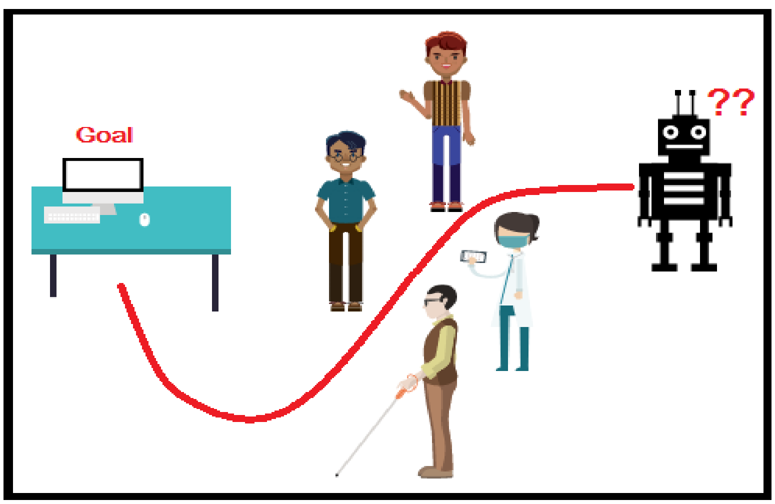
RBE 501: Robot Dynamics

Project Report

Robot Navigation and Path Planning

Heramb Nemlekar (hsnemlekar@wpi.edu)	Rishi Khajuriwala (rdkhajuriwala@wpi.edu)	Nishant Shah (nshah3@wpi.edu)



Instructor: Mahdi Agheli

Worcester Polytechnic Institute

ABSTRACT:

For mobile robots that operate in cluttered environments the selection of an appropriate path planning algorithm is of high importance. In this project we aim to explore several path planning algorithms to understand how each of them can be applicable in different situations. The motivation for this project comes from the need of dynamic motion planning in human-robot collaborative environments like hospitals, offices and homes. Therefore, we have analyzed the Dijkstra's and A algorithms through implementation and also through simulation on TurtleBot in a specially designed gazebo environment. This study will act as a stepping stone towards working on advanced projects on full SLAM implementations.*

INDEX

LIST OF FIGURES

INTRODUCTION:

Seamless integration of robots in a human environment is an unprecedented issue faced in any Human-Robot Interaction (HRI) project. Robots that are used in dynamic settings like manufacturing units, offices and households can be classified into two categories: (1) Manipulator Robots and (2) Mobile robots. The basic distinction between these two types of robots is that the mobile robots can change their position in an environment, while the manipulator robots are affixed to a support. This causes a mobile robot to have a large workspace and multiple dynamic entities to interact with in that space.

This brings us to the problem of moving a robot in an environment that may have several stationary or moving obstacles, especially humans. Here, the safety of humans is of utmost priority and hence it is vital that the robot can accurately sense its surroundings and decide its course of action. But sensor data is usually noisy and cannot provide a true estimate of the surroundings. Also as the objects in the scene are moving, there could be a lack of stationary landmarks for localization. But several probabilistic models are available for this state estimation and each have their own advantages and shortcomings.

We started with the study of path planners like Dijkstra's and A-star algorithms and move towards studying Simultaneous Localization and Mapping. The reason for experimenting with these two algorithms is that they are the most fundamental planners for navigation. Hence we tested them for different goal and obstacle locations and comment on their efficiency. For SLAM we would only provide an overview of its three types: Extended Kalman Filter, Particle Filter and Graph-based SLAM.

To embrace the practical side of this project, we shift our focus towards understanding the ROS environment for running Gazebo. This involved setting up the Gazebo, Rviz and Turtlebot in ROS. We also learned how nodes are created, topics are published or subscribed to and how python executables are compiled into a package. Using this knowledge we were able to create a world in Gazebo for navigation of Turtlebot and tele-operate it for generating maps. This map enabled us to re-implement the A-star algorithm in python and conclude a comprehensive study of these algorithms.

PROJECT OBJECTIVE:

The main goal of this project is to learn ROS and gain a substantial knowledge of implementing path planning and SLAM algorithms for mobile robots which will help us develop skills for further research in this field. Also testing algorithms in simulated dynamic environments will help us in developing robust algorithms for navigation tasks in social or medical settings.

LITERATURE REVIEW:

As background study for this project we looked at several existing path planning algorithms and read papers on motion planning and SLAM techniques. We also explored the ROS environment and learned how to create packages and run executables.

Grassfire Algorithm:

When a robot is moving in an environment of known dimensions, its floor space can be discretized into several blocks. By assigning a coordinate axis to one of the corners of this space we can represent each block with a pair of coordinates. Each block can be assigned a binary value depending on whether it is empty or contains an obstacle. Now we start from the goal location and mark it as 0 value. Each of the four neighboring cells that is empty is marked a consecutive value 1. Now the neighbors of the cells that are marked as 1, are given the next value 2. This goes on till you reach the start location. Now each value of the cells surrounding the goal location represents the number of steps required by the robot to reach the goal location from this cell. Therefore, successfully selecting the minimum neighboring value will lead the robot to the goal location via the shortest path.

The grassfire algorithm is also a complete algorithm, which means it considers all the possible cases of navigation. While moving from the goal to the start location, if all the reachable empty cells are numbered and yet the start location is not be assigned a value, it will indicate that no path exists from the start to the end location. As the grassfire algorithm accesses all the available cells, its computational efficiency is inversely proportional to the number of nodes.

Pseudocode[7]:

```
for each node n in the graph
    n.distance = infinity

create empty list

goal.distance = 0, add goal to list

while list not empty
    current = first node in list, remove current from list
    for each node n that is adjacent to current
        if n.distance = infinity
          n.distance = current.distance + 1
          add n to back of the list
```

Dijkstra’s Algorithm:
Grassfire algorithm works well when all the edge weights in the grid are equal. This means that in a Grassfire algorithm, the cost of moving from one cell to its neighbor is same for all cells. But when the edge weights are different, we use the Dijkstra’s algorithm.

Its flow is similar to the Grassfire algorithm, but starts from the start location instead of the goal location. We mark the starting cell as 0. Next we assign a value to the neighboring nodes according to its corresponding edge weight. As we move forward we also maintain a matrix of the parent cells for each node. When we finally arrive at the goal location we can trace back the path with the minimum sum of edge weights from the starting node. Here we don’t need to explore all the cells in the workspace, but we do need to explore all the neighboring cells until we reach the goal location.

Pseudocode[6]:

```
for each node n in the graph
    n.distance = infinity

create empty list

start.distance = 0, add start to list

while list not empty
    current = node in the list with the smallest distance, remove current
from list
    for each node n that is adjacent to current
        if n.distance > current.distance + length  of edge from n to
current
          n.distance = current.distance + length of edge from n to current
          n.parent = current
           add n to back of the list if there isn’t already
```

A*(star) Algorithm:
A* is a modification of Dijkstra’s Algorithm that is optimized for a single destination. Dijkstra’s Algorithm works well to find the shortest path, but it wastes time exploring in directions that aren’t promising. Greedy Best First Search explores in promising directions but it may not find the shortest path. The A* algorithm uses both the actual distance from the start and the estimated distance to the goal; A* finds paths to one location. It prioritizes paths that seem to be leading closer to the goal.

It is a best-first search algorithm, it solves problems by searching among all the possible paths to the goal location and selects the one path that incurs the smallest cost i.e. least distance travelled or shortest time taken to reach the goal location. A* is formulated in terms of weighted graphs, it starts from a specific node of a graph, and constructs a tree of paths starting from that specific node, it expands the paths one step at a time, until one of the paths ends at the goal node or goal position. A* expands paths that are already less expensive by using this function:

$$\boldsymbol{f}(\boldsymbol{n}) = \boldsymbol{g}(\boldsymbol{n}) + \boldsymbol{h}(\boldsymbol{n}),$$

where,

- n = the last node on the path
- f(n) = total estimated cost of path through node n
- g(n) = cost so far to reach node n
- h(n) = estimated cost from n to goal. This is the heuristic part of the cost function.

The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, which means that it should never overestimate the actual cost to get to the nearest goal node. The heuristic function can be calculated in various ways:

- **Manhattan Distance**:
  In this method h(n) is computed by calculating the total number of squares moved horizontally and vertically to reach the target square from the current square. Here any obstacles and diagonal movement are ignored.
  $$h = |x_{start} - x_{destinantion}| + |y_{start} - y_{destinantion}|$$
- **Euclidean Distance Heuristic:**
  This heuristic is slightly more accurate than its Manhattan counterpart. If we try run both simultaneously on the same maze, the Euclidean path finder favors a path along a straight line. This is more accurate, but it is also slower because it has to explore a larger area to find the path.
  $$h = \sqrt{(x_{start} - x_{destination})^2 + (x_{start} - y_{destination})^2}$$

- If h(n) = 0, A* becomes Dijkstra’s algorithm

All the above algorithms work under the assumption that we can accurately sense our surroundings and predict its states. But in practical applications, the sensor data has some inherent noise and we can never receive a true value of the location of the robot and its environment. Therefore, we need some form of a probabilistic model like a Kalman Filter to reasonably estimate the system states.

Pseudocode[8]:

```
function A*(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually
contain the
    // most efficient previous step.
    cameFrom := an empty map

    // For each node, the cost of getting from the start node to that
node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the
goal
    // by passing by that node. That value is partly known, partly
heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue            // Ignore the neighbor which is
already evaluated.

            if neighbor not in openSet      // Discover a new node
                openSet.Add(neighbor)
```

```
            // The distance from start to a neighbor
            //the "dist_between" function may vary as per the solution
requirements.
            tentative_gScore := gScore[current] + dist_between(current,
neighbor)
            if tentative_gScore >= gScore[neighbor]
                continue            // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] +
heuristic_cost_estimate(neighbor, goal)

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

Simultaneous Localization and Mapping

A SLAM problem can be defined as follows: The robot creates a map of the environment and dynamically estimates its own position with respect to the features in the map. The state of the robot at a time instant can be denoted by x(t). When the robot moves to x(t+1), it records a odometry reading u(t) and has an estimate z(t) of the surroundings m as shown in Figure 1.

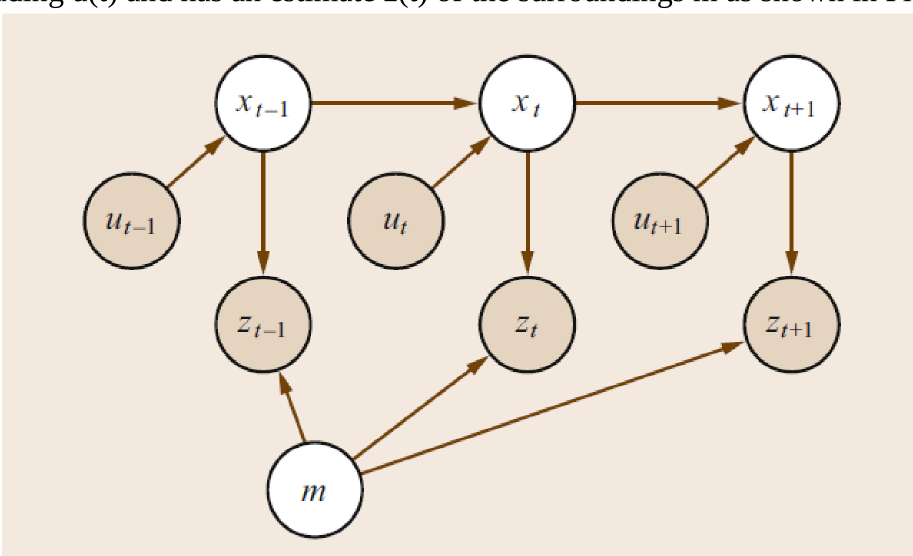


Fig. 1. SLAM Flow

There are three primary ways to address a SLAM problem: (1) Extended Kalman Filters (EKFs), (2) Particle Filters and (3) Graph-based Optimization Techniques.

Each technique has its own flaws. For using EKF we assume that the landmarks should be fully visible. Also in EKF the resultant covariance matrix is quadratic and hence difficult to evaluate. In particle filters every particle is a guess of the states of the system. At every step we evaluate the value of each guess and eliminate poor guesses. We create a new sample of guess in the next step and proceed. But there is a risk of particle depletion during resampling and fairly decent estimates may be lost. Also the complexity of particle filters scales exponentially as it depends on the number of particles and states in a system. Here we can employ Fast SLAM that combines three techniques: Rao-Blackwellization, Conditional Independence and Re-Sampling.

Lastly we can use Graph Optimization where x and m are the nodes in the graph. But graph based problems address the full SLAM problem and are hence not employed online.

In SLAM[9], the objective is to compute: $\boldsymbol{P(m_t\,, x_t|o_{1:t})}$**,** here $\boldsymbol{o_t}$ are series of sensor observations over time steps **t,** here we have to compute an estimate of $\boldsymbol{x_t}$**'s** location and a map of the environment $\boldsymbol{m_t}$**.**

By applying Bayes 'rule, we get a framework for updating the location posteriors, given a map and a transition function $\boldsymbol{P(x_t|x_{t-1})}$**,**

$$\boldsymbol{P(x_t|o_{1:t}, m_t) = \sum_{m_{t-1}} P(o_t|x_t, m_t) \sum_{x_{t-1}} P(x_t|x_{t-1}) P(x_{t-1}|m_t, o_{1:t-1})/Z}$$

For the map,

$$\boldsymbol{P(m_t\,|x_t\,, o_{1:t}) = \sum_{x_t} \sum_{m_t} P(m_t\,|x_t\,, m_{t-1}\,, o_t\,) P(m_{t-1}\,, x_t\,|o_{1:t-1}\,, m_{t-1}\,)}$$

Pseudocode:

- Project robot state distribution forward (robot motion model)
- Observe environment (laser scans)
- Update robot state by P(O|S)
- Update map (add new objects)
- Repeat

METHODOLOGY

1. Software Setup

The first steps in this project was to get the software stack ready. This course being one of the first courses in this program there was no prior setup of the required software packages. So a fresh install of Ubuntu 14.04 LTS was followed by installation of ROS indigo. This combination was chosen over Ubuntu 16.04 and ROS kinetic as a few of Turtlebot packages haven't been upgraded to Kinetic Kane. Performing a full installation of ROS ensured that Gazebo and Turtlebot were also installed with it. Following the Wiki ROS Tutorials were beneficial in learning how packages are created in ROS. A critical aspect of it was checking the dependencies and executables in CMakeLists.txt and package.xml. The next step in improving our skills in ROS was writing publishers and subscribers. This was essential for knowing how data can be sent and listened to from a topic while trying to move the Turtlebot.

MoveIt! is another set of packages and tools for doing mobile manipulation in ROS. There are many examples to demonstrate pick and place, grasping, simple motion planning, etc. MoveIt! contains state of art software for motion planning, 3D perception, collision checking, control and navigation. Apart from the command line interface, MoveIt! has some good GUI to interface a new robot to MoveIt!. Also, there is a RViz plugin which enables motion planning from RViz itself. But the need for using the C++ APIs of MoveIt! never rose as we stuck to teleoperation for observing motion in Gazebo.

For simulation of the planned motion, PR2 was also considered along with Turtlebot. The PR2 software system was installed and run for validating its use for our application.

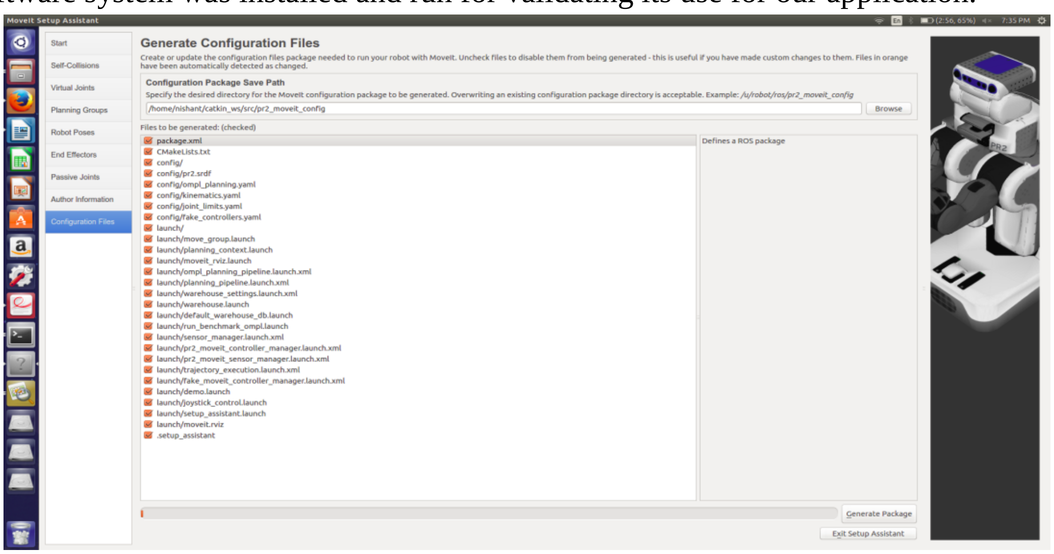


Fig. 2. PR2 MoveIt Setup Assistant

As we can see from figure 2 and 3 we need to setup the virtual joints, planning groups, robot poses, passive joints, end effectors to be used.

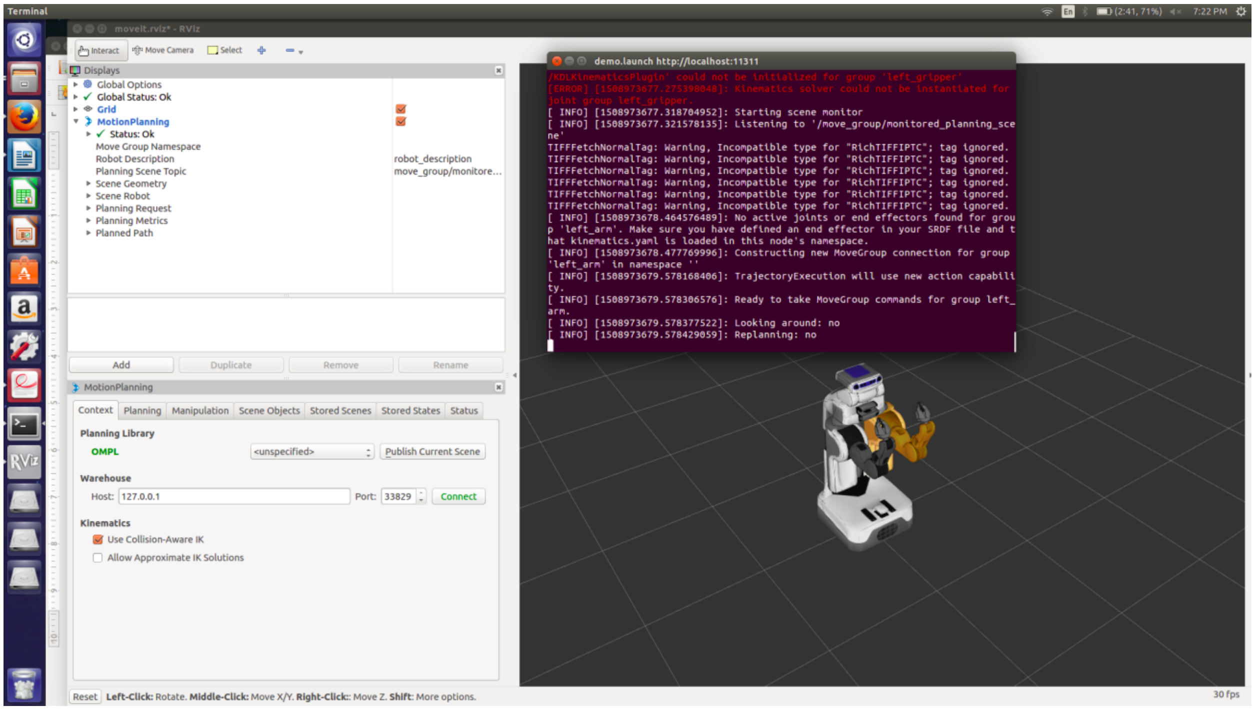


Fig. 3. PR2 Robot in Rviz

The Navigation stack for PR2 contained implementation of the standard algorithms, such as SLAM, A*(star), Dijkstra, AMCL, and so on, which can directly be used in our application. Navigation stack provides a node name move_base which receives a goal pose and links to many components to generate the output which is the command velocity which is then sent to the base controller for moving the robot for achieving the goal pose.

2. Turtlebot

TurtleBot (Figure-4) is a low-cost, personal robot kit with open-source software that uses a netbook as its main processor and Microsoft Kinect as its vision sensor. TurtleBot was specifically designed by Willow Garage to be used with ROS. The TurtleBot has many built-in sensors like bumper sensor, cliff sensor, wheel drop sensor, wheels encoder and gyro sensor. ROS provides demo code for the 2D SLAM process using TurtleBot. The Microsoft Kinect can be used to generate fake laser scan data from depth image, and using grid mapping for mapping processes. TurtleBot is designed to work with many computers, the netbook on the robot is used to read sensor data and control the robot movement. All heavy computation that is needed is to be done on another computer that has more processing power. Data between computer and the netbook is transmitted wirelessly through Wi-Fi. [2]

Fig. 4. TurtleBot

The TurtleBot can be simulated in Gazebo. Gazebo is an open source, multi-robot 3D simulator for complex environments. Gazebo is capable of simulation many robots, sensors and objects, it provides with both realistic sensor feedback and physically plausible interactions between the robots and the objects. Gazebo provides a TurtleBot simulator which can be used to provide a controlled environment for learning the TurtleBot motion. Figure-5 below shows the TurtleBot in Gazebo with some obstacles and also the teleoperation of the TurtleBot can be done using Keyboard in the gazebo.

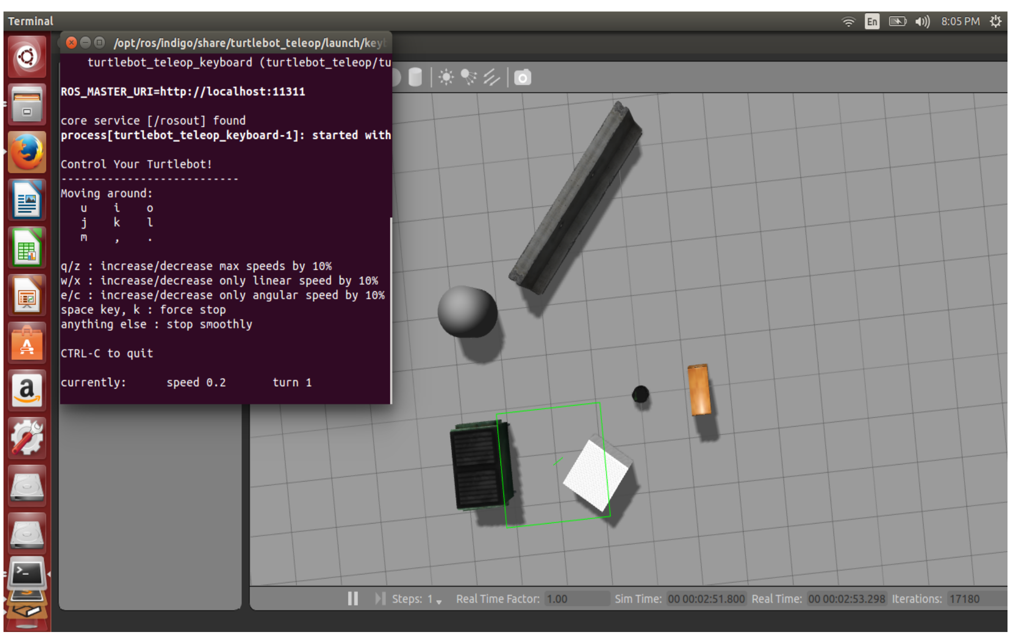


Fig. 5. TurtleBot in gazebo simulator and Tele-operating using keyboard with few obstacles

3. Dijkstra's and A* Algorithm

MATLAB was preferred for implementing the Dijkstra's and A* algorithms. We wrote a code for Dijkstra's algorithm in MATLAB and outsourced its visualization. Running the RunScript.m file will run the Dijkstra's or the A*(star) code. Figure 6 and 7 show the visualization of Dijkstra's and A* algorithms in MATLAB.

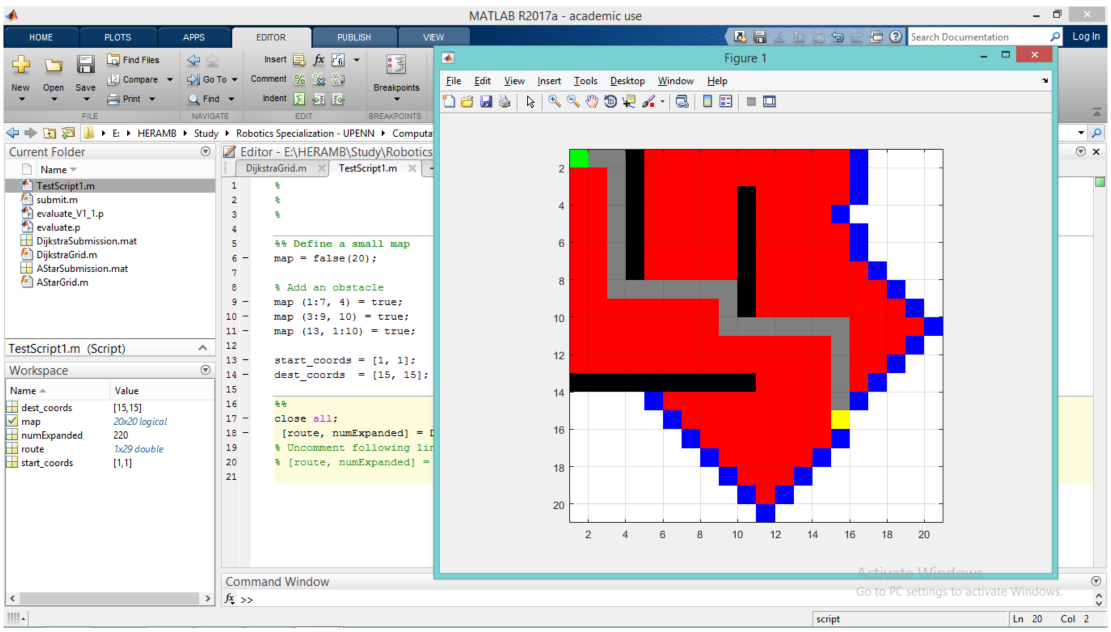


Fig. 6. Dijkstra's Path Planning Algorithm

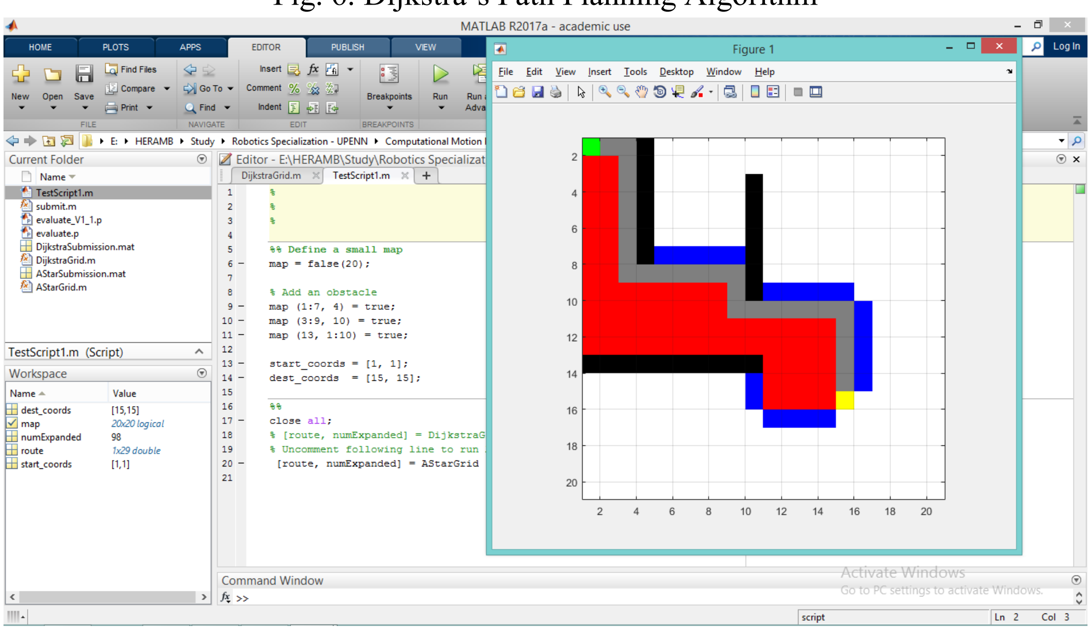


Fig. 7. A* Path Planning Algorithm

We observe that the A*algorithm visits far lesser cells in order to reach the goal location. This is because it does not search in all directions like the Dijkstra's algorithm but only in the direction of the distance heuristic. In the worst case scenario, A* will perform at par with the Dijkstra's algorithm.

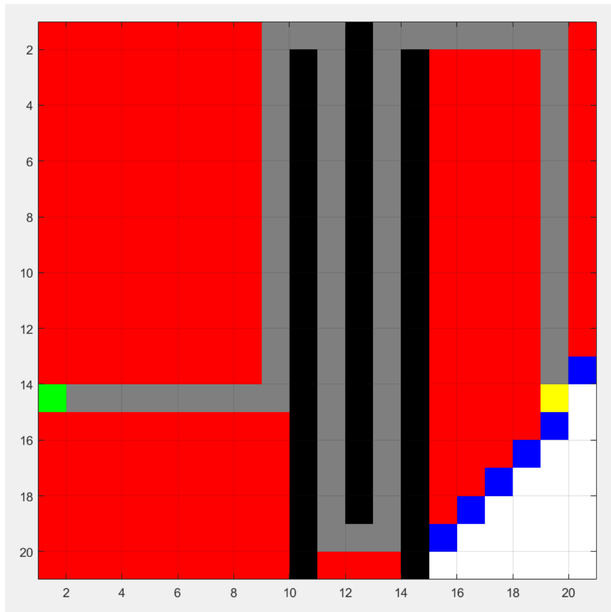


Fig. 8. Dijkstra's Algorithm for Case 2

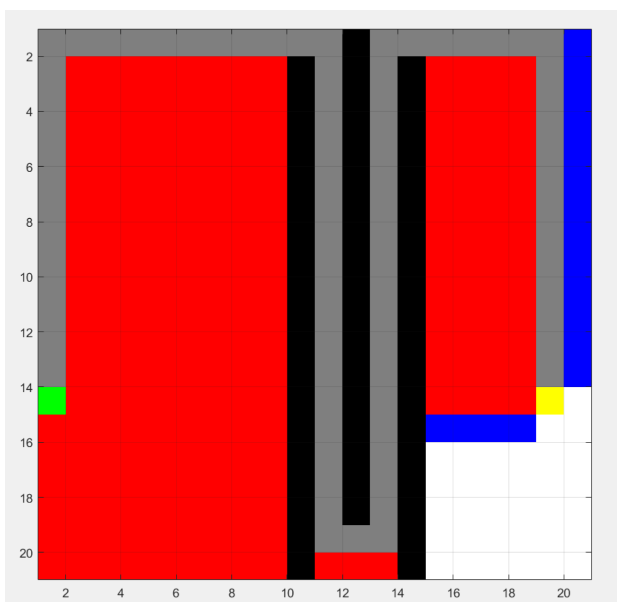


Fig. 9. A* Algorithm for Case 2

4. Creating the Gazebo World

Gazebo World can be used to create an environment, where one can place a collection of robots and objects such as buildings, tables, etc. and global parameters including the sky, ambient light and other physical properties. So, we decided to use it to make a dynamic environment where we can test our algorithms. Gazebo gives an option to create a new environment using an empty world as shown in figure - 10.

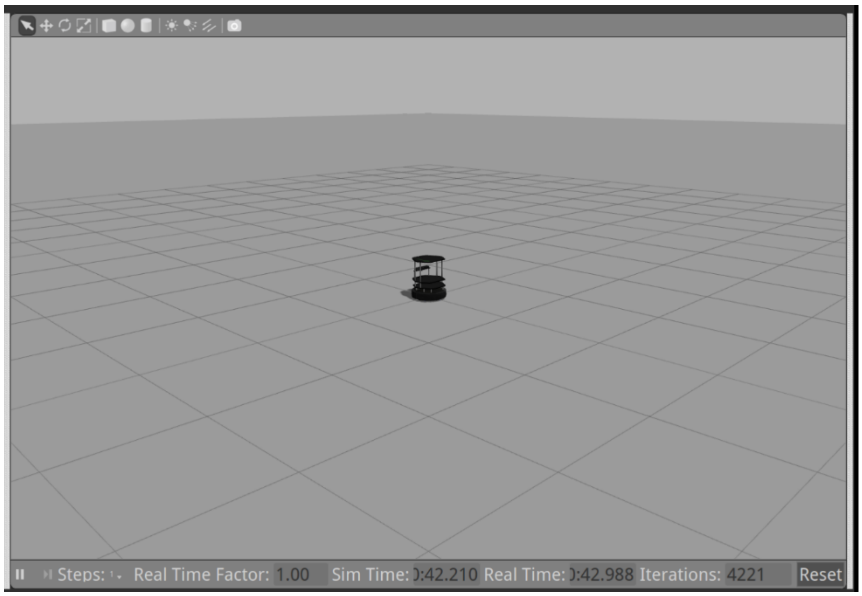


Fig. 10. : Turtlebot in an empty Gazebo world

So according to our problem statement we designed a dynamic environment in Gazebo as shown in the figure-11 , it consists of few daily used objects and human beings. The start location of the TurtleBot is in the bottom right corner and goal location is in the top left corner of the map. Gazebo gives an option to load the preexisting robots in it, so we loaded TurtleBot in our environment.

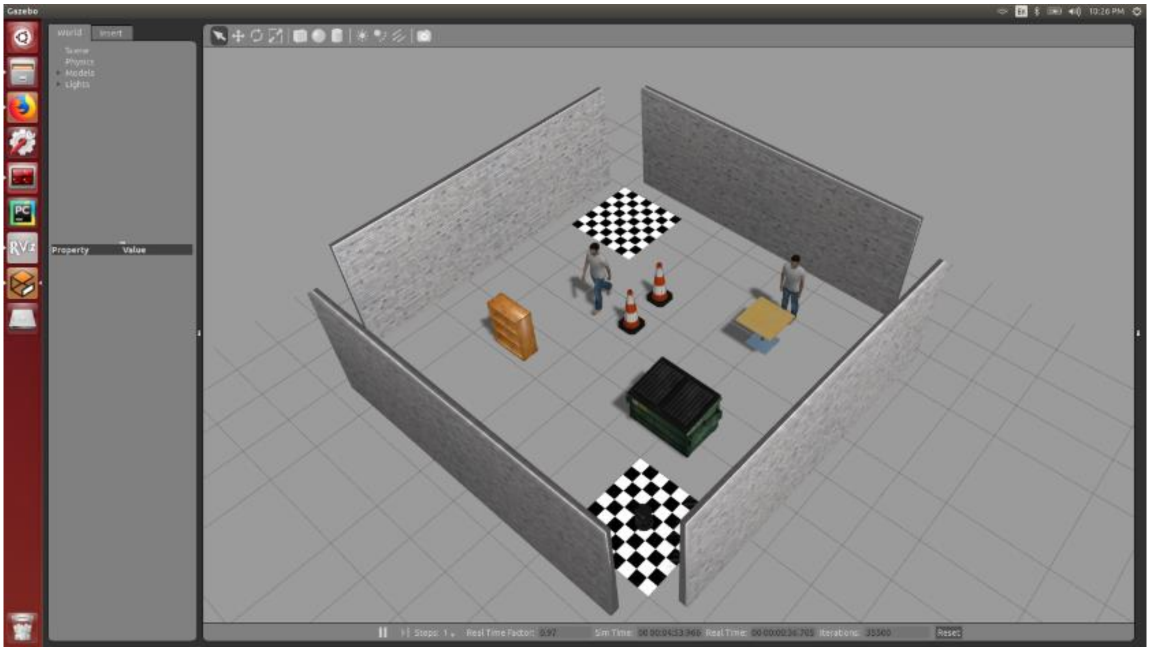


Fig. 11. Created Gazebo World

5. Gmapping

For path planning in this environment, a 2D map was required to be made for which we decided to use the GMapping tool in Gazebo. By teleoperating the robot in the created world we were able to generate the map by identifying free spaces from obstacles. This map was then used to create a binary map and store it in a text file.

To start the built map in Gazebo:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
world_file:=<file-path>/world_name.world
```

To start map building, type:

```
$ roslaunch turtlebot_gazebo gmapping_demo.launch
```

Use RViz to visualize the map building process:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

To save the map to disk:

```
$ rosrun map_server map_saver -f <map name>
```

To run the turtlebot in the mapped environment, close all the terminals and start the above process again and skip the building map step instead execute the below command to load the mapped environment:

```
$ roslaunch turtlebot_gazebo amcl_demo.launch map_file:=<full path to the map file>
```

To move the turtlebot to the desired location in the mapped environment, Rviz provides a feature named 2D Navgoal, where we need to select the desired position and orientation of the turtlebot. The turtlebot will try to implement the path planning algorithm and try to find the minimal cost path to reach the end goal. The visualization of this can be seen below:

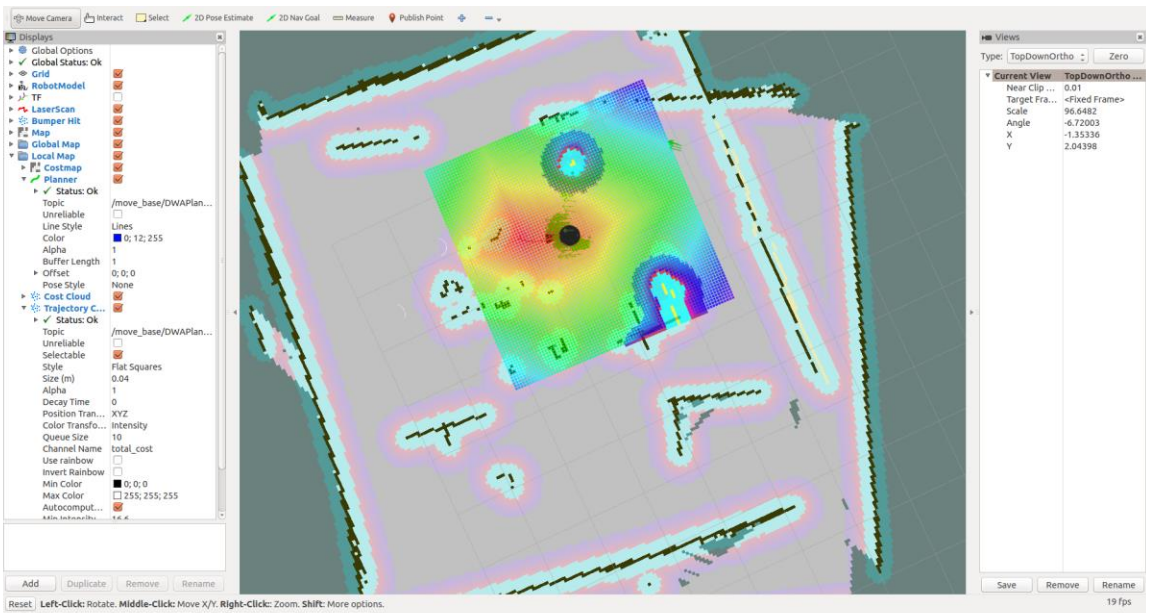


Fig. 12. Mapped Environment of the world in Gazebo

We also tried to plan the trajectory for the turtlebot between two fixed locations by using the cubic polynomial method. The code can be seen in the appendix. We defined the start and end locations and time frame as well.

Desired Position is given by below equation, where t is the timestamp, and a's are the unknown variables which can be obtained with the help of initial conditions.

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

Desired velocity is given by below equation

$$v(t) = a_1 + 2a_2t + 3a_3t^2$$

5. A* algorithm in Python

The A* algorithm written in MATLAB was ported to python by referring to the pseudo codes from [8] and [11]. The binary map created of the world is shown in figure 13.

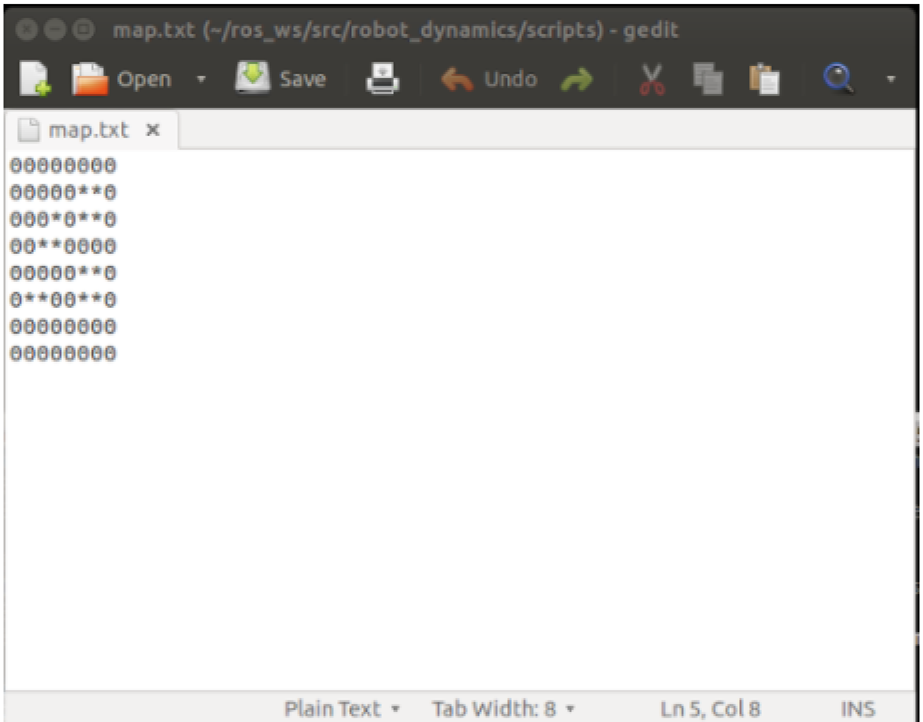


Fig. 13. Input map for A*

In this map the free locations are denoted by the value 0 and obstacles are denoted by an asterisk.

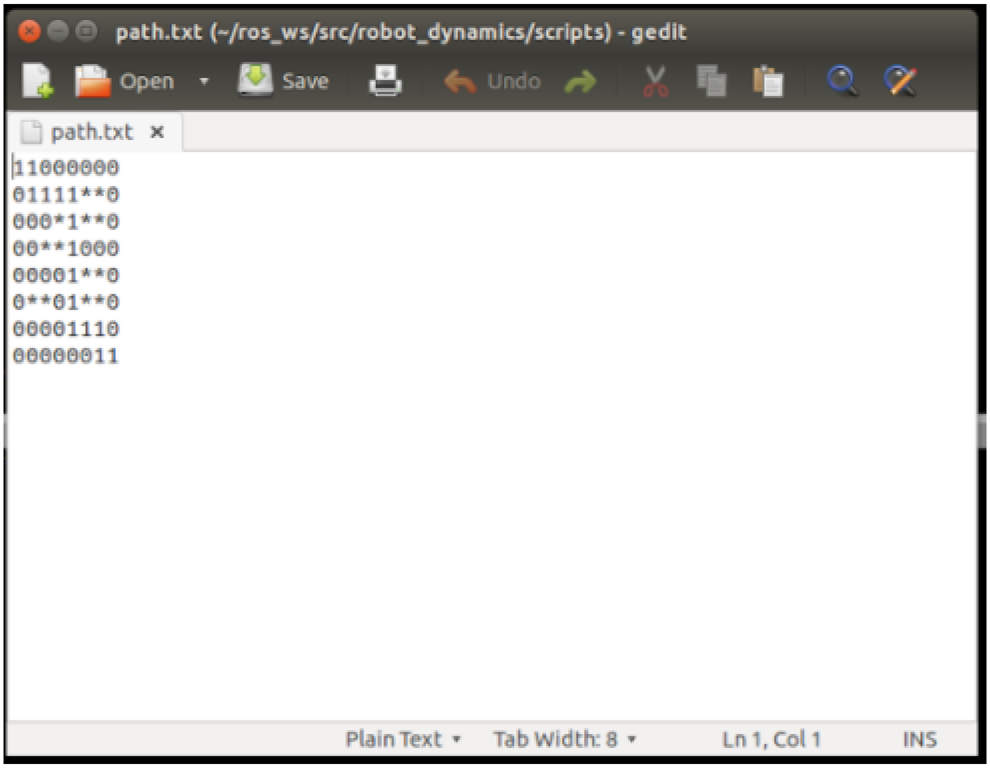


Fig. 14. Path found by A*

The path found by the A* algorithm is denoted by 1s and is the shortest path from the start to the goal location.

6. Turtlebot Motion Constraints:

For moving the Turtlebot in the Gazebo environment we wrote a python file that published the linear and angular velocities to the Turtlebot topic. The message here are of type Twist of geometry_msgs.msg. As the Turtlebot is a mobile robot, it only has three degrees of freedom: linear x and y in the horizontal plane and rotation about the z axis. Therefore by setting the 3 values to the Twist message and publishing it to *'/mobile_base/commands/velocity'* we can move the bot in the edited world. One issue faced during this execution was that the Turtlebot did not respond to linear velocities in the y direction. This lead us to examine the wheel constraints for the Turtlebot.

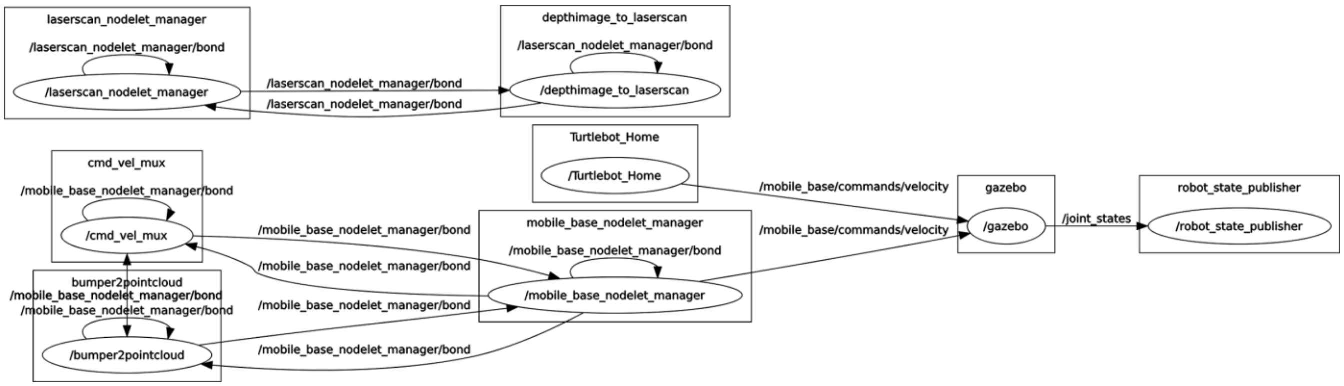


Fig 15: RQT plot

The turtlebot has two castor wheels and two standard wheels. Therefore the two standard wheels pose one sliding constraint each. The degree of mobility can hence be calculated as:

$$\delta_m = 3 - rank[C1(\beta_s)]$$

Therefore,

$$\delta_m = 1$$

As none of the wheels are steerable, Degree of Steerability:

$$\delta_s = 0$$

And finally, Degree of Maneuverability:

$$\delta_M = \delta_m + \delta_s$$

$$\delta_M = 1$$

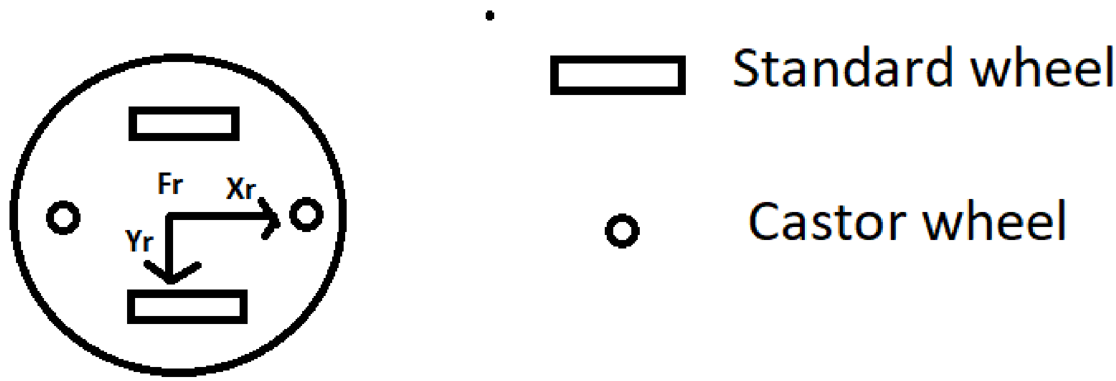


Fig 16: TurtleBot Wheel Constraints

CONCLUSION

Through this project we were able to accomplish several of our pre-defined objectives. We completed a comprehensive study of different path planning algorithms and also an overview of SLAM techniques. By the end of this project we gained substantial experience in ROS, which perhaps might be the most useful skill gained in this project. Unfortunately, we had to truncate our project aim of implementing SLAM as a group member had to leave midway through the course. Still we were able to do considerable work in creating a world and mapping it in Gazebo and simulating a few fundamental path planners in MATLAB.

FUTURE TASKS:

Having gained the required software skills in this project, it would be much easier now to implement the SLAM algorithms we studied. Also, the map generated was static in nature and should be replaced with a dynamic setting for a more accurate representation of a real world scenario.

As a topic for future research, a robust full SLAM algorithm can be pursued wherein the map is created online and hence is an unknown entity.

REFERENCES:

[1] A. Thallas, E. Tsardoulias and L. Petrou, "Particle filter — Scan matching SLAM recovery under kinematic model failures," *2016 24th Mediterranean Conference on Control and Automation (MED)*, Athens, 2016, pp. 232-237. doi: 10.1109/MED.2016.7535849.

[2] M. F. A. Ghani, K. S. M. Sahari and L. C. Kiong, "Improvement of the 2D SLAM system using Kinect sensor for indoor mapping," *2014 Joint 7th International Conference on Soft Computing and Intelligent Systems (SCIS) and 15th International Symposium on Advanced Intelligent Systems (ISIS)*, Kitakyushu, 2014, pp. 776-781.

[3] C. Stachniss, J. L. Leonard and S. Thrun, "Simultaneous Localization and Mapping" *2016 Springer Handbook of Robotics*, Chapter 46, pp. 1153-1174

[4] J. Minguez, F. Lamiraux and J. P. Laumond, "Motion Planning and Obstacle Avoidance" *2016 Springer Handbook of Robotics*, Chapter 47, pp. 1177-1201

[5] *http://wiki.ros.org/ROS/Tutorials*

[6]https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Pseudocode

[7]https://en.wikipedia.org/wiki/Grassfire_transform

[8]https://en.wikipedia.org/wiki/A*_search_algorithm

[9]https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping

[10]**http://gazebosim.org/tutorials?tut=build_world**

[11] https://github.com/wordswords

APPENDICES:

A* in MATLAB:

Contents


```
function [route,numExpanded] = AStarGrid (input_map, start_coords, dest_coords)
% color coding
```

```
% 1 - white - clear cell
% 2 - black - obstacle
% 3 - red = visited
% 4 - blue  - on list
% 5 - green - start
% 6 - yellow - destination

cmap = [1 1 1; ...
    0 0 0; ...
    1 0 0; ...
    0 0 1; ...
    0 1 0; ...
    1 1 0; ...
    0.5 0.5 0.5];

colormap(cmap);

drawMapEveryTime = true;

[nrows, ncols] = size(input_map);

map = zeros(nrows,ncols);

map(~input_map) = 1;
map(input_map)  = 2;

start_node = sub2ind(size(map), start_coords(1), start_coords(2));
dest_node  = sub2ind(size(map), dest_coords(1),  dest_coords(2));

map(start_node) = 5;
map(dest_node)  = 6;

parent = zeros(nrows,ncols);

[X, Y] = meshgrid (1:ncols, 1:nrows);

xd = dest_coords(1);
yd = dest_coords(2);

% Evaluate Heuristic function, H, for each grid cell
H = abs(X - xd) + abs(Y - yd);
H = H';

f = Inf(nrows,ncols);
g = Inf(nrows,ncols);

g(start_node) = 0;
f(start_node) = H(start_node);


numExpanded = 0;

while true

    map(start_node) = 5;
    map(dest_node) = 6;

    if (drawMapEveryTime)
        image(1.5, 1.5, map);
        grid on;
        axis image;
        drawnow;
    end
```

```
    [min_f, current] = min(f(:));

    if ((current == dest_node) || isinf(min_f))
        break;
    end;

    map(current) = 3;
    f(current) = Inf;

    [i, j] = ind2sub(size(f), current);

    % ********************************************************************
    % Self written code:

    [m, n] = size(min_f);

    if (current == dest_node)
        numExpanded = numExpanded + n -1;
    else
        numExpanded = numExpanded + n;
    end

    for I = 1:n
        i = i(m,I);
        j = j(m,I);

        if (i>1 & i<=(nrows+1) & j>=1 & j<=ncols & input_map(i-1,j)~=1 & map(i-1,j)~=3
& map(i-1,j)~=5)
            next_node = sub2ind(size(map),i-1,j);
            map(next_node) = 4;
            n_dist = g(i-1,j);
            if (n_dist > (g(i,j) + 1))
                n_dist = g(i,j) + 1;
                g(i-1,j) = n_dist;
                f(i-1,j) = g(i-1,j) + H(i-1,j)
            end
            parent(i-1,j) = current(m,I);

        end

        if (i>=1 & i<=nrows & j>1 & j<=(ncols+1) & input_map(i,j-1)~=1 & map(i,j-1)~=3
& map(i,j-1)~=5)
            next_node = sub2ind(size(map),i,j-1);
            map(next_node) = 4;
            n_dist = g(i,j-1);
            if (n_dist > (g(i,j) + 1))
                n_dist = g(i,j) + 1;
                g(i,j-1) = n_dist;
                f(i,j-1) = g(i,j-1) + H(i,j-1)
            end
            parent(i,j-1) = current(m,I);

        end

        if (i>=1 & i<nrows & j>=1 & j<=ncols & input_map(i+1,j)~=1 & map(i+1,j)~=3 &
map(i+1,j)~=5)
            next_node = sub2ind(size(map),i+1,j);
            map(next_node) = 4;
            n_dist = g(i+1,j);
            if (n_dist > (g(i,j) + 1))
                n_dist = g(i,j) + 1;
                g(i+1,j) = n_dist;
```

```
                f(i+1,j) = g(i+1,j) + H(i+1,j);
            end
            parent(i+1,j) = current(m,I);

        end

        if (i>=1 & i<=nrows & j>=1 & j<ncols & input_map(i,j+1)~=1 & map(i,j+1)~=3 &
map(i,j+1)~=5)
            next_node = sub2ind(size(map),i,j+1);
            map(next_node) = 4;
            n_dist = g(i,j+1);
            if (n_dist > (g(i,j) + 1))
                n_dist = g(i,j) +1;
                g(i,j+1) = n_dist;
                f(i,j+1) = g(i,j+1) + H(i,j+1)
            end
            parent(i,j+1) = current(m,I);

        end
    end

    %*********************************************************************

end
```

Construct route from start to destination by following the parent links

```
if (isinf(f(dest_node)))
    route = [];
else
    route = [dest_node];

    while (parent(route(1)) ~= 0)
        route = [parent(route(1)), route];
    end


    for k = 2:length(route) - 1
        map(route(k)) = 7;
        pause(0.1);
        image(1.5, 1.5, map);
        grid on;
        axis image;
    end
end
end
```

A* algorithm

Define a small map

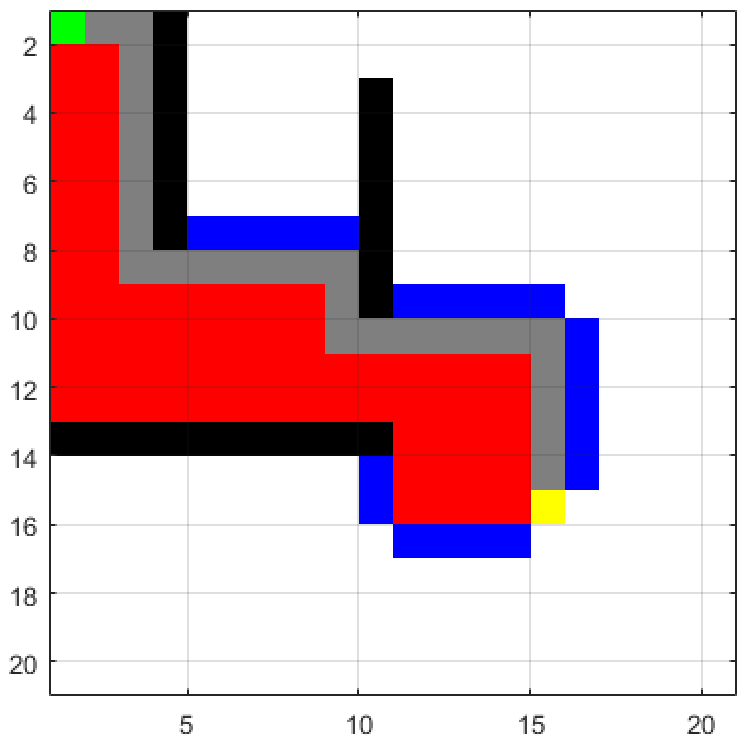
```
map = false(20);



% Add an obstacle
map (1:7, 4) = true;
map (3:9, 10) = true;
map (13, 1:10) = true;
```

```
start_coords = [1, 1];
dest_coords  = [15, 15];
close all;
% [route, numExpanded] = DijkstraGrid (map, start_coords, dest_coords);

% Comment previous line and Uncomment next line to run Astar
[route, numExpanded] = AStarGrid (map, start_coords, dest_coords);
```



Dijkstra's:

Contents

- [Construct route from start to dest by following the parent links]

```
function [route,numExpanded] = DijkstraGrid (input_map, start_coords, dest_coords)
% color coding
% 1 - white - clear cell
% 2 - black - obstacle
% 3 - red = visited
% 4 - blue  - on list
% 5 - green - start
% 6 - yellow - destination

cmap = [1 1 1; ...
        0 0 0; ...
```

```
        1 0 0; ...
        0 0 1; ...
        0 1 0; ...
        1 1 0; ...
         0.5 0.5 0.5];

colormap(cmap);

drawMapEveryTime = true;

[nrows, ncols] = size(input_map);

map = zeros(nrows,ncols);

map(~input_map) = 1;
map(input_map)  = 2;

start_node = sub2ind(size(map), start_coords(1), start_coords(2));
dest_node  = sub2ind(size(map), dest_coords(1),  dest_coords(2));

map(start_node) = 5;
map(dest_node)  = 6;

distanceFromStart = Inf(nrows,ncols);

parent = zeros(nrows,ncols);

distanceFromStart(start_node) = 0;

numExpanded = 0;

while true

    map(start_node) = 5;
    map(dest_node) = 6;

    if (drawMapEveryTime)
        image(1.5, 1.5, map);
        grid on;
        axis image;
        drawnow;
    end


    [min_dist, current] = min(distanceFromStart(:));

    if ((current == dest_node) || isinf(min_dist))
        break;
    end;


    map(current) = 3;
    distanceFromStart(current) = Inf;

   [i, j] = ind2sub(size(distanceFromStart), current);

   % *********************************************************************
   % Self witten code:
    [m, n] = size(min_dist);

    if (current == dest_node)
        numExpanded = numExpanded + n -1;
    else
```

```
        numExpanded = numExpanded + n;
    end

    for I = 1:n
        i = i(m,I);
        j = j(m,I);

        if (i>1 & i<=(nrows+1) & j>=1 & j<=ncols & input_map(i-1,j)~=1 & map(i-1,j)~=3
& map(i-1,j)~=5)
        next_node = sub2ind(size(map),i-1,j);
        map(next_node) = 4;
        n_dist = distanceFromStart(i-1,j);
        if (n_dist > (min_dist(m,I) + 1))
            n_dist = min_dist(m,I) +1;
        distanceFromStart(i-1,j) = n_dist;
        end
        parent(i-1,j) = current(m,I);

        end

        if (i>=1 & i<=nrows & j>1 & j<=(ncols+1) & input_map(i,j-1)~=1 & map(i,j-1)~=3
& map(i,j-1)~=5)
             next_node = sub2ind(size(map),i,j-1);
            map(next_node) = 4;
            n_dist = distanceFromStart(i,j-1);
            if (n_dist > (min_dist(m,I) + 1))
                n_dist = min_dist(m,I) +1;
            distanceFromStart(i,j-1) = n_dist;
            end
            parent(i,j-1) = current(m,I);

        end

        if (i>=1 & i<nrows & j>=1 & j<=ncols & input_map(i+1,j)~=1 & map(i+1,j)~=3 &
map(i+1,j)~=5)
         next_node = sub2ind(size(map),i+1,j);
         map(next_node) = 4;
         n_dist = distanceFromStart(i+1,j);
        if (n_dist > (min_dist(m,I) + 1))
            n_dist = min_dist(m,I) +1;
        distanceFromStart(i+1,j) = n_dist;
        end
        parent(i+1,j) = current(m,I);

        end

        if (i>=1 & i<=nrows & j>=1 & j<ncols & input_map(i,j+1)~=1 & map(i,j+1)~=3 &
map(i,j+1)~=5)
       next_node = sub2ind(size(map),i,j+1);
       map(next_node) = 4;
       n_dist = distanceFromStart(i,j+1);
        if (n_dist > (min_dist(m,I) + 1))
            n_dist = min_dist(m,I) +1;
        distanceFromStart(i,j+1) = n_dist;
        end
        parent(i,j+1) = current(m,I);
        end
    end

    %*********************************************************************

end
```

Construct route from start to dest by following the parent links

```
if (isinf(distanceFromStart(dest_node)))
    route = [];
else
    route = [dest_node];

    while (parent(route(1)) ~= 0)
        route = [parent(route(1)), route];
    end


    for k = 2:length(route) - 1
        map(route(k)) = 7;
        pause(0.1);
        image(1.5, 1.5, map);
        grid on;
        axis image;
    end
end
end
```

Define a small map

Dijkstra's algorithm

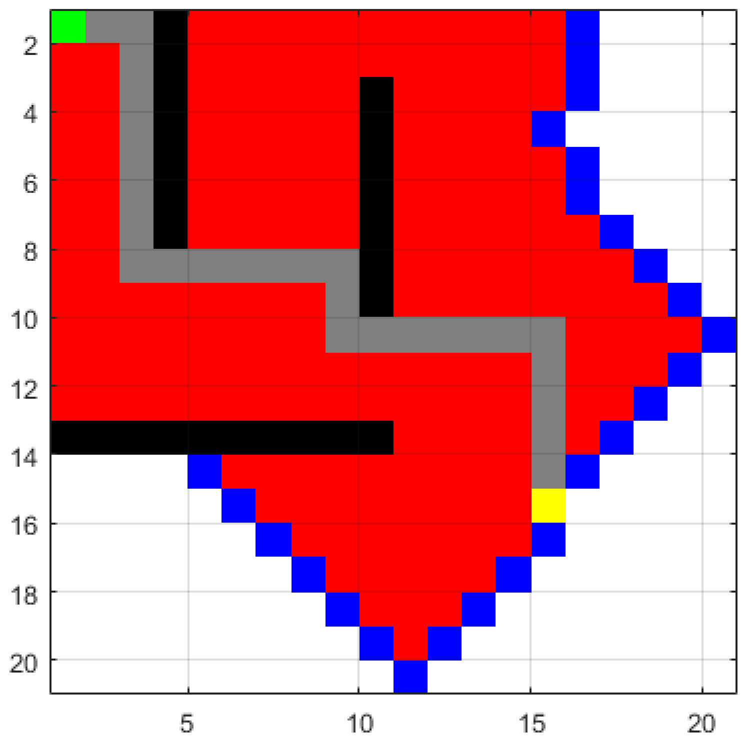
```
map = false(20);




% Add an obstacle
map (1:7, 4) = true;
map (3:9, 10) = true;
map (13, 1:10) = true;

start_coords = [1, 1];
dest_coords  = [15, 15];
close all;
 [route, numExpanded] = DijkstraGrid (map, start_coords, dest_coords);

% Comment previous line and Uncomment next line to run Astar
% [route, numExpanded] = AStarGrid (map, start_coords, dest_coords);
```



A* in python:

```
#!/usr/bin/env python
import math
import sys
import rospy
from block import block
from grid import grid

class a_star(object):
    closedSet = set()
    openSet = set()
    cameFrom, gscore, fscore = {}, {}, {}
    surface = ""

    def __init__(self):

        i = 0
```

```
        for sq in self.run():
            self.surface.units[sq.x][sq.y].state = '1'
            i = i + 1
      with open('path.txt','w') as file:
          file.write(str(self.surface))
        print self.surface

    def run(self):

        # setup grid
        self.surface = grid(sys.argv[1], 8, 8)
        start = self.surface.start
        goal = self.surface.goal

        # initialise start node
        self.gscore[start] = 0
        self.fscore[start] = self.gscore[start] +
self.heuristic(start,goal)
        self.openSet.add(start)

        while self.count(self.openSet) > 0:
            # pick an unevaluated node with the shortest distance
            fscore_sorted = sorted(self.fscore, key=lambda block:
self.gscore[block] + self.heuristic(block,goal))
            i = 0
            for i in range(len(fscore_sorted)-1):
                if(fscore_sorted[i] not in self.closedSet):
                    break
            current = fscore_sorted[i]

            if current == goal:
                return self.find_path(goal)

            try:
                self.openSet.remove(current)
            except KeyError,e:
                pass

            self.closedSet.add(current)
```

```
            for neighbour in self.neighbour_nodes(current):
                if neighbour not in self.closedSet:

                    temp_gscore = self.gscore[current] + 1
                    if (neighbour not in self.openSet) or
(temp_gscore < self.gscore[neighbour]):
                        # evaluate an unevaluated member or replace
its value with a smaller one
                        self.cameFrom[neighbour] = current
                        self.gscore[neighbour] = temp_gscore
                        self.fscore[neighbour] =
self.gscore[neighbour] + self.heuristic(neighbour,goal)

                        if neighbour not in self.openSet:
                            self.openSet.add(neighbour)


        print "Reached the end of nodes to expand, failure"

    def neighbour_nodes(self,node):
        """ Generate a set of neighbouring nodes """
        neighbours = set()

        if node.north != 0:
            neighbours.add(node.north)
        if node.east != 0:
            neighbours.add(node.east)
        if node.west != 0:
            neighbours.add(node.west)
        if node.south != 0:
            neighbours.add(node.south)

        return neighbours

    def distance(self,start_node,end_node):
        """ The distance in a straight line between two points on the
grid """
        x = start_node.x - end_node.x
        y = start_node.y - end_node.y
```

```
        return 1 * max(abs(x),abs(y))

    def evaluation_function(self,node,goal):
        """ Our evaluation function is the distance function plus the
cost of the path so far """
        return (node.self.distance(goal) + node.path_cost)

    def heuristic(self,start_node,end_node):
        heuristic = self.distance(start_node,end_node)
        return heuristic

    def find_path(self, current_node):
        """ Reconstruct the path recursively by traversing back
through the cameFrom list """

        try:
            self.cameFrom[current_node]
            p = self.find_path(self.cameFrom[current_node])
            path = []
            path.extend(p)
            path.append(current_node)
            return path
        except KeyError,e:
            # we have reached the start node
            return [current_node]

    def count(self,set_to_count):
        total_count = 0
        for i in set_to_count:
            total_count = total_count + 1
        return total_count

a_star()
```

Trajectory Planning using Cubic in python:

```
#!/usr/bin/env python
```

```
import rospy
from geometry_msgs.msg import Twist
import numpy as np
class Robot:
    def __init__(self):

        self._vel_pub =
rospy.Publisher('mobile_base/commands/velocity', Twist, queue_size=1)

        #TurtleBot will stop if we don't keep telling it to move.
How often should we tell it to move? 10 HZ
        r = rospy.Rate(10);
      # Twist is a datatype for velocity
        move_cmd = Twist()
      t = []
      for i in range(1,50):
          i *= 0.1
          t.append(i)

          #print(t)
        b = [[0.5],[0],[0.6],[0]]
      #for i in range(len(t)):
      A  = np.matrix( [ [1, 0, 0, 0],
              [0, 1, 0, 0],
              [1, t[1], t[1]**2, t[1]**3],
              [0, 1, 2*t[1], 3*(t[1]**2)]])
      a = np.array(np.linalg.inv(A)*b)
      qd = a[0] + a[1]*t[1] + a[2]*(t[1]**2) + a[3]*(t[1]**3)
      vd = a[1] + 2*a[2]*t[1] + 3*(a[3]*(t[1]**2))
      #move_cmd.linear.x = 0
      move_cmd.angular.z = 0.01
      #if move_cmd.angular.z == 1.57:
      move_cmd.linear.x = 0.1
      move_cmd.linear.y = vd
      while not rospy.is_shutdown():
          # publish the velocity
            self._vel_pub.publish(move_cmd)
          # wait for 0.1 seconds (10 HZ) and publish again
            r.sleep()
```

```
    def shutdown(self):
        # stop turtlebot
        rospy.loginfo("Stop TurtleBot")
      # a default Twist has linear.x of 0 and angular.z of 0.  So
it'll stop TurtleBot
        self.cmd_vel.publish(Twist())
      # sleep just makes sure TurtleBot receives the stop command
prior to shutting down the script
        rospy.sleep(1)

if __name__ == '__main__':
#    rospy.sleep(1)
    rospy.init_node('Move_turtlebot')
    turtle = Robot()

    while not rospy.is_shutdown():
        pass
```

Moving Turtlebot through python executable:

```
#!/usr/bin/env python

import rospy
import sys
from geometry_msgs.msg import Twist

class Home:
    def __init__(self):

        self.vel_pub =
rospy.Publisher('mobile_base/commands/velocity', Twist,
queue_size=100)

        r = rospy.Rate(1);
      self.i = 0
      while not rospy.is_shutdown():
           vel = Twist()
```

```python
        if self.i == 2:
            vel.linear.x = 0.0
            vel.angular.z = -2.6
        elif self.i == 4:
            vel.linear.x = 0.86
            vel.angular.z = 0.0
        elif self.i == 6:
            vel.linear.x = 0.0
            vel.angular.z = 2.6
        elif self.i == 8:
            vel.linear.x = 0.86
            vel.angular.z = 0.0
        else:
            vel.linear.x = 0.0
            vel.angular.z = 0.0
        vel.linear.y = 0.0
        self.i = self.i + 1
        print vel
        self.vel_pub.publish(vel)
        r.sleep()

if __name__ == '__main__':
    rospy.init_node('Turtlebot_Home')
    turtlebot = Home()

    while not rospy.is_shutdown():
        pass
```