Project Report
Intelligent Systems and Robotics
Group 4

By Daniel Bilc, Tomás Zilhão Borges,
Andrzej Miskow, Mohammed Tanvir Roshid

Daniel(sc19dab),
Tomás(sc19tpzp)

Andrzej(sc19am2),
Tanvir(ed18mtr)

Table of contents

Daniel(sc19dab), Andrzej(sc19am2),
Tomás(sc19tpzp) Tanvir(ed18mtr)

Requirements

Design:
Break down design in sub-tasks and explain them in detail
Justify decisions and show general knowledge of methods
Comment solution on: Efficiency, Robustness (likelihood of working in a wide range of situations), Novelty, Strengths, and Weaknesses

Implementation and Results:
Explain how the solution was implemented, use code screenshots (do not forget code comments)
Evaluate Efficiency, Accuracy and Performance at executing the task, use concrete evidence and present it nicely (numbers, figures, tables, diagrams, images, videos)

Real robot:
Describe what needed to be done to make the program work on the real robot.
Describe how the robot performed in the real world.
Provide link to a video showing a real robot attempt.

Daniel(sc19dab), Andrzej(sc19am2),
Tomás(sc19tpzp) Tanvir(ed18mtr)

Chapter 1 Introduction: Philosophy and ideas

The following section is meant to give a general representation of what our plan was. It is a high overview and requires no technical knowledge to understand. Technical details can be found in the next subsections.

The 'Room Identification' phase is the initial and most critical phase of the coursework. We have a map with two separate rooms, one of which should not get entered under any circumstances. The rooms are distinguished by a green and a red flag for the correct and incorrect rooms, respectively. There are two tasks for this part of the project, which are:

1. Identify the correct room and the corresponding coordinates.
2. Identify the objects surrounding the robot at the room entrance with the green flag, record the average distances and the robot's current angle, and select the minimum distance with its associated angle. The critical element is positioning the robot's view perpendicular to the entrance. We assume that the flag is on the same line as the entrance and use the minimum distance and angle for achieving this.

The second aim described above is crucial to obtain for the 'Path Planning' segment described below. The distance and angle are used for computation to seal the correct room's map to traverse and detect the Cluedo character without the robot leaving the room mistakenly. Failure to perceive the room with the green flag or computing an incorrect angle or distance will cause catastrophic issues for the remaining modules described below. For more details, check **Chapter 2**.

For our project, we opted to go for a more idealistic approach, where the already preconfigured 2D map of the environment would dictate the path of the robot. This requires the map to be accurate in its shape and size and some other small requirements which will be discussed later within the implementation section. Our approach takes in the map of the environment and tries to configure a path using the pixels of the map. We know that the pixels are configured such that each map will have a dark area that represents an object/wall, a grey area that represents the unknown, and white which is the space where the robot could theoretically be. This is encoded using grayscale colourspace.

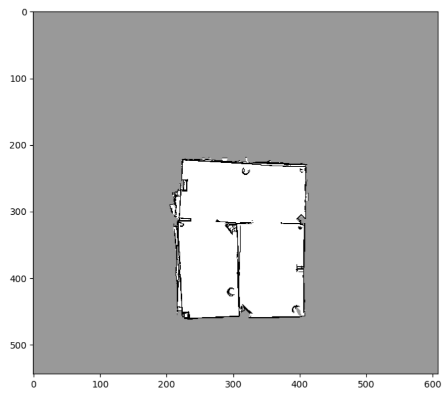


Figure 1.1 Map viewed as a 2d array in Matplotlib

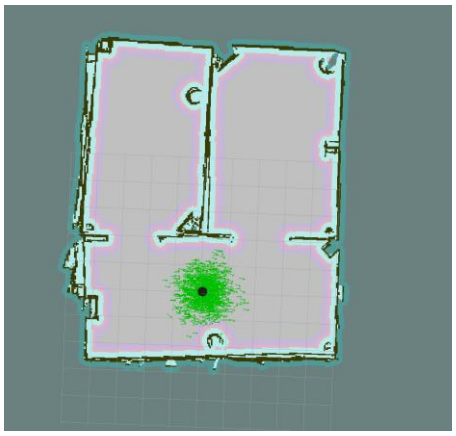


Figure 1.2 Map viewed in Rviz

Based on the dark pixels, we configure a path along them making sure that the robots has clearance on all sides, and covers all the dark pixels from the correct direction (if for instance you have an object in the middle of the room, the object can be seen from all 8 directions thus if the object is one pixel in its encoding the robot will have to have a corresponding position for all the directions and dark pixel configurations). An example of a path is **Figure 1.3**.

Once that path is computed, we try to remove redundant points such that if the robot would visit all the pre-planned points, it would not miss a spot on the wall/object with a specified overlap. The overlap is required to ensure that if the image is spotted, it would not be cut in half or any other way. **Figure 1.4** is a visual representation of this. With the assumption that the map is almost perfect in its recordings, we can assume that the method is 100% robust, such that it will always cover all the spaces. If the robot fits, it will go and look at all the places it needs to. Its other strength relies on the fact that no computation is needed during the search.

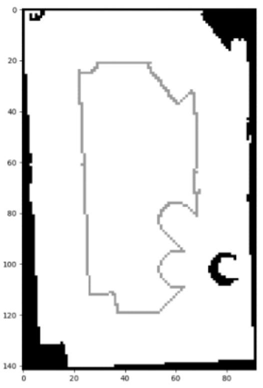


Figure 1.3 Room linear path in Matplotlib

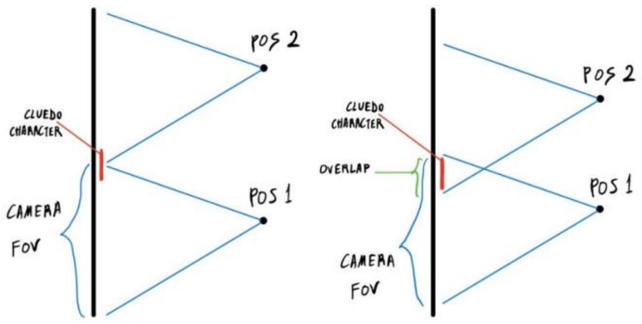


Figure 1.4 Point selection comparison

In terms of weaknesses, the reliance on the map is its biggest weakness. One mistake with the map and the entire process gives wrong answers. With that said, we were assured that we can trust the map, so the weakness falls with the task and not the implementation in this context. For further details, check **Chapter 3.**

Lastly, while the robot is traversing the room its goal is to detect the poster of the Cluedo character, take an image and classify which character was detected. To ensure that the robot has always found a character the two steps were merged, hence if a character is detected we know it has also been classified. To be able to execute the two processes in parallel the solution had to be efficient and require low computation cost. Additionally, for best results the detector should be executed in real-time to ensure a solution is found on the more difficult maps described in **Chapter 7**. For the following reasons three methods for this task were considered: detecting the dominant colour in the frame, OpenCV template matching and object detection using a convolution neural network. The following methods are described in depth in **Chapter 4**.

Chapter 2 Room Identification

2.1 Overview of the Room Identification phase

We have two rooms distinguishable by the flags and the colours at their entrances. We aim to avoid the room with the red flag and detect the room with the green flag, i.e., task one. We begin by sending our robot to the locations rotating 360 degrees clockwise for 10 seconds and applying a colour detector. Specifically, we are using colour thresholding to identify the green flag. We check both rooms regardless if one room fails to ensure robustness and avoid the possibility of both having red flags in front of them. Once we have identified the correct room, we need to complete aim two, i.e., making the robot face perpendicular to the wall in front. We achieve this by a 40-second 360-degree rotation at the entrance and using the laser scanner to record the robot's distances and the angles it makes. The assumption is that the shortest distance and the associated angle correspond to the wall in front of the robot, but one fatal flaw is present. Consider the scenario where an object is closer to the robot than the wall itself. We select this distance and angle from the laser scanner, making our robot no longer perpendicular to the wall. We avoid this possibility by only recording the values from the laser scanner where the green flag is present. Once we have the minimum distance, angle, and the correct room coordinates, the ‘Room Identification’ phase ends and jumps into the ‘Path Planning’ phase with these outputs chained in. An alternative to colour thresholding is to use a form of Machine Learning, i.e., a neural network trained to recognise the green and red flags if you require further robustness. We will now consider the individual elements that make up the whole Room Identification phase.

2.2 The Colour detection

The ColourDetector class is a generic class designed to detect a specified colour from the robot's camera feed. We subscribe to the camera topic and use a callback for detecting the contours, and if we identify a colour, we log this to the screen, but in an ideal situation, it would be preferable to either destroy the object or execute a separate callback when the colour is detected. Therefore, in the class constructor, we offer an additional callback parameter to run whenever the colour gets identified if a callback is even provided during the object's construction, as this is not a required criterion. We expect the supplied callback for successful detection to call the unregister_subscriber method provided if the caller wishes to use the same object again in the future, or they can directly delete the instantiated object as we do not pose any strict requirements for the class.

2.3 The distance recorder from the laser scanner

The DistanceLocator class can receive data from the robot's laser scanner via a callback. Our robot's depth sensor has a 60-degree range with the length of the ranges array set to 640. The ranges array is where the recordings from the depth sensor are stored, and the central digits of the ranges array directly indicate the distances detected from the front of the robot. We need to use the values from the middle of the laser scanner, so we use the 319th index, but we need to consider the average for robustness. Therefore, we use the upper and lower five values, i.e., 314 to 325 and calculate the average value from the laser scanner. If all readings from indices 314 to 325 are out of the minimum and maximum bounds from the range array, we set the distance to the maximum possible value, i.e., range_max. Otherwise, we calculate the average from the values read and divide by the number of valid values, i.e., those that meet the upper and lower bounds specified by range_min and range_max.

2.4 Other utility elements for the Room Identification phase

We considered the colour identification and the distance recording because they were critical elements with broad descriptions that needed outlining. We do have other components that contribute to the Room Identification phase that requires a small discussion.

A MathConversions class exists for containing mathematical functions defined to make development easier and quicker. An example is the conversion of Euler angles to Quaternions. Built-in libraries got used for robustness, and if a library did not support the needed functions, these got implemented via the internet from trustworthy sources. We also have a Point class to encapsulate data from the laser scanner, specifically storing the robot's current distance and quaternion from the laser scanner. Furthermore, a Robot class exists to provide an abstraction for calls to the Navigator class. We offer additional methods for rotation, stopping the robot and recording and returning the robot's current orientation. The Navigator class is just used to send commands from the robot and is adapted code from lab two. Finally, we have the YamlReader class designated to parse YAML files and return data, which uses a built-in Python library for parsing YAML files, so it is well-tested, and we are confident that it is 100% functional given a syntactically correct YAML file.

We would not like to consider the remaining classes in-depth because this would be an extensive and meaningless task. The choice of mentioning high-level descriptions at this stage was to show the readers the thought process for the project at a code level where robustness and other factors got considered. We have a set of decoupled, perfectly engineered classes that work together to perform aims one and two for our Room Identification phase.

2.5 Practical Overview

To complete aim two of the Room Identification phase, we began using the teleop and rostopic echo commands to move the robot manually and output the values displayed from the odom topic. The odom topic contains information about the robot's position in the world, and we recorded the robot's location and the values from the depth sensor when manually placing the robot perpendicular to the wall. We saw that the angle made relative to the original starting angle was approximately 89 degrees, and the distance from the wall was 0.8 m for world one. We noted this and began producing graphs showing the gap and angles recorded for a 360-degree rotation. We replicated this several times, changing the time for rotating to find an optimal rotation speed. We also colour encoded the graph to show time intervals where the green flag was present, and **Figure 2.1** is an example of the chart produced for a 360-degree rotation in 5 seconds. The figure confirms the findings from the teleop operation, i.e., the distance and angle being 0.8 m and 89 degrees, respectively and approximately. Once we had seen a working example of our code, we needed to repeat the rotation for different seconds to find the optimal value where the number of recordings was high, and the absolute difference of the distances and angles recorded was low relative to the teleop values found. We realised that the local minima from all graphs were always within the green region, i.e., when the flag was present, which makes sense since it is on the wall itself. Combined with the idea of objects nearby tampering with our results, we decided to use recordings only where the green flag is present, i.e., the fatal flaw discussed in **Chapter 2.1**. **Table 2.1** is an overview of the different timings used and our results. Ultimately, we chose 40 seconds as an optimal and safe choice for the robot. Finally, we tested the code on multiple worlds, including the evaluation worlds and confirmed that everything worked as intended.

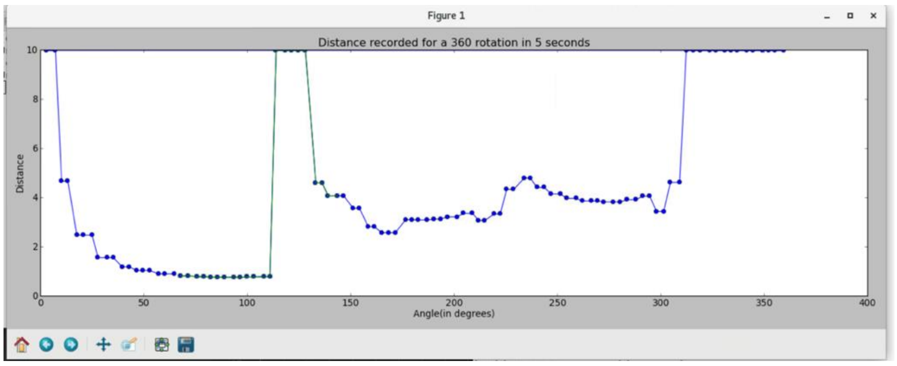


Figure 2.1: A graph to show the distances and angles recorded for a 360-degree rotation for 5 seconds and colour encoding intervals where the green flag is present.

Rotation time (seconds)	No. of recordings considering colour	Absolute difference relative to the teleop investigation
5	11	0.2
10	23	-1.09
15	32	0.74
20	44	-1.2
25	55	-0.1
30	66	-0.6
35	75	- 1.1
40	88	0.8

Table 2.1: A table showing the varying rotation speeds used, the number of recordings and the absolute difference of the values recorded relative to the teleop investigation.

Chapter 3 Path Planning

3.1 Method exploration and suggestions

For path planning inside the room, we initially wanted to use a live strategy, where the robot would constantly rely on its sensors to navigate the room until finding the picture. This was not possible due to the lack of a wider 3D sensor, making any attempt at using the given sensor invalid due to the time constraint. Most robots nowadays do most of its navigation live, using sensors, the difference in our case is the lack of features that we hope will be added to the robots for future generations such as a 360-depth sensor (or a full-blown lidar), a 360 camera, or a camera that can be dynamically moved through a mechanism etc. These will improve path planning strategies and are most of the time commonplace for all robot configurations.

3.2 Practical implementation

In practical terms, there are 4 different components: Room closer, Room cut, Proximity map, Point selection. Each component will be presented separately.

Before that, as input, the system needs the map image metadata from "/map_metdata" which describes the size and resolution of the image, as well as the corresponding bottom left corner of the map or 0,0, which can be later translated into the centre and from there any conversion between the systems (2D image plane and real world). After that, we extracted the map from the "/map" topic as a 1D array of values. This map could have values of -1 for unidentified space, 0 for empty space which can be used, and finally 100 for blocked space (which in theory stands for 100% an object there). Besides the information from the topics, we take data from the Room Identification phase as described above.

3.2.1 Room Closer

This module is responsible for drawing a line within the matrix map, such that the green room is enclosed in black pixels. This is done by taking the angle between the +X axis and the wall that has the green dot on it (this is usually 90 degrees). While the angle is taken, the distance to the wall is also recorded. To deal with inaccuracies, an additional distance is added.

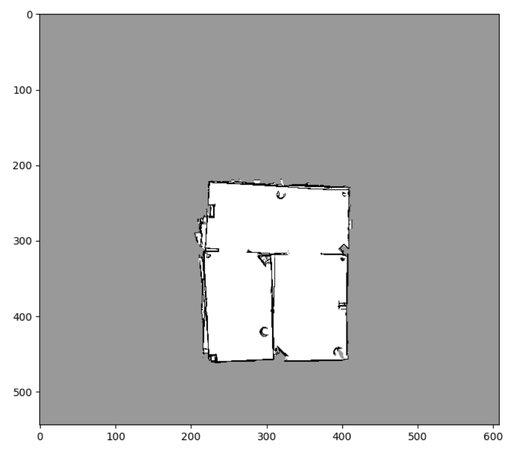


Figure 3.1: Map viewed as a 2d array in Matplotlib

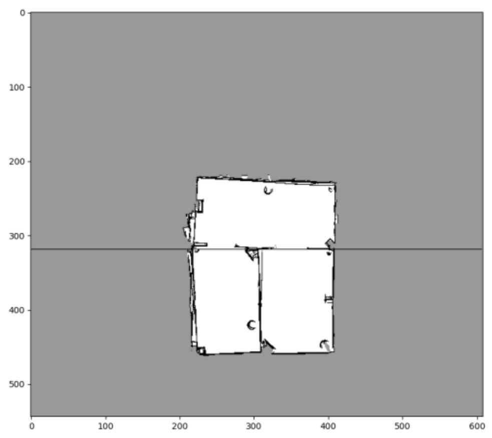


Figure 3.2: Map with blocked room entrances

However, before computing the line equation and getting **Figure 3.2** from **Figure 3.1**, we must figure out what are the coordinates for the point on the line where the wall is.

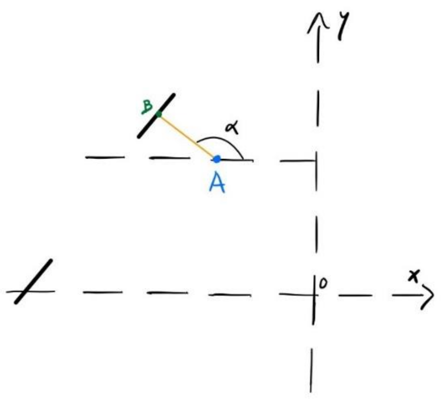


Figure 3.3 Diagram to help explain the line computation

From the image, we know the coordinates of A, the distance d (from measurements) and the angle alpha (from measurements). The angle alpha is given by the orientation of the robot compared to the ox axis, which represents 0 degrees. We can compute the line equation for AB using the point and the angle alpha, or rather the tangent of that angle. To obtain B's coordinates, we add to the coordinates of A, distance d multiplied by the cos of the angle alpha for x and then the sin of angle alpha for y.

With the two measurements, we compute a line equation that should cover the exit of the room. After that, we apply the line to pixel conversion like the graphics approach to drawing a line on a screen.

3.2.2 Room Cut

The room cut model is responsible for extracting the map of the room to make the processing faster and eliminate any other unwanted data (hallway and red room).

To achieve that, we make use of the centre of the green room (which we already know), in a recursive algorithm that visits all neighbouring pixels in the direction of up, down, left, and right if the base pixel is not black. With each pixel visited, an element inside a tracking array is set, so the same pixel is not visited twice and lets us know which spaces are empty and which are not. After getting this matrix, the map of the room itself can be generated. This step was not really required, but it provided a big advantage in terms of processing speed when working with a submap from the original map. An example of that is **Figure 1.3**.

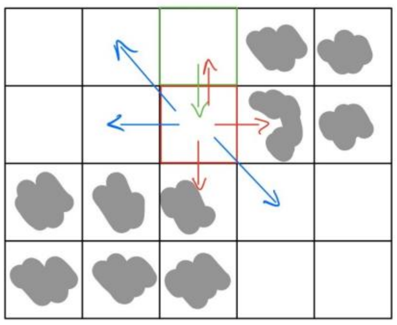


Figure 3.4: Recursive search escaping through a diagonal space

One problem that we encountered here was with the search direction. Initially, we also did a diagonal search besides the above. This had an issue in the case that a white pixel can be found diagonally even though the robot could never fit between the pixels. It can be seen in **Figure 3.4**.

3.2.3 Proximity Map

This is the hardest part of the process, which is the proximity layer added on top of the room map. There are 2 main thresholds: one is given by the size of the robot, such that the robot needs to be x meters away with its centre from a wall, so it does not hit that wall, and another measurement which is predetermined and is the distance we want the robot to be far from the wall. This distance can and is tweaked such that the robot will have to cover as little "stops" as possible while still always providing a clear image of the wall.

To create this overlay, we iterate 4 times over the matrix, once from left to right, second time from up to down, third time from top left to bottom right and finally from top right to bottom left. Each iteration will look for a black pixel and depending on how that pixel is reached, the procedure will either add a position point before or after it, considering the 2 specified distances. While the iterations are happening, on the same map, other points are added which determine the zones that are created by each measurement.

Sometimes, due to the iterative nature of the process, gaps will form, where points will be missing, thus creating a very fragmented path. An example of this is present in **Figure 3.5**.

The solution for this is to fill out the rest of the gaps with points that will have directions based on the neighbouring grey pixels. While this solution is not perfect by any means due to the many additional points that might not be worth checking, it provides reassurance that no surface is left unchecked. An example of this is **Figure 1.3**.

In **Figure 3.6**, the dark grey area is the zone where the centre of the robot cannot physically be as it will hit the wall, and the light grey area represents the arbitrary distance that was chosen such that the robot does not need to cover a substantial number of points. Finally, because it is a grey mode map, the dark line inside the map represents the "continuous" path that the robot should take to constantly be at the same distance from the wall.

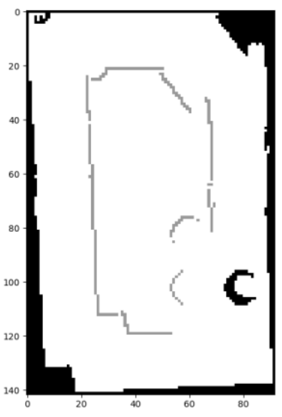


Figure 3.5: Map of the room with path gaps

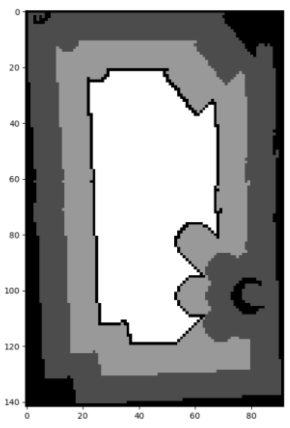


Figure 3.6: Map of the room with proximity overlay and path

Another important aspect to mention here is the fact that each of those points of "stoppage" has a direction. There are in total 8 arbitrary directions. In **Figure 3.7** there is a niche case in which this idea is presented. From the image, the position pixel is surrounded by a light grey area in all directions but one. This leaves us to believe that the stoppage time can come from any direction other than the one that it came in from. In total for this case, 7. Most of the time, this scenario does not happen, often there is only one direction or for corners, 3. This can be seen in **Figure 3.8**.

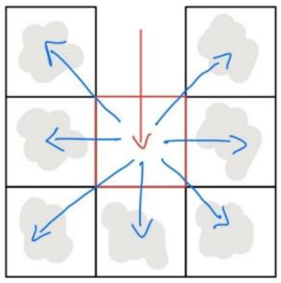


Figure 3.7: Niche case where the robot checks 7 directions in one point

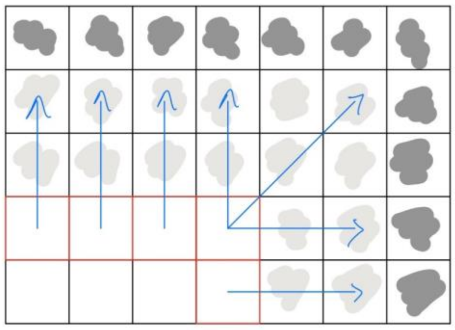


Figure 3.8: A corner structure from a path

3.2.4 Point Selection

Point selection is responsible for filtering out the points such that time is not wasted on visiting close points which render, effectively the same image.

The first idea we had was to keep points at an interval, such that every x'th point will be kept and the rest removed. This was flawed due to the predictability of the wall sizes. If the interval were chosen too big, we could skip important angles and if it were too small, we would have visited too many points for no reason.

Our final approach here was to predetermine an overlap. This overlap is used such that if the image of the character is between two wall shots from two positions next to each other in the same direction, the image is not sliced up. Refer to **Figure 1.4** image for a visual representation.

Methods	Original Points	Processed Points	Original Directions	Processed Directions
Fill gaps	313	37	340	40
Not filled gaps	265	23	265	23

Table 4.1: Points for each method

Table 4.1 references the number of points before and after the filter for each approach, giving another reassurance that the overhead of filling the gaps is not important in the grand scheme of things.

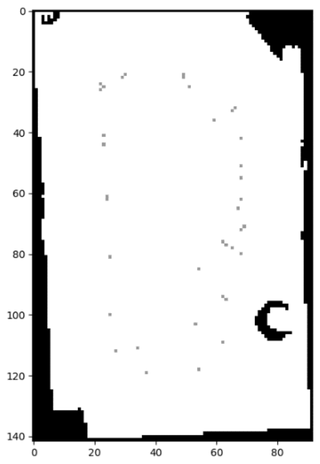


Figure 3.9: Map of the room with the final points

Next, the points are ordered based on their proximity to the previous points, making the list linear (points follow the path linearly). After that, the list is iterated and for each of the 8 directions, a check is performed such that between every 2 points in one direction there is one minimum distance. An example of the results can be seen in **Figure 3.9**.

Each of these points will act as a stopping destination where the robot will have to turn around and look in a maximum of 7 directions. This will guarantee that the image of the character is captured in the field of view by just turning the camera when the robot looks in one of the directions.

Chapter 4 Character detection and classification

4.1 Dominant Colour

The initial implementation looked at the simple case of detecting characters based on the dominant colour in the frame. The motivation behind this was that the most distinctive part about each character was the colour of their background and their clothes. Additionally, the colours present on each character was different from the object present in the room and the walls. This task could have been achieved in two ways: by counting the number of pixels in the image or by utilising k-means for a cluster of pixels.

The first method counts each pixel in the image and outputs the hex-code for the most dominant colour. This method had a major flaw because the most dominant colour in each image happened to be the thick black border. For this reason, k-means clustering was considered. K-means clustering partitions pixel colour data into k disjoint clusters, where the points in the same cluster are "more similar" than points in different clusters. The algorithm is relatively simple:

1. Take random k points (called centroids) from the image
2. Assign every point to the closest centroid, and the assigned points form a cluster.
3. For each cluster, find a new centroid by calculating a new centre from the points.
4. Repeat steps 2–3 until centroids stop changing.

The algorithm will output k clusters from this method that represent the most dominant colour in the image. Via experimentation, 5 clusters had the best results, with the most dominant colours being distinct for each character.

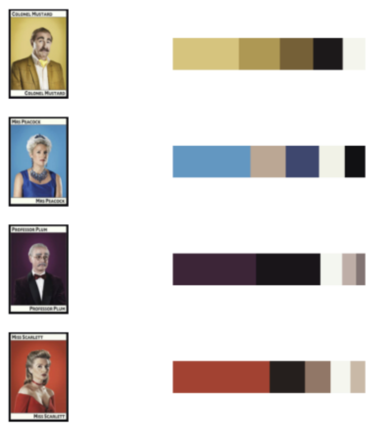


Figure 4.1: K-means dominant colour output

In addition to displaying the k dominant colours, a function was implemented to display the order of dominance, i.e., the clusters with the most data points. As witnessed in **Figure 4.1**.

However, the dominant colours method had its flaws. Firstly, it considered all the pixels present in the frame; hence, it could not have been directly applied to the camera feed. OpenCV contour detection was utilised to locate the poster in the image. Still, this method was not robust in the more complex maps due to the range of rectangular objects present in the image. The method also relies on the same lighting conditions, which could not have been guaranteed in the real-life robot scenario.

For the following reasons, more robust methods were considered that focused on features which are more than just colours. One of these methods was OpenCV template matching.

4.2 Template matching

Template Matching is a method for searching and finding the location of a template image in a larger image and can be seen as an extremely basic form of object detection. This method is implemented in OpenCV in a function cv.matchTemplate. The function slides the template image over the input image and compares the template and section of the input image under the template image. A metric is calculated to represent how “good” or “bad” the match is. The most effective method for this is the normalised correlation coefficient which determines how “similar” the pixel intensities of the two patches are. Results from the test can be seen in **Table 4.1**.

Method	Detection Accuracy	Classification Accuracy
CCoeff	75%	75%
CCoeffNormed	**90%**	**80%**
SqDiff	70%	60%
SqDiffNormed	85%	**80%**
CCorrNormed	**90%**	60%

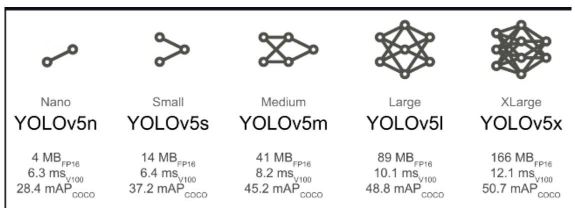
Table 4.1: Results for Character Detection

Through testing, this method performs better than the dominant colour method however it still has its flaws. The method is a basic object detection tool, where its performance correlates to the chosen templet matching method and hence is not fully robust. Depending on the chosen method, the detector can be sensitive to light, contours, or objects blocking parts of the image. Additionally, the program must compute template matching for each character for each frame which is not efficient for computation time and does not enable real-time character detection.

Since object detection worked well for the task of character detection, a method utilising state of the art object detector was considered; the current state of the art object detection method is done via a convolution neural network called YOLO5 implemented in PyTorch.

4.3 Object Detection using Yolo5

YOLOv5 [1] is an object detection architecture applied to a wide range of applications. The authors of the detector publicised five models that pertained to the COCO dataset and contained a different number of parameters. However, the models with more parameters will achieve better accuracy at a trade of memory requirement and computation speed.


Nano
YOLOv5n
4 MB FP16 6.3 ms V100 28.4 mAP COCO
Small
YOLOv5s
14 MB FP16 6.4 ms V100 37.2 mAP COCO
Medium
YOLOv5m
41 MB FP16 8.2 ms V100 45.2 mAP COCO
Large
YOLOv5l
89 MB FP16 10.1 ms V100 48.8 mAP COCO
XLarge
YOLOv5x
166 MB FP16 12.1 ms V100 50.7 mAP COCO
Figure 4.2: Yolo5 CNN Models [1]

The medium model was chosen for the project as it achieved considerably better accuracies than the small model. On the other hand, the difference inaccuracies between the medium model and the large were insignificant.

Training the model involves three steps data collection, data labelling and training. Data collection was done by taking screenshots of the posters at different map sections and a few background images. One hundred images were collected, with 20 belonging to each class and 20 background images. Each data sample was then hand labelled using Roboflow. However, for the object detector to have good accuracy, the authors recommend that training is executed over a dataset with 1500 images per class, with 10% of the images being background images. Labelling these many images would have been time-consuming hence data was artificially expanded via data augmentation. This method is used to create different image representations via transformations applied to the image. Examples of possible transformations are shifts, rotations, flips, brightness, contrast change, cropping or blur. It is vital to choose augmentations that will relate to the images the robot can encounter. For this reason, horizontal flip, rotation, zoom, brightness and contrast augments were used. Augmentation was done via the Augmentations library [2] was used as it enabled the augmentation of ladled yolo5 images, and the augmentation applied to the location of the bounding box.

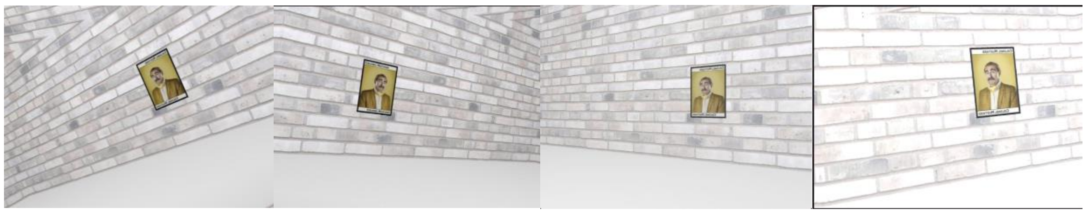


Figure 4.3: Output of different image representations through data augmentation.

Lastly, the training was executed on a batch size of 16 on 200 epochs at an image size of 640, which is the full resolution of the images produced by the robot.

The weights produced from the training were then utilised for the character detection process used on the robot. At the beginning of execution, the weights are loaded into the memory, and the image is fed into the prediction function. This function outputs if the character has been detected and, if so, its bounding box location and corresponding label. Due to the high accuracy of the detector set of constraints had to be set before the final screenshot was taken. These were a minimum area of the bounding box, and the location of the bounding box cannot be too close to the left or the right side of the screen. With these modifications, the character detection program achieved incredible results, with the ability to detect characters in the most challenging rooms. The detector software was also tested in the real-life robot scenario under different lighting conditions showing that the solution is robust and efficient.

Method	Average Execution Time (minutes)	Detection Accuracy	Classification Accuracy
Dominant Colour Detection	1.34	60%	40%
Template Matching	1.46	90%	80%
Yolo5	**1.34**	**100%**	**100%**

Table 4.2: Results comparing character detection and classification methods

Chapter 5 Testing

We approach testing in a systematic and methodical way so that we can evaluate and improve the quality of the solution prior to submission. We do so by first creating a set of worlds with gradually increased complexities, then we create an interface to easily switch between worlds and characters, and finally we time the runs for each world and verify for correctness.

World creation is done using standard practices acquired during this module. So for each world, we start by using Gazebo to design it and then proceed to scan a map out of it with Rviz and the Turtlebot's sensors. In addition, we use Gazebo to determine rooms' entrance and centre points.

For room design, we are especially concerned with placing the cluedo character in unpredictable places (e.g. on an object) and to test the robustness of path planning inside the room. We develop 3 additional worlds (2-4 in **Table 5.1**); **Figure 5.1** presents world 4 (and its corresponding map) and it is the one with highest difficulty (other world maps can be seen in appendix 1.1).

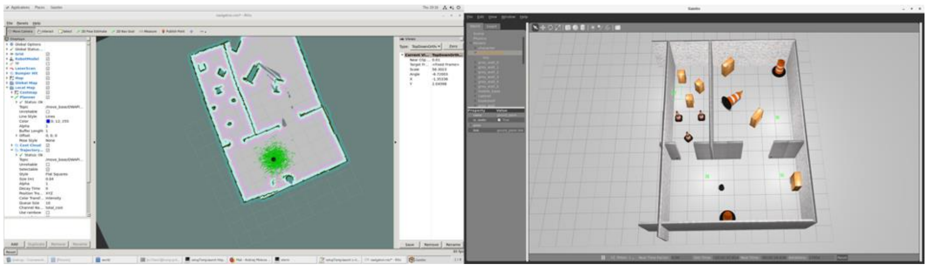


Figure 5.1: World 4 and corresponding scanned map

The interface for switching between world and characters is essentially a set of 2 shell scripts that takes care of renaming the files chosen by the user to standard names that are read from source code ('project_map.yaml', 'input_points.yaml', 'project.world' and 'Cluedo_character.png'). This is especially useful given the volume and verbosity of shell commands used for this project.

Finally, we run the proposed solution for each world and successfully reach to the end with the correct Cluedo character name specified in 'cluedo_character.txt' and a its (full) screenshot in 'cluedo_character.png'. The absolute path for these set of files is correctly chosen to match the evaluation requirements. As expected, we find that the running time increases with the world's complexity, but it stays under the 5-minute limit.
In addition, we run the mock evaluation provided (2 worlds) and successfully reach to the correct deliverables in time as well. **Table 5.2** presents the above-mentioned results, and we provide a demo video [3] for the mock evaluation's worlds.

World	Time (minutes)
1 (standard)	1:34
2 (custom)	1:51
3 (custom)	3:13
4 (custom)	2:43
5 (mock evaluation)	1:46
6 (mock evaluation)	3:54

Table 5.1: Timed successful runs on each map

Chapter 6 Real Robot Test

To be able to execute our code on the real-life robot we first had to create the map of the room. To do this we executed the following commands:

- roslaunch turtlebot_bringup minimal.launch.
- export TURTLEBOT_3D_SENSOR=astra
- roslaunch turtlebot_navigation gmapping_demo.launch.
- roslaunch turtlebot_rviz_launchers view_navigation.launch.

With the Rviz environment we walked with the robot around the rooms to map out the map. Setting up the map itself was difficult because the rooms had no boundaries defined, hence used tables to enclose the rooms and prevent the robot from mapping the outside. Additionally, the quality of the map produced was poor, so we measured the rooms manually and edited the map files in GIMP to improve the localisation of the robot [appendix 1.2].

We have also found that the commands for the real-life robot differ from the Gazebo simulation. To make the Turtlebot run, we need the following commands assuming a map got created:

- roslaunch turtlebot_bringup minimal.launch.
- export TURTLEBOT_3D_SENSOR=astra.
- roslaunch turtlebot_navigation amcl_demo.launch map_file:=<path to map>.
- roslaunch turtlebot_rviz_launchers view_navigation.launch.
- roslaunch of the python file.

The export and gmapping commands must get executed in the **same terminal** for mapping the robot. To run the robot in a defined map and be able to send goals and run code required the export and amcl_demo commands in the **same terminal.**

We also had a delay with being able to set up our code because our CNN required Python 3.9, and the laptops had an older version, which was also different to the Linux computers that we got advised to use for compatibility of our final solution. After solving the compatibility issues, we had issues with the detector camera feed not being displayed. Through trouble shooting we discovered that the reason for this came down to hardware limitation. Because of this we were unable to execute our character detection software in real time. Furthermore, the compatibly issues prevented us from fully executing our program. We were able to display the character detection working but could not create a full demo video. The character detection is shown in this online link [4].

References

[1] G. Jocher *et al.*, “ultralytics/yolov5: v3.1.” Oct. 29, 2020, doi: 10.5281/ZENODO.4154370.

[2] A. Buslaev, A. Parinov, E. Khvedchenya, V. I. Iglovikov, and A. A. Kalinin, “Albumentations: fast and flexible image augmentations,” *Inf.*, vol. 11, no. 2, Sep. 2018, doi: 10.3390/info11020125.

[3] *Group 4 Demo Video* : https://www.youtube.com/watch?v=G9BX0VqfeK8

[4] Group 4 Real Life Robot CNN Demonstration: https://www.youtube.com/shorts/mk9tmh5Nmrc

Appendix

1.1 Testing Maps

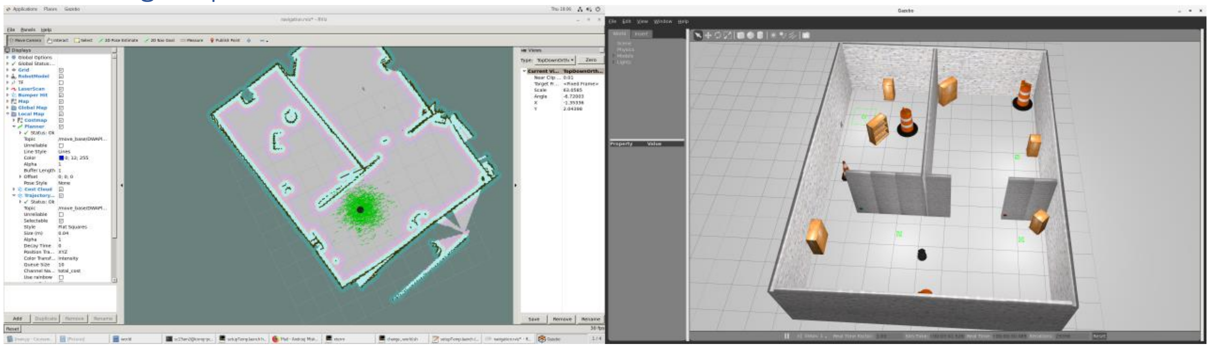


Figure A1.1: World 2 and corresponding scanned map

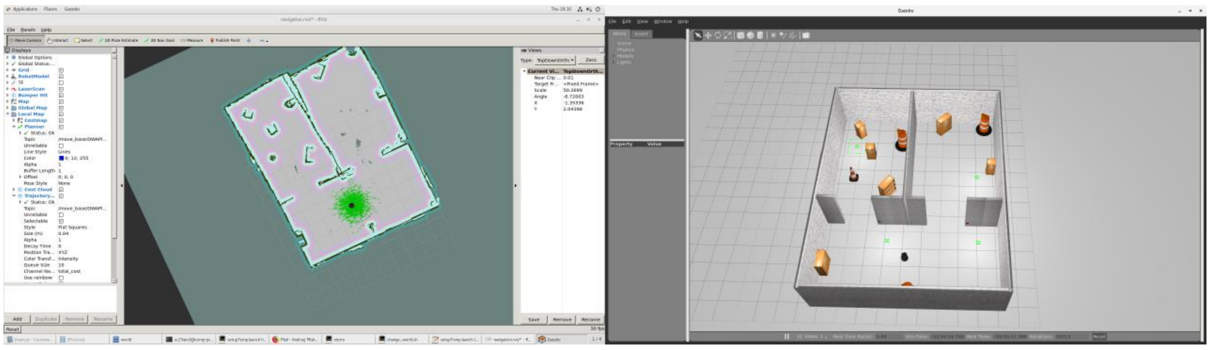


Figure A1.2: World 3 and corresponding scanned map

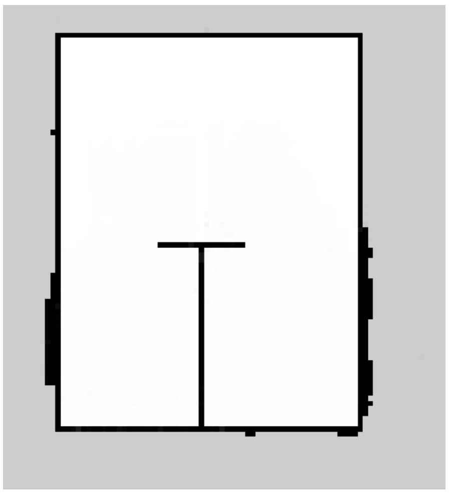


Figure A.1: K-means dominant colour output

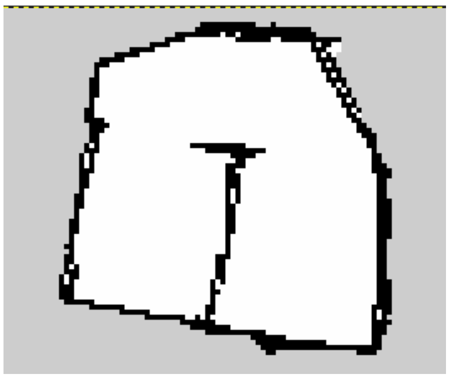


Figure 4.1: K-means dominant colour output