Path Planning for SLAM Based Maps

Adheesh Chatterjee
Maryland Robotics Centre
University of Maryland
College Park, MD 20740, USA
adheeshc@umd.edu

Andre Ferreira
Maryland Robotics Centre
University of Maryland
College Park, MD 20740, USA
andrelgf@umd.edu

Pablo Sanhueza
Maryland Robotics Centre
University of Maryland
College Park, MD 20740, USA
psanhuez@umd.edu

***Abstract* - This paper illustrates how maps developed from simultaneous localization and mapping (SLAM) methods can be used for path planning. In order to do so approaches to solve the SLAM problem are discussed and the properties of resulting maps are described. Furthermore, the principles of path planning are addressed. By using a TurtleBot, the possibility to appropriately deal with the properties of SLAM-based maps in path planning will be shown.**

Index Terms – SLAM, Path Planning, TurtleBot

I. Introduction

The robotics field is becoming more popular and gaining significant relevance in our lives and in the industry. The advent of robotic vacuum cleaners and lawn mowers make our lives easier and allow us to pass these tedious jobs to robots. Even in the industry, many companies have started to use automated guided vehicles and mobile manipulators in their warehouses. Apart from this, mobile robots have immense potential and applications in surveillance, reconnaissance, hazardous environment assessment, search and rescue, as well as planetary exploration and terrain mapping.

The main advantage of mobile robots is obviously, their mobility. The robot must be capable of moving from one position to another. However, to do this, it also needs to find a valid collision free path connecting the corresponding positions and configurations. Moreover, depending on the task at hand, there may be other requirements such as shortest, fastest and/or safest path.

In recent times, on-line path planning has received more attention from researchers since autonomous mobile robots must be capable of operating in dynamic environments. [1]

In order to determine such trajectory, the robot must be able to gather information about its surrounding environment, and about itself. These problems are known as localization and mapping which is fundamental for an autonomous robot.

Autonomous locomotion can be said to comprise of three steps – map building, localization and path planning. Map building reconstructs the environment surrounding the robot; localization estimates the robot's position on the map once it is made; path planning determines the best path to a destination while avoiding obstacles.

Truly autonomous navigation can be achieved using Simultaneous Localization and Mapping (SLAM) as it helps the robot to operate in an unknown environment. SLAM methods can create a map of an unknown environment and localize the robot in this map at once. With SLAM there is no more need to create a map and hand it to the robot in order to navigate it, since the robot develops it on its own. Slam can be summarized as the robot creating the map as it moves in the environment [2]

Since this map is created using robot sensors there might be a lot of noise which can lead to an inaccurate map and localization also known as the SLAM problem [3]

Essentially, SLAM works with probability-based Bayes filters or the Extended Kalman Filter. Due to this, SLAM based maps are not absolute but provide information about obstacles and the robots pose as probabilities. This means that SLAM maps have special properties that need to be addressed when trying to calculate an optimal path. There have been studies on this where references [4] and [5] discuss sample based path planning with pose estimation.

EKF-SLAM algorithm from [6] is considered as an approach to solve the SLAM problem using the extended Kalman filter.

Visual SLAM is also considered as well as Fast SLAM algorithms to improve the approach as In experimental design and 3D reconstruction it is desirable to minimize the number of observations required to reach a prescribed estimation accuracy. [7]

II Statement of the problem

For previous projects of this subject the students have been doing path planning on maps originated from instructions provided by the professor, where they are able to hardcode the environment creating a perfect representation of the real environment.

Such approach works in a satisfactory manner since the goal is to teach the students path planning, however on a real environment, autonomous robots usually build their maps based on sensors such as lasers, cameras and odometry readings to identify what surrounds them.

Robots that have camera sensors use images to generate the map. In order to properly identify the features present in those images, it is necessary to apply computer vison which will give the desired information to be used by the robot to replicate its environment.

Working with this type of sensor usually does not lead to perfect results as the information collected can have a considerable amount of noise.

The approach of this project is to create a map using the concept of slam using Gmapping which will return a picture of the environment and different from the previous projects of this class, the group will not create a hardcoded matrix to represent

the map. Using computer vision, the picture is going to be utilized to perform path panning.

III. Goals

The goals of this project are:

- Create a map through Slam using Gmapping
- Apply computer vision to the image generated in order to properly collect the necessary information of the environment.
- Calculate Minkowski sum to account for the TurtleBot radius and clearance using computer vision
- Perform path planning using A*

IV. Slam

Simultaneous Localization and Mapping is the process of the robot creating the environment map and simultaneously compute its own location in the constantly evolving map.
In order to incorporate the uncertainties, we first define certain variables –

- $x_t = (x_x, x_y, x_z)$ as the 3D state vector describing the pose of the robot at any time t
- y_t is the observation of the sensor at time t
- m is the map
- u_{t-1} is the control input at time t-1

Now, as per the definition of the SLAM problem, we need to compute the robot's state x_t and the map m depending on the previous conditions of y_t and u_{t-1}.

Essentially, we need to define a model that returns the current state x_t depending on the previous state x_{t-1} and the previous input u_{t-1}. One way of doing this is by using odometry data. However, odometry is very sensitive to errors and a small error due to noise or inaccurate calibration can cause the error to attenuate over time. The logic above is used to define the belief of the state as the conditional probability.

$$P(x_t) = P(x_t|x_{t-1}, u_{t-1}) \tag{1}$$

It is also needed to define a model that returns the current sensor output depending on the state x_t and the map m. Hence, it is defined as a conditional probability.

$$P(y_t) = P(y_t|x_t, m) \tag{2}$$

Both the models defined above are non-deterministic, as they will not always produce the same output going through the same states.

This obviously implies that the SLAM algorithm cannot be solved to an absolute value i.e. and therefore it is not able to compute the actual map, but what can be computed is the belief over the state and the map represented by the probability.

This leads to two cases, where there are several assumptions that have to be made. First, let's assume that the motion and models are defined, thus having all the values of states x_t, then the probability over the map can be computed as:

$$P(m) = P(m|x_{0:t}, y_{0:t}, u_{0:t-1}) \tag{3}$$

Second, let us assume that the motion and map are known, thus having all the values of m, then the probability over the state x_t can be computed as:

$$P(x_t) = P(x_t|m, y_{0:t}, u_{0:t-1}) \tag{4}$$

In SLAM however, the robot doesn't know about its current location or its current surroundings, and as per the definition and model defined, they both depend on each other. That leads to the conclusion that it is needed to compute the probability over the map m and the state x_t simultaneously. Additionally, the group also has to make sure to avoid divergence due to attenuation of errors.

Reference [8] provides possible solutions to the SLAM problem. The first solution seeks to recover the current pose x_t as:

$$P(x_t, m) = P(x_t, m|y_{0:t}, u_{0:t-1}) \tag{5}$$

The second solution estimates the whole trajectory as:

$$P(x_{0:t}, m) = P(x_{0:t}, m|y_{0:t}, u_{0:t-1}) \tag{6}$$

The two models defined above need to be solved to get appropriate results. Another requirement of SLAM methods is consistency and since the motion and transition model are usually non-linear, SLAM methods need to be able to deal with non-linearities.

V. MAP TYPES

A popular possibility to represent mapping results are feature-based maps. They are based on a set of landmarks to define the map. A point-landmark is only defined by its position. For a 2-D feature-map using points as landmarks, each landmark is a two-dimensional vector. Feature-based maps rely on predefined landmarks, which requires them to already know something about the structure of the environment in advance.

Another common approach is to use grid-based maps. They are based on a rigid grid of cells. The cells are either free, occupied or unknown depending on the state of the cell. Grid maps can describe arbitrary features and provide a detailed representation on the environment.

A grid-based map, however, would be a better approach for this project as the discretization is done as a separate step.

VI. KALMAN FILTER AND EXTENDED KALMAN FILTER

There are many approaches to solve the SLAM problem, most of which are based on the Bayes Filter [9].
The Kalman filter is just an extension of the Bayes filter, a probabilistic filter, used for state estimation that uses a Gaussian state and a linear Gaussian observation. It comprises of two steps – the estimation step and the correction step

The estimation step uses the process model to predict the next state
Estimation step $(x_t) : P(x_t) = P(x_t|x_{t-1}, u_{t-1})$

The correction step uses the observation model to correct the prediction
Prediction step$(\hat{x_t}) : P(y_t) = P(y_t|x_t, m)$

The final estimate is calculated by –

$$\hat{x}_t = x_t + \mathrm{K}\,(\mathrm{y_t} - \mathrm{Hx_t})$$

Where K is the Kalman gain and H is the coefficient from the observation model

The EKF is just an extension of the Kalman filter for non-linear systems. EKF SLAM methods primarily generate feature based maps but have difficulties with convergence at times.

VII. MONTE CARLO LOCALIZATION

Monte Carlo Localization uses a weighted measurement for each particle. In this case, each particle represents a pose of the robot. After each movement, the particles get sampled as per the motion model. This is a 4 step process that repeats over observations.

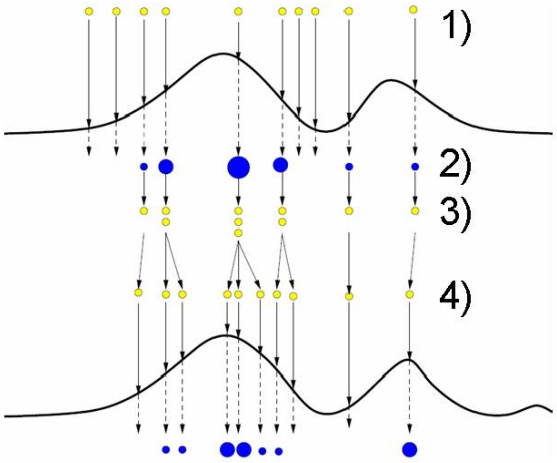

Fig. 1 Monte Carlo Localization. [12]

Step 1) sampling of the poses is done.
Step 2) The weights are estimated as per the gaussian.
Step 3) Resampling is done after which the whole process starts again with a new robot movement
Step 4) The sampling step is done again.
After several iterations the particles will gather on more likely poses of the robot.
The exact algorithm is explained below[12]

Monte Carlo Localization

function MONTE-CARLO-LOCALIZATION($a, z, N, model, map$) **returns** a set of samples
inputs: a, the previous robot motion command
z, a range scan with M readings $z_1, \ldots, z_M$
N, the number of samples to be maintained
$model$, a probabilistic environment model with pose prior $\mathbf{P}(\mathbf{X}_0)$,
motion model $\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0, A_0)$, and range sensor noise model $P(Z|\hat{Z})$
map, a 2D map of the environment
static: S, a vector of samples of size N, initially generated from $\mathbf{P}(\mathbf{X}_0)$
local variables: W, a vector of weights of size N

for $i = 1$ **to** N **do**
$S[i] \leftarrow$ sample from $\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0 = S[i], A_0 = a)$
$W[i] \leftarrow 1$
for $j = 1$ **to** M **do**
$\hat{z} \leftarrow$ EXACT-RANGE($j, S[i], map$)
$W[i] \leftarrow W[i] \cdot P(Z = z_j|\hat{Z} = \hat{z})$
$S \leftarrow$ WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S, W)
return S

Fig. 2 Monte Carlo Localization. [12]

VIII. GMAPPING

Gmapping is a grid-based Slam method that is used to create a discretized map. This method is already present in ROS using the ROS Gmapping package. This method will help to create a grid map using the laser and pose information collected by the robot with its sensors.

Gmapping implements Rao-Blackwellization particle SLAM method [10] which is a theorem that provides a process to improve efficiency of an estimator by getting the conditional expectation with respect to a sufficient statistic to create a grid-based map.

Essentially, Gmapping gets divided into two concepts – First is to do proposal distribution and the second is to adaptively determine when to resample.

The idea of the first concept is to compute SLAM as based on the models developed in the above section:

$$P(x_{0:t}, m|y_{0:t}, u_{0:t-1}) = P(x_{0:t}|m, u_{0:t-1})P(m|x_{0:t}, y_{0:t}) \quad (7)$$

The SLAM posterior gets divided into a path and a map posterior. It applies the concepts of the Kalman Filter and Monte-Carlo Localization to compute the first term and with the known poses it can compute the second term.

The second concept is achieved by using a Particle Filter. Here, samples with low importance get replaced by samples with higher probability. Resampling is a very important as the number of samples is limited.

The combination of both approaches described above leads to a highly efficient Rao-Blackwellized SLAM technique.

IX. COMPUTER VISION

Computer vision is the science of using a computer to process and understand images and video, also known as making a computer to "see".[13]

Open Source Computer Vision (OpenCV) is a programming library developed by Intel ,that contain methods to perform computer vision mainly in real time . This library is supported by many programming languages as Python, C++, Matlab and Java.

For this project the Language utilized will be Python, where we used the following function from OpenCV.

- cv2.imread – read the image

- cv2.CvtColor - Change the image to the desired color space
- cv2.threshold- Set the image pixels to 0 or 255 based on a desired threshold
- cv2.findContours- Find the all the contours of an image using Green's theorem.
- cv2.drawContours- Draw all the contours found by the previous function.
- cv2.Circle – Draw a circle in a desired location of the image.
- cv2.imShow – Show the desired picture

X. TurtleBot

The TurtleBot is a robot with open-source software that runs on ROS. It is ideal to use for testing algorithms dealing with localization and mobility. It was created by Melonee Wise and Tully Foote , with open-source software. The robot consists of a mobile base, distance sensors and a single onboard computer. It is ideal to use for testing algorithms dealing with localization and mobility .

Due to the precision in odometry readings and sensors, and its easy maneuverability, the TurtleBot is optimal to create precise SLAM maps and follow a planned trajectory. The open-source software for the TurtleBot also has a lot of pre-built frameworks, so less programming is required.

Fig. 3 TurtleBot3 Waffle Pi

XI. Dijkstra's Algorithm

Given a graph, in this case the map generated by SLAM, Dijkstra's algorithm allows to solve and find the shortest path from a single source node by finding the nodes that have minimum distance from its source.

Before explaining the algorithm, some variables and terminology will be explained. See below.

$$S \rightarrow Set\ of\ all\ nodes\ in\ a\ given\ map$$

$$Cost \rightarrow Distance\ between\ nodes$$

$$Q \rightarrow queue\ of\ all\ nodes\ in\ map$$

$$V \rightarrow Visited\ Nodes\ in\ map\ (initialized\ to\ be\ empty)$$

First of all, the algorithm initializes the cost/weight to an infinite distance, where the cost is the distance between each node of the graph. By common reasoning, it can also be determined that the source node has a cost or distance of 0. After initializing these first conditions the algorithm proceeds to recalculate the cost from each node to the source node, and this is done for several iterations until the shortest distance is found.

The variable Q, which represents the queue of all the nodes in the map, allows for the algorithm to search all possible nodes until Q is empty or until it has reached the desired node (goal node). The set V, allows for the storage of all nodes that have been visited, which allows the algorithm to detect repeated nodes to later determine if there is a shorter path for that current node to its parent. Once the algorithm reaches the desired goal or when Q is empty, it will give the shortest path from the source to the goal node.

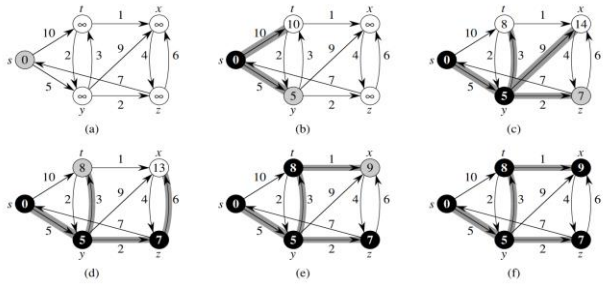

Fig. 4 Example of Dijkstra's algorithm in a graph. [11]

Shown below is the pseudo code for the algorithm that will be implemented in the actual testing stage of the project.

1. Initialize source node cost to 0 → cost(s_1) = 0
2. Initialize all other node cost to infinity → cost(S) = ∞
3. While Q is not empy, then perform algorithm

while Q:

a. X = getMinimum(Q) → Obtain the minimum cost node
b. $Q.remove(X)$ → Removes from the queue

c. ***if*** X == destination: → Break out of loop if goal was met
break;

d. ***for*** all S_i:
if X not in Visited:
Add X to Visited
$Q.insert(X)$ → Add X to the queue
Save the parent of x′, which is X
Cost(x′) = cost(X) + cost(X, U) → parent + current
else:
if Cost(x′) > cost(X) + cost(X, U)
Update cost(x′) to cost(X) + cost(X, U)
since its smaller than previous cost for that node
Save the parent of x′, which is X

Fig. 5 Example of Dijkstra's algorithm in a graph. [11]

XII. A* Algorithm

The A* star algorithm is one of the most well know algorithms in the field of path planning. This method is used in many real life situations such as in a map or games where

there are many obstacles. This algorithm separates from other algorithms, such as Dikjstra, where this method takes into account the cost to go for a given goal location.

A*, like Dikjstra, initiliazes all the cost to come as infinity, where the cost to come is the distance between the starting position to the current node that is being analyzed. Dikjstra and A* are very similar in nature, where the main difference is that that A* takes into account the cost to go, which is the distance between the current node to the goal node. This can be defined as a heuristic function. For purposes of this paper, the same notation will be used as in the Dijkstra section above (e.g. Q, V, S, and Cost). One important factor in a heuristic method is that it needs to be "admissible", in other words optimistic. This is defined below.

$$H(n) < H^*(n)$$

$$where\ H^*(n)\ is\ the\ true\ cost\ to\ a\ neares\ goal$$

One of the most ideal heuristic functions for real world application is by using the Euclidean distance method. Another reason to use this type of distance calculation is due to the fact the TurtleBot can move in many different directions, which means that the units can move at any angle. Euclidian distance is defined below.

$$p \rightarrow current\ node\ coordinate$$
$$q \rightarrow goal\ node\ coordinate$$

$$d(p,q) = \sqrt{(q_x - p_x)^2 + \left(q_y - p_y\right)^2}$$

Using the heuristic function as Euclidean distance will allow the group to obtain an optimized path from the SLAM map utilized.

Shown below is the pseudo-code that was implemented during this project, which includes the differential constraints of the TurtleBot.

1. Initialize source node cost to 0 → $\text{cost}(s_1) = 0$
2. Initialize all other node cost to infinity → $\text{cost}(S) = \infty$
3. While Q is not empy, then perform algorithm

while Q:

a. X = getMinimum(Q) → Obtain the minimum cost node
b. $Q.remove(X)$ → Removes from the queue

c. ***if*** X == destination: → Break out of loop if goal node met
break;

d. ***for*** all S_i:
$-\ Perform\ Movements\ from\ X\ node \rightarrow x'$
if x′ not in Visited:
Add x′ to Visited
$Q.insert(x')$ → Add x′ to the queue
Save the parent of x′, which is X
$\text{Cost}(x') = \text{cost}(X) + \text{cost}(X, U) + h(x')$
else:
if $\text{Cost}(x') > \text{cost}(X) + \text{cost}(X, U)$
Update $\text{cost}(x') \rightarrow \text{Cost}(x') = \text{cost}(X) + \text{cost}(X, U) + h(x')$
Save the parent of x′ → which is X

Fig. 6 Example of Dijkstra's algorithm in a graph. [11]

XIII. Previous Studies

Previous studies have also implemented SLAM and path planning algorithms to find the best and most efficient path to a specific goal. One of these papers was done by Yudai Hasegawa and Yasutaka Fujimoto, and titled "Experimental Verification of Path Planning with SLAM".

In this specific paper, the authors propose a method of path planning that can be used in autonomous mobile robots in which they utilize SLAM and the A* algorithm with a new heuristic function. Fujimoto and Hasegawa's method was first implemented by using SLAM to extract the raw data from the LRF sensor onto a grid-map and then using A* to solve for the best path to a specific goal. Their results compared to other research studies confirm that their method results in much smoother movements than regular A*.

The experiment and results of this paper will be compared to Fujimoto's and Hasegawa's paper. Even though the methodology of this paper is slightly different from Fujimoto and Hasegawa's research paper, it is still a comparable outcome with regards to how the mapping and planning is done. The main difference between this paper and Fujimoto's paper is that this project will be done using Dijkstra instead of A* and the results will be compared.

XIV. Results

To achieve the goal explained in previous sections we are going to use the following methodology.

Firstly, we are going to create an environment using Gazebo where the TurtleBot is going to be added to it with no information about its surroundings.

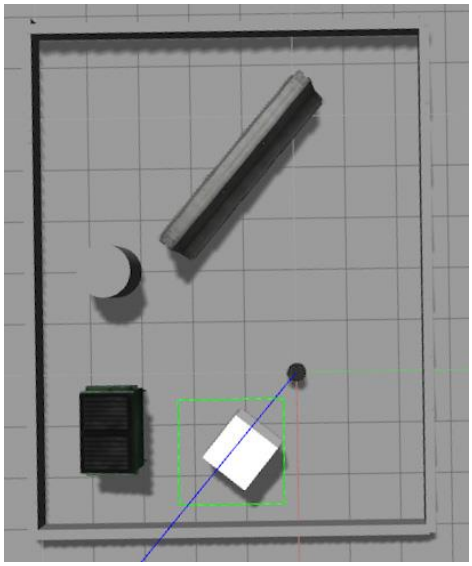

Fig. 7 Gazebo Environment

The second step involves performing SLAM in order to create a map and perform localization to properly operate path planning. SLAM will be implemented utilizing Gmapping.

Since the SLAM problem deals with probabilities, although there will be fast convergence, we deal only with the

belief of the environment which changes every time the SLAM is performed.

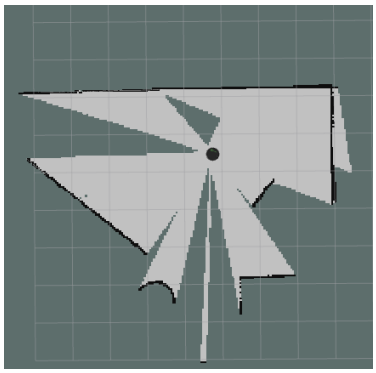

Fig. 8 RViz Environment

Once the localization and mapping is calculated using SLAM, the robot will have a probability of what is around it and its current location. However, we need to consider the resolution and dimensions of the TurtleBot

The most important step in our approach is to get rid of the probabilities and convert the SLAM based map into a discrete map. For this the group used thresholds for the probability, where the cells between the thresholds are considered unknown, the cells below the lower threshold are free and the cells above the upper threshold are obstacles.

```
image: /tmp/my_map.pgm
resolution: 0.050000
origin: [-12.200000, -13.800000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Fig. 9 Thresholds and parameters used for Gmapping

The final map obtained from the Gmapping is shown in Fig 8.

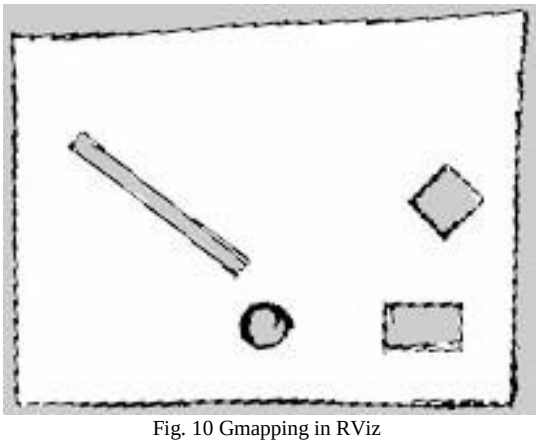

Fig. 10 Gmapping in RViz

The third step consists of implementing computer vision using OpenCv library to properly work with the image in order to be able to perform path planning.

The SLAM generated map is first read into the Python script editor. It is then resized and changed to grayscale to ensure the discretization of the values are optimal. A threshold is set to fill in holes and properly define obstacles.

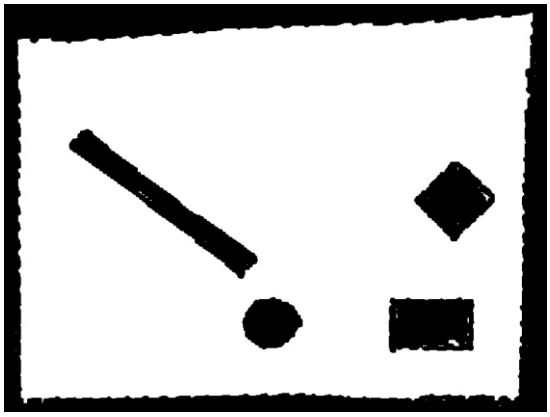

Fig. 11 After Thresholding

All the contours in the image are detected which gives a very good detection for all the obstacles. The Minkowski sum is then calculated by drawing circles around the obstacles by accounting for radius of the TurtleBot and the clearance.

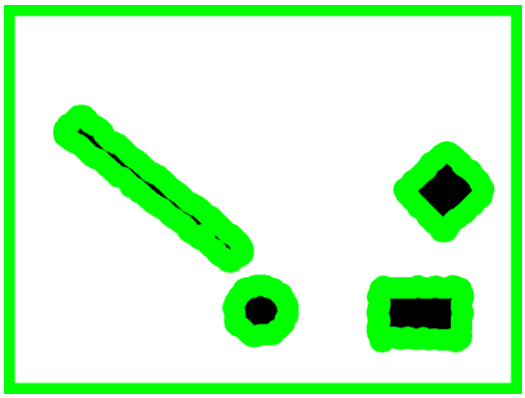

Fig. 12 Contours and Minkowski Sum

Finally, the robot has to use the A* algorithm to find the shortest collision free path between the current position and the desired target position. The robot has to take into account its own dimensions as well as its resolution as discretized grids may cause a part of the robot to be in an open cell while another part is in the obstacle cell.

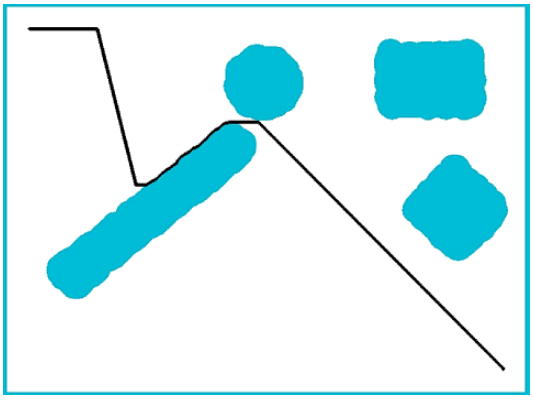

Fig. 13 A* Algorithm (weight = 0)

The A* algorithm is implemented for different weights. Weighted A star explores fewer nodes and completes much faster than with a weight of 1. Setting a weight of 0 reverts the A* back to Dijkstra.

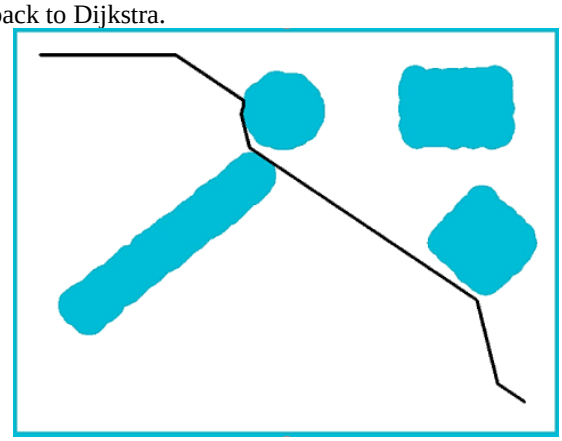

Fig. 14 A* Algorithm (Weight = 2)

XV. PRACTICAL IMPLEMENTATION

An environment was created to test the SLAM algorithm and the final path planning.

Fig. 15 Contours and Minkowski Sum

The TurtleBot3 was used with TeleOp commands to move around the environment and perform SLAM. Three obstacles were placed to be detected.

Fig. 16 Contours and Minkowski Sum

The output of the SLAM is shown in Fig 7. The idea was test the planning method developed on the real TurtleBot. However, a technicality prevented this test. In the ROS environment, the TurtleBot is operated through Teleop commands. Here, the commands are the velocity of the left wheels and the right wheels. Giving different velocities to the different wheels, the robot can be made to explore the environment. However, in the practical implementation, the TurtleBot both wheels move at the same velocity. The only two velocities that can be given are the linear velocity in x and angular velocity in z. Given together the robot can be made to explore the environment. Since the two methods achieve the same process but with slightly different inputs, a controller needs to be developed to account for this which is part of the future scope.

XII. PREVIOUS STUDIES

Previous studies have also implemented SLAM and path planning algorithms to find the best and most efficient path to a specific goal. One of these papers was done by Yudai Hasegawa and Yasutaka Fujimoto [A], and titled “Experimental Verification of Path Planning with SLAM”.

In this specific paper, the authors propose a method of path planning that can be used in autonomous mobile robots in which they utilize SLAM and the A* algorithm with a new heuristic function. Fujimoto and Hasegawa’s method was to first implement SLAM to extract the raw data from the LRF sensor onto a grid-map and then using A* to solve for the best path for a specific goal. Their results compared to other research studies confirm that their method results gives much smoother movements than regular A*.

The experiment and results of this paper will be compared to Fujimoto’s and Hasegawa’s paper. Even though the methodology of this paper is slightly different from Fujimoto and Hasegawa’s research paper, it is still a comparable outcome with regards to how the mapping and planning is done. . Fujimoto and Hasegawa’s paper serves as a good reference in understanding how the SLAM algorithm works and the authors of this paper were able to analyze and

compare relative outcomes of the SLAM and the A* planning algorithm.

XII. Future Scope

To ensure a fully autonomous behavior, it should not only be able to deal with static but also dynamic environments. The map generated is a static one and the A* algorithm cannot be used as a path planning algorithm. This could lead to a situation where a new obstacle is placed in the TurtleBot's computed path and as a result, the "optimal" path is no longer collision free. So the use of a local planner that updates dynamically and a different planning algorithm can be used instead.
As mentioned earlier, a controller needs to be developed to convert wheel velocities from simulation to linear and angular velocity for the real robot so that it can be simulated perfectly.

XII. CONCLUSION

From this project, the authors can conclude that the SLAM algorithm is a good method of identifying obstacles. To achieve fully autonomous behavior, a robot must be able to completely define the environment around it and find its position in it. Only then can path planning be initiated. Computer Vision is a very helpful tool in post processing of the output generated from SLAM. This ensures that any path planning algorithm can be implemented on the final map.

Acknowledgment

The authors of this paper would like to thank the TA for our planning course, Utsav Patel, for his constant support and encouragement as well as helping with the set up in the RRL lab, without whom this paper wouldn't have been possible. The authors would also like to thank the RRL staff for providing us with space to set up the environment and the use of the TurtleBot 3.

References

[1] P. Raja* and S. Pugazhenthi, "Optimal path planning of mobile robots: A review", DOI: 10.5897/IJPS11.1745, ISSN 1992 – 1950,Academic Journal, 2012
[2] S. Thrun, W. Burgard, and D. Fox, Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). The MIT Press. 2005.
[3] H.Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: PartI,"IEEERobotics&Automation Magazine, vol. 13, June 2006.
[4] R. Valencia, M. Morta, J. Andrade-Cetto, J. M. Porta. "Planning Reliable Paths With Pose SLAM." In: IEEE Transactions on Robotics 29.4 (2013), pp. 1050–1059 (cit. on p. 2).
[5] R. Valencia, J. Andrade-Cetto, J. M. Porta. "Path planning in belief space with pose SLAM." In: Robotics and Automation (ICRA), 2011 IEEE International Conference on. 2011, pp. 78–83 (cit. on p. 2).
[6] D. Fethi1, A. Nemra1, K. Louad and M. Hamerlain, Simultaneous localization, mapping, and path planning for unmanned vehicle using optimal control, Advances in Mechanical Engineering, DOI: 10.1177/1687814017736653, Vol. 10(1) 1–25,2018
[7] S. Haner and A. Heyden, Optimal View Path Planning for Visual SLAM, Centre for Mathematical Sciences, Lund University
[8] T. S. Ho, Y. C. Fai, E. S. L. Ming. "Simultaneous localization and mapping survey based on filtering techniques." In: Control Conference (ASCC), 2015 10th Asian. 2015, pp. 1–6 (cit. on p. 6).
[9] M. S. Arulampalam, S. Maskell, N. Gordon. "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking." In: IEEE TRANSACTIONS ON SIGNAL PROCESSING 50 (2002), pp. 174–188 (cit. on p. 7).
[10] G. Grisettiyz, C. Stachniss, W. Burgard. "Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling." In: Proceedings of the 2005 IEEE International Conference on Robotics and Automation. 2005, pp. 2432–2437 (cit. on pp. 7, 9, 23).
[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest. Introduction to algorithms. 24. print. Cambridge, Mass. [u.a.]: MIT Press [u.a.], 2000, XVII, 1028 S. ISBN: 0-262-03141-8 (cit. on p. 16).
[12] https://www.slideshare.net/YasirAhmedKhan/concept-17-sldes
[13] R. Szeliski. Computer Vision: Algorithms and Applications. Springer 2011.
[A] Y. Hasegawa and Y. Fujimoto, IEEJ Journal of Industry Applications, Vol 5, No 3, pp 253-260, 2015