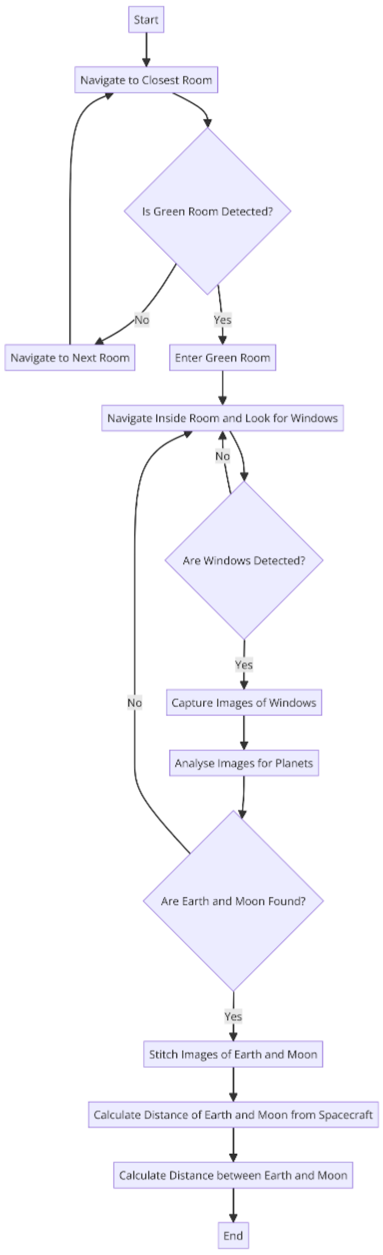
Table of contents

Introduction

In this project, our team designed and implemented a solution that autonomously controls a turtlebot in a simulated environment, alongside real-world testing. The solution involved navigating the turtlebot to the safe room locating the windows with views of Earth and the Moon, and capturing images to calculate the spacecraft's distance using these. Our team came up with various designs in order to solve the problem and after carefully considering potential implementations, we decided to implement a solution that is both robust and accurate. This report provides an overview of the various designs considered, the solution implemented, and the thorough testing and evaluation of the solution.

Project Tasks

Initially, our group came up with a Flow Chart in order to Outline how the proposed solution would work. The flow chart provided clear step-by-step tasks needed to complete the project goals.


Start
Navigate to Closest Room
Is Green Room Detected?
No
Yes
Navigate to Next Room
Enter Green Room
Navigate Inside Room and Look for Windows
No
Are Windows Detected?
Yes
No
Capture Images of Windows
Analyse Images for Planets
Are Earth and Moon Found?
Yes
Stitch Images of Earth and Moon
Calculate Distance of Earth and Moon from Spacecraft
Calculate Distance between Earth and Moon
End

Entering the Green Room: The robot should begin by autonomously navigating around the spaceship to locate the green room. Upon detecting the green room, the robot should enter the room.

Navigating the Green Room: Upon entering the green room, the robot should navigate around the room, while avoiding obstacles in order to locate windows.

Locating Windows: The robot should correctly locate windows within the green room and should capture images of the views outside to analyse them.

Planet Identification: The robot should analyse the captured images to detect the presence of planets, specifically Earth and the Moon.

Image Stitching: If the views from the windows contain both Earth and the Moon, the images should be stitched together to create a panoramic view.

Distance Calculation: Once the panoramic image is created, the distances between the spaceship and Earth, and the spaceship and Moon should be calculated. Lastly, the distance between the Earth and the Moon should be calculated.

Once all the distances were calculated, the robot would have successfully completed its task and the program could be stopped.

By following this structured approach, our team was able to develop a robust and accurate solution for navigating the spaceship, locating the necessary windows, capturing and analysing images, and calculating the required distances.

Design and Implementation

The following section proposes possible solutions to implement the tasks and justifies the chosen solution and how the proposed solution was implemented.

Entering the Green Room

There were multiple ways to implement the green room search, for both locating the buttons and checking their colour. One simple but robust method would have been to check the colour of the buttons for both rooms to prevent any false positives and to make sure the robot gets the correct room, however, as there was a time limit this method was deemed too slow. We instead decided to check a single room and use the result to determine which room was the green one. When detecting the buttons themselves the first idea we had was to look for red and green in the camera's view, however, the presence of objects like red fire hydrants meant that the robot could produce false positives. In this case, we opted to try and look for only single circles of each colour as that would be what is most likely a button, reducing the false positive rate.

When implementing the green room detection algorithm we begin by having the robot move to the entrance of module one, to make testing easier. As stated in our design to look for buttons without knowing which direction the door is in we have the robot spin 2π so that any buttons in the area are seen. The rotation method makes the robot spin at a rate of either positive or negative π/8 until it has reached the given angle.

When detecting the buttons we created a separate Python file to prevent having too much code in a single file. A green and red mask is created and applied to the image from the camera view, and then the image is grayscaled and blurred so that a Hough Circle method can be applied to it to find any green or red circles in the image. Different values are returned from the circle detection method depending if circles are present and what colour they are. This information is then used to decide if the current room is the red room or green room, if it is the red room the robot moves to the green room.

After this we improved the algorithm so that the robot moves to the closest room from its starting position instead of room one, this was calculated from odometry information from the odom topic. This provides the robots current pose so that it can be compared to the two room entrances to find which is closer. This information is not always actuated when compared to the map data so it needs to be transformed using a tf_buffer, this takes a little bit of time to query so the program does nothing for the first few seconds so that it can be correctly calculated.

Navigating the Green Room

The current implementation for navigating the room takes the centre of the room and doubles the distance to estimate the size of the room, this assumption holds true for square rooms but could break if a room uses a non-square solution. Using this information we move to both outer quarters of the room and then spin around. This allows us to try to catch a wide view of the room, circumventing rudimentary obstacles placed directly in front of any windows as we obtain a motion stereo view of the room, which is analogous to a binocular view.

If when trying to navigate to part of the room, an obstacle is encountered, it automatically sets a new valid location, and then attempts to navigate using the nav2 navigation stack. Once it has reached its appropriate position the robot faces the outer wall and proceeds to spin facing the outer wall. This has been made as an optimisation based on the assumption that a window containing either the image of the moon or the Earth does not exist on a wall adjacent to another room.

To understand that the robot is facing the outer wall, we take the midpoint of both rooms and then use this to create a vector facing away from the robot. We can use this information to face away from the robot or to detect when we may be walking towards a poster that is on the inner wall. Any poster on the inner wall could not contain a window with the actual Earth or moon and so this helps to remove redundant poster checks.

One obstacle we encountered when using nav2 goals was that the turtlebot would often start executing movement and rotation code before it had fully reached its navigation goal. This would make our implementation inconsistent, with the turtlebot not always rotating the correct amount or moving prematurely. To combat this, we made it so when the bot reached certain location error bounds, it would pass its current location and orientation as a nav2 goal to overwrite the previous one before continuing execution.

When an image is found within the centre of the bot's view we use lidar sensors to walk towards it in a straight line. The fact the image has to be in the direct centre of the view helps make sure we will not have an obstacle in the way when walking forwards. When sufficiently close enough to the wall / window we stop and snap the photo, then return to the previous location to see if there are any other photos we missed.

Another approach would have been a perfect wall-following algorithm, but we felt this would have taken more time and required stronger window detection, especially from shallow angles. With a certain inefficiency being constantly turning to look at the wall for photo taking, so instead we chose to keep our approach to enter parts of the room and do complete spins, which while not as robust, does take less time and works across moderate situations..

Locating Windows

To find a robust algorithm that could detect the windows in any case and under any circumstances, multiple algorithms were implemented and tested. Specifically, 3 types of rectangle detection were implemented:

- **Single mask detection:** In this algorithm, the image is converted to grayscale, then blurred using Gaussian blur method from cv2 Library. After, a mask is applied to the image to filter the chosen range of intensity. The result is a black and white picture where the white pixels represent the masked range. cv2.findContours is used to store the contours found from the masked image. If a contour is found and has four edges (found using cv2.approxPolyDP) then the bounding rectangle is calculated and if the ratio between the width and the height is more than 0.2 and less than 1.5 then a potential window is found. Although the ratio of the window is roughly 1.5 when the camera is perpendicular to the window, as soon as the camera angle changes, the shape of the window changes too, exhibiting a more squaric form the more angulated the camera is in respect to the window. The ratio between 0.2 and 1.5 allows the window to be detected from more angles. Notably, as the window colours are permanently white and black, a dark mask and light mask were used separately and tested. The results showed that the white mask is affected by the change of light and angle more than the dark mask. However, the dark mask pickles other objects in the room creating more false positives then the light mask. Overall, the dark mask performed better and is more robust.

- **Multiple mask detection:** This detection system is similar to the previous one. However, more masks need to detect the window to ensure a hit. This was done in two ways, however none of the two outperformed the single dark mask detection. First, we checked if any rectangle was found in the dark mask. If that was the case, the same check was done on the white mask. Although this approach was more accurate (fewer false positives than the single mask), slight change in light or camera angle changed one or both masks' detection which impacted the window detection. This could be due to the AND operation associated with the two detections. Hence, if one of the two detection systems did non find a hit, the algorithm would not consider the other as a hit. To avoid this, an OR system was implemented by blending the two masks together using bitwise OR operation. Now the resulting mask had white (255) pixels where originally the colour was extremely dark or extremely light. However, when the masks were combined, the noise was also combined, resulting in the worst detection amongst the three methods.

- **Shape comparisons:** This method looked at the difference in shape between the bounding rectangle and the contour of the detected objects to detect a hit. To achieve that, the contour's x and y coordinates of the centre were compared to the bounding rectangle's x and y coordinates of the centre. Then the distance between the two was computed. If the distance was less than 5 pixels, then the object was a perfect rectangle. Theoretically, this method should have worked. However, the amount of camera noise present in the frames did not allow for a reliable distance calculation, resulting in inaccurate results.

Hence, the single dark mask detector was used. To reduce the large amount of false positives detected, a more strict ratio was used. Specifically, the minimum ratio was changed to 0.9. However, the higher minimum ratio decreased the steepness of the angle from which the algorithm could detect the window. This happened because as the camera angle became steeper, the window rate lowered (in very steep angles, the window ratio is the same as a square). Lower ratios also picked bounding rectangles from not important objects such as bins, table shelves and rovers as well. Once the minimum ratio was increased, the issue was solved.

Planet Identification

To get the robot to identify the planets from the images, there were various approaches available. One of them was to use traditional image processing techniques, such as colour detection, contour analysis etc. However, these methods would not always work for the various planets due to overlaps and similarities, such as the Moon and Mercury looking similar. Hence, we decided to use a deep learning model using Convolutional Neural Networks (CNNs) to detect different planets. This method would ensure superior accuracy and robustness in detecting the various planets than the traditional techniques.

The deep learning model to detect planets from images was initially trained using transfer learning based on the Xception CNN architecture using TensorFlow and Keras libraries in Google Colab. The test accuracy for this was 100% and the test loss was 0.00029 to detect the various planets based on the dataset provided.

However, unfortunately, tensorflow and keras libraries were not included in the singularity image and only the Pytorch library was provided. So, the previous model had to be discarded and another model had to be trained using the PyTorch library. For this, the PyTorch library first had to be understood and learned. Then, a new CNN Architecture had to be chosen instead of Xception, this is because Xception architecture was not part of Torchvision. Hence, after preliminary research of various architectures, the DenseNet201 architecture was chosen for this task, because of its proven accuracy in other tasks such as in the ImageNet challenge.

In order to train the DenseNet201 CNN architecture to detect various planets and moons, the training process was carefully planned to ensure model robustness and accuracy. Firstly, in the pre-processing stage, data augmentation techniques such as random rotations, flipping, resizing, and brightness adjustments were applied to the images to ensure that the model would work well under diverse real-world conditions. Performing augmentation not only expanded the dataset but also added variations to the data that helped improve the generalisation of the model.

The DenseNet201 model, pre-trained on the ImageNet dataset was adapted by modifying the last layer to classify the 11 different planets and moons in the dataset. The training of the model was performed over 100 epochs with an early stopping mechanism based on validation loss. This approach was a form of regularisation and ensured the model didn't overfit and did not continue to train, wasting resources and time, if the performance was not improving.

Throughout the training process, the model consistently demonstrated a very high accuracy of 100% over several epochs. The validation loss continued to decrease, proving the effectiveness of the chosen architecture and the training process. Once the model was trained, it was tested on unseen test data. The model again proved its robustness by having an accuracy of 100% on the test data.

Image Stitching

Once the two pictures are saved, the method “from_frame_to_image_for_ml” crops and adjusts the picture to ease the stitching process. “resize_png_pictures” is then used to resize the pictures so that both have the same size. Finally, the method “perform_stitch” stitches the picture together using the “Stitcher_create” class from cv2. The stitched image is then saved and stored to be processed for calculating the distances with the Earth and the Moon.

The stitching tool was particularly sensitive to camera angle from which the picture was taken and to the distance from which the picture was taken. If the camera was too angled, the tool failed to find a good fit for the stitch. When the camera was too far, the retrieved picture would lose details and the tool would not be able to recognise the bounding circles of Earth and Moon.

The image stitching used was robust, however, another approach to solve the stitching would be to use the brute force used in the course to stitch 2 pictures togheter.

Distance Calculation

To calculate the distance of the camera to the Earth we used the equation:

$$\frac{f * realobjectheight\ (km) * imageheight(pixels)}{objectheight(pixels) * sensorheight}$$

This equation is used to get the distance of an object to the camera location. Where *f* & *sensorheight* is assumed to be the camera scaling factor of 3, with *imageheight* being the height of the camera feed.

Next, to calculate the camera’s distance to the moon we could use the same equation, with the caveat that the real height of the moon is not given to us. To calculate this we use a cv2 Hough circle to get the radius of the Earth in pixels, and since we know the real height of the Earth we can now get the kilometres per pixel. Using this information we can use the cv2 Hough circle function on the moon and translate its diameter in pixels to kms for use in the original distance to camera formula.

Lastly, to calculate the distance between both the Earth and the moon we use a different formula. Here we use the Hough circles to obtain the centre point of the Earth and moon alongside their radii. Using pythagoras theorem we can get the distance between the centre of both celestial bodies, and by taking away their radii, we know the distance between them, assuming the radius is constant throughout the planet/moon.

$$Dist.\ Earth\ to\ Moon\ =\ \sqrt{(Moon\ Center)^2\ +\ (Earth\ Center)^2}\ -\ Earth\ Radius\ -\ Moon\ Radius$$

Integration of Tasks

The various tasks were divided between the members of the team. Once all team members were done with their part, the code for all tasks was integrated together using Git and global flags to signal when one part of the program is complete and the next part can begin.

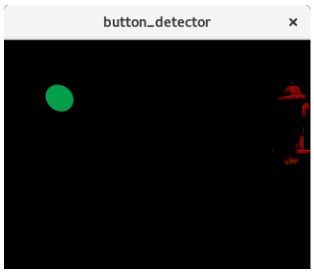
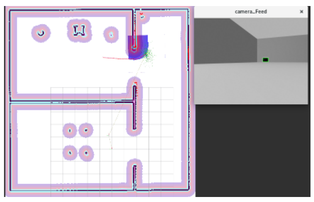
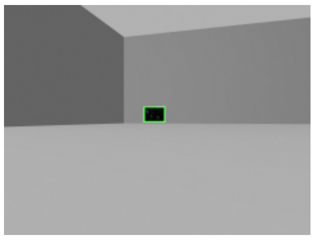
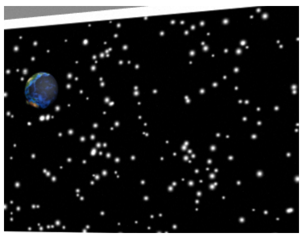
Git

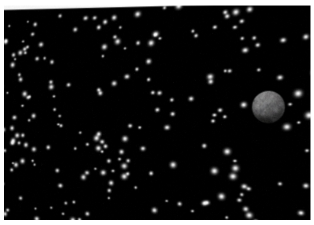
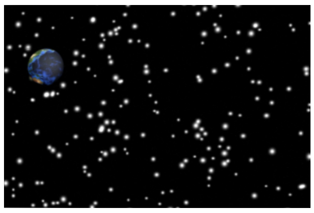
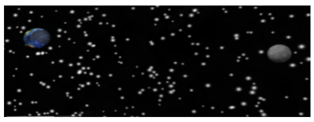
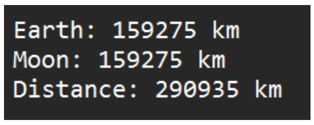
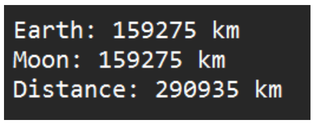
GitHub was used for the version control for this project. This allowed the whole team to work on the codebase together simultaneously, and at different times. We used branches primarily to help separate the workload and make integration easier afterwards.

Demonstration

A video of our program executing in the simulated environment can be found here.

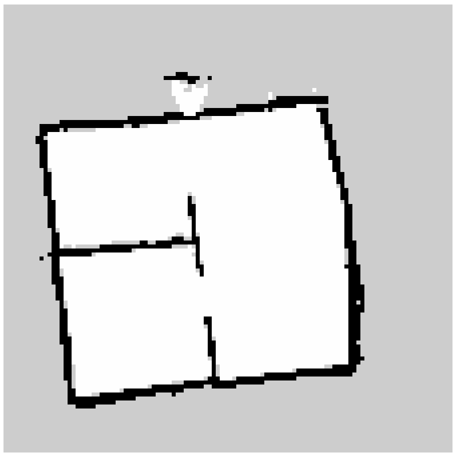
Testing

Task/Test Description	Performance Evaluation	Test Outcome (Successful/Unsuccessful)	Evidence
Does the program successfully locate the green room?	Robot efficiently locates the green room and enters it.	Successful	
Does the robot correctly move about in the Green Room	As designed, the robot correctly navigates the green room.	Successful	
Does the robot correctly find the windows?	The robot accurately detects the windows.	Successful	
Does the robot correctly move towards the windows detected?	Robot moves correctly towards the robot, but the window is not always perpendicular to the robot.	Partially Successful	

Does the robot correctly take a photo of the window view and crop it for analysing?	Robot successfully cropped the picture to better fit the model inputs.	Successful	
Does the robot correctly analyse the view and detect the planets?	Yes the deep learning model has a high success rate, with occasional errors on low quality or too distant images.	Successful	
Does the robot correctly stitch the images of earth and moon?	The stitcher works flawlessly when the pictures are taken perpendicularly or with a small angle. If the camera angle is tight, the stitcher would not always work.	Successful	
Does the program correctly find the distance from spaceship to earth and moon?	The distance from the spaceship to Earth and Moon are always the same.	Partially successful	 `Earth: 159275 km` `Moon: 159275 km` `Distance: 290935 km`
Does the program correctly find the distance between earth and moon?:	Distance found using Pythagorean theorem.	Successful	 `Earth: 159275 km` `Moon: 159275 km` `Distance: 290935 km`

Real Robot Testing

In order to test our solution in the real world, we had to set up our program to ensure it would work with the real robot. This involved connecting the computer to the robot using SSH, configuring the system, mapping the real world, getting the centre points and entrance points of both rooms and localising the robot.


Above image shows a map of the real world created.

Once the robot was configured, our code could be run on the real robot. Unfortunately, due to the lidar not working in the real-world robot and some other issues, we were unable to get our code to run on the real-world robot. The robot accepted goals but did not move and work as expected.

Due to the lack of time, we were not able to modify the code to make it successfully work on the real robot.

A video of our program attempting the real robot test can be found here.

Evaluation

Strengths:

- The robot can easily detect the green room.
- The code has been divided in small topic blocks, which allows for fast implementation of new features and fast debugging
- The implementation successfully stitches 2 pictures together and calculates the distances.
- The deep learning model successfully detected and categorised the pictures.

Weakness:

- Implementation is not as robust as it could be since the navigation of the room part only goes to the centre areas of the room and rotates. If there is a window that is being blocked by an object from the centre, the solution will not work.
- The program does not correctly work in the Real-World Test.
- The program does not align the view perpendicularly to the robot before taking a picture.
- Wastes time potentially looking at the same photo repeatedly.

Conclusion

Our solution performed individual tasks such as locating the green room, detecting windows, detecting planets and stitching the images well. However, the key part that held our solution back was accurately getting window images, our solution works well in the easy and medium test worlds but struggles in the hard test world and would have difficulty in more complex situations. However, given the circumstances of simply shaped rooms with a moderate amount of obstacles, our algorithm is able to adapt and may work given the specific map, even if it has difficulty completing others.