Robotics: Group Project

Names, IDs:
Sandra Guran, 201538092
Ayesha Rahman, 201522771
Natalie Leung, 201562361
Geeyoon Lim, 201553023

Table of Contents

INTRODUCTION

1.1 Aim

This assignment aims to develop a prototype of a solution to a system capable of determining the precise location of a spacecraft travelling from Earth to the Moon, which has been knocked off course due to an explosion in one of its two modules.

1.2 Objectives

1. Reaching the entrance and centre points of both modules.
2. At the entrance, look for green or red signs. If red, don't go into the room; go to the centre if green.
3. Navigate the room to detect windows.
4. Align the robot to the window and take a screenshot.
5. Stitch the screenshots together to create a panorama.
6. Detect Earth, Moon, Mars, and Mercury from the panorama.
7. Calculate the distance to locate the robot.

1.3 Overview of Solution

The proposed solution for the spacecraft navigation and detection system involves a multi-faceted approach to ensure precise location determination and navigation of a spacecraft's robotic unit. This system is designed to handle the spacecraft's unexpected deviation from its course due to an explosion in one of its modules. The solution comprises several key components:

1. Navigation Initialisation and Input Coordinates: The system begins by initialising the coordinates for the robotic unit, allowing it to navigate between specific points within the spacecraft. These coordinates are dynamically adaptable via a YAML file, enabling the robot to navigate different environments effectively.
2. Red and Green Sign Detection: At the entrances of the spacecraft modules, the robot detects red and green signs using image processing techniques. A green sign indicates safe entry, prompting the robot to proceed to the room's centre, whereas a red sign signals the robot to avoid the area.
3. Wall Navigation and Obstacle Avoidance: The robot employs LaserScan data to navigate along walls and avoid obstacles. This involves real-time distance measurement and PID controllers to maintain a consistent distance from walls and ensure smooth navigation.
4. Window Detection and Heuristic Prediction: The robot uses heuristic methods to predict potential window locations within the room. These heuristics are based on typical window placements, allowing the robot to locate windows efficiently before visual confirmation.
5. Screenshot and Image Stitching: Upon identifying a window, the robot aligns itself and captures screenshots. These images are then stitched together using a blending method to create a panorama, which is subsequently used for further analysis.
6. Planet Detection and Distance Measurement: The stitched panorama is analysed to detect celestial bodies such as Earth, Moon, Mars, and Mercury. The system measures the distances between these planets using their detected positions and sizes in the images, providing accurate spatial data.
7. Machine Learning Model for Planet Detection: A machine learning model, trained on a dataset of planetary images, is employed to identify different planets in the captured images. This model uses ResNet18 for robust and accurate classification.
8. Testing and Validation: The system undergoes rigorous testing in simulated and real-world environments. Different versions of navigation and detection algorithms are tested to ensure reliability and accuracy under various conditions.

By integrating these components, the solution provides a comprehensive system capable of autonomous navigation, obstacle avoidance, and precise location determination within the spacecraft. This robust approach ensures the robot can effectively navigate and perform its tasks even in challenging and dynamic environments.

DESIGN

2.1 Identifying Sub-Tasks

The development of the spacecraft navigation and detection system is divided into several key sub-tasks, each focusing on a specific aspect of the overall solution. This section outlines the primary sub-tasks, detailing each component's responsibilities and implementation strategies.

2.1.1 Input coordinates - done by Ayesha Rahman

The project began by initialising points for the robot to navigate around the modules. This initial step was crucial for enabling the robot to follow a defined path, laying the foundation for its navigation capabilities and understanding its environment. The RoboNaut class, inheriting from Node, initiates a ROS 2 node and declares a parameter to accept a YAML file containing target coordinates (entrance 1, centre 1, entrance 2, centre 2) for both rooms in the spacecraft. This allows dynamic adaptation to different environments by simply changing the YAML file. Additionally, it utilises ROS 2's action framework to send navigation goals to the nav2 navigation stack, enabling autonomous movement towards target locations.

A navigation sequence is a list of lambda functions encapsulating navigation commands for each point. This sequence dictates the order of navigation and proceeds automatically. The navigation logic includes functions such as 'Navigate to Next Point', which increments the index tracking the robot's position and invokes the following navigation command, and 'Navigate to Point', which sends a navigation goal to move the robot to a specific (x, y) coordinate by constructing a PoseStamped message. Goal response and result callbacks handle the acceptance and results of navigation goals. The result callback triggers navigation to the next point, allowing the robot to proceed autonomously. A feedback callback provides real-time feedback on the robot's current position, which helps monitor progress and debugging.

Later, the centre coordinate points of both rooms were removed as the project shifted focus to navigating along walls and predicting windows. These centre points are now used to calculate a fallback point in the room if the robot cannot predict windows or doesn't see any in its camera feed, as explained in the window detection using heuristics section.

2.1.2 Red and Green sign detection at entrances - done by Ayesha Rahman

In the next phase of our project, the team focused on developing a system for detecting red or green signs at the entrances to rooms. The robot's behaviour upon detecting these signs was defined as follows: if a green sign was seen, the robot would navigate to the centre of the room; if a red sign was seen, the robot would either move to the next entrance or stop, depending on its current entrance. While Sandra and Natalie initially worked on this task, their code faced some challenges. So through pair programming, ultimately refined the code, ensuring its robust functionality Ayesha managed to complete it.

The RoboNaut class inherits from Node and initialises essential components such as the CvBridge for image processing and the action client for navigation. It subscribes to camera images and sets up parameters and state variables to manage the robot's navigation and sign detection capabilities. The camera_callback method processes incoming images, applying colour filters to detect red or green signs. It uses contour detection to identify the presence of these signs based on specific criteria, such as area and circularity. When a sign is detected, the handle_sign_detection method is triggered, determining the appropriate response—either navigating to the centre of the room for a green sign or handling the red sign by initiating a rotation to check for green signs again.

The robustness of the system is enhanced through dynamic parameter handling, allowing easy adaptation to different environments by updating parameters without changing the core code. The action client in the ROS 2 framework facilitates asynchronous navigation goal handling, ensuring smooth and responsive movement even in dynamic environments. The system includes feedback mechanisms and error handling, particularly in image conversion processes, to maintain stability and provide real-time monitoring and adjustments.

A key aspect of robustness is the comprehensive colour filtering and contour analysis. The color_filter method creates HSV masks for detecting red and green colours, while the detect_sign method evaluates these masks to confirm the presence of signs based on their shape and size. This multi-step verification minimises false positives and ensures reliable detection. Specifically, the parameters used for shape and contour detection include area and circularity. The code looks for contours with an area greater than 500 and a circularity between 0.7 and 1.2. Circularity is calculated using the formula $\text{circularity} = \frac{4 * \pi * \text{area}}{\text{perimeter}^2}$. These parameters ensure that only sufficiently large and circular objects are considered as potential signs.

If a red sign is detected, the robot performs the following steps: Rotate the robot to scan for green signs after detecting red signs ->After rotation, reprocess the latest image data to check for a green sign -> Proceed to the centre of the detected entrance if a green sign is found -> If no green sign is detected, move to the next entrance.

If a green sign is detected, the robot follows these steps: Proceed to the centre of the detected entrance -> If the green sign is detected after rotation, proceed to the centre of the entrance -> If the green sign is detected before rotation, proceed to the centre of the entrance and increment the entrance number.

2.1.3 Wall Navigation and avoiding objects/obstacles using LaserScan - done by Ayesha Rahman

The next phase of our project involved enabling the robot to navigate walls and avoid obstacles using LaserScan data. This crucial functionality ensures the robot can manoeuvre safely and efficiently in its environment. This system was rigorously tested across four different worlds—easy, moderate, hard, and a custom-designed environment—to ensure its robustness and reliability. The RoboNaut class, inheriting from Node, initialises various components and subscribes to the LaserScan topic to receive real-time distance measurements from the robot's surroundings. These measurements are critical for detecting obstacles and maintaining a safe distance from walls.

2.1.3.1 Key Components for Wall Navigation and Obstacle Avoidance

The class subscribes to the LaserScan topic (/scan), receiving data on the distances between the robot and nearby objects. This data is stored in the laserscan attribute and is used to inform navigation decisions. To ensure reliable data communication, we used a QoS profile. These settings ensure that the robot receives all critical LaserScan data reliably, maintaining efficiency and accuracy in navigation:

qos_profile = QoSProfile(reliability=QoSReliabilityPolicy.RELIABLE, history=QoSHistoryPolicy.KEEP_LAST, depth=10)

- ReliabilityPolicy.RELIABLE: Ensures that all messages are received without loss, which is crucial for navigation data.
- HistoryPolicy.KEEP_LAST: Keeps only the most recent messages to manage memory usage.
- depth=10: Maintains the last 10 messages in the queue.

To maintain a consistent distance from walls and ensure smooth, controlled movement, the class utilises two PID controllers—pid_lat for lateral (side-to-side) control and pid_lon for longitudinal (forward) control. The PID controllers are initialised with specific gain values:

- **pid_lat (Lateral Controller):** kP=0.5, kI=0.0, kD=0.2, kS=10
- **pid_lon (Longitudinal Controller):** kP=0.1, kI=0.001, kD=0.05, kS=10

These gain values were chosen based on extensive testing to balance responsiveness and stability, ensuring the robot maintains a steady path along walls without excessive oscillations or sluggishness. The robot_controller_callback method is periodically invoked (every 0.1 seconds) to process LaserScan data and adjust the robot's movements accordingly. This method calculates errors based on the desired and actual distances from the walls and obstacles, then computes control signals using the PID controllers. The robot_controller_callback function checks the front, left, and right LaserScan readings to determine the robot's position relative to walls and obstacles.

The robot follows walls by maintaining a specific distance from them, as indicated by the LaserScan data. The desired distance from the wall is set to 2.0 metres. The control logic calculates the cross-track error (CTE) as the difference between the left and right errors, which represent deviations from the desired distance. The PID controller then uses this error to adjust the robot's angular velocity, ensuring it stays on course.

When an obstacle is detected within 1.0 metre in front of the robot, the avoid_obstacle method is triggered. This method initiates a sequence where the robot first backs up slightly and then determines the direction with more space (left or right) based on the LaserScan data. The robot turns in the chosen direction to avoid the obstacle and then resumes forward movement. The backup distance and turn duration were fine-tuned through testing, with the robot backing up at -0.2 metres per second for 1 second and turning at $\pm\pi/6$ radians per second for 1 second, before moving forward again at 0.2 metres per second for 2 seconds.

Several helper functions support the main logic:

- **get_valid_scan:** This function filters out invalid scan data (e.g., infinite or NaN values) and returns a safe high value (3.5 metres) if no obstacle is detected.
- **rotate_for_duration:** This function rotates the robot for a specified duration and speed, helping it reorient when necessary. For example, rotating at 0.2 radians per second for 90 degrees.
- **shutdown_procedure:** This function saves a heuristic map of the robot's path and performs clean-up actions upon shutdown.

It should be noted that we used a reference [1] for this part of the code using LaserScan for the wall navigation which we tuned to our robot and added additional features. The wall navigation is later changed during the project to add grid navigation heuristic predictions used for window detection. Here the LaserScan is used to avoid obstacles and navigate along the wall when the window is closer to the wall so as to not collide while the robot aligns itself to the centre of the window. This has been explained in detail in the **robot alignment** section of the report.

Issues with LaserScan

We encountered several issues while working with LaserScan data. In the simulation, the scan data included both finite and infinite values. As previously mentioned, the robot ignores the infinite values and only considers the finite ones. However, when we tested on the real robot instead of the simulation, we found that the laser data received was in a different format with empty values. Another issue was that the real robot's LiDAR sensor rotates at varying speeds and can sometimes change its rotation speed. This variability caused the LaserScan data to be unreliable, resulting in the robot

coming to a standstill. One potential solution to this problem could be extending the LaserScan data beyond the received data for more accurate predictions. Although we were unable to implement this, the method is explained in detail in the **LaserScan Stretching** section of the report.

2.1.4 Window Detection - done by Sandra Guran

Once the robot is in the green module, it starts looking for the window. To detect the window, it looks for the contour of a polygon and checks for 4 points to determine if it's a rectangle. Next, it checks for a white border to differentiate the window from other rectangular objects like trash bins. A white threshold method is used to find the border. Additionally, it checks the black ratio of the area of the detected rectangle, as all usable windows are predominantly black. This way we assure that the posters with multiple planets will not be accidentally detected. When all conditions are met and the camera of the robot sees: a rectangle, a white border, and the rectangle has more than half black ratio; the robot sends a message that the window is seen. When the window is out of view or only a small fraction of it is seen, it sends a message that the window is not visible.

2.1.5 Window Prediction using heuristics - done by Ayesha Rahman

The window prediction phase of our project aimed to enhance the robot's capability to predict and locate windows within a room using heuristic methods. This task was essential for improving the efficiency and effectiveness of the robot's navigation system, particularly in environments with multiple potential window locations.

The heuristic approach for window prediction relies on predefined assumptions about typical window placements in a room. By leveraging this knowledge, the robot can anticipate potential window locations even before visually confirming them. This predictive capability allows the robot to navigate more efficiently, reducing the time and computational resources spent on searching for windows.

In our implementation, the prediction function uses heuristic rules to estimate window positions based on the room's layout. The heuristic is based on the assumption that windows are typically placed uniformly along the walls of the room. Specifically, we predict four primary window locations, one on each wall, assuming the room is approximately square. For instance, if the room's centre is known, the heuristic places windows at fixed distances along the room's left, right, top, and bottom walls. The room size was assumed to be 5 metres by 5 metres for simplicity.

The predicted window locations are as follows:

- Left wall: (room_center.x - room_size / 2, room_center.y)
- Right wall: (room_center.x + room_size / 2, room_center.y)
- Bottom wall: (room_center.x, room_center.y - room_size / 2)
- Top wall: (room_center.x, room_center.y + room_size / 2)

Once these potential window locations are predicted, the robot navigates to each predicted spot sequentially, checking for the presence of windows using its camera and image processing capabilities. The camera callback function processes the images to detect windows based on contours and other image processing techniques. For example, it identifies rectangular shapes with specific characteristics, such as having a white border and a significant black area, indicative of a window.

When a window is detected, the robot aligns itself properly using its PID controllers to ensure it is positioned correctly relative to the window, allowing for an optimal screenshot to be taken. This process involves adjusting the robot's orientation and position based on the window's detected coordinates and angle.

The heuristic approach to window prediction enhances the robot's ability to efficiently navigate and locate windows by leveraging predefined assumptions about their placement. This method allows the robot to preemptively direct its movements towards likely window locations, improving its navigation efficiency and reducing search time.

2.1.6 Robot alignment to window and screenshots - done by Ayesha Rahman

The robot's alignment to windows and capturing screenshots is a critical task that involves several steps to ensure accurate documentation and efficient operation. Initially, the robot processes incoming images through the camera_callback method. This method converts the raw image data using CvBridge and applies colour filtering and contour detection to identify windows as mentioned earlier in section Window Defining. It looks for specific characteristics, such as the presence of a white border and a significant black area within the contour, to confirm the detection of a window. When a window is detected, its coordinates and angle are stored for further processing.

Upon detecting a window, the robot begins the alignment process. The approach_window method initiates this by moving the robot towards the window while continuously checking its position using LaserScan data to maintain an optimal distance. If the detected window is at an angle, the robot adjusts its orientation using the adjust_orientation method to align perfectly with the window. This involves calculating the required rotation angle based on the window's coordinates and the robot's current position.

Once the robot is aligned, it ensures it is at the correct distance from the window by using the check_distance_and_take_screenshot method. This method calculates the distance to the window and makes fine adjustments if necessary. It ensures the robot is positioned precisely 2.0 metres away from the window, allowing for an optimal view. If the robot is not at the correct distance, it adjusts its position by moving closer or further away until the desired distance is achieved.

Finally, the robot captures the image of the window. The take_screenshot method crops the image around the detected window area and saves it to a specified directory. This step involves locking the current image data to ensure

that the captured screenshot is accurate and corresponds to the detected window. The robot logs the capture and saves the image with a unique filename to prevent overwriting previous screenshots.

This systematic approach ensures that the robot can reliably detect, align, and document windows within the spacecraft. The precise alignment and controlled distance management are crucial for capturing high-quality images, which are essential for creating accurate panoramas and further analysis.

2.1.7 Initial Planet Stitching - done by Sandra Guran, Geeyoon Lim

We first did stitching using the FLANN method from Lab 5, with the following steps : initialising the SIFT detector, applying the FLANN-based matcher, using Lowe's ratio test, and computing at least 4 matches to compute homography. However, in the panorama one of the planets would become elongated and deformed compared to its perfectly round shape, as shown in Figure 1.1. Therefore, we tested and used a different method.

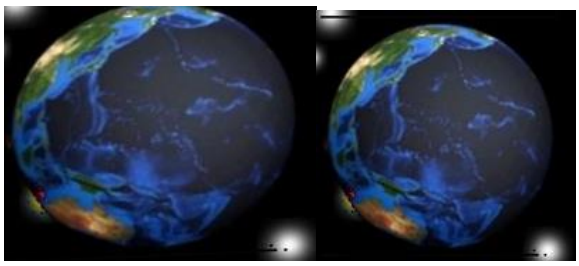

Figure 1.1: The first image shows the deformed method using FLANN, and the second one shows a good image using the Blended Image method.

The method we used involves direct image manipulation and blending rather than feature-based stitching[3]. The images are loaded, and we crop the image with majority black and stars from the white border. We check for image resizing if the heights of the two images are not equal. The images are directly placed next to each other on a new canvas (an array of zeros). The width of the new canvas is calculated to be the sum of the widths of the two images minus the width of the overlapping region.

A linear blend is created for the overlapping region where the two images meet. This blending is achieved by using a linear gradient (fade) to smoothly transition from one image to the other across the overlap width. This gradient is applied separately to each colour channel (RGB), combining the corresponding areas of the Earth and Moon images. The formula used for blending is: □□□□□□□ □□□□□ = □□□□*h* □□□□□ × □□□□ + □□□□ □□□□□ × (*1* − □□□□) , where "Fade" is a linear gradient going from 1 to 0 across the overlap width. The remainder of the second image is placed next to the blended region to complete the panorama.

2.1.8 Planet Detection - done by Sandra Guran

At this stage, the images have already been stitched, with the possibility of containing 2-4 planets: Earth, Moon, Mars, and Mercury. The detection algorithm begins by locating the planets in the big picture and cropping a square around each planet from within the window, an example of a cropped planet is Fig. 1.1. These cropped images are then processed by the model to identify each planet, and the respective window is named viewPlanet_name. The detection involves searching for circular shapes in the window and storing their coordinates (x, y). For example: Moon was detected at (x=1880, y=538) with a height of 168 pixels.

The detected planets are highlighted in the final image, check Fig. 1.2, with different colours outlining each planet. This visual confirmation ensures accurate detection, because later in the measurement formula we will use the exact height of the planet in pixels to determine the distance in km.

2.1.9 Planet Measurements - done by Sandra Guran

To calculate the distance to each planet, we use the formula explained in the Computer Vision lecture: Stereo & 3D Perception I: $\text{□□□□□□□□□\_□□} = \frac{\text{□□□□□\_□□□□□}h \times \text{□□□□\_□□□□□□□□□\_□□} \times \text{□□□□□\_}h\text{□□□}h\text{□\_□□□□□□}}{\text{□□□□□□\_}h\text{□□□}h\text{□\_□□□□□□} \times \text{□□□□□□\_}h\text{□□□}h\text{□}}$, where planet_diameters = {'Earth': 12742, 'Mars': 6794, 'Mercury': 4878, 'Moon': 3475}, are real life planet diameters; sensor_height = 3, focal_length = 3, standard values. This formula calculates the distance by considering the real diameter of the planet and scaling it based on the image and sensor dimensions. This will output the following result in the log, for example, "Moon was detected at (x=1880, y=538) with a height of 168 pixels and a distance of 23,559.67 km."

For calculating distances between planets, the formula used is: $\text{□□□□□□□□} = \sqrt{(\square_2 - \square_1)^2 + (\square_2 - \square_1)^2}$, where x, y are coordinates of the planets compared. This formula calculates the Euclidean distance between the coordinates of two planets. If the image contains only two planets, the program outputs their distance directly in the

requirements.txt file. Additionally, we have considered the case where there are more than two windows stitched together, so we calculate the distance and output the distance between each pair, making our method more robust and reusable.

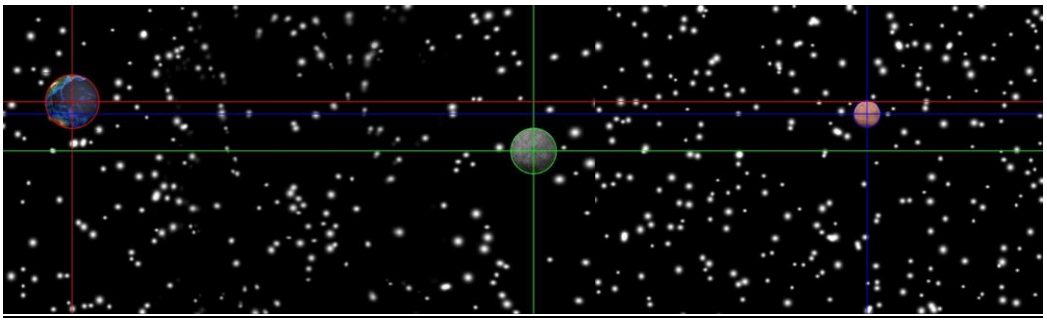

Figure 1.2. A panorama image with 3 detected planets: Earth, Moon, Mars

Fig 1.2 Output Example of background code information:
Earth detected at (x=3583, y=356) with a height of 198 pixels and a distance of 73,298 km.
Moon detected at (x=1880, y=538) with a height of 168 pixels and a distance of 23,559 km.
Mars detected at (x=649, y=401) with a height of 98 pixels and a distance of 78,962 km.
Distance between Earth and Mars: 58,190 km.
Distance between Moon and Mars: 16,538 km.
Distance between Earth and Moon: 45,480 km.
Fig 1.2 Output Example in measurement.txt:
Earth: 73298 km
Moon: 23559 km
Mars: 78962 km
Distance between Earth and Mars: 58190 km.
Distance between Moon and Mars: 16538 km.
Distance between Earth and Moon: 45480 km.

As seen above the program also arranges the planets first seen in the panorama, first written in the output file based on their positions (x, y coordinates) within the panorama image

2.1.10 Reusability of image stitching, main code linking - done by Sandra Guran
After the robot has finished capturing the screenshots of the images in the green module, all of the images window#.png, are cropped to exclude the white border and any potential wall fragments. The screenshots are renamed with the format viewPlanet_name using the process of planet detection explained in section 2.1.8. Once the images are renamed, they are ready to be stitched using the method explained in section 2.1.7. We also set a specific order for stitching the planets: Earth, Moon, Mars, and Mercury. This ensures that the panorama is not randomised, which could lead to incorrect distance measurements. The code is robust enough to stitch more than two images. After stitching, the next step is planet measurements, which detects where each planet is, using the method explained in section 2.1.9. If no images or planets are detected, we have added useful comments to the log to indicate the problem.

Results with Earth and Moon

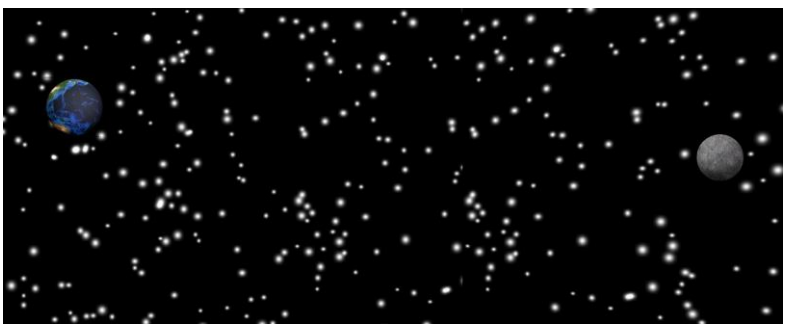

Figure 1.3. A panorama image with 2 detected planets: Earth, Moon

Fig 1.2 Output Example in measurement.txt:

Earth: 91855 km
Moon: 23559 km
Distance: 45480 km

Testing

In terms of testing, we created panoramas with up to four different windows, making all possible image combinations and flipping them to test measurement consistency. We tested on lower-quality images and images showing different sides of the same planet. Our code ensures that if it detects two images of the same planet, it only picks one to stitch with. As seen in section 2.1.9, we have also ensured that more than two planets can be measured and their distances calculated.

This approach is robust compared to only testing on two planets because it covers a wide range of scenarios, including different image qualities and perspectives. Flipping the images and testing for consistency ensures accuracy regardless of orientation. Handling different sides of the same planet tests the algorithm's adaptability. By selecting only one image when duplicates are detected, we avoid redundancy and errors. Ensuring the code can handle more than two planets makes it scalable and adaptable for future expansions. Calculating distances between multiple planets tests the code's ability to handle complex computations reliably. Overall, this comprehensive testing strategy ensures our method is robust, reliable, and accurate under various conditions and scenarios.

Limitations

In section 2.1.7, not using the FLANN-based matcher has the disadvantage of not being able to find common points and stitch the panorama perfectly. However, we prioritised planet measurements over the visual perfection of the panorama. This method is also better suited if the images are captured at an angle. For future work, we plan to test more and improve this aspect.

The planet detection and measurements work perfectly with clear high-resolution windows or even lower resolution images that could have been obtained in the window detection using heuristics, as described in section 2.1.3. However, with the current solution for window detection in section 2.1.4, the images are taken from far away, making it hard for them to be cropped correctly. Due to the very poor resolution, it is almost impossible for the model to detect them accurately. The stitched panorama is based on the cropped windows and planet detection, so the current results in the simulated worlds are not realistic or useful. With better window detection, the process would be much more useful and effective. Improved detection would enhance the accuracy and reliability of the results, making the method more applicable in realistic scenarios.

2.1.11 Data training - done by Natalie Leung

2.1.11.1 Training the model

The machine learning model is used to detect planets. A CNN model using Tensorflow and Keras was first created. However, the model was not compatible with the data. Therefore, we tried to use different approaches to improve the model's performance. Subsequently, we created another machine-learning model using Pytorch. It is a machine-learning library based on the Torch library. Pytorch can handle different deep-learning models, image transformations, and file operations. ResNet18 is used in data training. It is a famous convolutional neural network architecture that has been pre-trained on a large dataset (typically ImageNet) and then fine-tuned for a custom classification task with different output classes.The dataset used for training the model was sourced from the internet and GitHub (https://github.com/emirhanai/Planets-and-Moons-Dataset-AI-in-Space), and it contained images of Earth, Moon, Mars, and Mercury.

To ensure the model's robustness, the data was split into 80% for training and 20% for testing. During the data processing stage, the function applied several transformations to the data. These included colour jitter for brightness and contrast adjustments, resizing the images to 224x224 pixels, Gaussian blur for regularisation by reducing high-frequency noise, and normalisation using specific mean and standard deviation values, typical for models trained on ImageNet.

Some adjustments were also made to the model to align with our specific goal. The fully connected layer was removed and replaced with a new linear layer to adapt the model to a custom number of output classes. A CrossEntropyLoss function is used as the loss criterion, which is suitable for multi-class classification problems. Moreover, the Adam Optimiser was chosen to adjust the weights, which is a popular choice due to its efficiency in dealing with large datasets and parameter spaces.

The training process has the flexibility to perform in different places. It will move to the GPU if it is available. Otherwise, it uses CPU. During the training process, for each batch of data, the model performs a forward pass to generate predictions, calculates the loss, and performs a backward pass to update the model weights. This process is repeated for a defined number of epochs, and the model's state is saved upon completion. The model is then set to evaluation mode, which disables certain layers like dropout for consistent performance. The function iterates the model through the test data, keeping track of the total and correctly predicted instances to calculate the accuracy. After training and testing the model, it is saved as model.pth and ready for future uses.

The training process was divided into 10 epochs, with the loss printed for each epoch. The loss is calculated using PyTorch's Cross Entropy Loss function, which is used for classification tasks. The loss decreased from 0.5434 to 0.1845. The consistent decrease in training loss indicates that the model is learning and the training process is successful.

2.1.11.2 Making prediction

The program loads a trained ResNet18 model to perform predictions. The final fully connected layer of the model is modified to match the desired number of output classes, which correspond to different image categories. During processing the images, the function applies a sequence of transformations to standardise and prepare the image for the model. These transformations include resizing, tensor conversion, and normalisation to match the training data's preprocessing. The image is then passed through the model to predict the class with the highest likelihood. The four classes are Earth, Mars, Mercury and Moon. After making a prediction, it returns the index of the predicted class.

2.1.12 Robust testing

The initial requirements of this task were only to be tested on 2 possible images, Earth and Moon, with outputting the distance between the 2 planets. We have made a robust algorithm of detecting 4 different planets, we have tested the model on panoramas with plus 3 planets, that measures different distances between all planets, we have tested on all possible combinations of planets in a image and have flipped the panoramas to show that our code will output the right order of the planets measurements in the detection.

Limitation:

The blended image method of stitching is effective for scenarios where images have similar lighting and the scene or objects in the images can be aligned without complex transformations. It is simpler and faster than feature-based methods but is less flexible in handling images with large differences in perspective or scale. The linear blending helps create a seamless transition between images, making it particularly useful for creating panoramas where artefacts at the edges would be visually disruptive. If any other planets, for example Jupiter, is added then the model wouldn’t be able to detect what planet that is so it will end up saying one of the 4 already known.The model's testing accuracy is only 92.59%, likely due to the limited availability of training data.

The model may struggle to learn meaningful patterns of different planets, potentially leading to inaccuracies in predicting planets. Also, validation is not implemented, making it challenging to monitor the training performance due to the lack of training data. Moreover, when the screenshot from the robot is far and not clear enough, the robot struggles to predict the difference between Mercury and Moon. When the images are far and not clear, the fine details that could help distinguish between Mercury and the Moon are lost. The model relies on these details to make accurate predictions.

2.1.13 Custom world

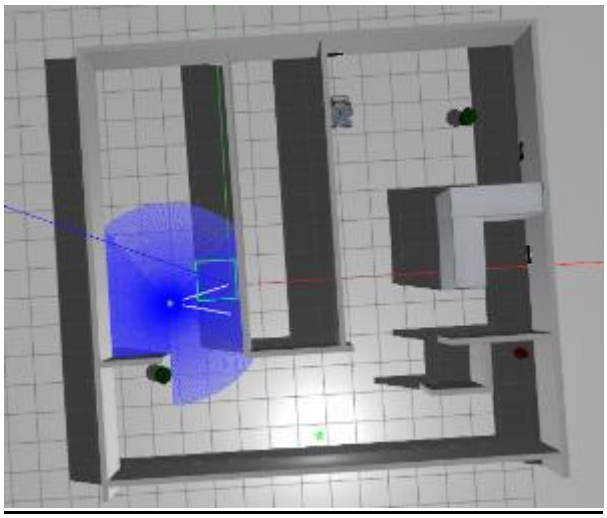

Figure 1.4

The custom world is saved as our_world. The design is based on the moderate world with a few modifications to make it more difficult for the robot to find and reach the windows. In the green room, space boxes are used to create

confined areas, blocking the robot's view of the windows and adding obstacles in front of them to affect the visibility during rotation. Additionally, entrance 2 is surrounded by walls, making it difficult for the robot to detect the sign from a distance.

Therefore, the robot must move close to the entrance to identify the green sign.

In the moderate world, the robot can detect the green sign from a distance and enter the room without needing to navigate directly to the coordinates of entrance 2. However, in this world, the walls around the sign force the robot to go to the coordinates of entrance 2 to detect the sign. It changes the robot's path entering the room compared to the moderate world. After entering our_world, the robot performs 360-degree rotation to capture windows. It successfully captures two of the windows but fails to capture the third window, which is obstructed by the surrounding boxes.

Additionally, if there are obstacles in the centre of the room, the robot will not be able to avoid them. For future improvement, we plan to integrate our current code with laser scan and heuristic algorithms to improve the robot's robustness so that it can use heuristic algorithms to find the third window.

2.1.14 Simulation Videos

Easy, Moderate, Hard and Our Own World Simulation: https://youtu.be/qmPNFvL9QM4

Real Life robot Testing Simulation: https://youtu.be/HHJa7MI64B8

IMPLEMENTATION AND RESULTS

3.1 Real Robot Testing - done by Ayesha Rahman, Sandra Guran, Natalie Leung

The final phase of our project involved testing the system on a real robot. This crucial step was conducted by Ayesha Rahman, Sandra Guran, and Natalie Leung. The testing aimed to validate the performance and robustness of the navigation and obstacle avoidance system in a real-world environment, ensuring that the robot could handle real sensor data and dynamic conditions effectively.

The testing was done with different versions of the code:

1. Simple navigation around the room and taking screenshots of the detected windows
2. Heuristic Grid Navigation using LaserScan

3.1.1 Simple Navigation

In the simple navigation test, the robot performed the following steps:

- It navigated to the entrance and scanned for signs.
- Upon detecting a green sign, it entered the room and moved to the centre.
- The robot then rotated, scanning the room for windows.
- As it rotated, the robot took one screenshot of each detected window.

This version of the code focused on verifying the basic functionality of sign detection, room entry, and window detection, ensuring that the robot could perform these tasks accurately in a real-world setting. This method primarily tested the robot's capability to perform fundamental tasks like detecting specific signals and capturing visual data of the environment. However, it highlighted a limitation: the robot's performance could be compromised by obstacles that obstruct its view, preventing it from detecting and photographing windows. This points to a potential area for improvement in the robot's obstacle detection and handling capabilities to enhance its functionality in real-world environments. Due to this we used LaserScan and Heuristic methods for scanning the room. This has been explained in detail earlier in the report. The following section explains its implementation in the real world.

3.1.2 Heuristic Grid Navigation using LaserScan

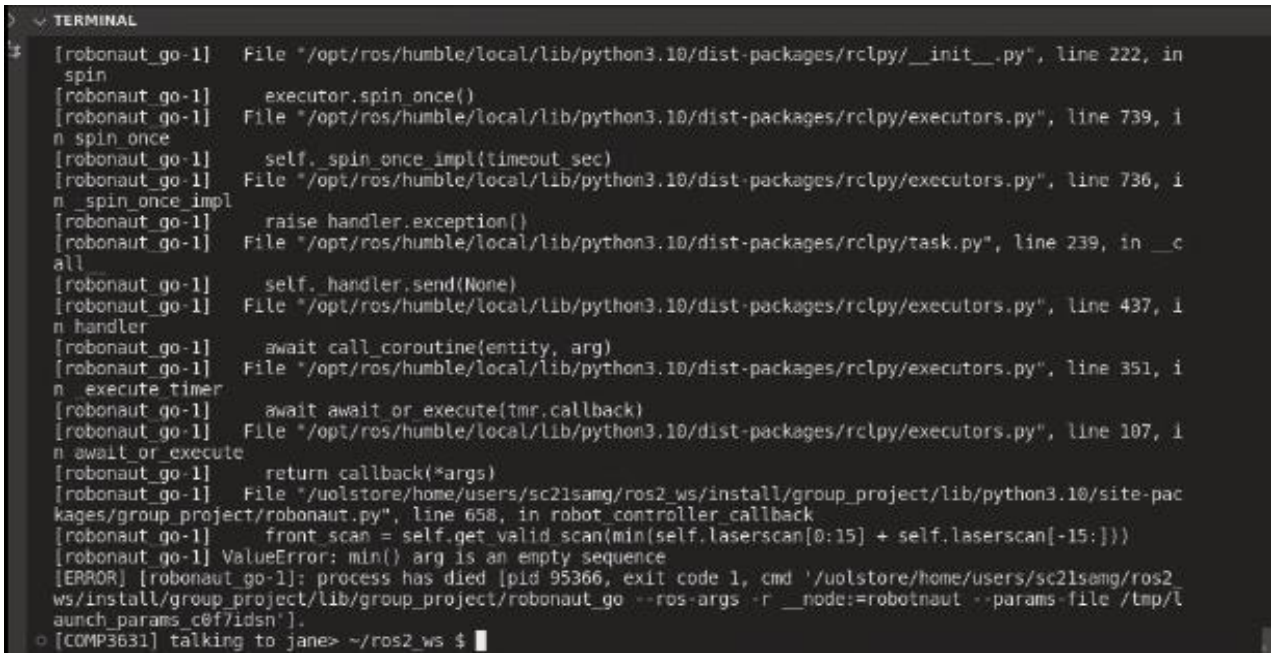

Figure 1.4: Real Robot implementation error

In the heuristic grid navigation test, the robot utilised LaserScan data to navigate more complex environments. This test aimed to evaluate the robot's ability to handle dynamic conditions and navigate using a heuristic grid. Due to the issue mentioned in the report earlier where the data received from the real robot had invalid empty values, one method to fix this is using LaserScan Stretching. Below are the details on how this can be implemented.

3.1.2.1 LaserScan Stretching

One proposed method to enhance the robustness of the LaserScan data processing is "LaserScan Stretching." This technique involves extending the LaserScan data beyond the directly received values to make more accurate predictions and fill in gaps caused by sensor limitations or inconsistencies.

3.1.2.1.1 Concept of LaserScan Stretching

The idea behind LaserScan Stretching is to create a more comprehensive representation of the environment by extrapolating the existing LaserScan data. This extrapolation helps in filling gaps and providing a continuous data set that the robot can use for better navigation and obstacle avoidance.

3.1.2.1.2 Implementation Steps:

1. Data Extrapolation: Extend the finite LaserScan data points by interpolating or extrapolating values to cover areas where data might be missing or marked as infinite. This process involves estimating the distances based on nearby finite values and the known characteristics of the environment.
2. Smoothing Algorithms: Apply smoothing algorithms to the extrapolated data to reduce noise and ensure a more consistent and reliable data set. Techniques like moving average or Gaussian smoothing can be used.
3. Validation Checks: Implement validation checks to ensure that the extrapolated data remains within plausible limits and does not introduce significant errors. These checks can compare the stretched data against known obstacles or features in the environment.

Although we were unable to implement LaserScan Stretching during this project phase, the method offers a promising approach to enhancing the robustness and reliability of LaserScan-based navigation systems. For reference for different methods to do laser stretching, we have looked at the multiple methods used [2].

REFERENCES

[1] Tinker-Twins. 2023. Autonomy Science And Systems Capstone Project. [Online]. [Accessed 20 April 2024]. Available from: https://github.com/Tinker-Twins/Autonomy-Science-And-Systems/tree/main/Capstone%20Project/capstone_project/capstone_project/capstone_project

[2] IRALabDisco. 2023. IRA Laser Tools. [Online]. [Accessed 05 May 2024]. Available from: https://github.com/iralabdisco/ira_laser_tools

[3] Rosebrock, A. 2018. Image stitching with OpenCV and Python. [Online]. [Accessed 24 April 2024]. Available from: https://pyimagesearch.com/2018/12/17/image-stitching-with-opencv-and-python/