be ready for testing.

To facilitate this process, the following strategy was used. First, a search for a framework was done to find a tool that would help accelerate the setting up process of the model. The search resulted in a GitHub repository known as "Segmentation Models Pytorch" (SMP) [1]. This library provides several semantic segmentation models along with ways to train, validate, as well as test them. This package uses a library known as “Pytorch lightning,” [2] which itself is built on top of PyTorch and simplifies the process of training neural networks by abstracting much of the boilerplate code for training loops, validation, and checkpointing. It helps users focus on model design by offering a clean separation between model and training logic while supporting hardware flexibility (CPU, GPU, TPU) and distributed training. As an extension of PyTorch, it retains all of PyTorch’s flexibility and power, making it easier to scale models and integrate features like logging and callbacks. The second strategy is to find a pre-trained semantic segmentation model that could potentially work in the environment in which the robot will be placed. Since the models are pretrained, there is no need to use a basic dataset like Cityscapes to train the model before adding real-world data, which could save a lot of time and computational power. Huggingface provides several pre-trained semantic segmentation models that can be installed and used on local machines. One good family of models is the NVIDIA’s Segformer family, especially those fine-tuned on the Cityscape dataset. These pre-

trained models can also be fine-tuned locally, which could help them perform in a specific environment. The bulk of the workload of the first month is focused on training and running tests on both of these model types. Since it is extremely tedious to collect and label data, which in this case is to take pictures of the scenes that the robot could be operating in and use some tools to make a ground truth segmentation mask, the first phase of tests was done without this real-world data.

The purpose of the first phase of the test is to find out three things. First, how does the pipeline of the models work as well as what is required to train/run the models? Second, how much time does a model take to evaluate an image (tested on an RTX 3050 Laptop GPU)? Lastly, how good are those models with no training data? Note that there is no good way to quantify how "good" a model is at segmenting an image without building a ground truth segmentation mask, therefore the goodness will be judged comparatively to each other models as well as visually compared to the input image.

To train the SMP models, a dataset called Camvid was used instead of Cityscapes, as Camvid is much simpler in complexity and smaller in size compared to Cityscapes. This difference in nature makes Camvid worse at being the actual dataset behind a working model than Cityscapes but is a much better choice in quick prototyping and cross-testing several models. There are many models provided in the SMP library, however, three were focused on their speed and performance. These models are "FPN", "U-net", and "DeepLabV3+", which were introduced in these papers respectively: [3], [4], [5]. All of these models run on an image of resolution (720x960) with a slow frame rate of 0.3 FPS, however, with a resolution of (192x256), the framerate of around 2-3 FPS was consistently ached which just barely passed the speed benchmark. For performance, the IoU tests on the Camvid testing set returned similar results of around 0.77 when trained at 150 epochs, and 0.8 when trained to 300 epochs, note that all of the models' IoU does not increase beyond 0.8 even after many more epochs. For test runs on real-world images taken, all the models did poorly. The segmentation masks are highly inaccurate in both coverage and class identified. This is to be expected as the real environment holds little resemblance to the training set. If the SMP models were to be used in the final project, it would require a lot of training using real world data. Fortunately, with the abstractions provided by Pytorch Lighting, the training should not be difficult.

Model	FPS	IoU on val	IoU on test	Remark on Real
FPN	3.15	0.88	0.79	Bad
DeepLabV3+	3.15	0.84	0.79	Bad
Unet	3.15	0.87	0.79	Bad

Table 2. The Performance of the SMP models (300 epochs) on the CamVid and Real set

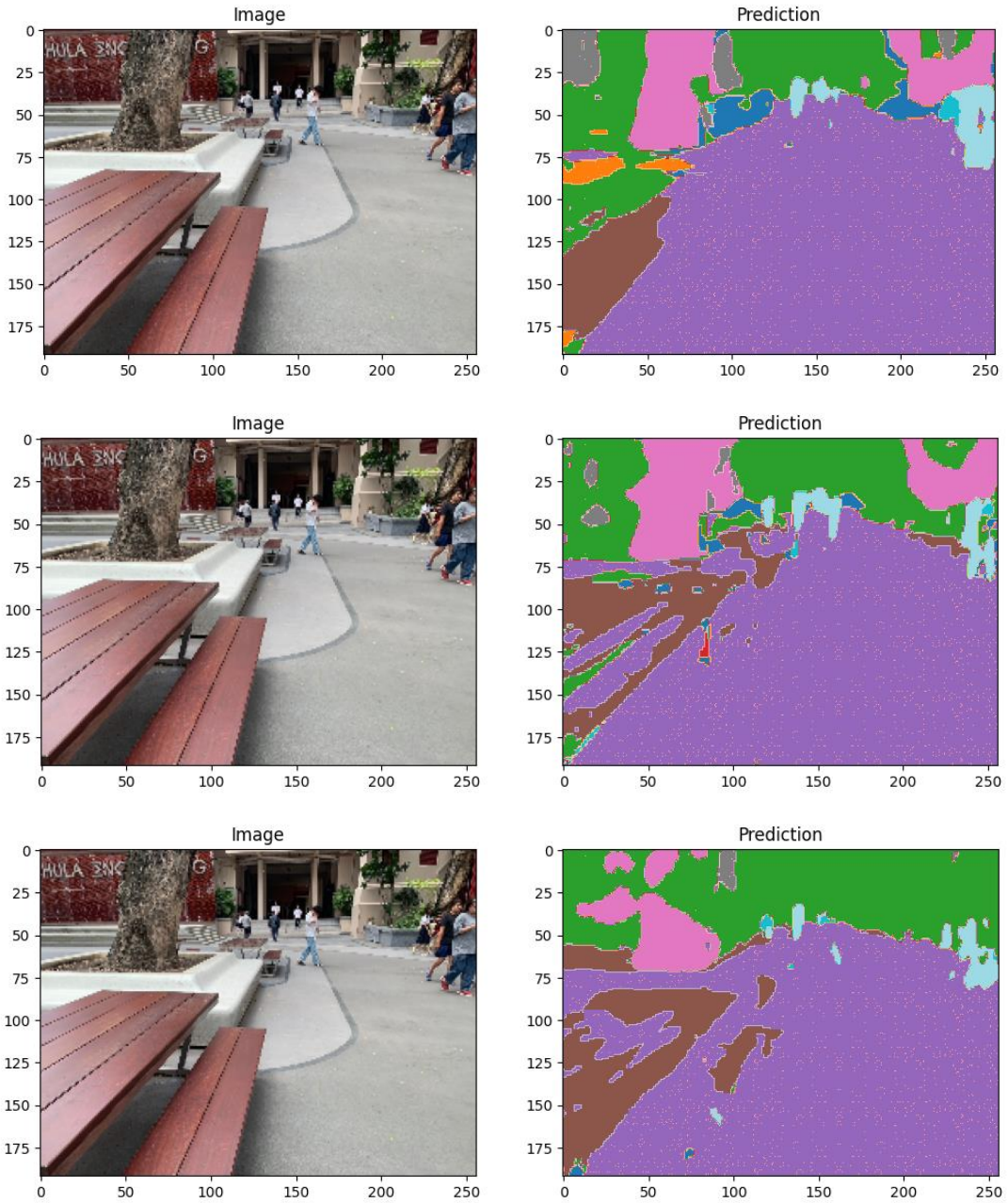

Fig 3. The segmentation output of FPN, DeepLabV3+, and Unet respectively

Since they are pre-trained on the Cityscapes, there is no need to worry about training the Segformer models at this stage. Their performance on the Cityscape test set differ over their model size, of which there are several varieties named b0-b5. The smallest model is known as b0, which is so lightweight that it can evaluate a (1024x2048) image at 5-10 FPS with great results, while the largest model known as b5 is so heavy that it took 10 seconds to evaluate just a frame of the same size, but returned an almost perfect segmentation mask. Since a fast frame rate is necessary, small models such as the b0, and b1 were considered candidates for the final project, however, b5 testing will also be done to show the difference in performances. The performance on the real data of the b0, and b1 models were bad, but not as bad as the SMP models, whereas those from b5 were already at the required level of accuracy, at least with visual judgment. To use these pretrained models in actual projects, fine-tuning using real-world data is necessary.

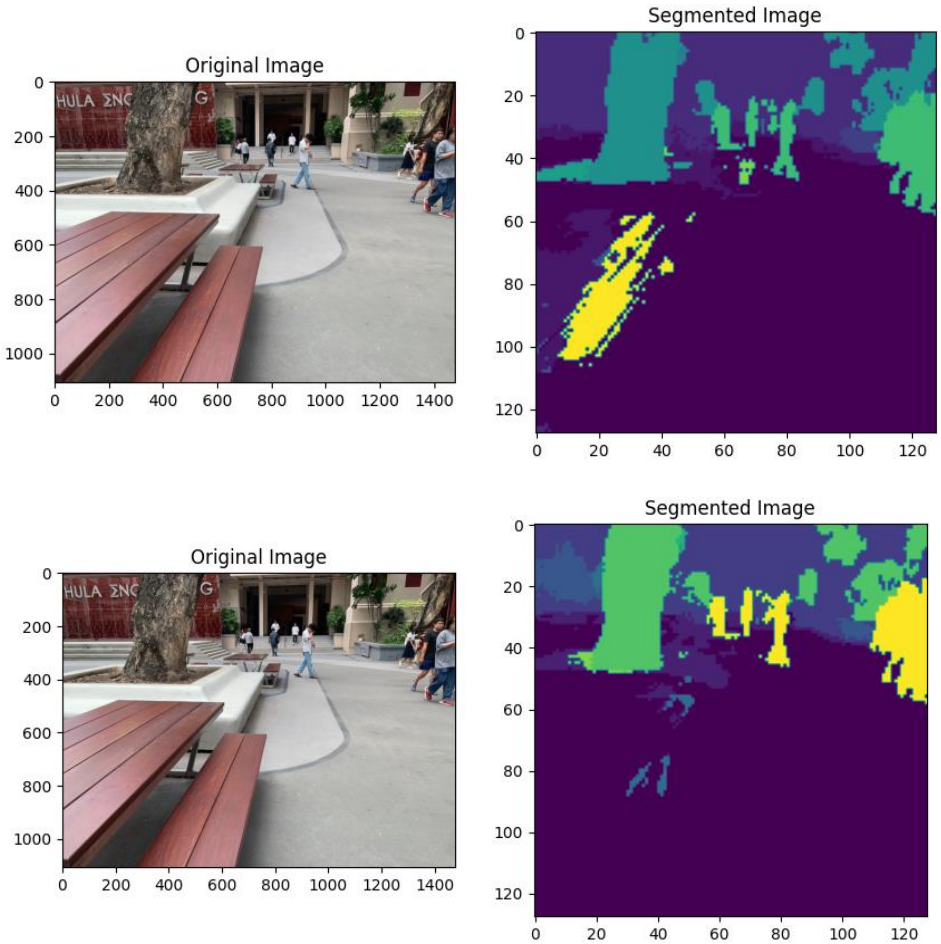

Fig 4. The segmentation mask of b0 512 and 1024 respectively. Note that the color difference in the mask is not due to misclassification but rather Matplotlib color range-based display.

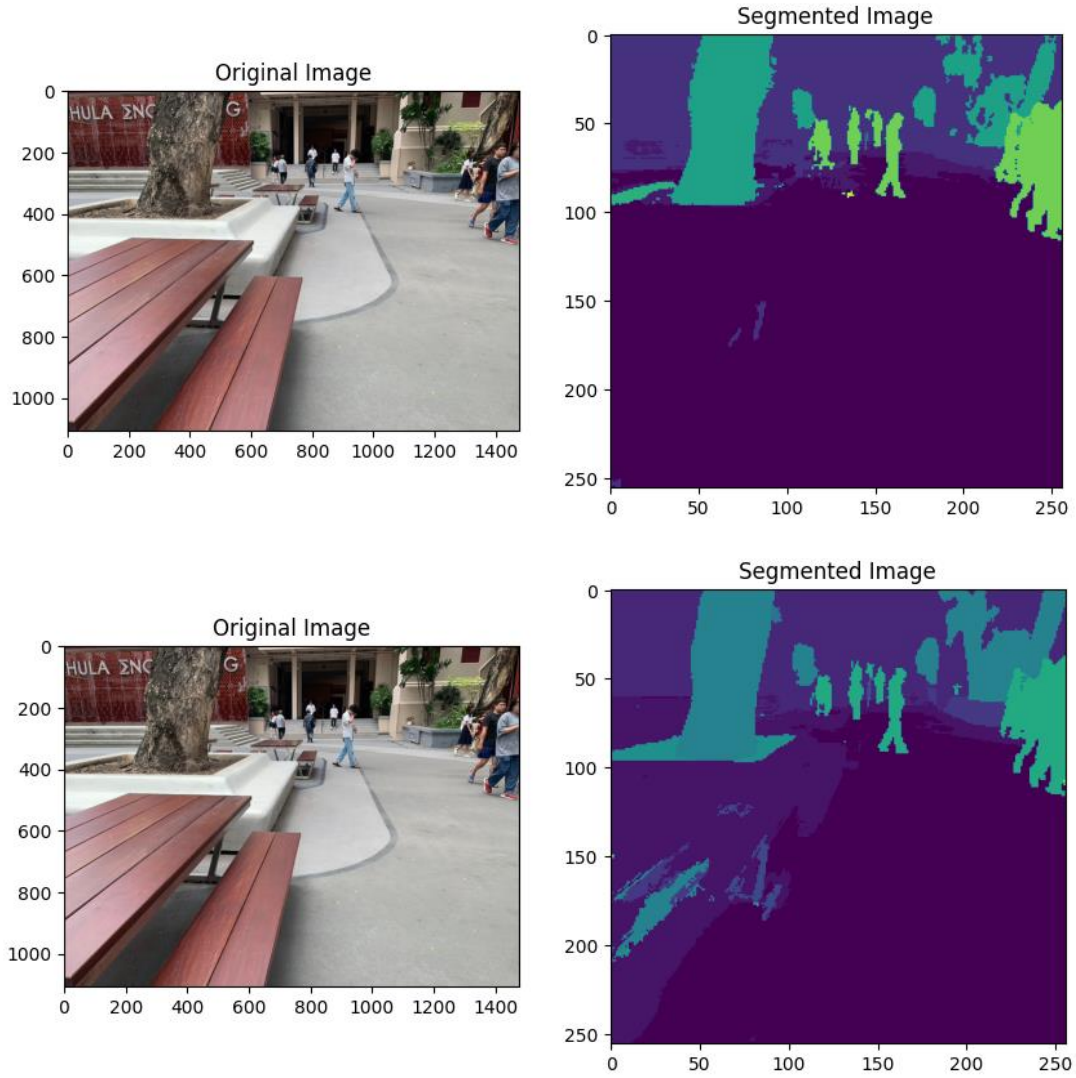

Fig 5. Segmentation mask of b1 and b5 respectively

Model	FPS	IoU on val	Remark on Real
b0 512x1024	2.3	0.67	recognizable
b0 1024x1024	2.4	0.67	recognizable
b1 1024x1024	0.77	0.79	good
b5 1024x1024	0.17	1	great

Table 3. The performance of the Segformer models on the Cityscape and real dataset

From the first phase of the test, the following conclusions can be drawn. First, the use of real-world data in the training of these models is imperative. Second, the performance of the pre-trained models is much better on the real-world data than those of the SMP models with clear contours of objects shown as opposed to the blobs of the smp's.

With such conclusions, the next steps would be gathering real-world data and creating the segmentation masks to finetune the Segformer models and train the SMP models. Whichever model responds best at this next stage will be the one selected for use.

ROS

This work in this section is under the responsibility of and is worked on by Thana Wanavit.

In this project, the robot will rely on the Robot Operating System to navigate in its environment. The version of ROS used is ROS1 Noetic, and the program for simulation includes Rviz and Gazebo.

ROS concept

To understand how a robot can use ROS for navigation, let's begin by exploring the fundamental concepts of ROS and how it operates. ROS has a graph-like structure that comprises nodes and their relationships. A *node* in ROS is an executable that communicates with other nodes using ROS protocols. A *message* is the data that a node can send to or receive from other nodes. Nodes can publish messages to *topics*, which are referred to as publishers. Conversely, nodes can also subscribe to topics, known as subscribers. Publishers send messages to a topic, and subscribers that are subscribed to that topic receive messages from the publisher. For example, the robot can determine its speed and orientation based on the information it receives from its wheels.

Every ROS application must start with the initialization of the ROS master, which manages all nodes and enables them to communicate with each other. Without the ROS master, other nodes cannot function. This means that if the ROS master is accidentally closed while there are nodes running in ROS, those nodes will also stop operating. The command `roscore` is used to run the ROS master. Once the ROS master is running, nodes can be launched using the command `rosrun`. Examples of nodes include a simulation node that executes the simulation program, a controller node used to manage the robot in the simulation, and a graph node that opens a graphical interface displaying the workflow or the messages exchanged between nodes.

Navigation in ROS

For the robot to be able to navigate in its environment, it must accomplish five key tasks:

1. The robot needs a global map.

2. The robot must determine its location on the map.
3. The robot needs to be able to move around using its map.
4. The robot must be able to avoid obstacles that are not represented on the map.

These tasks collectively form what is known as the Navigation Stack, which enables the robot to autonomously navigate from one point to another while avoiding obstacles in its path. The Navigation Stack takes inputs such as the robot's current location, the target destination it must reach, and data from sensors used to ascertain its position. In turn, it provides velocity information as output, instructing the robot on how to move safely toward the goal position.

Mapping

The robot can obtain a map of its surroundings through sensors. In this project, LiDAR will be used as a laser sensor. It functions by emitting laser beams in multiple directions. When a laser beam strikes a surface, it reflects back to the sensor. The sensor then calculates the distance from itself to the surface by measuring the time it takes for the laser beam to return, as described by Eq.1.

$$Distance = (Speed\ of\ Light \times Time) \ / \ 2 \qquad (1)$$

Gmapping is a ROS package that enables the robot to create a 2D map of its surroundings using SLAM (Simultaneous Localization and Mapping). SLAM is an algorithm used to construct a map of an unknown area while simultaneously tracking the robot's location on that map. The gmapping package includes a *slam_gmapping* node, which subscribes to the *scan* topic to receive laser information from the sensor and publishes the *map* topic to allow other nodes to access the generated map. Essentially, gmapping reads data from LiDAR and converts it into a 2D map.

The generated map is temporarily stored in the robot's cache. To save the generated map permanently to a file, the *map_saver* node from the map_saver package is used. Additionally, the *map_server* node from the same package is used to read the map file and publish it via ROS to other nodes that require map data.

For accurate map generation, the configuration of the robot must be specified. This includes the position of the LiDAR sensor on the robot, whether it is placed on top, in front, or at any

other location. Without proper configuration, the robot cannot accurately determine its position, as the origin of the map is based on the LiDAR sensor rather than the robot itself.

The robot must move around to generate a map of its surroundings, which typically needs to be done only once, provided that the robot's environment remains consistent. However, a limitation of this approach is that the generated map is static. The robot cannot navigate into areas not previously included in its map, meaning it cannot enter spaces it has never explored before. Additionally, if the robot's environment changes significantly, the existing map becomes invalid, requiring the robot to generate a new map of its surroundings.

Localization

After the robot has generated a global map of its surroundings, the next step is for it to determine its position and orientation on that map simultaneously. This can be achieved using the LiDAR sensor again, but this time not to generate a new map, but rather to match the detected surfaces with the global map.

The robot estimates its position by comparing the surfaces detected by the LiDAR sensor to the known global map. It calculates its position and orientation as a probability distribution over possible locations, known as a point cloud, on the map. As the robot moves and gathers more data about its surroundings, it refines this estimate by filtering out positions that do not match the sensor information. Over time, the probability distribution converges to the most likely position of the robot. This process is known as the Monte Carlo Localization (MCL) algorithm, also referred to as particle filter localization [8].

The AMCL (Adaptive Monte Carlo Localization) package provides the *amcl* node, which uses the MCL algorithm to determine the robot's location on the map. The node subscribes to both the *scan* and *map* topics to receive laser and map data, and it publishes the *amcl_pose* topic, which contains the estimated position and orientation of the robot on the map.

Path Planning

In ROS, the move_base node from the move_base package will find the path for the robot to move from its current position to the goal position. The node subscribes to the *goal* topic to receive the goal position, and publishes the *cmd_vel* topic that stream the velocity value for the robot to command on its wheels or other parts for it to move.

In ROS, the *move_base* node from the move_base package is responsible for finding the path for the robot to navigate from its current position to a designated goal position. This node subscribes to the *goal* topic to receive the target location and publishes the *cmd_vel* topic, which streams the velocity commands necessary for the robot to move.

The *move_base* node utilizes the global path planner library from the nav_core package to formulate the robot's path, referred to as the global plan. Before the robot begins its movement, the global planner calculates a global costmap based on the global map and determines the optimal path known as the global plan.

The costmap is essentially a grid divided into cells, with each cell assigned a value between 0 and 255. A value of 0 indicates that the area is safe to move through, while higher values represent an increased likelihood of collision, with 254 indicating a certain lethal collision. Because costmaps function similarly to weighted graphs, the most optimal path between two points can be determined using A* and Dijkstra's algorithm.

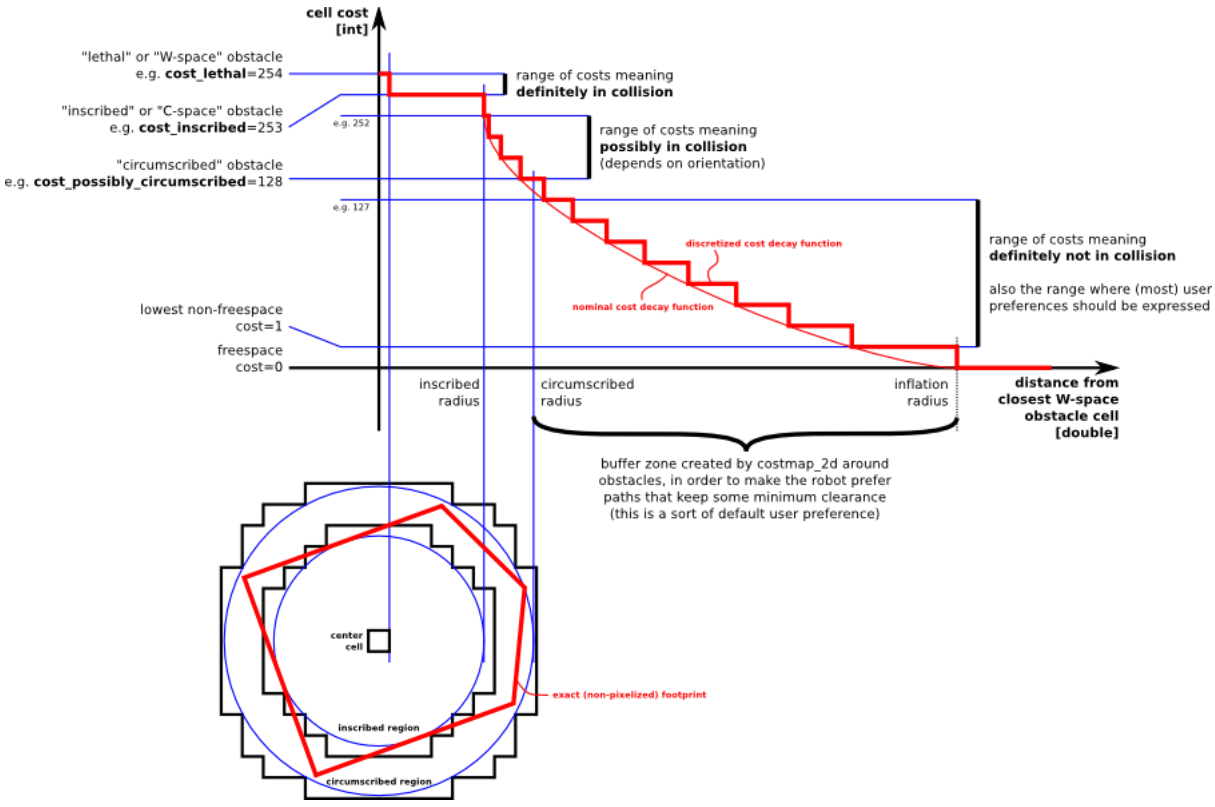

Fig 5 - Inflation of Costmap Values

Obstacle Avoidance

The robot must be capable of avoiding obstacles that may appear in its path, which were not originally present on the global map. These obstacles may include a person walking in the area or a piece of furniture placed after the robot has generated its map.

To safely avoid collisions, the robot needs to understand its own boundaries. This configuration is provided to the robot through a URDF (Unified Robot Description Format) file, which contains information about the various parts of the robot. The URDF defines the connections between each part and their dimensions, allowing the robot to calculate its own boundary for collision avoidance.

The *move_base* node also includes a local path planner that generates a local costmap and computes a local plan. This planner segments small portions from the global plan and utilizes the robot's sensor data to recalculate a new path, ensuring that it can navigate around obstacles. The output of the local path planner consists of velocity commands, which are sent to the robot's controller and translated into the robot's movements.

Simulation using Gazebo

This work in this section is under the responsibility of and is worked on by Yossaphat Kulvatunyou.

The objective of this project is to simulate the behavior and performance of a rocker-bogie rover within the Gazebo simulation environment. The rocker-bogie mechanism, developed by NASA-JPL for planetary exploration, is well-suited for traversing challenging, uneven terrain. This simulation focuses on evaluating the rover's kinematics and its ability to handle various terrains—such as concrete, rubber, and grass—within a simplified environment. The project utilizes a passive suspension system, meaning there is no additional control algorithm or sensors to dynamically adjust the angles of the rover's joints. As a result, the rover relies entirely on its mechanical structure to navigate terrain without any active intervention, limiting the simulation to purely mechanical responses.

The decision to implement only a passive suspension system simplifies the simulation while still allowing for a thorough evaluation of the rover's locomotion. The absence of control algorithms reduces complexity but places greater emphasis on the physical design of the rover's joints, friction parameters, and damping values to achieve stability and performance on rough surfaces. This design aims to reflect real-world mechanical behavior without relying on automated adjustments or control logic.

Gazebo, a widely-used robotics simulator, has been employed to model and test the rover. The simulation work began with a simpler model, involving the creation of a two-wheel robot within Gazebo’s edit/build mode. This robot included integrated sensors from the Gazebo library, allowing it to autonomously follow the closest object in the environment. These initial experiments facilitated a deeper understanding of Gazebo’s joint and link mechanisms, such as the use of parent and child link joints to connect the chassis to the wheels. Key parameters, including collision properties, visual elements, and inertia values, were modified to observe their impact on the simulation and ensure realistic interactions within the environment.

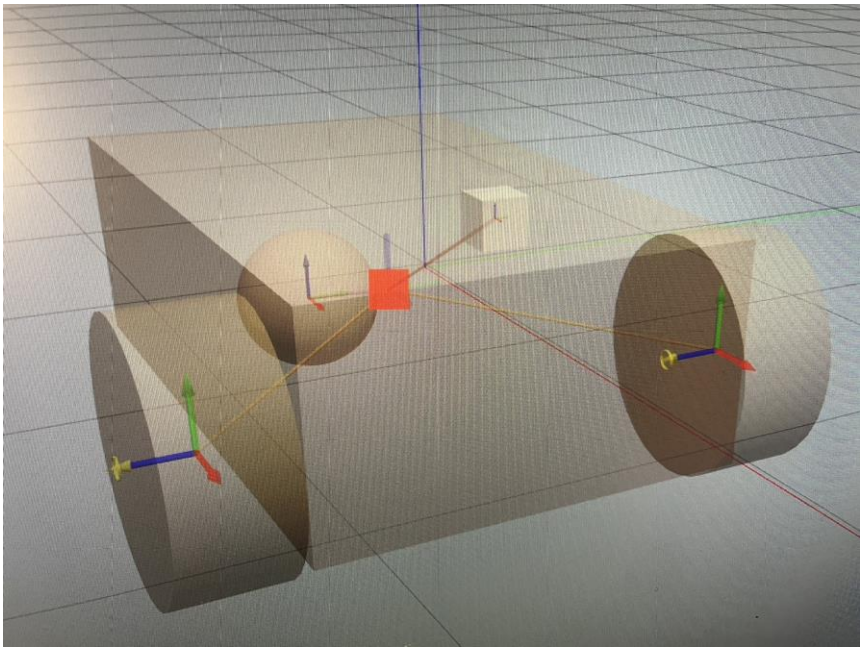

Fig 6 - Two wheels robot link

Additionally, preliminary work was carried out to simulate sensor models, although they were not intended for the final rover design. These exploratory simulations provided essential insights into the Gazebo environment, helping to refine the understanding of joint behaviors, physical parameter adjustments, and sensor integration. Although the models tested in these early phases differ from the rocker-bogie system, they were valuable in building a foundational understanding of Gazebo's simulation capabilities.

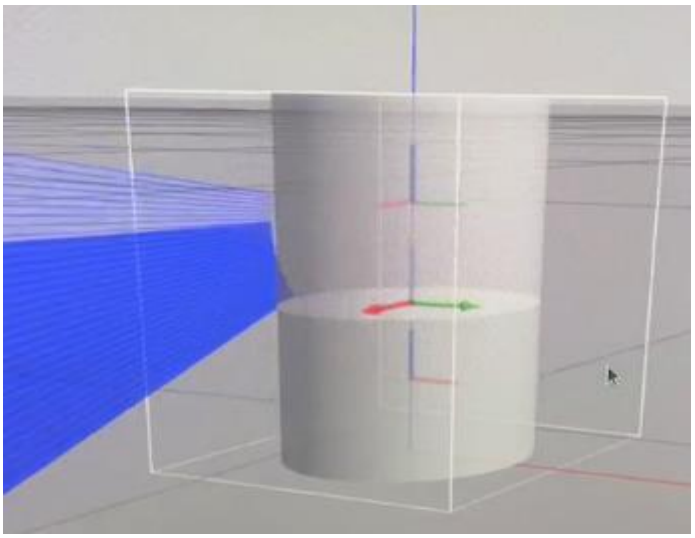

Fig 7 - LiDAR simulation

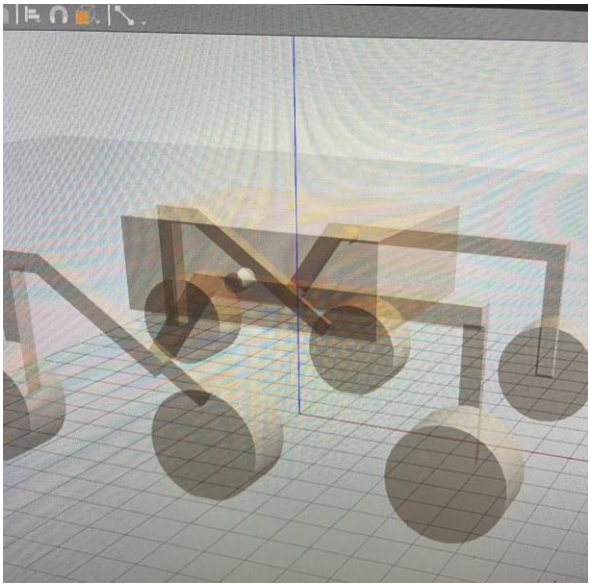

Fig 8 - Rover design from CAD in Gazebo

The next step in the rover project involves constructing the complete rover model in Gazebo by defining and configuring the necessary joints to connect the wheels, chassis, and rocker-bogie linkages. This includes setting up appropriate `revolute` or `continuous` joints for the wheels and `fixed` or `prismatic` joints for the linkages as required. A critical focus will be on ensuring model stability by correctly tuning the physical parameters such as mass, inertia tensors, and center of mass (CoM) for each link. This will ensure the model behaves stably under gravity, especially on flat terrain.

Further tasks involve integrating wheel dynamics to allow actual movement, either by utilizing Gazebo's built-in `ODE` physics engine for motorized joints or by implementing sensor-driven locomotion using a camera sensor for object tracking and autonomous movement. The plan might also involve implementing steering control using a velocity or position PID controller if time permits.

Reference

[1] P. Iakubovskii, "Segmentation Models Pytorch" GitHub, 2021. [Online]. Available: https://github.com/qubvel-org/segmentation_models.pytorch. [Accessed: Sep. 11, 2024].

[2] Lightning AI, “Pytorch-lightning,” GitHub 2024. [Online]. Available: https://github.com/Lightning-AI/pytorch-lightning. [Accessed: Sep. 11, 2024].

[3] A. Kirilov, K. He, R. Girschick, P. Dollár, “A Unified Architecture for Instance and Semantic Segmentation”. [Online]. Available: http://presentations.cocodataset.org/COCO17-Stuff-FAIR.pdf. [Accessed: Sep 11, 2024]

[4] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 9351, pp. 234-241, 2015. Available: https://arxiv.org/abs/1505.04597.O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 9351, pp. 234-241, 2015. Available: https://arxiv.org/abs/1505.04597.

[5] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, “Encoder-decoder with atrous separable convolution for semantic image segmentation,” in *Proc. European Conf. Computer Vision (ECCV)*, 2018, pp. 801–818. Available: https://arxiv.org/abs/1802.02611.

[6] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 3213-3223. Available: https://arxiv.org/abs/1604.01685.

[7] Jet Propulsion Laboratory. (2024). JPL Open Source Rover Project. September 22, 2024. Available: https://open-source-rover.readthedocs.io/en/latest/

[8] F. Dellaert, D. Fox, W. Burgard and S. Thrun, "Monte Carlo localization for mobile robots," Proceedings 1999 IEEE International Conference on Robotics and Automation, Detroit, MI, USA, 1999, pp. 1322-1328 vol.2, doi: 10.1109/ROBOT.1999.772544. Available: https://www.ri.cmu.edu/pub_files/pub1/dellaert_frank_1999_2/dellaert_frank_1999_2.pdf