Home Service Robot

Ross Lloyd

Abstract—The ROS navigation stack, as well as a wall follower node and c++ nodes that provide goal locations and manipulate a virtual marker, are used to model "home service robot" behaviours. The pickup / dropoff behaviour utilises the built in ROS result message type and effectively uses goal messages to drive a correct sequence of events. A correctly configured Slam Gmapping node provides mapping services and creates a good quality map for the robot to localize itself within. Bash scripts are utilised to control the launch flow of each of the nodes.

Index Terms—SLAM, ROS Navigation Stack, Udacity, Dijkstra.

1 Introduction

PATH planning and navigation are the "decision making" steps of mobile robotics. Given a map, and the robot's location within it, it is now necessary for the robot to be able to autonomously derive and follow a navigable path that will allow it to reach its goal pose safely. With this latter point in mind, a simultaneous goal is *obstacle avoidance*. Of course, obstacle avoidance will modify the path. Respectively, these two problems can be seen as *strategic* and *tactical*. Note that obstacles may move over time, especially in environments featuring humans. As a field, its applications continue to grow, from basic home service robots such as vacuum cleaners, self driving cars, through assistive care robots and planetary rovers. Each robot type must accomplish their goals in the safest way possible with increasingly high margins of safety.

1.1 Path Planning Algorithms

A path planning algorithm can be assessed as to its effectiveness by considering whether it is complete and / or optimal. An algorithm is complete if it is able to find *any* solution where one exists. It is considered optimal if it is able to find the *best* solution, according to some definition of best, such as shortest path or shortest time. There follows a brief overview of three varieties of path planning - *Discrete*, *Sample Based* and *Probabilistic Path Planning*. Note that the diagrams here are taken from the Udacity Robotics Nanodegree classroom, except for the Minkowski sum diagram.

1.1.1 Discrete Path Planning

Discrete Path Planning (DPP), also known as *Combinatorial Path Planning*, is a computationally expensive, but extremely accurate path planning process that discretizes the robot workspace into a connected graph, then applying a *graph search algorithm* in calculating the optimal path. Its accuracy can be finely tuned by specifying how finely the workspace is to be discretized, with concomitant increase in computational cost. It is not well suited to high dimensionality problems. The process of DPP involves three steps:

1) *Create the Configuration Space*, or *C-Space*. This takes the design and configuration of the robot into consideration and better facilities the application of discrete search algorithms.
2) The C-Space is discretized.
3) The discretized space is represented by a graph.

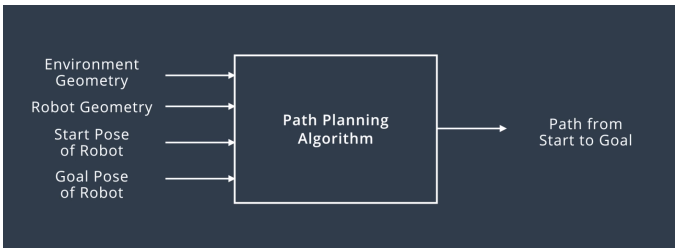

Fig. 1. Path Planning Process with Inputs and Outputs.

Then a search algorithm can be applied to find the best path. There are a broad array of approaches to solving each of these steps, a selection of which will be considered here. In *continuous representation*, all obstacles are considered and "inflated" by the radius of the robot. This creates a C-Space that ensures a path will not be selected which causes a collision with some part of the robot. The C-Space is the space of all possible robot poses, divided into free space and obstacle space.

1.1.2 Minkowski Sum

In the Minkovski Sum, obstacles are again increased in size by an amount proportional to the size of the robot (see figure below), though in this case the increase can be thought of as "painting" the outside of the shape with a brush whose radius is equal to that of the robot. This is however inefficient for non-convex shapes such as those with holes and gaps. The Minkowski Sum is calculated according to

$$A + B = \{x + y | x \in A, y \in B\}$$

Of course, for non-circular robots, the Minkowski Sum would be different for varying rotations and translations of the robot. For this reason a bounding circle can be applied to the robot model which encompasses the entire robot profile. This ensures that no collisions can take place, but does mean that any algorithm using it would not be complete - it would

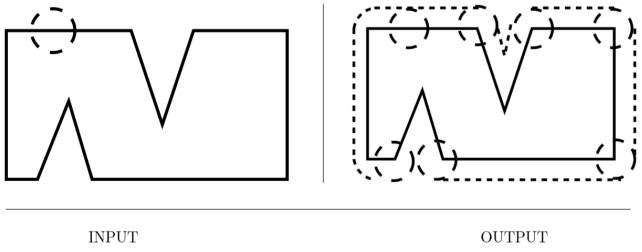

Fig. 2. Minkowski Sum *Credit: Computational Geometry, De Berg et al.*

reject some paths that the robot could traverse in a certain rotation. This can be solved by adding an axis of rotation to the configuration space, creating a *3D C-space*.

1.1.3 3D Configuration Space

In order to create a 3D configuration space, the new dimension z may be added. Then, for each degree of rotation of the robot, a new 2D configuration space is created, which is then stacked vertically on top of the previous one, leading to a geometry similar to the figure below: In this space, a

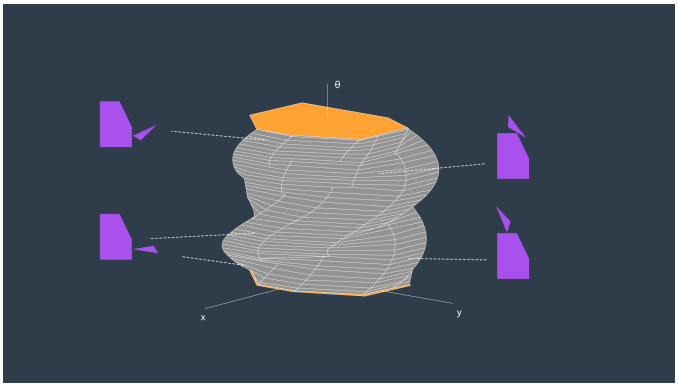

Fig. 3. Stacked 3D Configuration Space.

rotation of the robot is represented as moving upwards in the positive z direction. The contact surface of this helical shape now represents all areas where the robot will not be in contact with the obstacle.

1.2 Discretization

The configuration space must be divided in such as way that it is computationally efficient to plan a path. The process for doing this is *discretization*. Some methods for accomplishing this are outlined by the group they belong to:

1.3 Roadmap

Roadmap approaches represent a discretized configuration space through the use of graphs. There are two phases to the roadmap process - the time and computation heavy construction phase, and the faster and lighter query phase. An example of two processes used in the construction phase are the *visibility graph* and the *voronoi diagram*. In the *visibility graph* method, the configuration space is discretized by connecting the vertices of all nodes - start, object and goal - together. Then the shortest path is found along a series of these connections. This presents a complete and optimal solution. In the voronoi diagram, a roadmap is again constructed, but this time *clearance* of obstacles is maximised. This is accomplished by drawing a line between all obstacles that is equidistant to each. It does not contain the optimal solution, but it is complete.

1.4 Non-roadmap methods: Cell Decomposition

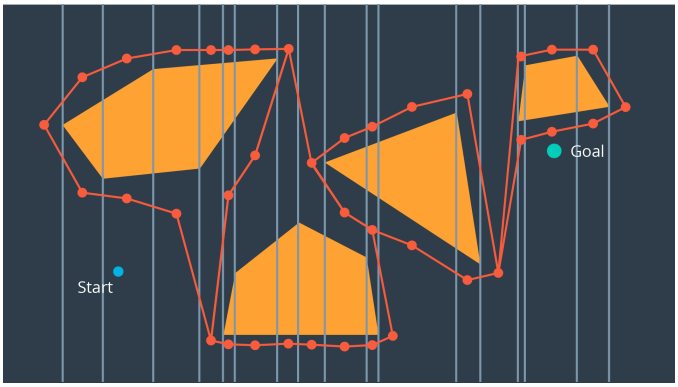

Fig. 4. Exact Cell Decomposition.

In the *cell decomposition method*, as the name suggests, the c-space is broken down into individual cells, with each cell becoming a node. There are two ways of doing this - exact decomposition and Approximate Cell Decomposition. In *exact cell decomposition* as shown in the figure above, a vertical line is drawn through every vertex of every obstacle. This divides the space up into non-overlapping cells and subdividing the obstacles into triangles and trapezoids. Once decomposed in this way, the graph can be extracted and the shortest path computed. In the *approximate cell decomposition* method, the c-space is divided up into much simpler, and readily computable, regular shapes such as squares and triangles. In *iterative decomposition*, the space is steadily decomposed into smaller and smaller quadrants of squares. This is done by marking each quadrant as full, empty, or mixed, with the latter used to represent cells partially occupied by an obstacle, but also contain some free space. The algorithm will attempt to find a path, and if one is not found, the mixed cells will be further decomposed into four quadrants. This process repeats until a path can be found to the goal. For the 2D representation, *quadtrees* are used to carry out this recursive subdivision, and for 3D space, *octrees* are employed. An illustration of how mixed spaces are further decomposed in octrees is illustrated below:

1.5 Potential Field

Another type of discretization is the Potential Field Method. Here, two functions are applied to the C-space, one which attracts the robot to the goal, and another that repels it from obstacles. By summing the two functions and then using an optimization method such as gradient descent, the robot can reach its goal.

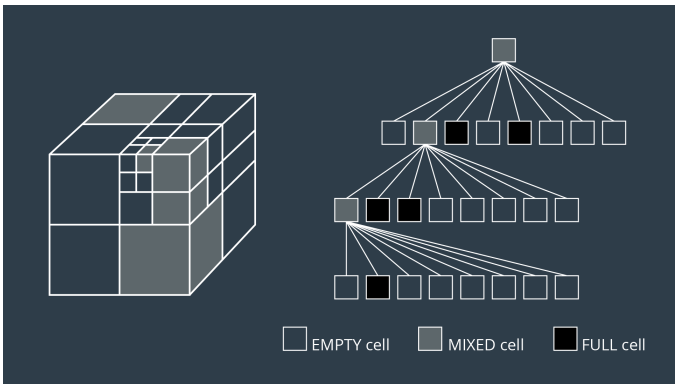

Fig. 5. Decomposition of Mixed Space in Octree Space Representation.

1.5.1 Attractive Field Potential

Here, a simple function (such as a quadratic) with the goal at a local minimum can be used to 'attract' the robot by having it follow the direction of steepest descent. It can be described as follows

$$f_{att}(x) = v_{att}(||x - x_{goal}||)^2$$

Where x is the robot's present position, x_{goal} the goal and ν is some scaling factor.

1.5.2 Repulsive Field Potential

In order to present a 'repulsive' force around obstacles and a neutral one in free space, a function is required where that function equals zero in free space, and grows to some large number near obstacles. Such a function is

$$f_{rep} = \begin{cases} \mathrm{v}_{rep}(\frac{1}{\rho(x)} - \frac{1}{\rho_0})^2 & if \rho \leq \rho_0 \\ 0 & \text{if } \rho \geq \rho_0 \end{cases}$$

where $\rho(x)$ is a function that returns the distance x from the robot to the nearest obstacle. ρ_0 defines how far the 'repulsiveness' of the function may extend (a scaling parameter), and ν is a scaling parameter. The attractive and repulsive potential fields can be summed to produce a new potiantial field, capable of guiding the robot to the goal from any location on the map. The below is an illustration of the concept, where the top left image is the attractive potential, the top right is the repulsive potential, and the third, bottom image is the sum: The gradient of the function defined above

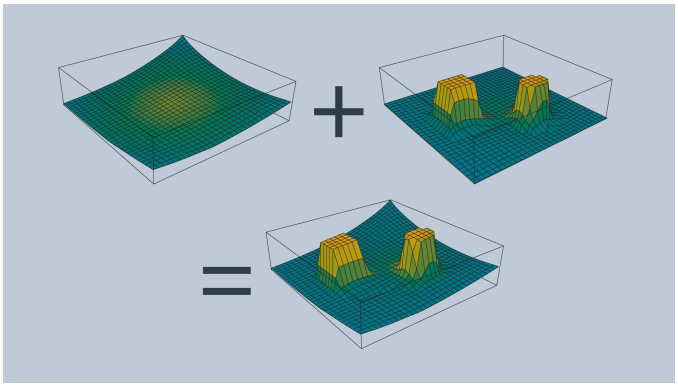

Fig. 6. Summation of attractive and repulsive fields.

dictates the direction the robot will move, and it is possible to control the speed via a constant or a scaling factor related to the distance remaining between the robot and its goal.

1.6 Advantages and Disadvantages of Discretization Methods

The *visibility graph* has the advantages of being complete and optimal, but does not leave any room for error due to the fact the path must traverse along the very edge of an obstacle. For a robot that has length and width, this is not a useful solution, though the method may be suitable in animation or video games. The *voronoi diagram* has the advantage of providing our robot with the maximum possible clearance, which is desirable for our application. One drawback however is that it cannot provide the optimal solution, due to the equidistance of each line between the objects. In *exact cell decomposition*, because the algorithm forms such neat polygons, the c-space is divided up exactly into free and non-free space. They exactly represent the space and so must contain an exact path if one exists. It is complete. However, the irregular shapes it produces cause a large computational overhead. Irregular shapes may be difficult to compute. For this reason, *Approximate Cell Decomposition* is usually preferred in robotics. A disadvantage of ACD is that it does not necessarily return an optimal solution, as it will provide a path before all possible cell decompositions have taken place. It also loses its computational strengths in high dimensionality environments. The *Potential Field* method has a number of drawbacks, namely that as it uses the calculation of local minima to guide the robot, it can get stuck in false minima rather than the goal minimum. This can be resolved by adding "random walks", but this is an added complexity and contributes to the lack of optimal path calculation. In the event the random walk fails to resolve the prbolem, the method is also not complete as the shortest path may not follow the path of steepest descent. As the output of each of these discretization methods is a graph, a graph search algorithm must be employed.

1.7 Graph Search Algorithms

There are two broad varities of search algorithm - *Uninformed* and *Informed* Searches. An uninformed search is not aware of where the goal node is until it is one cell away from it, and conversely an informed search can assess whether any node on a potential path is better or worse for reaching the goal. Three examples of **uninformed** search are, *Breadth First Search*, *Depth First Search* and *Uniform Cost Search*. An example of an informed search is *A* Search*. For each algorithm, as well as Completeness and Optimality, the *Time Complexity*, *Space Complexity* and *Generality* of each must also be considered, that is how long it takes to converge, how much memory it uses and the range of applications it can be used with.

1.7.1 Breadth First Search

In the breadth first search, as depicted in Figure 8 below, the breadth first search explores nodes on one level of the tree before moving onto the next level, exploring nodes in the order shown. The subsequent images illustrate how this works practically.

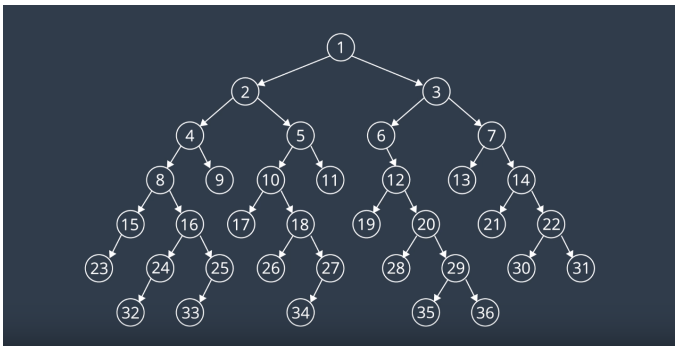

Fig. 7. Order of node search in breadth first algorithm.

In the next image, the process begins at the start node and searches for unexplored nodes Up, Right, Left and then Down (if applicable) from the current position. This search order is applied throught the entire process.

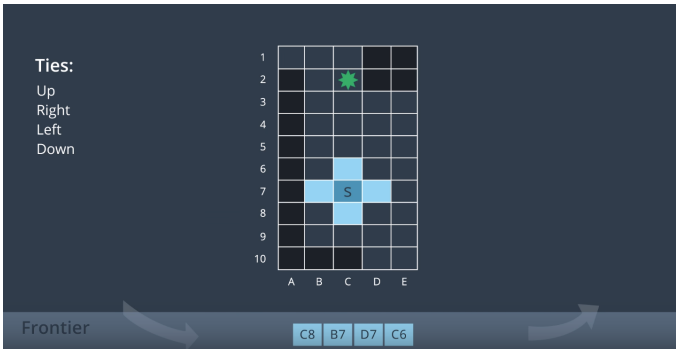

Fig. 8. Breadth First In Action - exploring start node area.

Each of the explored nodes is added to the "Frontier", a first-in-first-out (FIFO) queue which dictates the next node to be explored. The algorithm explores each of the queued nodes, expanding again the space around them: The process

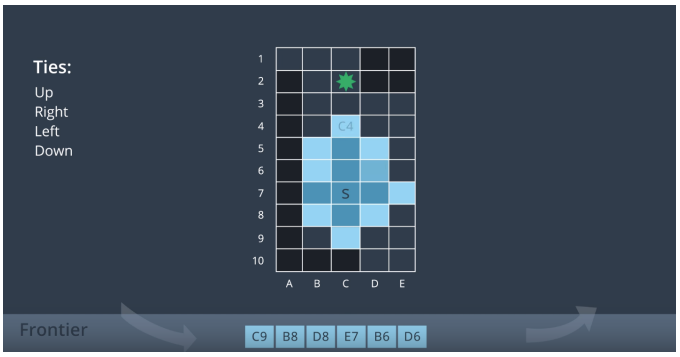

Fig. 9. Breadth First In Action - continuing exploration.

is repeated until the goal node is discovered.

1.7.2 Depth First Search

The Depth First Search explores child nodes first, moving 'down' the tree as illustrated by the following:

Practically, the exploration is similar to breadth first, except this time the queue is explored last-in-first-out (LIFO). This means that the algorithm leaves nodes very near the start position unexplored until near the end of the complete search.

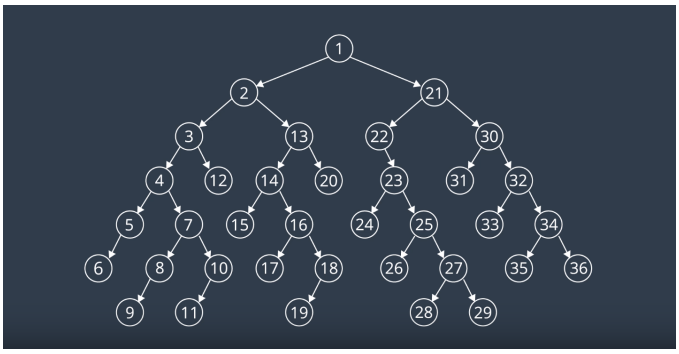

Fig. 10. Order of node search in depth first algorithm.

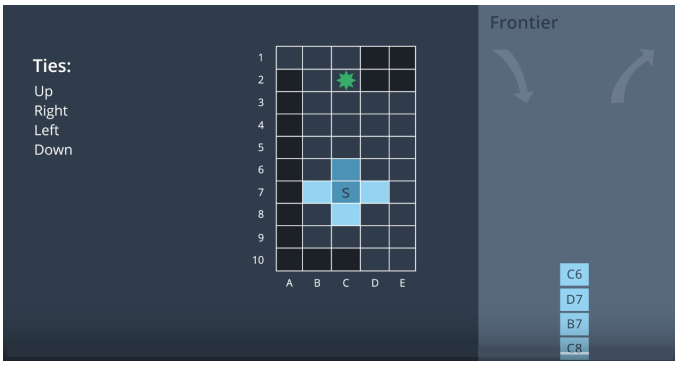

Fig. 11. Depth First In Action - exploring start node area.

1.7.3 Uniform Cost Search

Uniform Cost Search is an expansion of Breadth First Search that allows a cost to be assigned to edges between nodes. An practical example might be a path that takes the robot over rough terrain, which would be assigned a higher cost, than a path over smooth terrain. It expands the shallowest nodes as with BFS, but with the addition of expanding those paths first that have the lowest cost. First it expands paths with cost 1, then paths with cost 2, then 3 and so on. This builds in optimal path finding. Its queue system is a priority queue where those paths with the lowest cost have the higher priority.

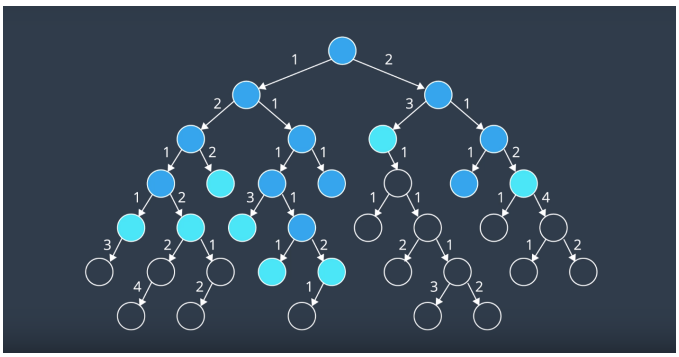

Fig. 12. Path costs and tree in uniform cost algorithm.

As each level is explored, so the shortest path to each node may be updated, and this path stored. This takes place all the way to the goal node, as illustrated below:

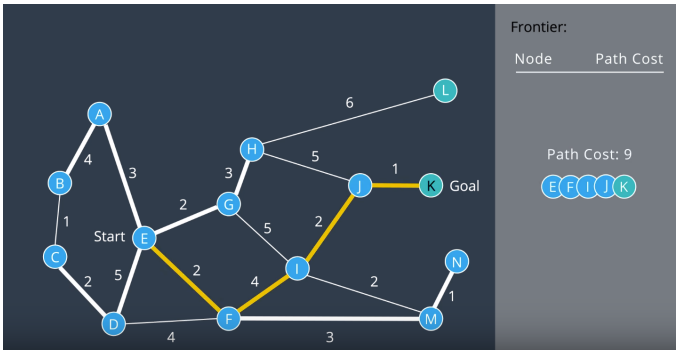

Fig. 13. Uniform Cost In Action - exploring start node area.

1.7.4 A* Search

A* search is an *informed search*, which means it has information about which paths are more desirable than others, removing the need to search every node. This is accomplished through the use of two heuristic functions

$$h(n) = distance \quad to \quad goal$$

where $h(n)$ is an estimate of distance and

$$g(n) = path \quad cost$$

. A* chooses the path that minimises the function

$$f(n) = g(n) + h(n)$$

effectively favouring the shortest path, and paths in the direction of the goal. An example of one such heuristic is shown in the figure below. As before each edge has a cost, but each node is also assigned a value according to its euclidean distance, or its straight line distance to the goal as shown by the purple line from node I to node k:

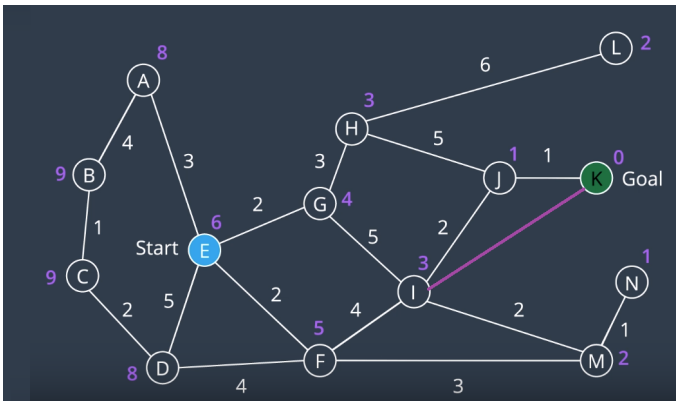

Fig. 14. A* heuristic, given by euclidean distance from node to goal.

The function $f(n)$ can now be calculated for each node one step from the start node by adding the edge cost and the heuristic cost:

The value of f(n) is used to determine the priority queue. As the most promising paths are explored, so the queue is updated as new nodes are explored and shorter paths updated.

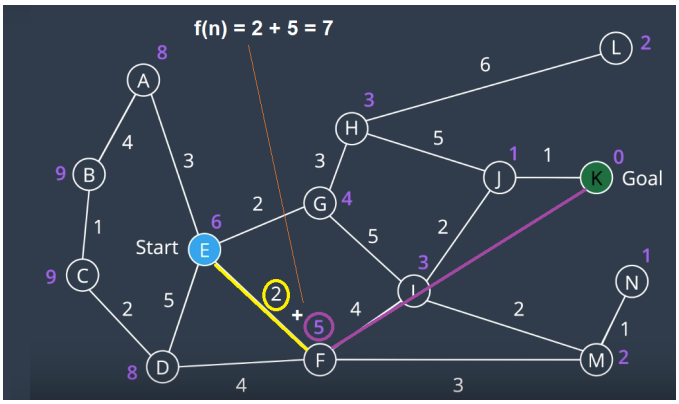

Fig. 15. A* calculating f(n).

1.7.5 Criteria affecting optimality of A*

A* is optimal only under a very specific set of conditions:

- Every edge is required to possess a cost larger than some value, ϵ. If this is not the case, the algorithm may become stuck in an infinite loop, and the search would not be complete
- The heuristic must be consistent, that is obey the *triangle inequality theorem*.
- The heuristic function must be *admissible*. That is, h(n) must be less than or equal to the true cost of traversing the path to the the goal from every node i.e. h(n) must not overestimate the true path cost.

Using the euclidean distance ensures that the heuristic is always admissible. There are variants available for A* which can be used in dynamic environments. Still others adapt the algorithm to be more useful in very large environments.

1.8 Advantages and Disadvantages of Graph Search Methods

The *Breadth First Search* method is optimal and complete, as it always finds a solution and it finds the shortest path due to the fact it always expands the shallowest nodes. However it is very inefficient as it must explore every node in a gradually radiating manner from the start node. The *Depth First Search* is neither complete nor optimal, as for an infinitely large configuration space it will explore one branch indefinitely. It is also not optimal as it may find a longer path down a branch it explores before a later, shorter one. The Uniform Cost Search is complete and optimal, as it prioritises the shortest path by design. It is however still not efficient, as it must still explore in all directions. A* is much more efficient in most mapping environments and applications. However, if the path to the goal initially goes in the opposite direction, it will perform worse than other algorithms.

1.9 Additional concerns with search algorithms

With *Bi-directional search*, in which two searches - one from the start node, and one from the goal node - are carried out, the number of nodes that need to be expanded by a search is significantly reduced. The two paths will meet indicating a path between the goal node and start node.

Path Proximity to Obstacles: Discretization methods such as cell decomposition do not differentiate empty cells from one another, and as a result the robot can pass very close to obstacles as it follows the optimal path. Obstacles can be marked with a higher cost than free cells, causing the robot to pass much further away. This process is called *smoothing*.

Paths Aligned to Grid: As the robot will try to follow the cell boundaries in discretized spaces, this can lead to the robot zig-zagging when a more direct path is available. Again a smoothing can be applied to increase the distance that the robot passes along these edges.

1.10 Sample Based and Probabilistic Path Planning

Sample Based Planning is a graph based planning algorithm. Though it may not produce the optimal path, it is able to rapidly produce a feasible path by taking a limited number of samples and builds a discrete representation of the workspace. Unlike discretization, it takes a representative sample of the configuration space. It is much better adapted to higher dimensionality environments and with robots that possess a number of degrees of freedom. It is also more useful in the case where there are constrained dynamics and complex systems that rely on derivatives of robot variables, such as velocity, i.e. non-holonomic robots. Sample based path planning relies on *weakened requirements*. Rather than looking for a complete search algorithm as in discretized methods, instead sample based searches are ones that are *probabilistically complete*. If a path exists, its probability of finding it increases to one as time goes on. Similarly rather than an optimal solution, sample based methods instead look for a *feasible* path. This is one that obeys all obstacles and motion constraints. A feasible path shows that a path exists, and then local optimzations can be found in order to improve performance. These weakened requirements lead to methods that are far more efficient, if not as accurate. A major disadvantage to the method is that insufficient samples may lead to no path being found.

1.10.1 Probabilistic Roadmap (PRM)

In the PRM method, in the **learning phase** the algorithm creates a set of sample nodes and then checks to see if any of those nodes are inside of an obstacle. If they are, they are removed. Once it has a sample space, it will try to connect the nodes via an edge. It can search for nearby nodes via various means, such as a standard radius search or a k-means search. Then for a given pair of nodes, it determines if a collision exists by creating another set of nodes between them and again determining if any of these intersitial nodes are in an obstacle. If the samples return "no collision found" then a new edge can be added to the graph. In the illustration below, nodes have been found with no obstacle, and one with an obstacle between them. The edge with the obstacle will be removed.

The *query phase* then follows. The following parameters must also be considered when optimising the learning phase for a given application:

- **Number of iterations** - this controls how detailed the resulting graph will be. In a very open space with few obstacles, a low number of iterations may be acceptable. However for a map with thin passageways, insufficient detail may lead to no path being found.
- **Node discovery method** - as mentioned above, there are a number of methods for locating nearby nodes, such as k-means and radius search.
- **Local planner method** - this is the approach used for finding free paths or obstacles between individual nodes. A fast and efficient planner is preferred for most applications though in some cases another PRM may be desirable to find the interstitial nodes.

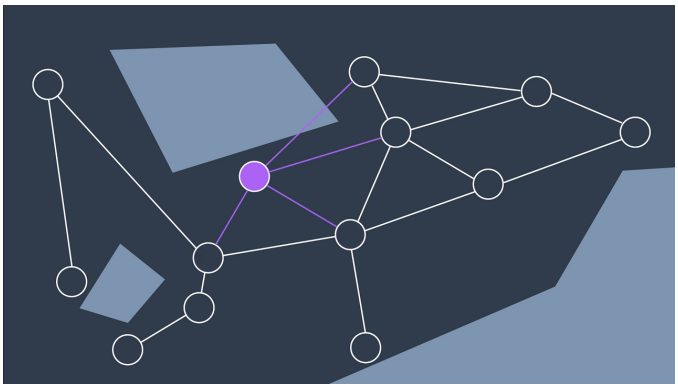

Fig. 16. PRM finding edges between nodes.

```
Initialize an empty graph
For n iterations:

   Generate a random configuration.
   If the configuration is collision free:

      Add the configuration to the graph.
      Find the k-nearest neighbours of the configuration.
      For each of the k neighbours:

         Try to find a collision-free path between
            the neighbour and original configuration.
            If edge is collision-free:
                Add it to the graph.
```

Fig. 17. PRM pseudocode for learning phase.

PRM is descibed as a multi-query planner, that is once the graph is built during the learning phase, multiple queries can be made of it. Though some environments change too rapidly for this to be exploited, in general this is a strong advantage of the PRM method.

1.10.2 Rapidly Exploring Random Tree Method (RRT)

In the event that the environment changes too rapidly, the *Rapidly Exploring Random Tree Method (RRT)* can be used instead. The difference with this method is that it is single query, creating a new graph for every query. The new graph is smaller but more directed. The process is as follows:

- Unlike PRM which adds in the start and goal nodes as it builds the graph, RRT includes them from the start.
- RRT builds a tree instead of a graph, where each node has only a single parent. Lateral node connections are not considered. A new node is added to the tree.

- If the new node is within some distance δ and not in collision, then it is added to the tree.
- If a new node, for example called x is farther than this distance, then a different new node, for the sake of example called y, is created in the same direction but at distance δ

```
Initialize two empty trees.
Add start node to tree #1.
Add goal node to tree #2.
For n iterations, or until an edge connects trees #1 & #2:

    Generate a random configuration (alternating trees).
    If the configuration is collision free:
        Find the closest neighbour on the tree to the configuration
        If the configuration is less than a distance δ away from the neighbour
            Try to connect the two with a local planner.
    Else:
        Replace the randomly generated configuration
            with a new configuration that falls along the same path,
            but a distance δ from the neighbour.
        Try to connect the two with a local planner.

    If node is added successfully:
        Try to connect the new node to the closest neighbour.
```

Fig. 18. RRT pseudocode for learning phase.

Tunable parameters include:

- **Sampling Method** - The search can be made "greedy" and so create more samples around the goal, or alternatively search the entire space. Biasing towards the goal is especially beneficial in the case of RRT as it is a single query method. Tuning this parameter too high can cause the process to get stuck in local minima.
- δ - this will dictate the growth rate of the graph, as new nodes will be closer to or further away from their parents. Too high and the graph may miss important details, and lead to nodes that cannot be connected due to collisions. Too small and the graph will be too dense and therefore more computationally demanding.

Both PRM and RRT are well suited for robots with many degrees of freedom. RRT is much more efficient in high dimensional spaces than PRM.

1.10.3 Path Smoothing

The output of RRT and PRM is not usually a nice smooth path, and for this reason it is desirable to apply some form of path smoothing. Also known as a *path shortcutter*, such an algorithm can identify and attempt to connext two non-neighbouring, and collision free, nodes together. This path may still not be optimal but does rerutn a feasible path. Other factors, such as smoothness and safety, can also be smoothed for. Its not just path length that may be used as a factor. A large downside to path smoothing is that the smoothing process may take longer than the initial feasible path identification process.

1.10.4 Sample based path planning summing up

Though powerful, sample based path planning is ultimately not complete, as it may fail in certain environments when

```
For n iterations:

   Select two nodes from the graph
   If the edge between the two nodes is shorter than the existing path
     between the nodes:

        Use local planner to see if edge is collision-free.
        If collision-free:

           Replace existing path with edge between the two nodes.
```

Fig. 19. Path smoothing pseudocode.

the sample distribution is not fine enough. Alternate methods such as bridge or gaussian sampling may help in some environments, but not all. Additionally, as the graph building stage of sample based programming does not return the most optimal path choices, an algorithm such as A* will never be able to find the optimal path either.

1.11 Probabilistic Path Planning (PPP)

PPP allows us to build in uncertainty into our path planning, taking into account sensor noise. This allows for a much higher margin of safety as now the robot is less likely to accidentally come too close to dangerous obstacles. This can be achieved by using a Markov Decision Process (MDP). Terms used in an MDP are:

- Set of states, S
- Initial State, S_0
- A set of actions, A
- The transition model, $T(s, a, s\prime)$, which is the probability of reach state $s\prime$ from state s by taking action a.
- A set of rewards, R

The *Markov Assumption* states that the probability of making the transition given by the transition model is independent of the path taken, and only influenced by the starting state s. In MDP's for path planning, the robot is aware of state, actions, transition model, rewards. Below is an illustration of a simple transition model: The figure suggests that when

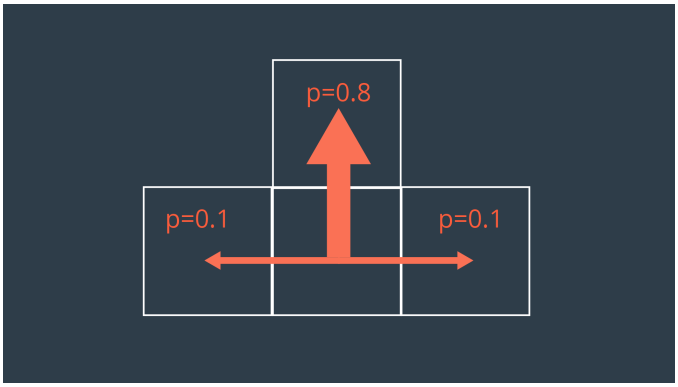

Fig. 20. Illustrative transition model

commanded to move forward, there is an 80% chance the robot will do as intended, a 10% chance the robot will go left, and a 10% chance the robot will go right. An additional constraint might be that bumping into a wall causes the

robot not to move from the cell it currently occupies. It is then possible to apply a rewards scheme similar to the illustration below. Here we see that each new cell moved to has a small negative reward (to maintain robot speed), each obstacle has a larger negative reward, and potentially "lethal" obstacles have an extremely negative reward. The goal has a high positive reward: Note that this will have

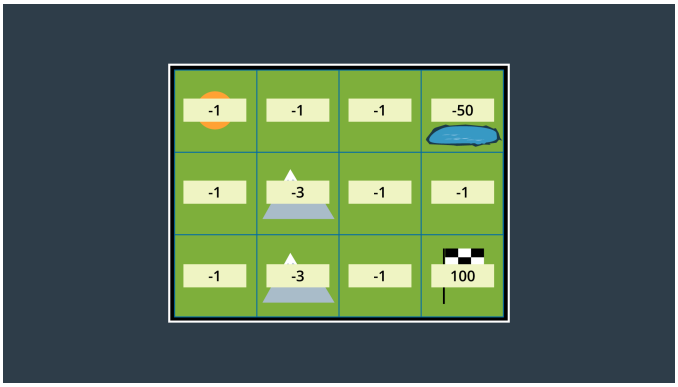

Fig. 21. Illustrative reward scheme for MDP in PPP.

a very different result than if a path was found using A*, which would likely take a route past the mountains and near to the pond, potentially resulting in the robot sinking due to positional inaccuracy.

1.11.1 MDP Policies in PPP

Solutions to the MDP are termed *policies*, and denoted by π. The purpose of a policy is to inform the robot of the correct decision to make for any given state, and the optimal policy is denoted by π^*. Given the illustrative transition model and the reward mapping above, it can be stated that, the probability of the robot moving from the bottom mountain cell, to the cell one place to its right correctly when commanded, is calculated as

$$expected\ reward =$$
$$0.8 * (-1) + 0.1 * (-3) + 0.1 * (-3) = -1.4$$

as there is an 80% chance of reaching the correct cell, a 0.1% chance of going up and 0.1% chance of bumping into the wall below.

1.11.2 State Utility

The aim of using the MDP is to find the *optimal* decision to make at each state. For each state, the State Value Function gives the expected return if the robot begins in that state and subsequently acts according to the policy for all available time steps. The state value function is given by

$$U^{\pi}(s) = E(\sum_{t=0}^{\infty} R(s_t)|\pi, s_0 = s)$$

where

- $U^{\pi}(s)$ represents the utility of state s
- E represents the *expected value*
- $R(s)$ represents the reward for state s

Summing all the rewards that the robot would accrue if it starts at state s and follows the policy to the goal is the *utility of the state*. It is thus an iterative process, and the utility of any given state is dictated by the policy. The equation can be rearranged as

$$U^{\pi}(s) = R(s) + U^{\pi}(s\prime)$$

where $s\prime$ is the next state. It follows that the optimal policy can be given by

$$\pi^*(s) = \underset{a}{argmax}\, E[U^{\pi}(s)]$$

In the example above, starting from the lower mountain cell and taking the most rewarding path, the utility of state of going one step to the right, and then another to the goal is -1.2 + 79.8 + 0 = 78.6. This is because the UOS is arrived at by beginning at the goal and working backwards: If already at the goal, there is 0 reward. Moving from the cell adjacent to the goal would yield a reward of 79.8, and moving from the lower mountain cell to the one adjacent to the goal would yield -1.2 once the above policy is applied. Finally, discounting must also be applied - a method which leads to the robot to value rewards in one time frame over another, for example valuing rewards it gets sooner over those it receives later. The discounting factor is given by γ:

$$U^{\pi}(s) = E(\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi, s_0 = s)$$

This is useful since in robotics, future timesteps bring greater uncertainty.

1.11.3 Value Iteration Algorithm (VIA)

The *Value Iteration Algorithm* is used to find the optimal solution to the MDP automatically, and is given by

$$U(s) = R(s) + \gamma max_a \Sigma_{s\prime} T(s, a, s')U(s')$$

by first initializing the state value function to some arbitrary value such as zero. It then iterates, gradually converging on the most accurate value for the SVF. The algorithm for the VIA is given by:

$U' = 0$

loop until $close\text{-}enough(U, U')$

$U = U'$

for s in S, do:

$U(s) = R(s) + \gamma max_a \Sigma_{s'} T(s, a, s')U(s')$

return U

Fig. 22. Value Iteration Algorithm pseudocode.

The algorithm also uses an additional ”close-enough” function, which detects when the algortihm has converged on a value within some given tolerance. *Probabilistic Path Planning* is a more capable form of Path Planning, which can factor in the uncertainty of the particular robot design and sensors. This is especially useful in environments requiring higher margins of safety. Unlike discretization, it takes a representative sample of the configuration space. It is much better adapted to 3d environments. It should be noted that path planning is an area of active research, and many authors are currently studying Asynchronous Advantage Actor-Critic (A3C) algorithms, a type of Reinforcement Learning approach.

1.12 Dijkstra Algorithm

Dijkstra’s Algorithm is a robust graph based path planning algorithm that finds the shortest path in a weighted graph, a weight being some value such as distance or traversability. The shortest path than becomes the one which returns the smallest sum of weights. The ROS Path Planner uses a variant of this algorithm.

1.13 ROS Path Planner

For the Home Service Robot project, the ROS navigation stack will be used to allow the virtual turtlebot to navigate to various points on the map.

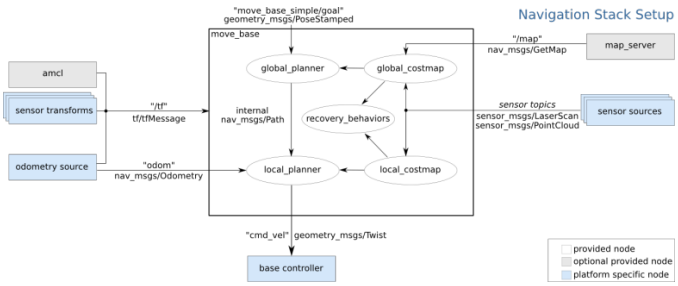

Fig. 23. Overview of ROS navigation stack.

With reference to the above diagram from the ROS wiki, the section of interest here is the area within the box, specifically the global and local planners. The outputs needed to drive a robot are generated by the **move_base** node. The node contains a global and local costmap, which it then uses to create global and local plans. A global plan is created using the **navfn** package, which in turn uses the dijkstra algorithm (or A* as an option) to find the lowest cost path to the goal given the current known global map. Then a local plan is generated by the DWAPathPlanner (Dynamic Window Approach). This factors in the dynamics of the robot to ensure it can comply with the intended path given its speed, and produces a stream of velocity commands. It continuously updates the robot’s intended direction in order to ensure it adheres as closely as possible to the global plan, within certain tolerances set by the user. In the picture below, the red line is the local plan, the green is the global plan, the small ’cone’ protruding from the robot represents the trajectory analysis. The colourful square is the grid representing the local costmap. The inflation layers can be seen on the map walls:

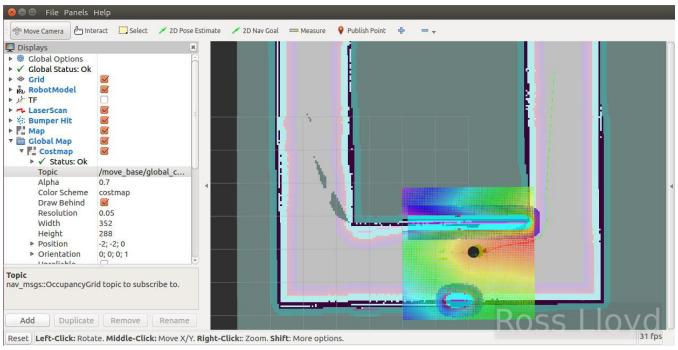

Fig. 24. Illustration of navigation elements within RVIZ.

move_base also provides an implementation of an **action server**, which uses supplied goal pose data to define its goal. The action client is an elegant way of broadly tracking the status of goals with simple messages. For an overview of important topics, see the packages section of ”Localization and Navigation” below.

2 Project Sections

The standard Turtlebot packages were used for this project, the relevant ones for each section are listed below:

2.1 Gazebo World

The following *u_world.world* file was made for the robot to navigate, using the gazebo building editor.

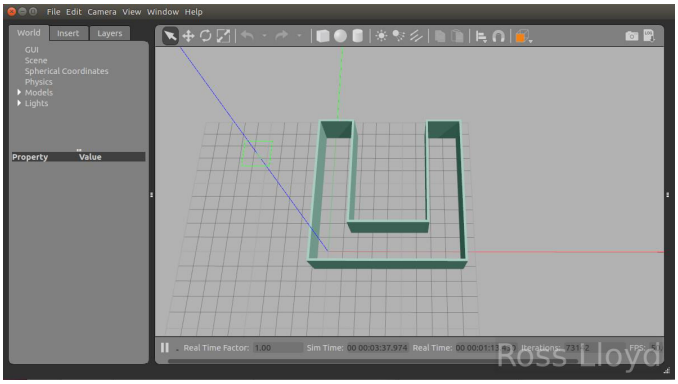

Fig. 26. Gazebo world created in Building Editor.

2.2 Testing SLAM and creating a map

2.2.1 Packages Used

All necessary packages for SLAM testing were launched using the *test_slam.sh* bash script. The mapping with wall follower step was launched with the *wall_follower.sh* script.

1) *turtlebot_gazebo turtlebot_world.launch* : This file was modified to use the custom *u_world* gazebo world. It starts the world with a turtlebot robot.
2) *gmapping_kinect_gmapping.launch* : This is a new file created and added to launch the mapping node, instead of using the gmapping_demo.launch. It was more straightforward to create it and add the necessary parameter lines than work with the original

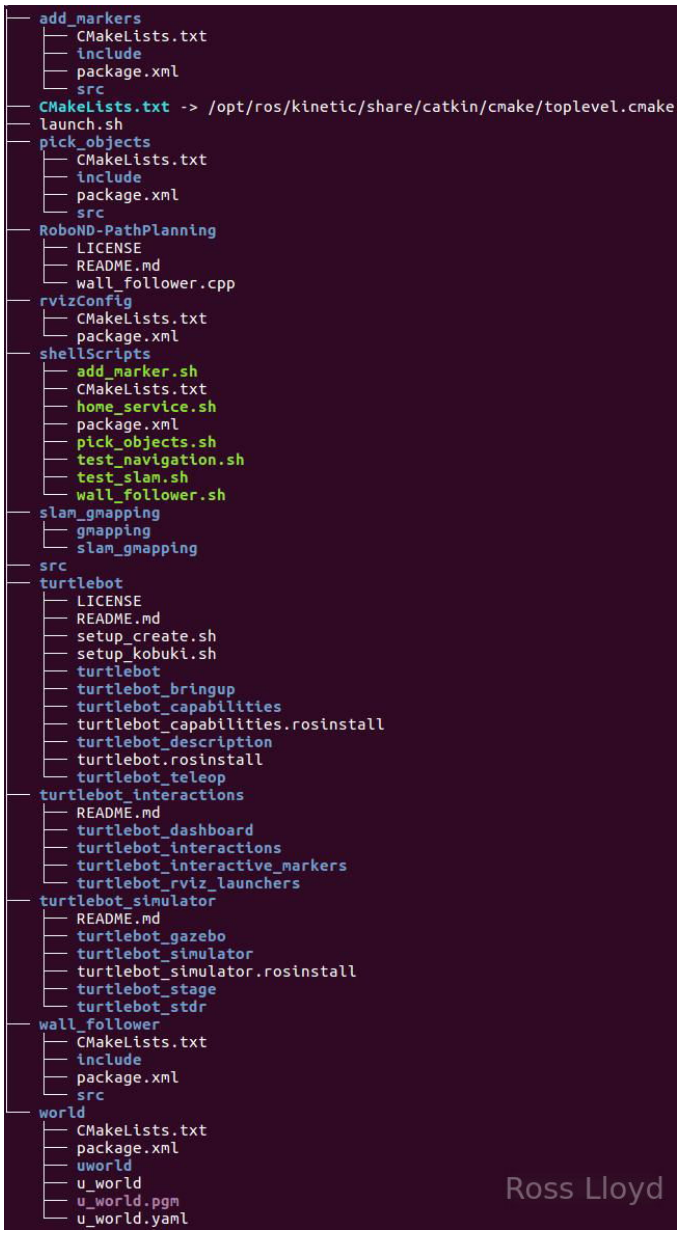

Fig. 25. Overview of Catkin workspace directory structure.

xml "include" format of the demo file. Important topics are:

- **/map** - the output occupancy grid published by the node
- **/scan** - virtual laser scan created from the kinect depth data using *depthimagetolaserscan*.

3) *turtlebot_rviz_launchers view_navigation.launch* - links to *turtlebot_rviz_launchers)/rviz/navigation.rviz*, an rviz configuration file for the turtlebot.
4) *wall_follower_node* - A c++ node which uses a left hand wall following algorithm to autonomously map the area.
5) - *keyboard_teleop* node, to allow control of robot with keyboard.

2.2.2 Parameters

This part of the project focused on creating 2D maps with the built-in ROS slam-mapping tool, *slam_gmapping*. In order to create good quality maps, the following parameters were adjusted in the *kinect_gmapping.launch* file:

- *minimumScore* was set to 10000. This value drives the scan matching algorithm which localizes the robot in order to correct for odometry. However, when this value is set too low in featureless environments, it can lead to the robot 'jumping' back and forth in the map and leading to high levels of inaccuracy in the map dimensions. Setting this high ensures that only very high probability matches are accepted. Until this value was set it was extremely difficult to produce an accurate map.
- *particles* was set to 40. Again part of the scan matching / localization. This value did not have as much of an impact as minimumScore, but does improve the map somewhat. Increasing its value over 80 begins to impact the performance of the mapping algorithm, which leads to worse maps.
- *linearUpdate* was set to 0.2
- *angularUpdate* was set to 0.2 . Combined with the above setting, this ensures much better coverage of the map area, with less 'dead' areas.
- *map_update_interval* was set to 1 second. Again this helps prevent gaps in the map.

2.3 Localization and Navigation

In this section, the map created by the mapping process is used to test localization and navigation on the turtlebot. Goals were sent to the move_base node by using the Set 2D Nav Goal button on the RVIZ GUI. The nodes were launched with the *test_navigation.sh* bash script.

2.3.1 Packages Used

All necessary packages were launched using the test_navigation.sh bash script

- *turtlebot_gazebo turtlebot_world.launch* - as above.
- *turtlebot_rviz_launchers view_navigation.launch* - as above
- *turtlebot_gazebo amcl_demo.launch* - initializes the map_server node, which uses the map generated by the mapping step. Also launches the amcl node by linking to another amcl launch file in the system-installed *ros-kinetic-navigation* package. Important topics:
  1) **/particlecloud** - publishes the amcl particle cloud to be visualized in the pose array.
  2) **/scan** - subscribes to the *depthimagetolaserscan* generated laser data.
- *move_base* - carries out the path planning and execution for the robot. Important topics:
  1) **/navigation_velocity_smoother/raw_cmd_vel** - actuation commands to the robot from the node.

2) **/move_base/goal and /move_base/current_goal** - contain the current target pose for the robot.
3) **/move_base/global_costmap/costmap** - global costmap data based on the map file / inflation settings etc.
4) **/move_base/local_costmap/costmap** - as above but for the local costmap.
5) **/move_base/NavfnROS/plan** - this is the global plan in x.y.z etc points generated based on the costmap and provided goal
6) **/move_base/DWAPlannerROS/(multiple)** - DWA provides the local plan / velocity commands derived from the plan and kinematic model.
7) **/map** - the generated map being published by the map server
8) **/odom** - subscription to wheel encoder position feedback
9) **/scan** - subscription to laser scan data

2.3.2 Parameters

The default parameters were used for all packages, except passing the generated map of the new gazebo world to the map server.

2.4 Home Service Functions

In this section, all the previous steps were combined with a custom written node - *add_markers_goal.cpp* - to publish markers at the previously defined goal locations as well as a node that sends the same goal positions to the move_base node, called *pick_objects.cpp*. The robot was to drive to the pickup point, whereupon the marker would disappear as if being picked up. There would be a 5 second pause, after which the robot would navigate to the dropoff point and "drop off" the marker, i.e. the marker will appear at the dropoff point. All packages were launched with the *home_service.sh* bash script, though the more basic testing of goal navigation behaviour was achieved with the *pick_objects.sh* bash script.

2.4.1 Packages Used

Packages are the same as for the navigation and localization step, with the addition of:

- *pick_objects pick_objects* - sends a goal request to the move_base server through a SimpleActionClient, in this case a pickup and dropoff point. It starts the **move_base** action client to keep track of the goal status.
- *add_markers add_markers_goal* - this node was written for the project. As the task requirement for this part allowed for a lot of freedom in its execution, the *move_base/result* topic, which is only published when the robot reaches a goal, was used to trigger a callback. The callback identifies which goal has been reached, and then uses some simple logic within the main **while** loop to call a function which either hides the marker in the case of goal 1 (pickup) or calls the pose of the dropoff marker for goal 2. *move_base/result* has a message type of move_base_msgs/MoveBaseActionResult. The status/goal_id/id field carries the goal 1 or 2 data, and a c++ search function identifies a substring. Using this goal driven approach eliminated the need for thresholds with an odometry approach, which can take a great deal of trial and error. Important topics:

1) *visualization_marker* - publishes the marker
2) *set_sw_case* - publishes a variable used in the marker logic
3) *move_base/result* - subscription to the result topic.

In addition, though not used in the final application, an *add_markers.cpp* node was created that instantiates a marker at the pickup point for 5 seconds, hides it for 5 seconds, and then makes it appear at the dropoff point.

2.4.2 Parameters

See localization and navigation section.

3 Results

3.1 Testing SLAM and creating a map

It can be seen from the pictures below that the testing of slam functioned well, and was improved by the setting of parameters in the *kinect_gmapping.launch* file. Controllability of the robot was increased by setting the angular speed of the keyboard teleop node lower:

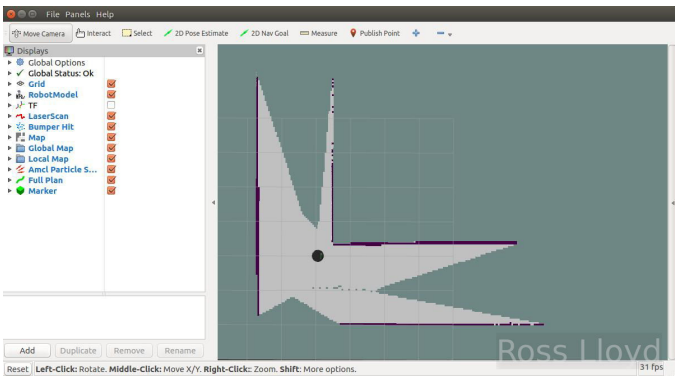

Fig. 27. Map during test of SLAM functions, manual input.

The picture below shows the robot and the map it is creating whilst using the wall follower node. It can be observed that the robot tends to miss some parts due to the field of view of the sensor and the robot's movements. Most of these can be filled in by setting the update rates (temporal and translational / rotational) to be faster.

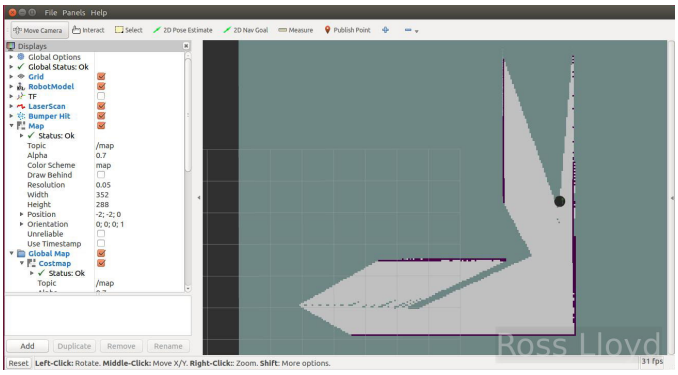

Fig. 28. Mapping whilst using the wall follower node.

Below is an illustration of the gaps left in the map when the update rates are too high:

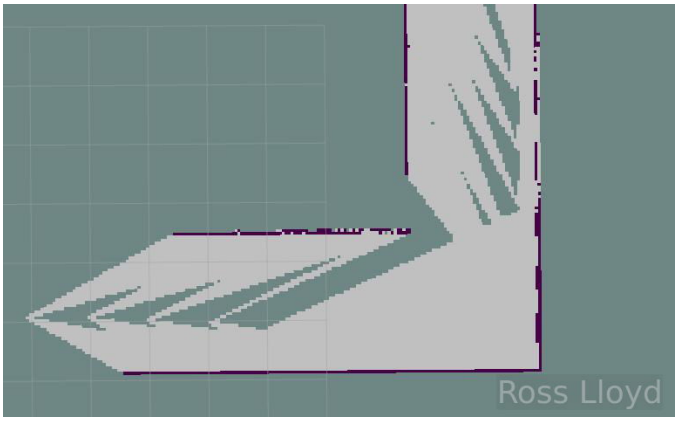

Fig. 29. Effect of setting update rate too long.

On the top right of the page is an illustration of the final map produced by the mapping algorithm. There is some slight misalignment which is introduced when the robot rounds the corner from the right-hand upright, into the horizontal ”crossbar” of the U shaped. It can also be observed that there is a small hole still remaining in the map which is caused by the wide turn the wall follower node takes on these types of corner (also observed on the left most upright). Before setting the minimum_score parameter to a high value, the dimensions of the map were very poor, with one upright of the U being shorter than the other, or the map compacted into a small square, depending where the robot started. Making the scan matching / localization aspect of gmapping require a very high score improved the map and prevented gmapping wrongly localizing the robot and introducing large errors into the map.

3.2 Localization and Navigation

The robot was able to navigate smoothly to provided 2d Nav Goals, and did not appear to struggle with any portion of the map. From time to time the recover behaviours would initiate but this was not repeatable. At right-middle of the page is the scene observed on starting the packages. The robot can be seen at the origin, surrounded by the AMCL particle cloud. The inflation layers for both global and local costmaps are seen.

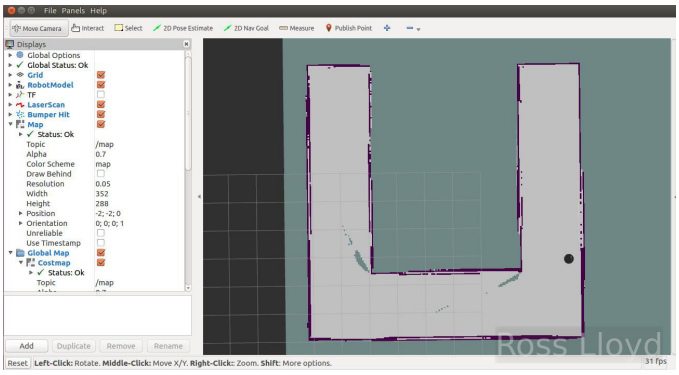

Fig. 30. Final Map.

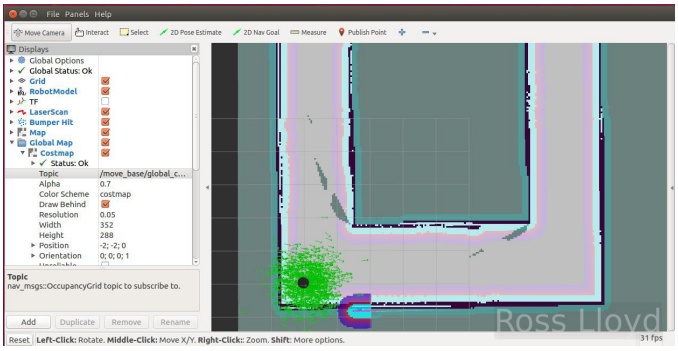

Fig. 31. Testing navigation.

The picture below illustrates the cost cloud visualization for the local planner.

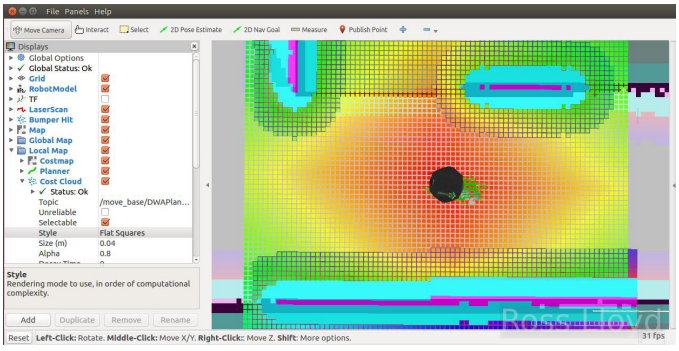

Fig. 32. The colourful region is the Cost cloud.

The picture below shows the robot navigating to a goal set by the 2D Nav Goal function on the RVIZ GUI. The red path is the local plan and the green is the global plan.

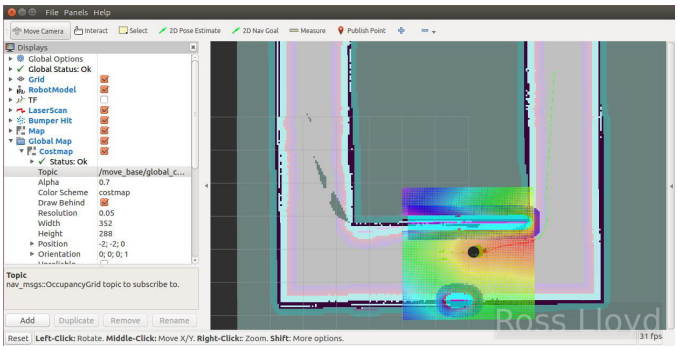

Fig. 33. Robot navigating to a point set by using the 2D Nav Goal function.

Below: Illustration of converged AMCL cloud, which was adequate for this section of the project. The grey cone is the trajectory cloud of the DWA path planner.

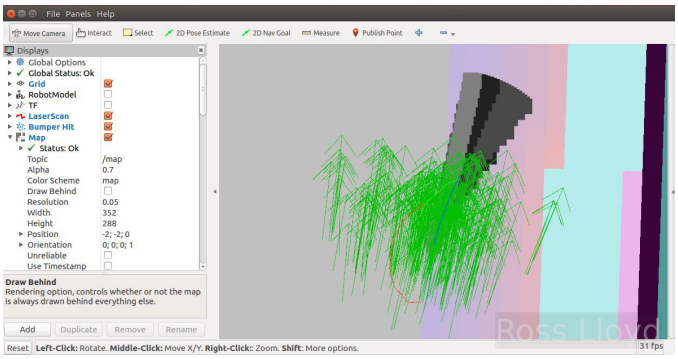

Fig. 34. AMCL Particle Cloud.

Below: In the areas where the map was not 100% accurate, it can be observed that the inflation layers are not completely aligned with the real world. However, this did not impact navigation.

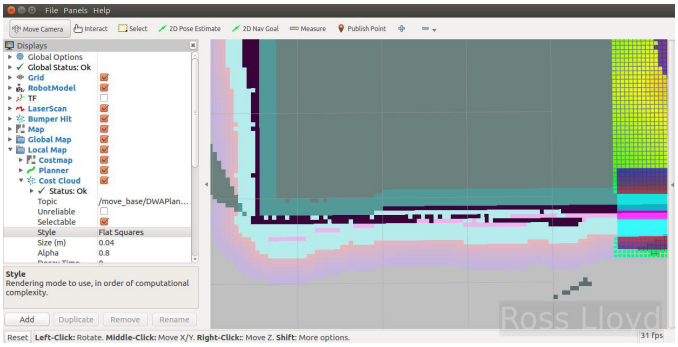

Fig. 35. Errors in inflation layer.

Below: Comparison of the local inflation layer with the global layer above.

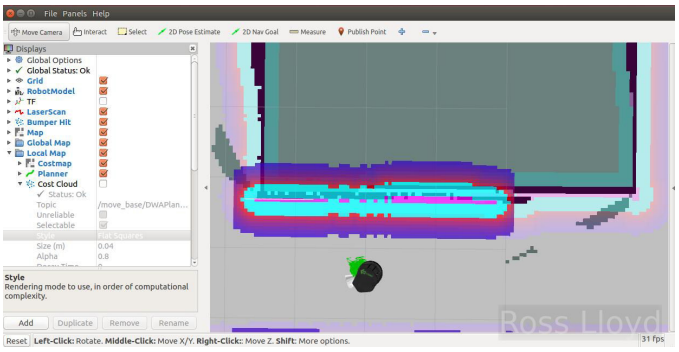

Fig. 36. Comparison of local inflation layer with global inflation layer.

3.3 Home Service Functions

The pickup zone marker was successfully published. The robot successfully navigated to the pickup zone, and the goal result message triggered the pickup behaviour (marker disappears). The robot waited 5 seconds, and then the second goal was sent to the robot, which it navigated to. Again the goal result message lead to the marker appearing at the dropoff point.

Below: Testing the *pick_objects* node which sends the pickup and dropoff locations to the robot. At this stage there are no markers as the add_markers node is not running.

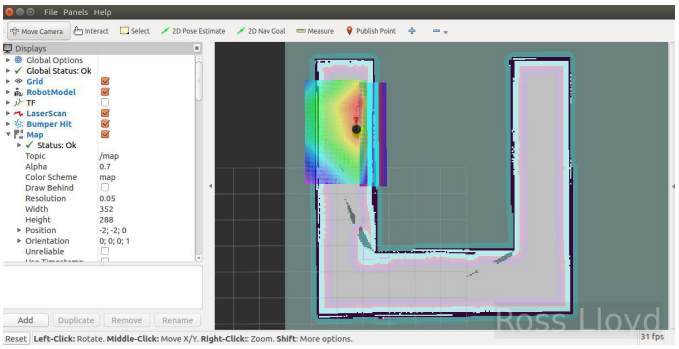

Fig. 37. Robot navigating to pickup zone.

Below: Now with the marker node running, the robot navigates to the pickup point with the blue marker present:

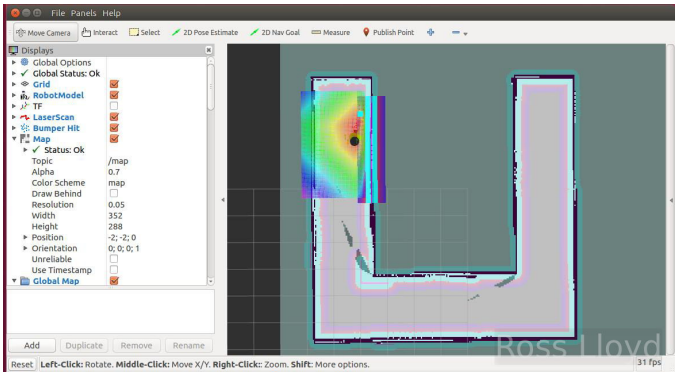

Fig. 38. Robot navigating to pickup point.

Below: Then, the marker disappears, simulating a pick up. The robot pauses for 5 seconds.

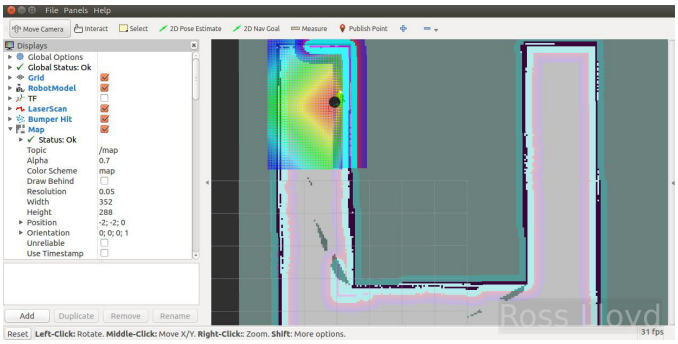

Fig. 39. Robot at pickup position.

Below: After navigating through the map, the robot arrives at the goal position.

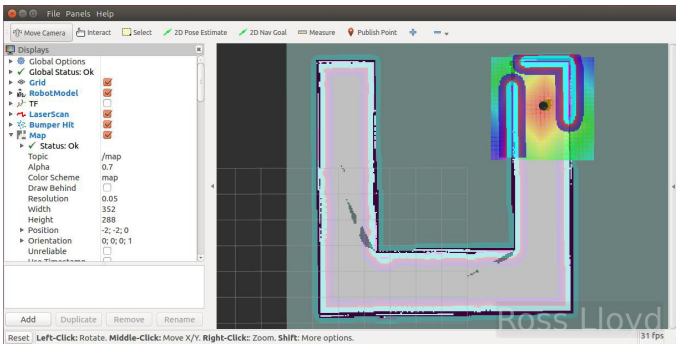

Fig. 40. Robot at goal position.

Below: After which it 'drops' the blue marker.

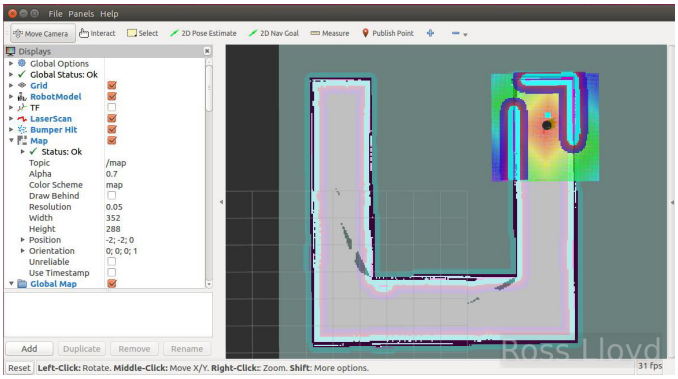

Fig. 41. Robot 'dropping off' the marker at the goal.

4 Discussion

The robot successfully carried out the required tasks. The mapping process was fairly robust once the correct parameters were found. It likely that the environment used in mapping greatly affects the outcome of the gmapping process due to the way the scan matching localization works. In the rather featureless hallways of the created world, there is very little for the algorithm to use to accurately localize, requiring a very high minimum score setting to prevent incorrect matches / localizations from introducing large errors into the map. A more feature rich world would perform better with a lower score, and may even produce a better quality map as the scan matching / localization would help align mapped areas. This may have been useful with the horizontal misalignment of the "crossbar" of the U.

The goal message approach used for the add marker / pickup-dropoff behaviour worked well and has the advantage over odometry based approaches in that it does not require additional measurement tolerances to be introduced. Instead it leverages the functionality already built into the move base goal behaviour as well as serving as a means of communicating exactly when and which goal has been reached. The use of global variables in the code is not considered best practice, and instead functors could have been implemented instead. However for such a small program, global variables are acceptable.

The path planning algorithm used in the ROS navigation stack performed well and took a path that was not too close to the wall, but also not needlessly long by sticking to the middle of the corridor. It would be interesting to try using the A* alternative path planner provided in the **navfn** package which creates the global maps for the move base node. Results might be further refined by optimising the AMCL node parameters, as the convergence of the particle cloud was not as precise as is possible.

Navigation is a fundamental ability of any mobile robot and the applications of it are immense, from warehouses to office spaces, security robots, supermarket stock level monitoring robots and so on. With obstacle avoidance enabled the robots could navigate well in structured environments. However, there are some concerns about using path planning algorithms were large, heavy robots can come into contact with humans, and at present many heavy industrial applications still feature separated areas for robots and people as well as added infrastructure to enable safe navigation.

5 Conclusion / Future work

In conclusion, the mapping, localization and navigation aspects of the project worked well, and the robot was able to create a reasonably accurate map and localize, then navigate within it. The **add markers goal** and **pick objects** nodes successfully passed goal data to the robot, detected when those goals were reached, and controlled the appearance and disappearance of the markers as intended. The move base node will be implemented in real hardware which was built for the Map My World Udacity Robo ND project, which at present is capable of producing 3D maps using rtabmap and localizing within them.

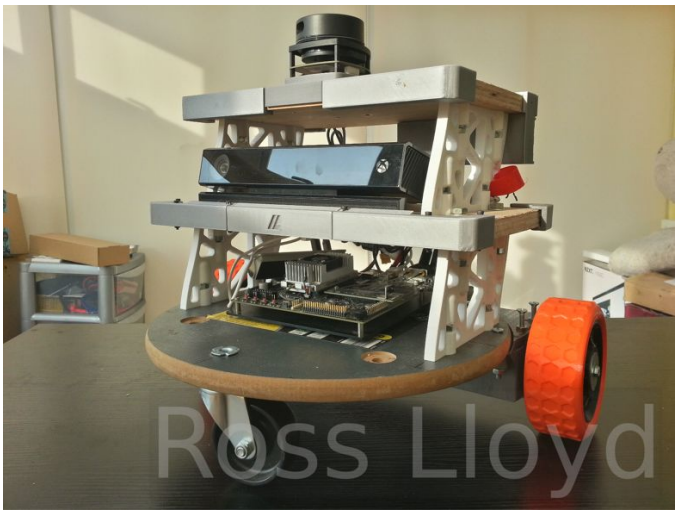

Fig. 42. Gbot robot.

This can be done by creating a config for the move base node within the robot package and then setting the parameters appropriately.

6 REFERENCES

- Udacity Robotics Nanodegree materials
- ROS Wiki
- Geek to Geek website
- ROS Answers
- Stack Overflow