COMP4034
Jordan Lovett - 20236464

Contents

1 minitask5

1.1 SLAM_localiser

Localises the robot's position and orientation and translates this into world space.

Publishes to "/minitask5/pose" using Pose2D.

Subscribes to

- **/odom** using **Odometry**

Class variables

- **pose** (Pose2D) : Stores the current position and orientation of the robot in world space
- **buffer** (tf2_ros.Buffer) : A buffer of transform callbacks
- **listener** (tf2_ros.TransformListener) : Listener object that sends transforms to the buffer with time stamps

Subscriber callback methods

- **odom_callback** : When odometry publishes a message, we translate that relative to the map transform to convert it to world space. This includes position and orientation.

Node methods

- **SLAM_localiser()** : The constructor for this node

1.2 frontier

Finds the shortest path to the nearest unexplored area

Publishes to "/minitask5/path" using Path.
Publishes to "/minitask5/cost_map" using OccupanyGrid. (For RViz only)

Subscribes to:

- **/map** using **OccupancryGrid**
- **/minitask5/pose** using **Pose2D**
- **/minitask5/obstacle_grid** using **OccupancryGrid**
- **/minitask5/vision_map** using **OccupancryGrid**

Class variables

- **rate** (rospy.Rate) : The rate at which we publish messages
- **resolution** (float) : Size in m of a single grid cell in the maps
- **map_width** (float) : Width of the map in m
- **map_height** (float) : Height of the map in m

- **pose** (Pose2D) : Position and orientation of the robot in world space
- **cost_map** (OccupancyGrid) : Inflated map of the walls to mark how safe areas are to drive within
- **obstacle_map** (OccupancyGrid) : Map of all the obstacles that can't been seen with lidar
- **vision_map** (OccupancyGrid) : Map of the areas that have been looked at with the robot's camera

Subscriber callback methods

- **map_callback** : Receives the global map and calculates an inflated cost map and stores this for later use
- **pose_callback** : Stores the pose for later use
- **obstacle_grid_callback** : Stores the obstacle grid that was obtained with the camera
- **vision_map_callback** : Stores the map of seen areas of the map

Node methods

- **frontier()** : The constructor for this node
- **find_path () (Path)** : Uses the member variables for the most recent maps and pose to find the shortest path to the closest frontier
- **flood_fill (start_cell, fill_grid, cost_map, ignore_threshold, target_threshold) (cell)** : Performs a flood fill to locate the position of the nearest target cell
- **pathfind (map, start, end, threshold, allow_unknowns) ((x,y)[ ])** : Uses A* algorithm to find the shortest path from the start to end location through the given map. This can't travel through anything above the threshold, but has the option to travel through unknown areas based on allow_unknowns

1.3 localization

(Deprecated) Localised the robot within the map.

Publishes to "/minitask5/grid_pose" using GridPose.
Publishes to "/minitask5/localization_map" using OccupanyGrid. (For RViz only)

Subscribes to:

- **/minitask5/map** using **OccupancryGrid**
- **/scan** using **LaserScan**
- **/odom** using **Odometry**

Class variables

- **rate** (rospy.Rate) : The rate at which we publish messages
- **map** (OccupancyGrid) : The global map
- **grid_pose** (GridPose) : Position of the robot within map space

- **grid** (OccupancyGrid) : The local map visible with Lidar
- **pose** (Pose2D) : Position and orientation of the robot in world space

Subscriber callback methods

- **map_callback** : Receives the global map and stores it for later use
- **lidar_callback** : Given the local LaserScan, create a local map and find where in the local map looks most similar to localise on the centre.
- **odom_callback** : Stores the estimated world position based on odometry

Node methods

- **localization()** : The constructor for this node

1.4 mapping

(Deprecated) Build up a global map as the robot moves.

Publishes to "/minitask5/map" using OccupancryGrid
Publishes to "/minitask5/localization_map" using OccupanyGrid. (For RViz only)

Subscribes to:

- **/scan** using **LaserScan**
- **/odom** using **Odometry**

Class variables

- **map_rate** (rospy.Rate) : The rate at which we publish messages.
- **pose** (Pose2D) : Position and orientation of the robot in world space
- **grid** (OccupancyGrid) : The local map visible with Lidar

Subscriber callback methods

- **lidar_callback** : Given the local LaserScan, add to the global map
- **odom_callback** : Stores the world position of the robot

Node methods

- **mapping()** : The constructor for this node

1.5 movement

Controls the movement of the robot to follow the given path by publishing to cmd_vel

Publishes to "/cmd_vel" using Twist.

Subscribes to:

- **/minitask5/path** using **Path**

- **/minitask5/pose** using **Pose2D**

Class variables

- **path** (Path) : The path to follow
- **rate** (rospy.Rate) : The rate as which we publish messages
- **pose** (Pose2D) : Position and orientation of the robot in world space
- **max_angular** (float) : The maximum m/s we permit rotations
- **max_linear** (float) : The maximum m/s we permit linear movements

Subscriber callback methods

- **pose_callback** : Stores the pose for later use

Node methods

- **movement()** : The constructor for this node

1.6 object_detection

Uses the camera to detect the items we are looking for within the scene and mark them in RViz using MarkerArray

Publishes to "/minitask5/markers" using MarkerArray.

Subscribes to:

- **camera/rgb/image_raw** using **Image**
- **camera/color/image_raw** using **Image**
- **camera/depth/image_raw** using **Image**
- **camera/aligned_depth_to_color/image_raw** using **Image**
- **minitask5/pose** using **Pose2D**

Class variables

- **map_width** (float) : Width in m of the map
- **map_height** (float) : Height in m of the map
- **real** (boolean) : If the node is running on a real robot or not
- **bridge** (CvBridge) : Used to convert camera images into openCV data
- **depth_image** (Image) : The depth image returned by the camera
- **pose** (Pose2D) : The position and orientation of the robot
- **rate** (rospy.Rate) : The frequency we publish MarkerArray
- **markers** (MarkerArray) : Array of markers that identify the seen objects

- **resolution** (float) : Size in m of each grid cell in the maps
- **green_grid** (OccupancyGrid) : Probability grid of suspected green objects
- **red_grid** (OccupancyGrid) : Probability grid of suspected red objects
- **marker_pub** (rospy.Publisher) : Publisher for sending marker array to RViz

Subscriber callback methods

- **color_image_callback** : Finds objects within the color image, samples the most recent depth at the centre of the objects, and calculates there location in the world. This is then converted into grid space and increases the probability of an object there within our object detection grids. Only works on objects above a certain size to remove noise.
- **depth_image_callback** : Converts the depth image into the correct format and stores this in a member variable for later
- **pose_callback** : Stores the pose for later use

Node methods

- **object_detection() ()** : The constructor for this node
- **update_markers() ()** : Sets the MarkerArray using both the green and red probability grids
- **get_markers(grid, start_id, ns, color) (MarkerArray)** : Thresholds the given grid, and then finds the connected components within it. The centre of each connected component is the centre of the marker. The marker id's start at "start_id", are placed within the namespace "ns" and are of the color "color"
- **to_world(origin, resolution, grid_coord) (Point)** : Converts the grid_coord into world space around the map origin and given the resolution
- **to_grid(origin, resolution, world_coord) (Point)** : Converts the world_coord into grid space around the map origin and given the resolution

1.7 obstacle_detection

Uses the camera to detect the obstacles that should be avoided but can't be seen with the lidar

Publishes to "/minitask5/obstacle_grid" using OccupancyGrid.

Subscribes to:

- **camera/rgb/image_raw** using **Image**
- **camera/color/image_raw** using **Image**
- **camera/depth/image_raw** using **Image**
- **camera/aligned_depth_to_color/image_raw** using **Image**
- **minitask5/pose** using **Pose2D**
- **map** using **OccupancyGrid**

Class variables

- **map_width** (float) : Width in m of the map
- **map_height** (float) : Height in m of the map
- **real** (boolean) : If the node is running on a real robot or not
- **bridge** (CvBridge) : Used to convert camera images into openCV data
- **depth_image** (Image) : The depth image returned by the camera
- **pose** (Pose2D) : The position and orientation of the robot
- **rate** (rospy.Rate) : The frequency we publish MarkerArray
- **resolution** (float) : Size in m of each grid cell in the maps
- **map** (OccupancyGrid) : Map of the locations we have seen to be unsafe through the camera
- **map_pub** (rospy.Publisher) : Publishes the grid of objects to avoid

Subscriber callback methods

- **color_image_callback** : Finds any obstacles within the image and samples the most recent depth image at each pixel of the obstacle objects. Each is then converted into grid space using the robots position, orientation, location within the image, and depth. This erodes to remove noise and reduce issues with synchronisation issues
- **depth_image_callback** : Converts the depth image into the correct format and stores this in a member variable for later
- **pose_callback** : Stores the pose for later use
- **map_callback** : Caches the size and resolution of the costmap so that we ensure they are perfectly lined up

Node methods

- **obstacle_detection() ()** : The constructor for this node

1.8 vision_mapping

Builds up a map of the locations the robot has seen through the camera

Publishes to "/minitask5/vision_map" using OccupancyGrid.

Subscribes to:

- **scan** using **LaserScan**
- **map** using **OccupancyGrid**
- **minitask5/pose** using **Pose2D**

Class variables

- **real** (boolean) : True if the node is running on a real robot
- **rate** (rospy.Rate) : The frequency we publish the vision map

- **pose** (Pose2D) : The position and orientation of the robot
- **map** (OccupancyGrid) : Map of the locations we have seen

Subscriber callback methods

- **lidar_callback** : Builds up the map of regions we have seen using Lidar to determine the distance to objects that could be blocking the view. This is only frontal cone based on the realsense2 FoV.
- **pose_callback** : Caches the position and orientation of the robot
- **map_callback** : The first time this is called, it creates the vision map of the exact same size and resolution as the costmap to ensure they are perfectly aligned

Node methods

- **vision_mapping() ()** : The constructor for this node

1.9 util

Provides a number of utility functions that are used throughout the different nodes

Methods

- **get_cell(map, coord) (float)** : Gets the cell from an occupancy grid using it's internal size.
- **set_cell(map, coord, value) ()** : Sets the value in the occupancy grid of the given cell to the value provided.
- **to_grid(map, world_coord) ((x,y)** : Converts a world space coordinate to grid space.
- **to_world(map, grid_coord) (Point)** : Converts a grid space coordinate to world space.
- **convert_coord(src, dst, grid_coord) ((x,y))** : Converts a coordinate from one grid space to another.
- **get_line(start, end) ((x,y)[])** : Given a start and end point of coordinates, find all coordinates on the line that connects them.
- **lerp(x1, x2, y1, y2, x) (float)** : Linear mapping of value x from between ranges (x1,x2) to (y1,y2).
- **getangle(src, dst) (float)** : Calculates the angle in world space between 2 coordinates.
- **inradius(src, dst, radius) (boolean)** : Return true if the distance between the src and dst points are less than the given radius.
- **quaternion_from_theta(theta) (Quaternion)** : Converts a Euler angle to a quaternion object.

2 Architecture & Behaviour

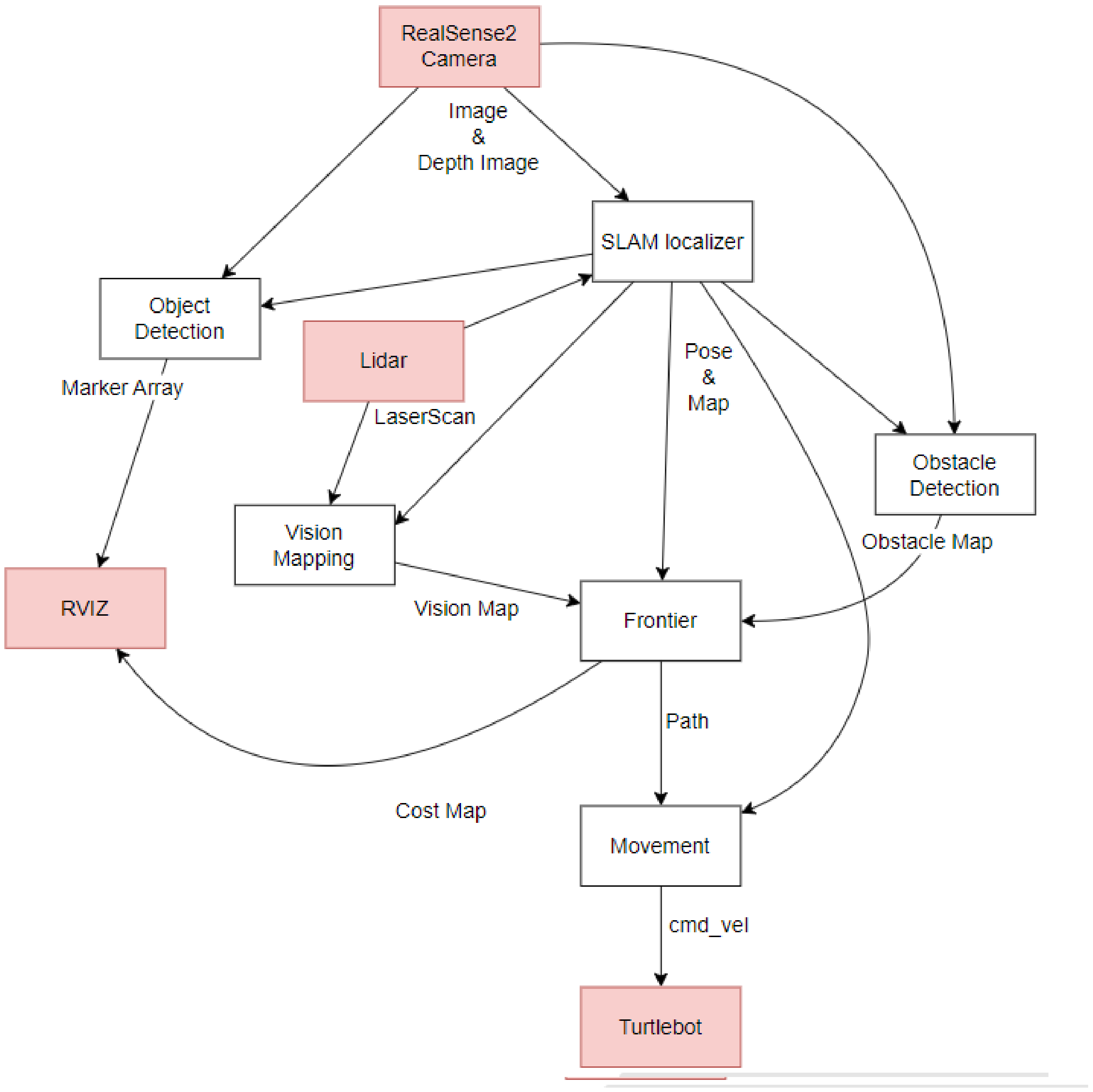

Figure 1: Architecture Diagram

The main flow of data throughout our nodes can be see in [Figure 1]. We have designed and built it as such so that there each node has a single responsibility as is recommended with ROS programming. Additionally, each node only publishes to a single topic (other than for visualisation). The entirety of this project was completed in unison using paired programming, so that the emergent behaviour can be fully understood and design decisions are fully informed. Therefore I will explain the design, implementation and testing completed for section.

2.1 Data Collection

The first nodes that are worth discussion are those who take input data from the robot and prepare this for decision making. These are all the nodes that subscribe to the Image, Depth Image and LaserScan topics.

2.1.1 Image & Depth Image

Those which subscribe to Image and Depth Image use these callbacks to identify objects within the environment based on colour. One use of this is to identify the coloured objects we are looking for within the environment. This is done within the "Object Detection" node, which uses openCV to identify contours of the desired colour (using Hue from the input image). We then sample a cached depth image at the midpoint of this coloured object and use this with the robots pose, and a number of other parameters to map the object into a real world position. This location is then marked in a probability grid as a possible location for an object, with increasing likelihood the more hits we get.

As mentioned, this node uses the current Pose of the robot as part of the object location calculation, so it must also subscribe to the pose callback from SLAM_localizer.

The location is only an estimate because the depth image and colour image callbacks are not perfectly synced up due to the concurrency of the nodes and callbacks. Additionally, the localisation can itself have issues at times, meaning our actual robot position can deviate from it's true position. This is why we build up a probability grid of objects rather than simply marking them with 100% confidence. After updating the probability grids, we threshold to remove any background noise / false positives, and then identify the connected components. For each connected component, we create a marker and publish it within the marker array, these are then identified as "found", which is the main intended behaviour of the bot.

"Obstacle Detection" is another node that uses the images, and it works almost identically to the Object Detection node. The only real difference comes with the marking of the objects, since we only care about avoiding the objects meaning we can skip some steps for efficiency. Instead, we sample each pixel within the identified object and convert this into a coordinate in an obstacle map which we will publish for the frontier node to use later. The sampling and conversion is done the same way using a cached depth image, and hence has the same issues. We chose to put this behaviour in a separate node since we wanted to publish different data than the "Object Detection" node. Additionally, because the avoidance of blue squares is a very high priority behaviour, having this being the only responsibility of a single node means that we will get quicker feedback on the matter. This was a very important design decision, as the callback on the real-sense camera is very slow, and if we are moving forward in the environment we need to identify any upcoming blue as quickly and accurately as possible.

2.1.2 Lidar

The other information we get from the robot is in the form of the 360 Lidar callback. We use this in 2 nodes: "SLAM Localizer" and "vision mapping". SLAM Localizer is responsible for liaising the information from the gmapping node we run within the launch file and converting this to a position in the world using transforms. The gmapping runs off SLAM, which uses Lidar and the image callbacks to find the location we have the highest probability of being at within the map, at the same time as building up the map. We did implement this behaviour ourselves originally, using the "Mapping" and "Localizer" nodes, but we chose to use the built in gmapping for efficiency after testing our own implementation. Both the custom and built in localsation nodes publish the same things, the current map and the estimated pose.

The ”Mapping” node uses the Lidar callback and uses the depth at each degree as the endpoint for Bresenham’s Line Algorithm. This, along with our current pose, was then used to build up a map of the environment as we moved through it.

The ”Localizer” node we built then uses image processing to identify where in the global map the robot is. This works by converting the global map into a main large image and creating a smaller local area image based on the Lidar callback we receive. We then scan this local image over the entire global map image and find at which point the sub image is most similar. This worked incredibly well in terms of accuracy, and we even discussed possible ways to improve this by addition additional probability scaling based on current velocity, but the main issue was with the speed at which the image processing took place. Therefore we decided to instead switch to using the build in SLAM which was far quicker given the time frame available. More on the testing will be discussed later.

”Vision Mapping” is an additional node that we built to take into account the FoV (Field of View) of the Turtlebot. This was created to ensure that the robot physically faces all the open space within the environment. Since we can only detect the objects we are searching for within the scene using the camera, we need to make frontiers based on the locations which we have not seen, not just those that haven’t been mapped yet. To do this, we build up a map of the exact same size and location of the cost map and mark the locations of the front 60 degrees of the Lidar (since we can’t see through / behind objects like walls).

2.2 Decision Making

All the decision making is done as part of the frontier node. Since we have already processed most of the input data, all we have to do is combine the different sources and make our decisions based of these. We use the assumption that all the other nodes have worked correctly, meaning that we know exactly which areas are safe to drive using the different maps, and the locations which need to be explored from the vision mapping.

When this map receives the global map, we build a cost map using by dilating and blurring an image of the map and converting this back into an occupancy grid. These are based on the size of the robot to provide an accurate representation of the unsafe region, as well as giving proportionally higher weights to regions extremely close to the wall, so we avoid them if possible during the pathfinding [Figure 2].

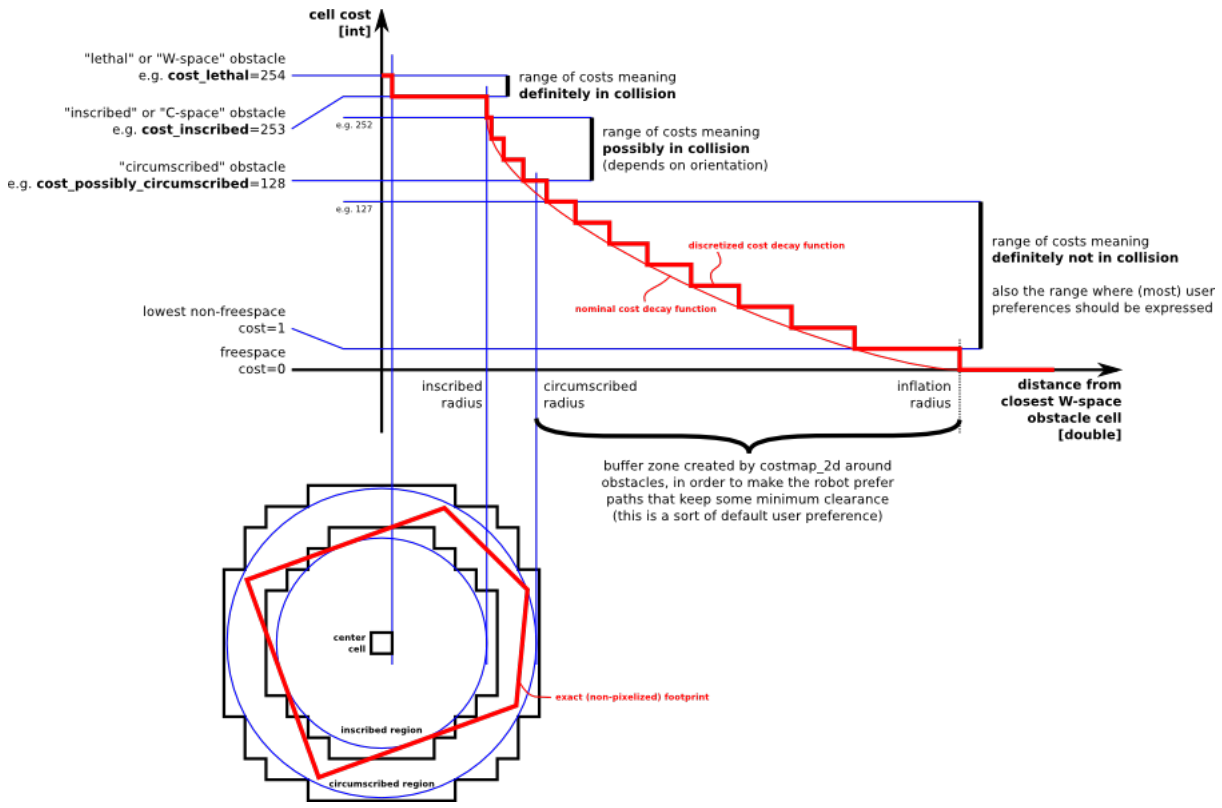
cell cost [int]
"lethal" or "W-space" obstacle e.g. cost_lethal=254
"inscribed" or "C-space" obstacle e.g. cost_inscribed=253
e.g. 252
"circumscribed" obstacle e.g. cost_possibly_circumscribed=128
e.g. 127
lowest non-freespace cost=1
freespace cost=0
range of costs meaning definitely in collision
range of costs meaning possibly in collision (depends on orientation)
range of costs meaning definitely not in collision
also the range where (most) user preferences should be expressed
discretized cost decay function
nominal cost decay function
inscribed radius
circumscribed radius
inflation radius
distance from closest W-space obstacle cell [double]
buffer zone created by costmap_2d around obstacles, in order to make the robot prefer paths that keep some minimum clearance (this is a sort of default user preference)
center cell
exact (non-pixelized) footprint
inscribed region
circumscribed region

Figure 2: Cost map inflation

The first step in the main loop is a flood fill from our current position in the cost map to the nearest frontier. This frontier is based on the FoV map, so we combine these 2 maps to perform this. This gave us the quickest way to identify the geometrically closest frontier. We then use this as a destination and perform an A* search to find the shortest route to this location within the cost map, taking into account safe and unsafe regions as explained above. If the robot identifies that it starts within an unsafe region at this point, it will first pathfind to the closest safe region, and then again from this safe location to the frontier, maximising the safety of the robot and environment before moving towards to desired goal. This is because we considered this to be a more important behaviour than a slightly quicker route. The path we get at the end is then sampled at 0.15m intervals to give a number of points to drive between, each with location and desired direction that will take us through a quick and safe route to a previously unseen location.

2.3 Action

When it comes to actually moving the robot, we assume that the path we have created will be efficient and include all the intended behaviours. The path is made of Points, representing a series of goal positions and orientations towards to closest frontier. The ”Movement” node is then responsible for moving the robot along this path by publishing Twist messages to cmd_vel to guide it along that path. This is done by treating the path as a queue, first moving towards the nearest goal, and once within a threshold of distance and orientation, we pop it from the queue and aim towards the next Point. The actual velocity we provide to the robot is based on how close the target is to being in front of the robot, and lerping the angular and linear velocity based on this angle. This is a rather simple node, because we do all the preparation before hand, the only important choices are based on the ranges for the velocity at each stage to balance an efficient speed whilst still providing accuracy.

3 Discussion

3.1 Implementation

The mapping and frontier nodes were a logical starting point since we could use the "teleop" command to manually drive the robot and view how the robot mapping behaves based on these changes. Testing and tuning could therefore be done on the single node and we could perfect these before moving onto adding the autonomous behaviour. At this point, we moved onto using SLAM instead of our custom localisation since the benefits of quick processing outweighed the hit to accuracy. Frontier and SLAM were deliberately chosen after a discussion with my partner, as we wanted to create a solution that could effectively work within any environment without existing knowledge or a map of the environment. We knew at this stage that we would suffer from lower accuracy regarding the localisation, as well as potentially taking longer to explore the environments compared to an implementation that could calculate the shortest route through the entire map. However using SLAM ensures that the robot is not overfit to a specific testing environment or map configuration, in addition to being far more adaptable to randomly allocated objects, hence our decision. Using these mapping and frontier nodes, we could then publish the shortest path to the nearest frontier based on the dynamically growing map.

Once we were publishing a path, we began implementation of a basic path follow in the "Movement" node. This uses the lerp function to make the linear speed and rotation proportional to the direction of the robot. At this stage we didn't implement the individual Point orientation goals since we add the vision mapping towards the end of the project, so the movement node just drives the robot to the points, regardless of orientation.

Once we had a robot that could crudely follow the path, we decided to add the image processing nodes. The object detection node was first, since it wouldn't change the behaviour of the robot. We implemented it almost exactly as designed, making only minor improvements in the future. We designed this to simply run in the background of the robot and mark the locations of the objects as we move around the environment, since we only wanted the movement to be based on the frontier search. This did mean we had to ensure the image processing techniques were efficient enough to not impact the refresh rate, otherwise while the robot was turning we could have missed objects. As an alternative to this, it's possible to include the camera vision as part of the movement behaviour, and stop to turn and look more confidently at the possible objects when they're detected during a movement. This would increase the accuracy of the actual object position, but has a number of other issues. Firstly, it makes the movement behaviour vastly more complicated, requiring a state machine like architecture, which is contradictory to our initial design. Additionally, it would hugely impact the speed at which the robot would be able to complete it's task, so we decided it would be more beneficial to have this node quickly processing in the background.

We then implemented the obstacle detection node, which works very similar to the object detection but we decided to keep this as a separate node. This is because the obstacles need to be taken into account for the movement so the robot doesn't drive over the blue tiles, so we publish a map of identified objects for processing in the frontier node, when makes the map of traverse-able regions. This has to be done extremely quickly, as the camera has a small FoV as well as slow refresh rate, so we decided to keep this as a separate node with just this one behaviour to reduce processing time. Any other configuration of this would result in a lower response rate to the blue tiles, other than simple stopping movement if we saw blue tiles. Although having such a directly responsive behaviour has massive implication for the robot behaviour as a whole, since it again results in a state machine type structure which we had planned to avoid due to it being so susceptible to issues.

At this stage we had a rough solution for the whole behaviour, so we did another loop through each of the nodes to make improvements, such as improving cost map inflation, adding depth changing based on height in an image and also adding heuristics to the pathfinding. We then noticed that the current frontier search wouldn't be able to fully view the whole map with the camera, so we decided to add the FoV based frontier search by implementing the vision mapping node. This node was designed as an additional input to the frontier, but also required the addition of orientation checking to each point within the movement node so that we could ensure the robot was turning correctly at the end of a path, rather than simply sitting at a frontier facing the wrong direction. This was a effective and creative solution that fitted well into our existing architecture, giving us a complete solution that followed a dynamic subsumption-like behaviour to complete that task within all the specific criteria.

As stated previously, we implemented all of the nodes and designed the full architecture using paired programming. That being said, there were a few sections which one person took more of a lead. For example, I was more leading on the image processing sections, such as with the Obstacle & Object Detection nodes, as well as coming up with the idea of adding the FoV map to improve the search behaviour to be more fitting to the task.

3.2 Testing

When it came to testing each of the nodes, we would implement the smallest measurable amount for the current node and iteratively test these alongside the implementation. This was chosen because we knew as we added more and more nodes it would become harder to understand the emergent behaviour, and this meant we could fully understand each node and ensure it was working as intended before moving on. We also knew that a large majority of the code would be very hard to quantify in terms of performance, since the nodes are so interrelated. Therefore we used a combination of integration and acceptance testing on the behaviour before committing satisfactory code and progressing onto new sections. This was performed after running the

One of the most important parts of the testing was the use of RVIZ to view the various maps and paths being built and calculated in real time. You can see all of these in our final solution, which provided great insight into the different "spaces" the robot would take into account for the decision making and its understanding of the environment. We also used RVIZ for outputting identified objects and the paths we intended to take at each section, as this would allow us to quickly compare the input data with the processed data, and finally the outcomes of the processing, allowing quick identification of issues or areas of improvement.

A few of the functions were quite simple to test, such as the A* pathfinding, or the util functions. With these being purely mathematical functions, we just logged all the inputs and outputs to the console and could quickly check these are working correctly rather than relying on deep behavioural understanding. Although we knew that this wouldn't be possible for all of the functions, which is why we opted for paired programming for the entire project alongside the use of RVIZ to help identify and understand the emergent behaviour.

Another useful part of the testing procedure was the use of openCV for the image processing. OpenCV allows the various images we use for processing to be viewed in real time, which allowed us to visualise the object and obstacle detection as the robot was moving throughout the environment. We would check each step of the pipeline for these nodes, from masking and thresholding the original input images to highlighting the centroids within the original image. This was vitally important in ensuring the image processing techniques we used to remove noise and correctly identify the desired objects were working correctly.

3.3 Parameter Tuning

When it came to tuning exact values of parameters and constants within the code, this again took a highly iterative approach, with constant changes and improvements as we implemented various sections. There were a few constants that we need to keep the same across multiple files, such as physical characteristics of the robot and camera, which we kept in a helper file so we could change all instances of them in a single point as to not forget any. These can be seen within the ”constants.py” file of the ”minitask5/util” subpackage, which remained consistent for the majority of the project.

The other parameters however, such as image processing techniques and frontier calculations, took a lot of tuning. The image processing started with us looking at colour ranges for masks and thresholds for converting to binary. Size we know that the test environment, we knew that the only red and green would be the actual objects we were trying to detect, making it possible for us to manually pick the hue colour ranges easily. We considered using AI on the input images of the objects to create a highly accurate model for identifying the required objects given input images we could take around the scene, but this was completely unnecessary since we were able to use HSV to such success.

There were a few other image processing parameters that required tuning across the various nodes, namely in the size and shapes of kernels. Kernels were used to erode noise in the background of input images, as well as inflate the cost map. The cost map inflation kernels were best tuned to be proportional to the size of the physical robot, but we did investigate smaller and larger sizes for this. A larger kernel resulted in the robot being overly cautious, taking a much longer route around edges in addition to not traversing through narrow corridors. Alternatively, have the kernel too small resulted in the robot clipping the walls at corners or whilst turning. Based on both of these, and observations about the robots behaviour, we did expand the inflation kernel to be a single cell larger than the physical robot size for the map inflation, since we knew there could be error in localisation and odometry, and this worked exceptionally well for most cases.

The object detection kernels were a little more complicated to tune. Since the size of the actual objects could change depending on the distance and angle at which we were looking at them, we had to find the balance of a large enough kernel to remove incorrectly identified colours as part of the threshold, but not so big as to remove true positives. Therefore, we had to adjust the image processing to only consider objects detected within the advised safe bounds of the camera (as provided on the real-sense website). Given these bounds on accuracy, we could determine the relative sizes of the objects at the upped bound, and use this to guide our tuning to aggressively remove all the background noise without removing the true objects. We also looked at training a model for this, providing some labelled images and finding the perfect kernel size, but since we wanted the camera to only function without the recommended bounds anyway to ensure the depth reading was accurate to calculate object position, we decided to just tune this in ourselves. After adapting our code to the recommended depth ranges of the camera, we learnt that the camera was only accurate within 2%, which caused some unavoidable leeway.

There were then only a few other parameters that required tuning for our solution to be complete. These were things like thresholds for acceptable speeds of movement, or radius for conditions being met. These were much harder to tune, since they could only really be tested with acceptance testing by watching the robot perform within the environment. The method we took for finding the best values for these was taking each of the values to the most extreme we thought it could still perform and evaluating the changes. This would then be repeated a few times until we hit a value for these parameters that was satisfactory for the intended robot behaviour. For example, we had values to determine when the robot was close enough to a point in the path for it to be considered visited. This included euclidean distance and difference in angle. We started with these both as 0, so the robot had to be in the perfect position for this to be accepted, but we found that this was far too precise and resulted in the robot going back and forth countless times to perfectly align itself. So we increase this to 0.5m, which had its own issues, and after a few iterations arrived at values of 0.1m and 20 degrees.