AUTONOMOUS PLANT WATERING ROBOT

by

Charmin Pritesh Desai (50351140)

Spring 2023

Advisor

Prof. Dr. Farshad Ghanei

A project report submitted to the
faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfillment of the requirements for the
degree of

Master of Science
Department of Engineering Science - MS in Robotics

TABLE OF CONTENTS

Acknowledgements

I would like to express my appreciation to all those who provided me with the possibility to complete this project.

A special gratitude I give to my project advisor Professor Dr. Farshad Ghanei, whose contribution in teaching Robotics Algorithms & ROS in my master's curriculum and stimulating suggestions and encouragement, helped me to coordinate my project.

Furthermore, I would also like to acknowledge with much appreciation for my program director Professor Dr. Vojislav Kalanovic, who gave the permission to work on this project and contributing immensely towards my degree in terms of teaching and providing knowledge about automation and robotics .

A special thanks goes to my program coordinator MS Kaeleigh Peri for always guiding me towards my graduate degree in Robotics.

Brief Description

This project is about developing a plant watering robot. The goal is for the robot to water household plants autonomously. It should explore the unknown environment and build a map, and scan for possible candidate targets (plants) at the same time. It should validate the data with the user, through a friendly interface. Once the map is finalized, the robot should be able to navigate in the map and reach the plants avoiding obstacles, then water them according to the user's preferences.

1) Introduction

The project focuses on watering household plants. Therefore, it has been developed through ROS and on a Gazebo simulator. So, a package from ROBOTIS website called "Turtlebot3" has been installed and utilized. This package offers different gazebo environments, of which the house environment has been used in compliance with the project description.

SLAM

First the robot needs to know about the environment. For that we will perform SLAM (simultaneous localization and mapping). SLAM process here builds a 2D map of the environment

Thus, using the TURTLEBOT3 SLAM package the turtlebot3 is moved around in the gazebo house environment, while in this process the SLAM (gmapping method) node builds the map and returns an image file which can be used later for navigation.

Now we have the SLAM image of the environment, therefore we need to set up a path planning algorithm for our robot to reach its target locations. We will implement the A* path planning algorithm.

Then, the next step would be to enable the robot in a way that it can reach different locations in the gazebo house environment after planning the path. Those locations will include plants.

Occupancy Grid Formation

While now, to write A*, an occupancy grid of the gazebo house environment is needed. Hence the SLAM image is used to create the occupancy grid. The SLAM image will be a greyscale image. Each grey, black, and white pixel in the image will be an indication of an unexplored area, obstacle, and free space respectively.

We will convert the image into a NumPy array. The array will only have numeric data. Grey and black pixels will be set as “1” and white pixel will be set as "0”.

Each digit (0 & 1) of occupancy grid array will have a corresponding coordinate of the house environment. This occupancy grid directly represents our house environment. The A* algorithm plans the path in the occupancy grid choosing only “0’s” from start to goal location.

Coordinate Relations

But because our robot will traverse in real world environment, we will need to convert the location of each cell of occupancy grid frame into world coordinate frame and vice- versa.

Path Planning using A* algorithm

Now when we assign the start and goal location to the A* algorithm, it first converts the world coordinate into corresponding occupancy grid coordinate, then plans the path in the occupancy grid coordinates. Now the planned path has many coordinates in occupancy grid frame of reference. Thus, it will be then converted into a world coordinate frame for the robot. This will be the final path for the robot to traverse and eventually reach the goal.

ROS AprilTags

Going one step further, if we look back the robot was relying on its real time position from subscribing to the odometry topic. But in the real world, there is no such thing. Therefore, we need to set a system from which the robot can estimate its true position and correct itself if needed.

For this we can make use of AprilTags. AprilTag is a visual fiducial system, useful for a wide variety of tasks including augmented reality, robotics, and camera calibration. Targets can be created from an ordinary printer, and the AprilTag detection software computes the precise 3D position, orientation, and identity of the tags relative to the camera.

In ROS gazebo, we will set up multiple AprilTags at multiple locations in the gazebo house environment. We will also define the pose of each tag with respect to the world frame of reference. Also, each AprilTag has a unique tag ID. We can also attach AprilTag on the pot of the plants and define the tag ID to the robot.

The robot will detect the AprilTag and determine if it is a plant or not using predefined tag-ID. It will then calculate the transformation from its camera frame to the AprilTag frame. Already knowing the other transformations such as from its base to camera, and world frame to each tag frame, we can easily estimate the pose of the robot in the world frame.

Now our robot can reach the target location relying on pose estimation from AprilTags. This solves our localization from AprilTag detection. Using frame transformation mathematics, we have also estimated the pose of an AprilTag.

Robot Driver (localization from odometry)

Odometry is a ROS Gazebo topic from which the pose of the robot can be fetched. Hence the robot can be localized from /odom topic. However, this shouldn't be used as no such topic exists in real world applications. But we first create a robot driver node that drives the robot from one location to another such that robot relies on /odom topic for its position.

Robot Driver (localization from ArpilTag detection)

We talked above about localization from AprilTags. Hence instead of /odom topic, we rely on AprilTags for the robot's position. And the driver programming remains the same for driving the robot.

But the drawback is that if the robot does not see any AprilTag then it cannot localize itself. And it is not a good practice to mess up the environment with hundreds of AprilTags. So AprilTags should be used to rectify the robot's pose and not to be fully relied upon.
Therefore, there should be other means to robot localization, as discussed below.

Robot Driver (localization through velocity estimation)

One of the best practices of robot localization is to enable the robot to estimate its own pose with the help of its own velocity. So, we need to know the robot's velocity and use it in a constant loop with time and thus mathematically estimate its position. This will give us an estimate that could be about 70-80% accurate. The accuracy is made > 95% while an AprilTag is detected, and the pose is updated.

Plant Pose Estimation

The pose estimation of AprilTags is also important because plants can easily be identified if an AprilTag is placed on their pot. That way the robot can find if there is plant located somewhere and estimate its accurate position. Rather than we give the pose of each plant to the robot.

K-Means Clustering for Exploration

Now comes the robot exploration phase. Here through programming, we come up with multiple points in the environment covering it all up. Such that when robot reaches out to each point it has explored the environment and, in the process, if it detects an AprilTag (plant) then it estimates its position and saves the data offline.

We use the K-Means Clustering algorithm to group different points together into distinct areas of the environment. So that we can distinctively differentiate different areas of the environment.

World Exploration and Plant Finding

This is useful when the robot is exploring the environment. Ideally this process should be done when the robot is performing SLAM. But this project begins with plants (AprilTags) put in the environment and robot exploring the environment after it has completed SLAM and thereafter occupancy grid is created.

Hence from the above concept of plant pose estimation, while the robot explores the world, it looks for plants in parallel, and saves their location with their identity into its memory.

Robot Watering the Plants

The last part is watering the plant. Where the robot loads the plants data that was saved during the exploration phase. Now the robot can use this data to reach out to all the plants and water them. The robot sorts the points into an order that will minimize the navigation time and distance to water all the plants.

2) Robot Operating System (ROS)

What is ROS ?

ROS is a flexible framework or middleware for writing robot software, and collection of modular software tools and libraries that help in creating complex and robust robot behavior.

ROS is a middleware that provides high level abstraction between low level hardware, drivers, and high-level software API's.

ROS provides a sub-environment within an OS like ubuntu and allows multiple robot software files
to communicate with each other seamlessly.

ROS Architecture :-

ROS is composed of nodes; nodes are processes running into a computer. Nodes register themselves to a ROS-Master which is the core node of ROS Ecosystem. First roscore starts the ROS-Master then nodes register themselves to ROS-master and communicate among each other.

ROS defines different Client Libraries both in C++ & Python programming languages. It uses TCP and UDP for inter-process communication.

When/Why to use ROS ?

e.g.,

Building an obstacle avoiding robot, it might be easy by using simple Arduino and a single software file. But imagine building the same obstacle avoiding robot with additional capabilities of mapping its environment using additional and advanced sensors like LiDAR and knowing its exact location within that map (this needs a lot more programming than a single software file and the ROS community provides such advanced software libraries that can be used to make such a robot, yes this is very cool !!!).

e.g.,

Robots using Arduino is simple, but more the sensors, actuators, controllers you add on your application the more complex it becomes until you reach a point where everything is mixed up and you can’t add any code for a new sensor without having a huge headache. ROS is here to help you develop powerful and scalable robotic applications. You can use it whenever you want to create a robot software which needs a lot of communication between its sub-programs or has some functionalities that go beyond a very simple use case etc., so for a robot that will just open a door with servo motor when it detects a movement maybe you don’t need ROS, but for a mobile robot which you want to control with a GPS and a camera, in this case ROS might really help you.

e.g.,

The difference between running a piece of code on robot vs running on ROS is, a piece of code doesn't provide you flexibility to add multiple things to it while assuring it will work perfectly. While ROS provides you with an open-source platform with a large community and numerous frameworks and libraries to help you create your own robot according to your needs.

What is more interesting is that there is software called GAZEBO that runs alongside the ROS environment and helps users to build and simulate robots virtually. This means you need to have a good computer with ROS and GAZEBO running on it so that you can build your dream robot without spending any money on it.

Main 3 ROS terms :

1) Master - It is the core of the ROS environment it initiates the ROS environment starts listening for any request or data from the oscillator software modules. It manages and monitors everything that happens inside an ROS system. It provides naming and registration services for ROS nodes and helps them to locate one another. It's both HR and a manager.

2) Node - It is any robot software file built to work under the ROS master. Usually there will be lots of nodes pre-built in ROS network ready for us to access and use them. Every node in a ROS environment communicates with the same and only master on whether they are expecting input or giving output along with other personal stuff like their names, type of input/output they give, their recipient's name, senders name, etc. So, if you want to know what all nodes are currently running and what they are doing just ask the master.

3) Topic - They are channels of data transfer between nodes. When node "a" wants to output data that node "b" can use as input, both the nodes register a common name with the master as their channel of communication. And the master takes this name and creates a channel of communication with that name, and this channel is called a topic. A topic can have any number of subscribers but only one publisher.

The modular way of programming and easy communication protocols between multiple robot software files have made it easier to make advanced and complex robots using ROS

Advantage of using ROS :-

At any time, it gives the user the possibility to send commands to make the robots move and it will give you as a developer the ability to read the state of the robot at any time. state = (position, image from the camera sensor, location of an object from the laser scanner). Thus, you can easily get any information from the robot and can easily send any kind of command to the robot you want to control. Thus, to make communication between different nodes and different processes inside the robot, thus, to develop a particular application.

ROS Limitations :-

a) It is limited to single robots and not swarm robots. So, if you want to do multi robot cooperation with ROS either you need to develop your own packages for multi robot communications or there are some contributed packages by some researchers. But there is nothing native that will allow the robots to communicate b/w each other using ROS

b) It is not real time. It is based on UDP and TCP, so from processes perspective all the processes would have the same priority. And so maybe we need to differentiate b/w the priority of processes running into a robot.

c) It needs a very reliable network, because ROS will be composed of several nodes and all these nodes will communicate b/w each other. And every node will consume a bandwidth. So, you will assume that you have a high and reliable bandwidth connection network connection b/w the different processes running inside the robot. And maybe this is something not easy to provide every time.

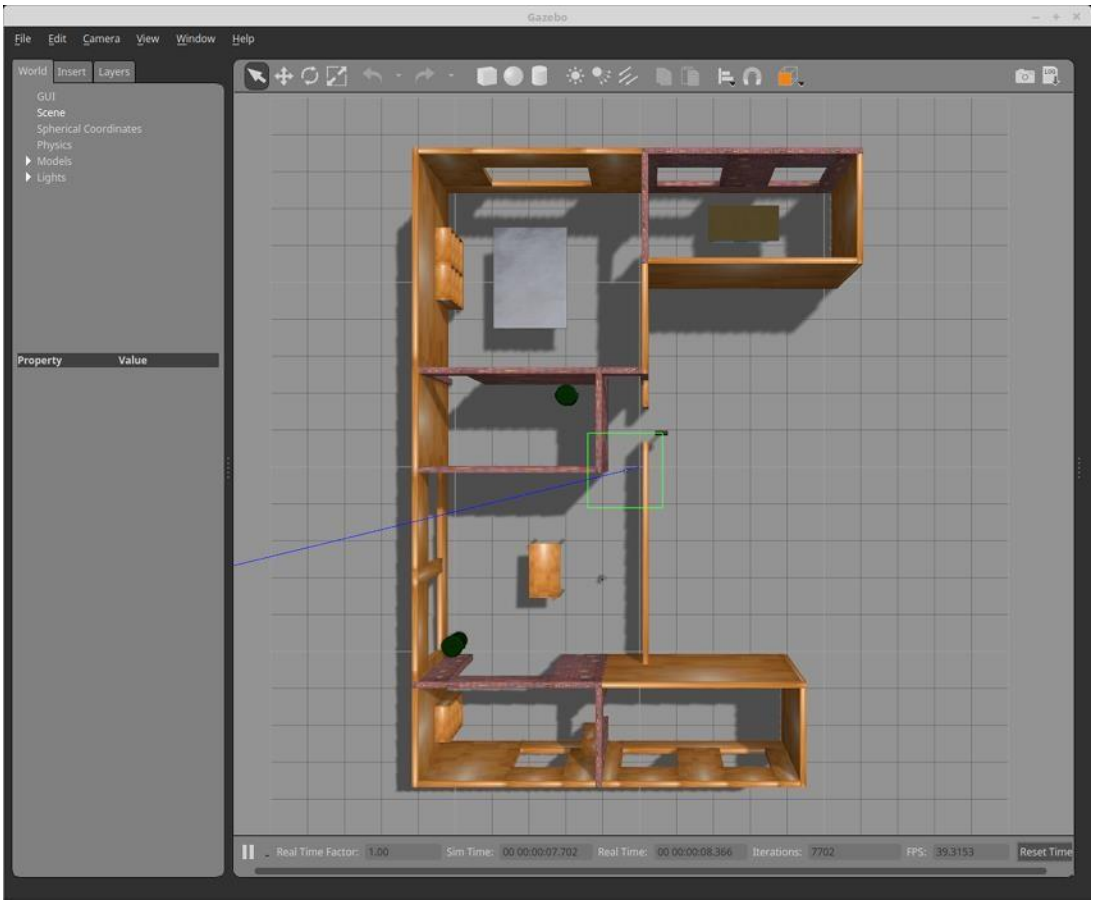
3) About the Turtlebot3 ROS Package and Gazebo Simulator

- TurtleBot is a ROS standard platform robot. TurtleBot3 is a small, affordable, programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping.

- The goal of TurtleBot3 is to dramatically reduce the size of the platform and lower the price without having to sacrifice its functionality and quality, while at the same time offering expandability.

- The TurtleBot3's core technology is SLAM, Navigation and Manipulation, making it suitable for home service robots. The TurtleBot can run SLAM (simultaneous localization and mapping) algorithms to build a map and can drive around your room.

- It offers a Gazebo simulator with a variety of pre-build environments and a capability to customize more and load different versions of turtlebot3 robot.



- https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/
- https://emanual.robotis.com/docs/en/platform/turtlebot3/slam/

4) SLAM (gmapping)

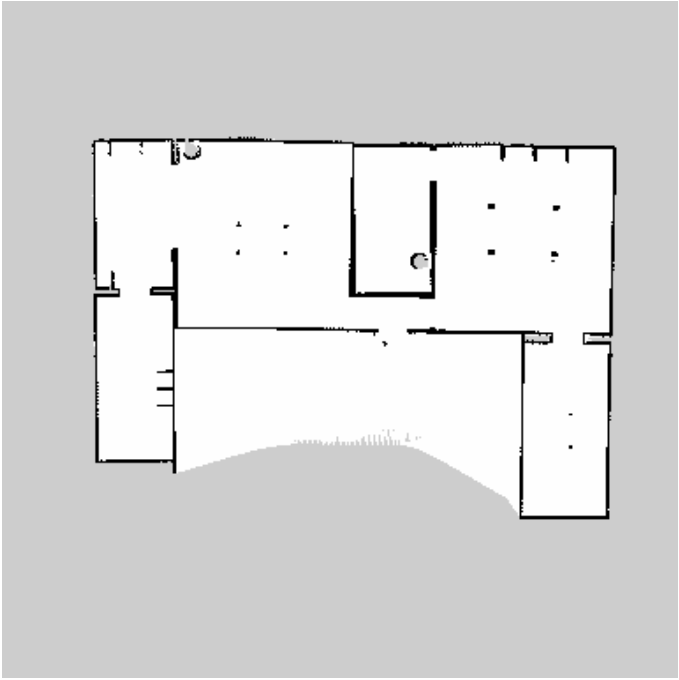
Simultaneous localization and mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. The slam gmapping node takes in sensor_msgs/LaserScan messages and builds a map

- roslaunch turtlebot3_gazebo turtlebot3_house.launch
  - This command loads the gazebo house environment.

- roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
  - This command launches the slam gmapping node for creating 2D map.
  - "Gmapping" methodology which is a laser-based SLAM algorithm that generates a 2D image map.

- roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch. Using this command, we can move the robot around to create the 2D map.

  - rosrun map_server map_saver -f ~/tb3_house_map
  - https://www.youtube.com/watch?v=KP_FbT5kZKc
  - Once a map is built, we need to save it.
  - map_server is a package for publishing and manipulating maps.
  - This command generates two files; tb3_house_map.pgm which is the map image, and tb3_house_map.yaml which contains the metadata about the map; including the location of the image file, its origin, resolution, occupied cells threshold, and free cells threshold.
  - However, when it was run, the image did not turn out to be good. Hence the below image has been referenced from; https://github.com/aniskoubaa/ros_course_part2/tree/master/src/topic03_map_navigation



house map (SLAM generated image)

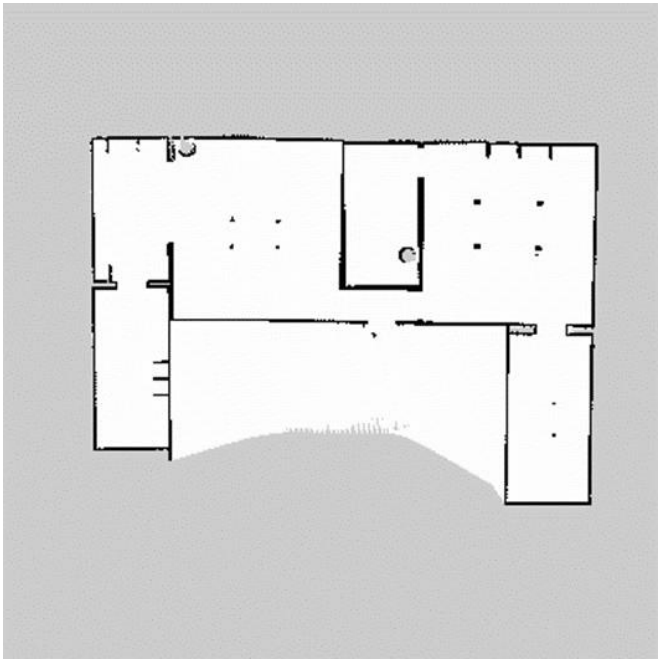
- Properties of the SLAM image :-
  - The image has black, grey, and white pixels.
  - The image is of shape 384x384 pixels and has 3 channels.
  - Value of black pixel = 254
  - Value of grey pixel = 205
  - Value of white pixel = 0

- rosrun eas561_50351140 House_Properties_old.py
  - Load the SLAM image and learn about its properties.

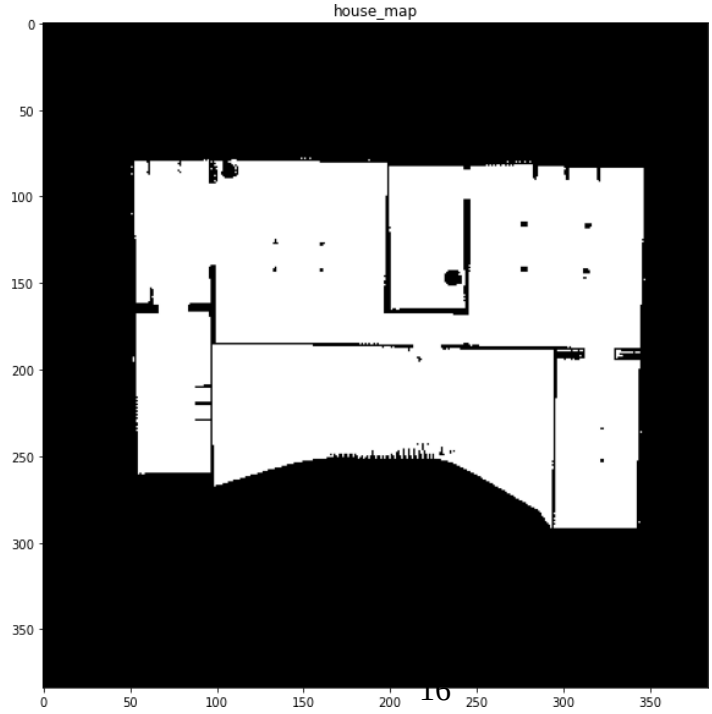
5) Occupancy Grid Formation

From occupancy grid image we form a matrix which has only 0's and 1's. 0's are free spaces, and 1's are obstacles.

1. Load the house map generated from performing SLAM.



2. Replace the grey pixels (unexplored area) to black pixels (obstacles)

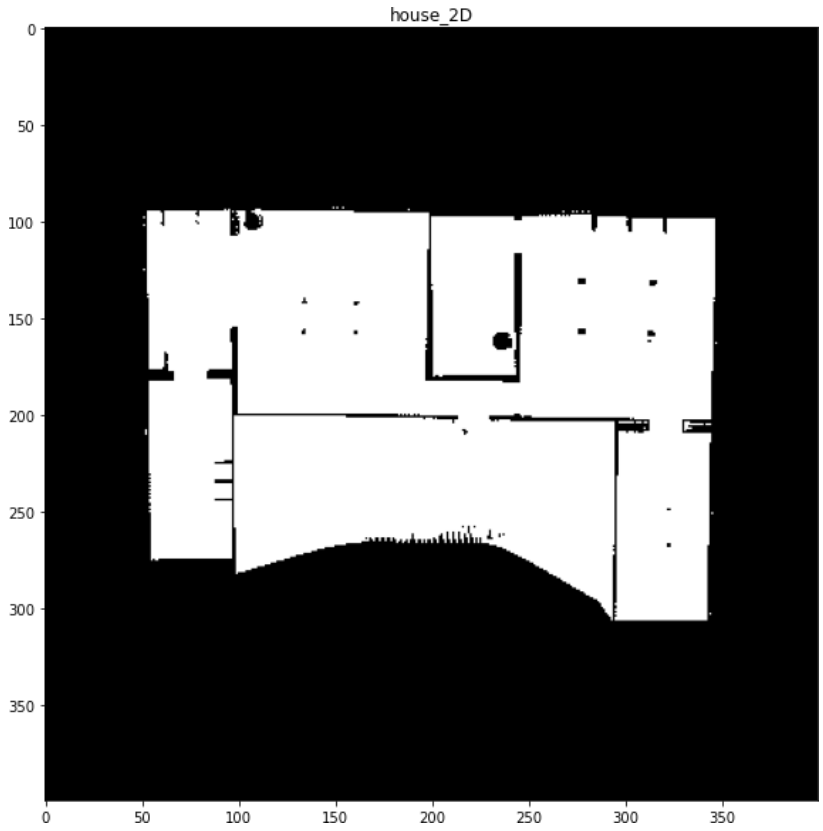


3. The above image is a 3-channel image. We reduce it to a monochromatic image. This reduces the image size for processing.

4. We see that the shape is (384, 384). But we want it to be (400, 400) for improving accuracy. So, we add borders.

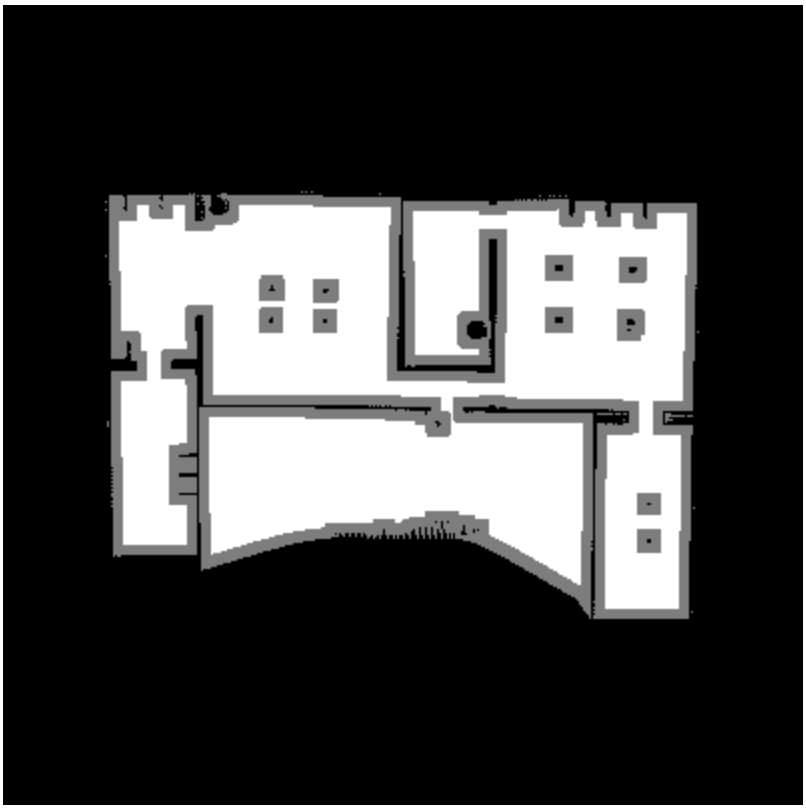
5. Top border = 15 rows of black pixels
   Bottom border = 1 row of black pixels
   Right vertical border = 16 columns of black pixels

    - Thus, creating a (400, 400) shape occupancy grid that can accurately resemble (20m, 20m) shape gazebo environment.
    - The below image has a shape of 400x400 pixels.



6. We will have to add margin besides obstacles/walls otherwise the robot will crash/bang with them because of its own dimensions. Robot's dimensions are 30cm X 30cm in gazebo. 1 pixel in occupancy grid corresponds to 5cm X 5cm. So, we consider the 5 pixels margin from every obstacle.

    - In "house_2D" since we considered "0" as obstacle and "255" as empty, for margin we assign the value "125". For this we created a new image called "PathPlanning_Map" to modify the "house_2D" image.



- The above image has a shape 400x400 pixels.

- There are grey, white, and black pixels having values 125, 0, and 255 respectively.

- To create an occupancy grid out of this image. We create an empty python list and append the pixels values one by one into the list. First converting it into python integer before appending. Because for storing it later into a .json file, it cannot be NumPy integer.

- Black, grey, and white pixels have been assigned values 2, 1, and 0 respectively into the list.

- Now finally we save the list as an occupancy_grid.json file to use it later anytime.

  The occupancy_grid.json file can be called later anytime and converted into NumPy array and reshaped to 400x400 representing the gazebo house environment.

7. roslaunch eas561_50351140 Occupancy_Grid.launch

   - This launch file executes Occupancy_Grid.py file and performs the entire occupancy grid formation process as explained above.

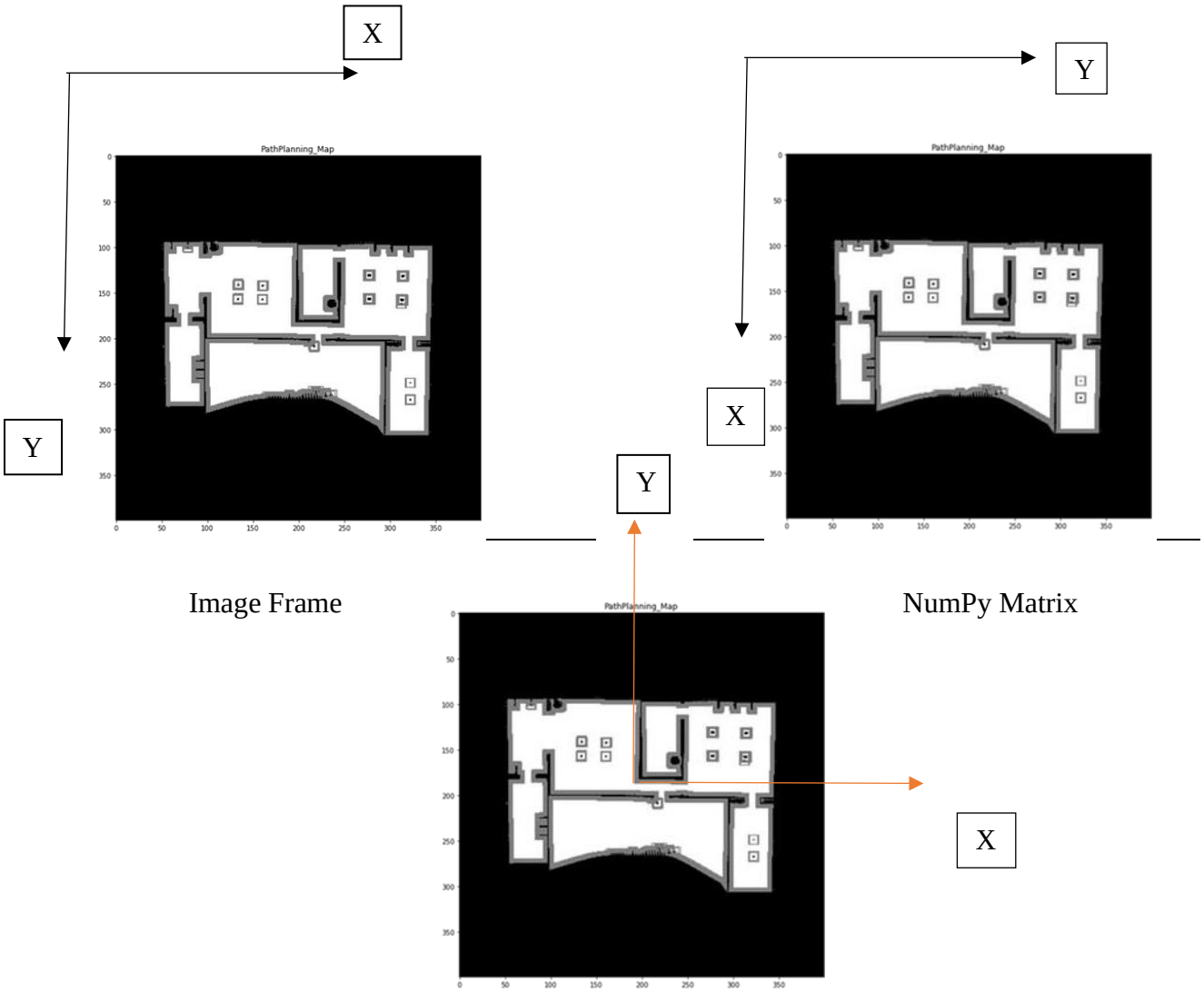
6) Coordinate Relations

- We remember that the start and goal location that we assign for path planning are always with respect to world coordinate frame.

- Hence, we need a mathematical relation from world coordinate frame (20x20 m) of reference to occupancy grid (400x400 pixels) of reference.

- The path planned returns the path in the occupancy grid cells locations. But our robot must traverse in the real world. Thus, we again need an opposite mathematical relation from occupancy grid (400x400 pixels) of reference to world coordinate frame (20x20 m) of reference.

- rosrun eas561_50351140 Coordinate_Relations.py
    - This program performs the calculations to do so.



7) Path Planning using A-star algorithm

- Here, we consider two environments, the 'turtlebot3-gazebo-house' and the 'occupancy-grid-matrix'.

- This piece of code utilizes the coordinate relations program, A* path planning algorithm, and occupancy grid matrix to plan an obstacle free path from start to goal location.

- This program takes start and goal locations, as numeric tuples, and checks if they are not obstacles, using a special cell status function defined. If both locations are not obstacles, then the program will return a planned path in the world coordinate frame.

- But to let this happen there are multiple steps in between. First when the path is returned it is exceptionally long. Say sometimes greater than 300 sub-goal locations on the way.

- This type of path slows down the robot and takes too much time to reach the goal. It is because sometimes two cells can have nearly the same world locations. So, we need to minimize the path, removing repeated locations.

- Next, when the robot needs to move in a straight line, either vertically or horizontally; either the x-coordinate or y-coordinate remains the same and the other one changes. Thus, again stopping the robot multiple times in a straight line reducing its speed.

- To solve this, a program is created to remove multiple in-between sub goal locations.
  Now the robot speeds up.

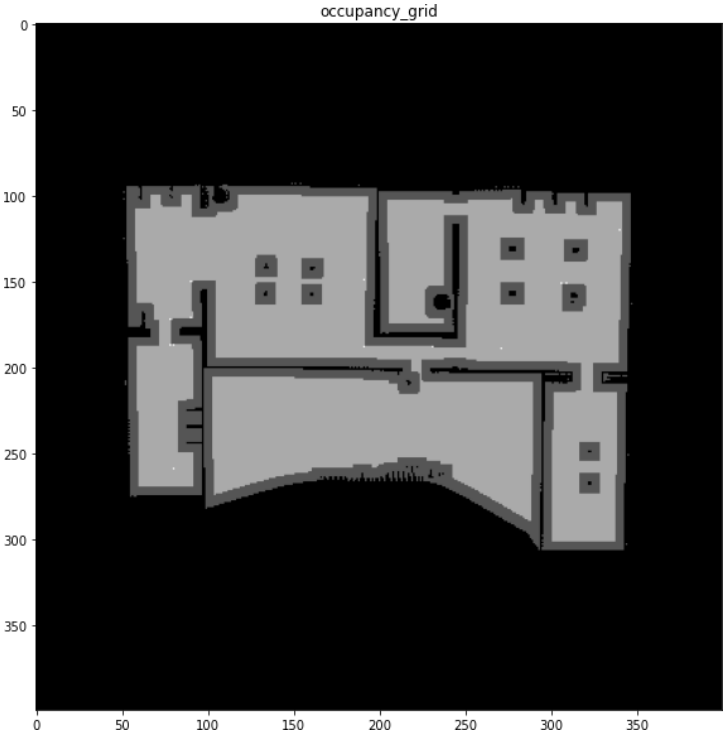
- Next, sometimes the robot follows a step-step (z type) path to its goal. This again wastes time in execution to reach goal. Again, to solve this behavioral complexity we write a program.

- This entire process happens in the path planning program and returns a short path with sub-goal locations as much as possible to improve execution.

- For example, START = (-3, 1) and TARGET = (6, -3).
  - Length of Grid Path = 484
  - Length of Gazebo Path = 13

- Now, once the path is planned. It must be executed. Therefore, we need to write a node/program that can drive the robot to its goal location. We will see later different ways of implementing it.

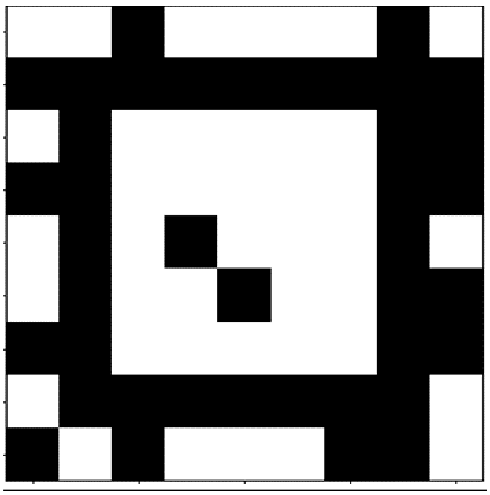
- Path planned in as indicated by white dots.



- Reducing the path complexity, as compared to the one before (notice white pixels = path).

8) ROS AprilTags

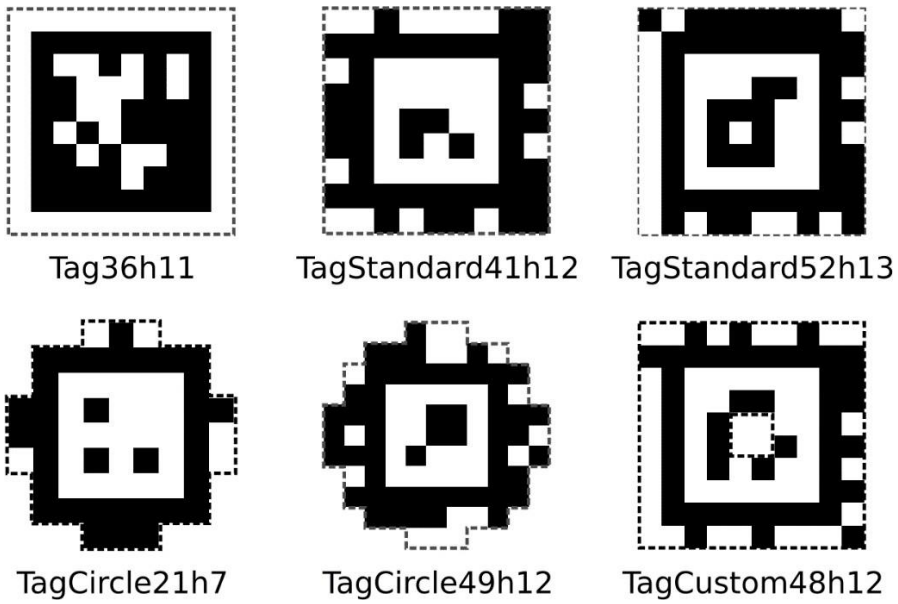
- AprilTag is a visual fiducial system, useful for a wide variety of tasks including augmented reality, robotics, and camera calibration. Targets can be created from an ordinary printer, and the AprilTag detection software computes the precise 3D position, orientation, and identity of the tags relative to the camera.



Tag ID = 1

- There are different families of AprilTags. Technically they are called tag standards. Each AprilTag has a unique ID number. We use AprilTag in this project for pose-estimation/localization.

- In this project TagStandard41h12 is used. Some common one's are,



- The AprilTags from TagStandard41h12 are not directly simulated into gazebo house environment. Before that some experiments have been performed on AprilTags to learn how to work with them and what data do they generate.

- First installed AprilTags ROS package.
  - http://wiki.ros.org/apriltag_ros
  - https://github.com/AprilRobotics/apriltag_ros
  - apriltag folder is the library and apriltag_ros folder is the AprilTags ROS wrapper.

- The behavior of the ROS wrapper is fully defined by the two configuration files config/tags.yaml (which defines the tags and tag bundles to look for) and config/settings.yaml (which configures the core AprilTag 2 algorithm itself). Then, the following topics are output:

  - /tf: relative pose between the camera frame and each detected tag's or tag bundle's frame (specified in tags.yaml) using tf. Published only if publish_tf: true in config/settings.yaml.

  - /tag_detections: the same information as provided by the /tf topic but as a custom message carrying the tag ID(s), size(s) and geometry_msgs/PoseWithCovarianceStamped pose information (where plural applies for tag bundles). This is always published.

  - /tag_detections_image: the same image as input by /camera/image_rect but with detected tags highlighted. Published only if publish_tag_detections_image==true in launch/continuous_detection.launch.

- In this project, we rely more on /tag_detections topic. For tag ID and pose relative to camera frame.

- The main idea is to fill out config/tags.yaml with the standalone tags and tag bundles which we would like to detect (bundles potentially require a calibration process) and config/settings.yaml with the wrapper and AprilTag 2 core parameters. Then, we simply run the continuous or single image detector.

  - http://wiki.ros.org/apriltag_ros/Tutorials
  - http://wiki.ros.org/apriltag_ros/Tutorials/Detection%20in%20a%20video
  - In short, we need to fill out the settings.yaml file in the apriltag_ros package. Specifically, we need to set the tag standard of interest.
  - In the tags.yaml for detecting standalone tags, we need to define the ID, size, and name for each tag.

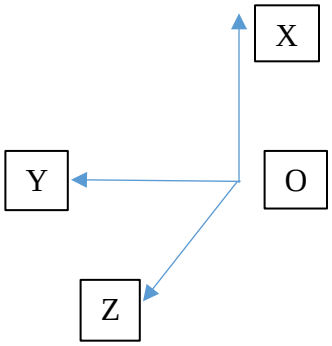
- The tags of interest can be downloaded from,
  - https://github.com/AprilRobotics/apriltag-imgs

- The ID of each tag image is in the name of the image downloaded from above.

- However, to detect AprilTag ID, useful information, and if the AprilTags were to be detected from a camera, the size of the AprilTags must be specified in the tags.yaml file. To do this the following guide can be used,

  - https://github.com/AprilRobotics/apriltag/wiki/AprilTag-User-Guide#pose-estimation

- The coordinate system of each AprilTag seems to be wrong in the guide link just above. The correct system from experimentation is as follows,



- Now first I try to detect AprilTag live with laptop's usb camera.

  - For that I need to calibrate the laptop's usb cam. So, I installed calibration package of OpenCV "usb_cam" in ROS.

    - sudo apt-get install ros-noetic-usb-cam

  - I downloaded 10 AprilTag images, of size 9x9 total 81 pixels. Very tiny couldn't be seen properly, so I visualized using matplotlib and saved the bigger image in AprilTags directory.

  - Wrote a code to detect AprilTags (taken from references put above) and got some results e.g., 'ID', etc. Using that data, added it into config/tags.yaml.

  - To launch ROS AprilTags detection node we need to run,
    - roslaunch apriltag_ros continuous_detections.launch

  - A few changes to be done are, to change;
    - "camera_name" to "/usb_cam" and default "image_topic" to "image_raw".

  - Because the usb_cam ROS package when run, web_cam opens it creates the above

topics for "camera_name" and "image_topic".

- Next is to create a file camera_head.yaml into /home/”your_user_name”/.ros /camera_info/ containing usb camera intrinsics.

- To calculate camera intrinsics, a chessboard pattern of 9x6 is used. And the following command was used,

    - rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.024 image:=/usb_cam/image_raw camera:=/usb_cam

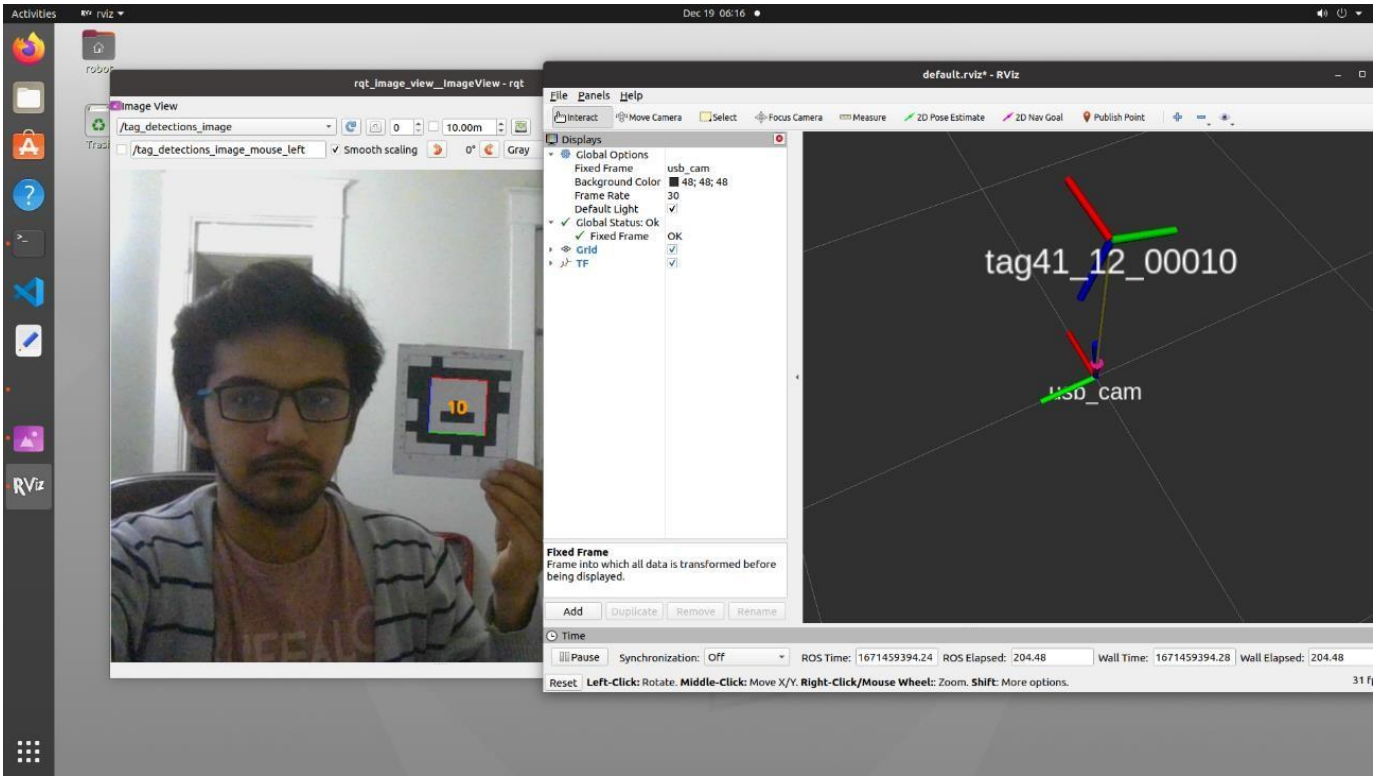
    - It generates camera, distortion, rectification, and projection matrix.

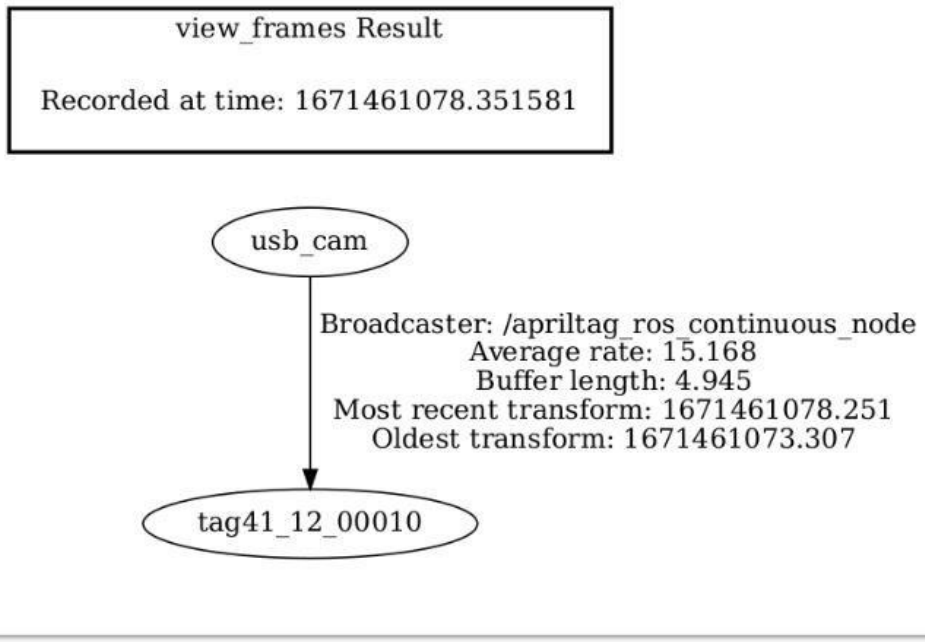
    - These all data is then passed into the /home/”your_user_name”/.ros/camera_info/camera_head.yaml file.

- Now finally upon running the below file, which launches apriltag_ros continuous detection node, RVIZ, usb_cam, ATdata.py file that prints out the necessary detection data on screen.

    - roslaunch eas561_50351140 apriltagWebcam.launch

- The all above methodology was learnt from the videos below,
    - https://youtu.be/UGArg1kQwFc
    - https://youtu.be/MrKYawVvHzE
    - https://youtu.be/gQeVc2JJ1Ag
    - https://youtu.be/UYVRbSBtZfU


Activities
rviz
Dec 19 06:16
rqt_image_view__ImageView - rqt
Image View
/tag_detections_image
/tag_detections_image_mouse_left
Smooth scaling
Gray
default.rviz* - RViz
File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
usb_cam
Background Color
48; 48; 48
Frame Rate
30
Default Light
Global Status: Ok
Fixed Frame
OK
Grid
TF
tag41_12_00010
usb_cam
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Duplicate
Remove
Rename
Time
Pause
Synchronization:
Off
ROS Time:
1671459394.24
ROS Elapsed:
204.48
Wall Time:
1671459394.28
Wall Elapsed:
204.48
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.


view_frames Result
Recorded at time: 1671461078.351581
usb_cam
Broadcaster: /apriltag_ros_continuous_node
Average rate: 15.168
Buffer length: 4.945
Most recent transform: 1671461078.251
Oldest transform: 1671461073.307
tag41_12_00010

9) AprilTags Detection and Simulation into Gazebo

- In short, we can draw/create a shape/object/structure into gazebo and load it as a 2D/3D model with a fixed defined size.

- Here we are supposed to insert the AprilTags images into gazebo, to be detectable by the robot's camera.

- Now each of such model has many files that define it in some of its folder. One of such is an SDF (Simulation Description Format) file. This file defines the size and shape of the object/model and its pose in the gazebo world.

  Reference,
  - https://classic.gazebosim.org/tutorials?tut=build_world#PositionModels
  - http://classic.gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros

- To create a folder that contains necessary files that define it and load it in gazebo, the following reference is used,

  - https://github.com/koide3/gazebo_apriltag

  - The above resource downloads a data folder/package that contains a file "generate.py". This file creates a necessary folder containing important files for simulation in gazebo.

- For this project, 201 AprilTag images from 1-201 have been passed into the "generate.py" file.

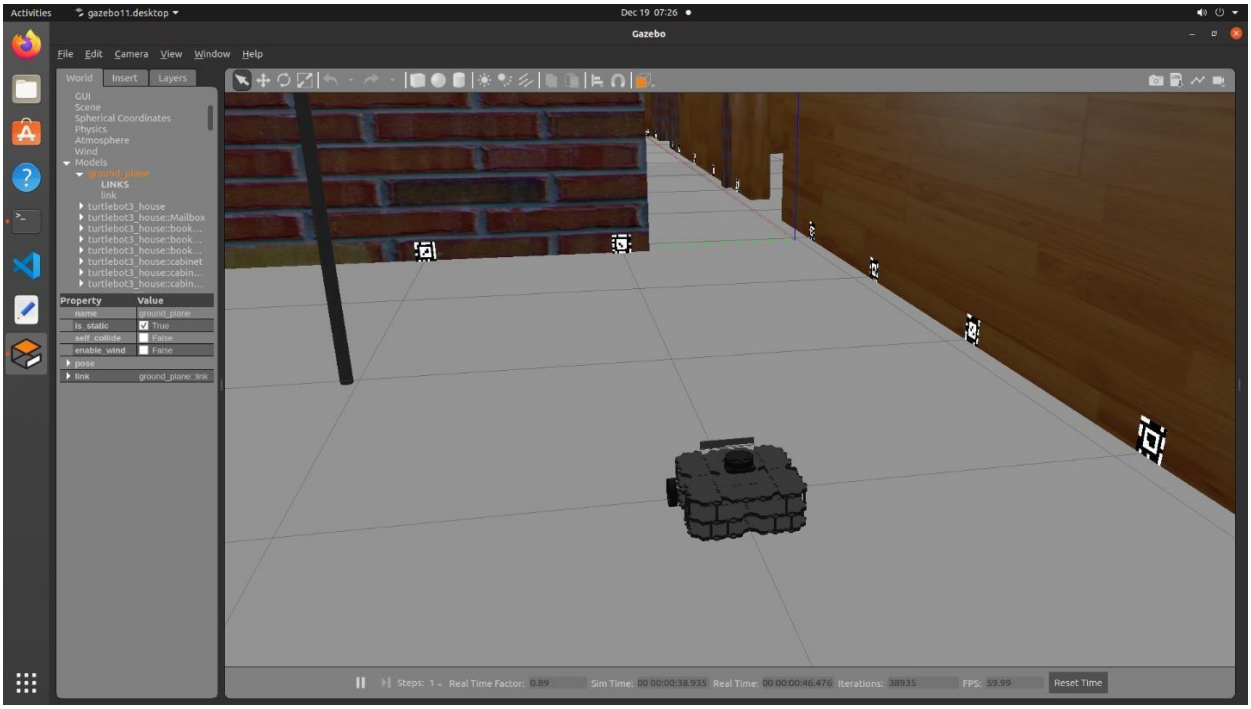
- Once these folders are created, these need to be copied to /home/"your_user_name"/.gazebo/models folder. Basically, the ROS gazebo loads any models from this directory that have been defined in whatever .world file is launched.

- The pose and size of the models need to be defined in the .sdf file in the .gazebo/models directory. Then these .sdf files need to be called in the .world file of interest. This process will load as many objects as possible into the gazebo environment.

- Open apriltag_ros/continuous_detection.launch and change,
  - "camera_name" to "/camera/rgb"

- Now we will create a new launch file, to launch the gazebo house with AprilTags. We use the existing turtlebot3_house.world file format and data and create a new tb3GazeboHouse.world file by adding 201 AprilTags .sdf data. Then we created a new tb3GazehoHouse.launch file and, called the tb3GazeboHouse.world file.

- Launching the tb3GazehoHouse.launch loads gazebo house with AprilTags.
  - roslaunch eas561_50351140 tb3GazehoHouse.launch


Activities
gazebo11.desktop
Dec 19 07:26
Gazebo
File Edit Camera View Window Help
World Insert Layers
GUI
Scene
Spherical Coordinates
Physics
Atmosphere
Wind
Models
ground_plane
LINKS
link
turtlebot3_house
turtlebot3_house::Mailbox
turtlebot3_house::cabinet
Property Value
name ground_plane
is_static True
self_collide False
enable_wind False
pose
link ground_plane::link
Steps: 1
Real Time Factor: 0.89
Sim Time: 00 00:00:38.935
Real Time: 00 00:00:46.476
Iterations: 38935
FPS: 59.99
Reset Time

10) Robot Driver (localization from odometry)

- The file ODPE_robotDriver.py subscribes to the pose of the robot from subscribing to the /odom topic. And utilizes necessary functions and algorithms from formulas_algorithms.py file to drive the robot from start to the target location.

- Run;
  - roslaunch eas561_50351140 tb3GazeboHouse.launch
  - roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
  - rqt_image_view (for seeing what the robot is seeing)
  - rosrun eas561_50351140 robotDriver_ODPE.py

- Finally, all the above nodes can be run at once using the launch file,
  - roslaunch eas561_50351140 robotDriver_ODPE.launch

11) Robot Driver (localization from ArpilTag detection)

- The pose of the robot x,y,z coordinates and roll, pitch, yaw angles can be calculated if we have the transformation from odom frame to robot's base_footprint.

- The transformation from robot's camera frame to each AprilTag is subscribed from /tag_detections topic, which also gives tag ID.

- When RVIZ is launched, it launches the URDF model of our turtlebot3 robot. Which gives us the transformations among all the frames within the robot.

- When the tb3GazeboHouse environment is launched, the transformation from odom to each AprilTag is automatically generated.

- Therefore mathematically, knowing the fixed transformation from odom to AprilTag, robot's camera to its base footprint; and continuously detecting transformation from robot's camera to AprilTag, we can easily compute our required transformation, i.e., from odom to robot's base_footprint.

- Thus, we created programs "broadcaster_server.py", simple_listener_client.py, explore_listener_client.py, and water_listener_client.py. All three types of listener client files do almost the same job.

- The simple_listener_client.py is the basic one, it estimates the pose of the robot (localization) and pose of a plant. The explore_listener_client.py file additionally saves the location of a detected plant into plants_data.json file.

- What specifically additional is water_listener_client.py file does will be discussed later. All the listener client type files will randomly choose maximum of 4 AprilTags out of all detected and then estimate the pose of the robot from all detected tags. It will then average out the pose of the robot from all and publish this data to ATPE_Pose topic.

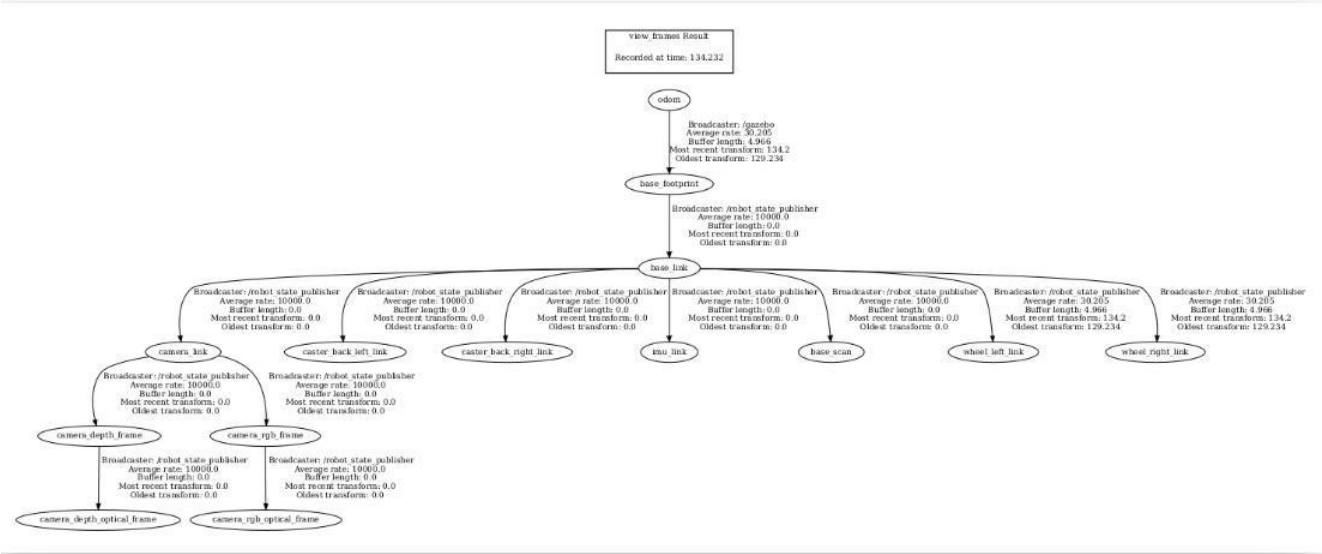
- The broadcaster_server node acts as a tf broadcaster and as an ROS server. This node accepts a request from simple_listener_client/explore_listener_client/water_listener_client node.

- The request is about a detected AprilTag whose TF tree model must be created as shown below. Once this tree is created either of the listener client type nodes listens to the transformation from world to the robot's base footprint. This must be done because the communication must be synchronous and cannot be asynchronous just by using publisher/subscriber/topics model.

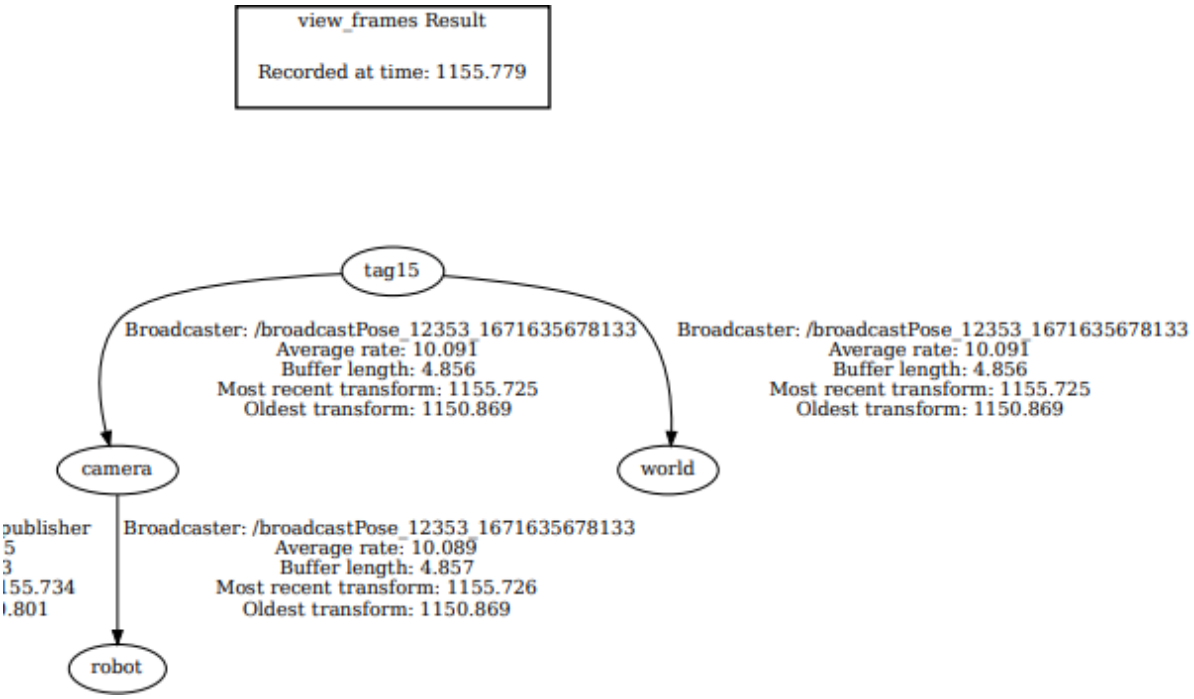
- To only perform pose estimation, we load gazebo house, broadcaster_server, and simple_listener_client node, RVIZ, teleop node all using one launch file,
    - roslaunch eas561_50351140 poseEstimation.launch

- Earlier we used to drive the robot to target location while fetching its pose from /odom topic. Now we fetch the pose from /ATPE_Pose topic. And with success we are now able to localize the robot only from detecting AprilTags, estimating the pose and driving it to target location.

- Thus, we run the below nodes.
    - roslaunch eas561_50351140 tb3GazeboHouse.launch
    - roslaunch apriltag_ros continuous_detection.launch
    - roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
    - rqt_image_view (for seeing what the robot is seeing)
    - rosrun eas561_50351140 broadcaster_server.py
    - rosrun eas561_50351140 simple_listener_client.py
    - rosrun eas561_50351140 robotDriver_ATPE.py

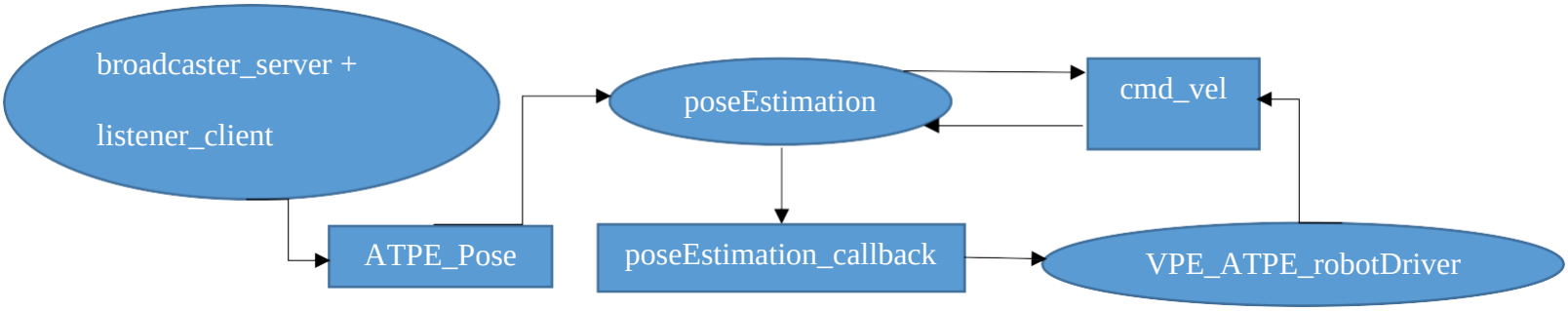


Turtlebot3 Robot frames



Turtlebot3 Pose Estimation Model

12) Robot Driver (estimating pose also through its own velocity)



- Technically AprilTags are and should be used to improve the localization of a robot in this case. A robot relying only upon AprilTags for its position is not a good practice. It is because, doing so would require hundreds of AprilTags to be put in the world (201 in numbers as done earlier) and its data (pose) to be logged into robot's memory. Doing so would be a waste of time and resources.

- Despite doing so if a robot doesn't find an AprilTag then it will not know where it is and will be called a trapped robot (usually happens when robot is in a corner). This is a highly undesirable case. Thus, there should be some other means of robot localization, and one of that is to enable the robot to estimate its position using its own velocity (both linear and angular).

- So, the file poseEstimation.py and best_poseEstimation.py (both almost the same) consider the linear and angular velocity of the robot, its previous position and through trigonometry it estimates the current position of the robot.

- This way the robot can localize itself anywhere without having to rely upon AprilTags. But practically localization from velocity can give us about 70-90% accuracy as found after performing it.

- Therefore, there should be at least some AprilTags randomly put around in the world environment, so that if an AprilTag is detected then pose estimate it derived from AprilTag detection and thus robot will update/improve its pose.

- We can see this by running robotDriver_VPE_ATPE.py file, which is just like the other robot driver nodes, but this node relies on pose of the robot from poseEstimation.py/best_poseEstimation.py node (both almost same).

- poseEstimation.py or best_poseEstimation.py (both almost same) should be used for robot localization. An exception to our localization model is that the localization is not initiated until the robot detects an AprilTag. If that's the case, then an improvement to our poseEstimation node(s) is that then the robot rotates 360 degrees expecting that it will see an AprilTag and that will thus initialize localization. Until then our robotDriver_VPE_ATPE.py will wait.

- Now the only difference between poseEstimation.py and best_poseEstimation.py is that in the latter one unlike the former one the trigonometric calculation in main function and ATPE_Pose_callback function run one at a time. That is, the ATPE_Pose_callback function is not allowed to be executed until the calculation in the main function has been completely executed. Therefore, it is better to run best_poseEstimation.py.

13) Plant Pose Estimation

- Pose estimation of a plant is also performed and we use AprilTag to detect if it is a plant. The files simple_listener_client.py, explore_listener_client.py and water_listener_client.py need to be given what AprilTag ids are plants.

- Once we have the pose of robot in world frame (world to robot transformation) from estimation, then we listen to the transformation from robot to plant (AprilTag).

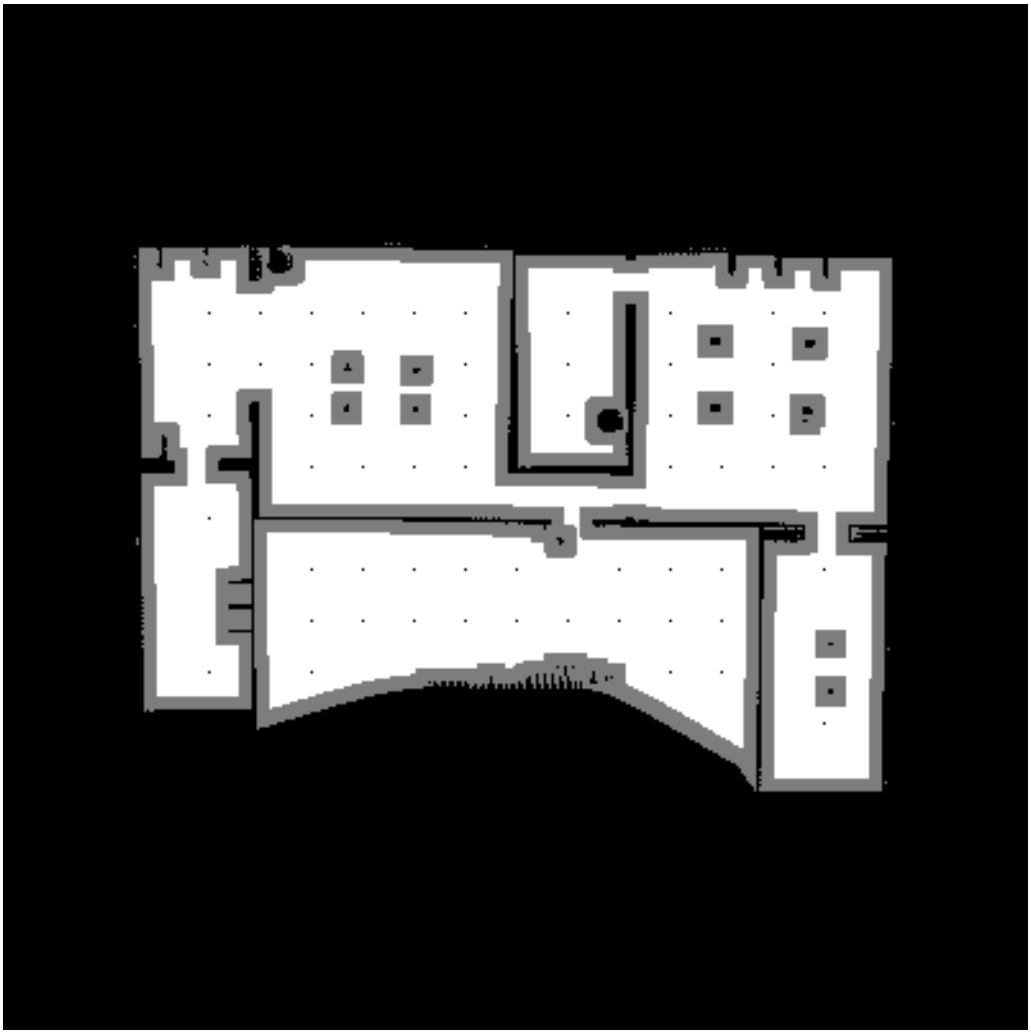
- Now taking the matrix multiplication of world_to_robot and robot_to_plant will give us the transformation from world to plant. Multiplying this computation with a zero vector will give us the coordinates of the plant in the world frame.

14) World Exploration and Plant Finding

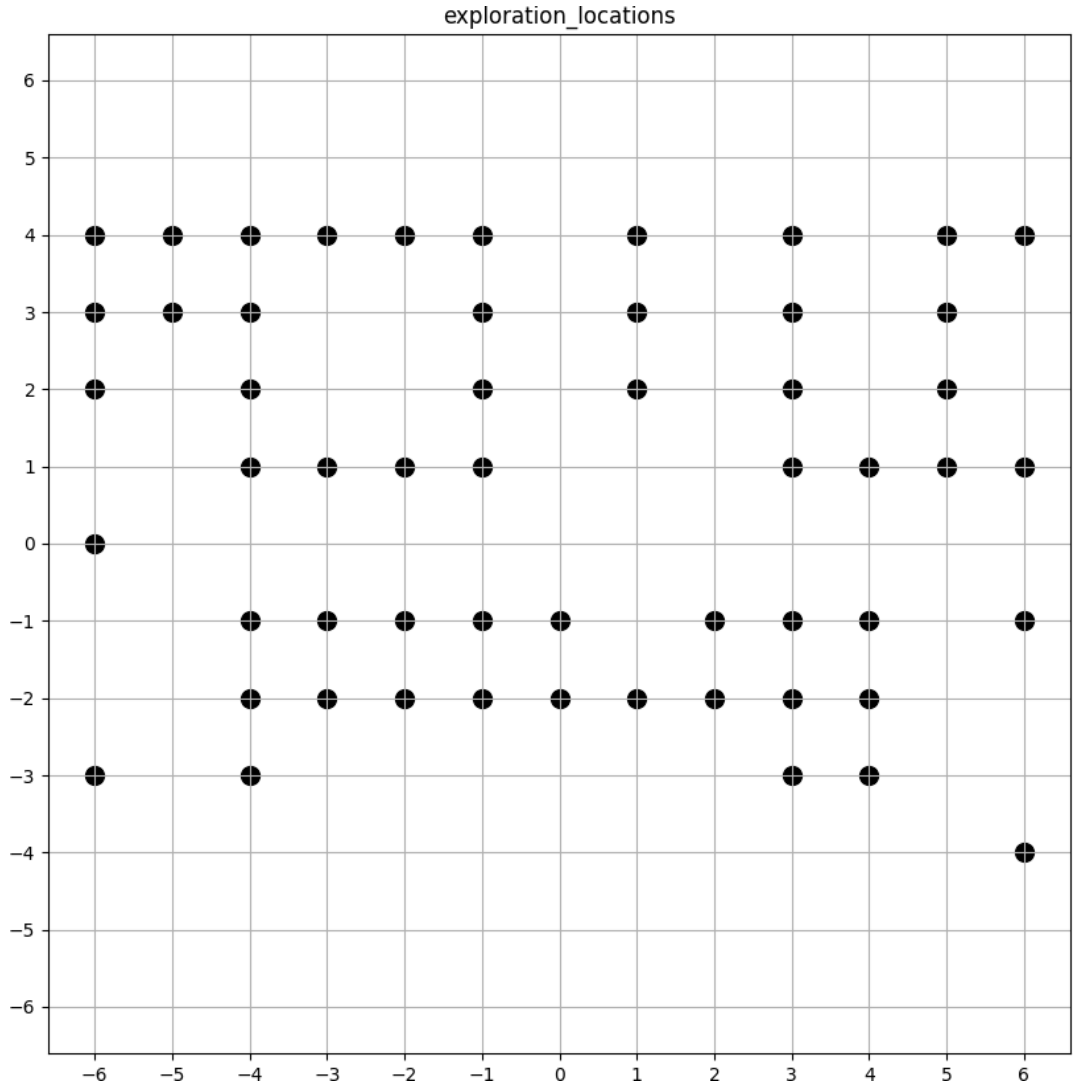
- Now the robot should explore the environment and look parallel for plants so that it can save their position offline into a plants_data.json file.

- The robot should automatically create points in the environment and travel to all those points in a way so that it will have the environment fully explored.

- This has been done programmatically by create_exploration_points.py file.



- Black dots represent exploration points.

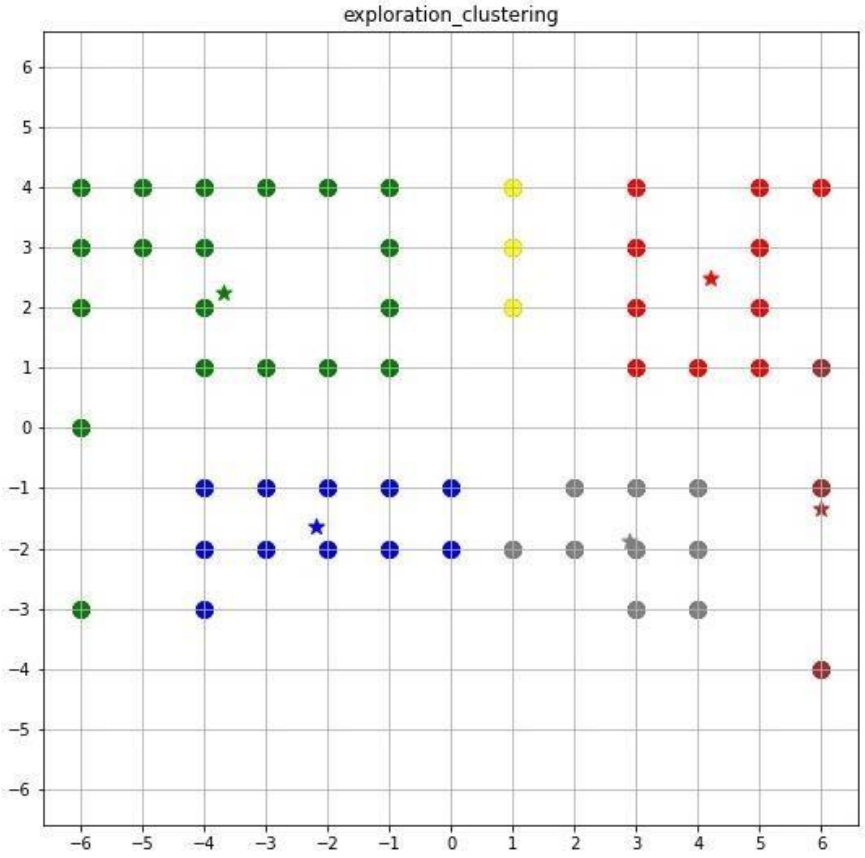


- To explore all the locations there must be a way planned so that the robot travels around the world environment taking the shortest possible route and time.

15) K-Means Clustering for Exploration

- To do this we have simplified our exploration and planned it area-wise. For that kmeans clustering algorithm has been implemented to cluster the points into 6 distinct clusters.

- The kmeans clustering algorithm has been devised from scratch to implement A* planning as a distance metric rather than L1 or L2 norm which cannot be used because we have walls and obstacles.



- To implement it, run kmeans_exploration.py. Then the clustering is done so, two files are created, centroids.json and clusters.json. These two files are used in Exploration_robotDriver.py file to plan a short way for the robot to visit each cluster one by one thus exploring the environment.

16) Robot Watering the Plants

- Now the robot loads the plants_data.json file that was created while exploring and sorts all the plants in a manner that it takes the shortest route and time to reach out to all the plants and water them. To run Watering_robotDriver_VPE_ATPE.py file to make the robot drive to all plants and water them.

17) ROS Nodes, Topics, and Services

- The most basic example of running this project demonstration is for a robot to begin localization first, then have a target location where it will plan its path and drive towards it to reach.

- In this process there are several ROS nodes, Topics, and Services initiated.

- ROS Nodes;

    - / robotDriver_ VPE_ATPE_22366_1683673895916
    - /apriltag_ros_continuous_node
    - /best_poseEstimation_20305_1683673841403
    - /broadcaster_server_19556_1683673810461
    - /gazebo
    - /gazebo_gui
    - /robot_state_publisher
    - /rosout
    - /rqt_gui_cpp_node_19463
    - /rviz
    - /water_listener_client_19677_1683673825547

- ROS Topics of interest;
    - /ATPE_Pose
    - /camera/rgb/image_raw
    - /cmd_vel
    - /odom
    - /poseEstimation
    - /rosout
    - /rosout_agg
    - /scan
    - /tag_detections
    - /tag_detections_image
    - /tf
    - /tf_static

- ROS Services of intereset;
    - /broadcast_apriltag_service
    - /gazebo/reset_world

18) Conclusion and Summary

1) The objective of this project was to develop an autonomous plant watering robot using ROS and Gazebo.
2) The scope and task of the robot was to explore and map an unknown (specifically household) environment.
3) This involved SLAM, that was carried out through SLAM gmapping built-in package.
4) The implementation began with creating an occupancy grid out of processing the SLAM generated image.
5) Then after calculating mathematical conversions between the world environment and occupancy grid, path planning through A* algorithm was implemented.
6) To drive the robot, the driver node was created.
7) Localization was first solved AprilTag detection and eventually from using the robot's own velocity to calculate and estimate. Finally integrating both ways to create a final poseEstimation concept.
8) Well, soon after AprilTag detection and pose estimation, through programming the pose of an AprilTag is also estimated.
9) Now again through programming we came up with exploration points in the image for world exploration.
10) Also, the points were grouped through K-Means Clustering algorithm to distinguish different areas in the world.
11) Then the robot was driven to reach to all the points so that it will have the environment explored and, in the meantime, save the locations and id of all the plants detected in the way.
12) Finally, now having the locations of all the plants the robot could drive to each one of them to water them.

Thus, we can conclude that, the project had 6 important pillars :-

1) SLAM
2) Path Planning
3) Localization
4) Driving the Robot
5) World Exploration (part of SLAM, performed separately afterwards)
6) Plant Pose Estimation

19) Scope for Improvement

1) We can also perform SLAM from scratch.

2) Generally, we do have dynamic obstacles in our environment, so our robot can be enabled to avoid dynamic obstacles through obstacle avoidance concept. And Bug algorithms can be implemented to do so.

3) This project has been done on software simulation mode using ROS and Gazebo. This project can be further implemented on hardware.

20) References

- https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/
- https://emanual.robotis.com/docs/en/platform/turtlebot3/slam/
- https://www.youtube.com/watch?v=KP_FbT5kZKc
- ros_course_part2/src/topic03_map_navigation at master · aniskoubaa/ros_course_part2 · GitHub
- http://wiki.ros.org/apriltag_ros
- https://github.com/AprilRobotics/apriltag_ros
- http://wiki.ros.org/apriltag_ros/Tutorials
- http://wiki.ros.org/apriltag_ros/Tutorials/Detection%20in%20a%20video%20stream
- https://github.com/AprilRobotics/apriltag-imgs
- https://github.com/AprilRobotics/apriltag/wiki/AprilTag-User-Guide#pose-estimation
- https://youtu.be/UGArg1kQwFc
- https://youtu.be/MrKYawVvHzE
- https://youtu.be/gQeVc2JJ1Ag
- https://youtu.be/UYVRbSBtZfU
- https://classic.gazebosim.org/tutorials?tut=build_world#PositionModels
- http://classic.gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros
- https://github.com/koide3/gazebo_apriltag