SLAM and Landmark Detection Based on Turtlebot2

Shutong JIN* Ke GUO† Konstantin Akhmadeev‡

Abstract

In this project, a Turtlebot2 with Kinect camera and Hokuyo lidar is used to map the environment, as well as detect ArUco markers and save their coordinates with their mark ID in a 2-dimensional array. Then, on user's request, the TurtleBot will go to one of the markers and is able to travel between them in the requested order. Based on the framework of ROS, the map construction is found out via the gmapping_hokuyo_demo package, which is a integrated package in ROS with a more suitable design to this project based on the Rao-Blackwellized particle filter algorithm. After obtaining the coordinates of the Turtlebot and pose returned from cv::aruco::estimatePoseSingleMarkers, the specific location of the ArUco marker, together with its ID, are save in a 2-dimensional array, on which the future instructions are based. Finally, the movement between different target marker points is accomplished by the ROS navigation framework with the core principle, adaptive Monte Carlo localization.

Keywords: SLAM, Landmark detection, Navigation, Route planning, Pose estimation

1 Introduction

Mobility is an critical prerequisite for social, cultural, and financial improvement and for social participation. It is also becoming affordable for an ever-growing quantity of human beings worldwide. Motorized private transport has become the most important and most frequently used mode of transport, and forecasts indicate that this will continue to be the case in the future. Efficiency and safety are the two topics most concerned by the travel industry. With the rise of deep learning and computer vision technology, autonomous driving provides new solutions for improving traffic safety and efficiency. Autonomous driving integrates multiple technologies such as artificial intelligence, communications, semiconductors, and automobiles. During the last decade, the autonomous driving has been the hottest topic in the industry. Because autonomous driving is extra efficient, more secure

*Wuhan University *Email address: 2017301200232@whu.edu.cn*
†Harbin Institute of Technology at Weihai *Email address: 170600304@stu.hit.edu.cn*
‡The supervisor

and more environmentally friendly compared with the conventional automobile industry.[1] More and more countries put the development of autonomous driving technology at the core of industrial research. In 2018, almost 8,000 patents related to autonomous driving were applied globally, as shown in figure 1(data source: IncoPat).

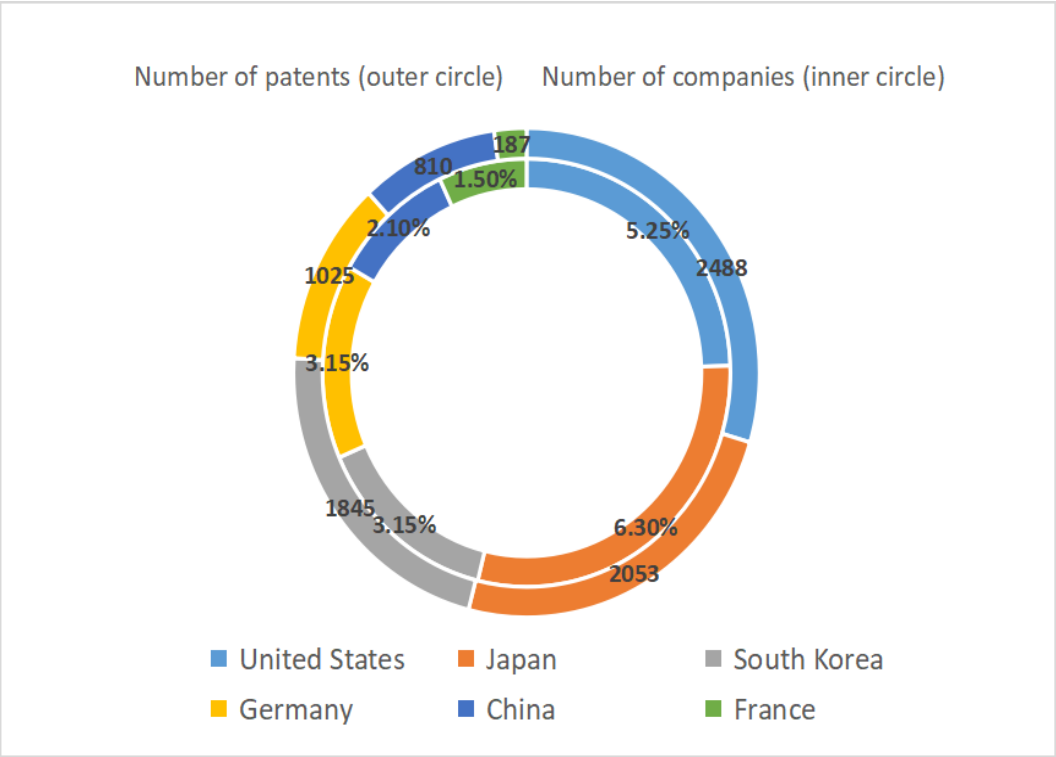

FIGURE 1: The distribution of global autonomous driving patent applications in 2018

Autonomous driving can be simply divided into two elements, the environment recognition and the navigation. In this project, we implement these two modules on the Turtlebot2 to complete a simple autonomous driving simulation, which is obviously expandable and of practical significance.

2 Project Framework

The framework of this project is shown in figure 2:

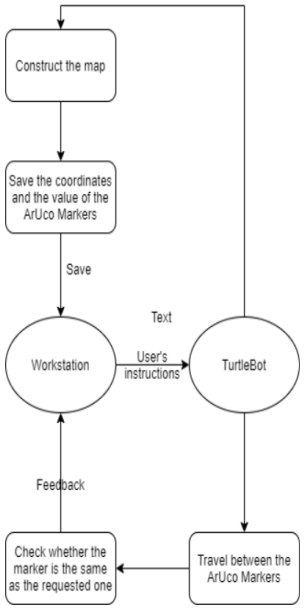

FIGURE 2: framework

In order to permit the robot to move flexibly, we use two computers, one computer is called the workstation, and the other laptop is called the turtlebot, to control the robots based on the router and synchronous communication mechanism . The workstation is responsible for issuing instructions and accepting the returned detection data, as well as for visualization and data analysis. Turtlebot takes charge of executing instructions.

3 ROS and Turtlebot2

3.1 ROS

Robot Operating System (ROS or ros) is an open source robotics middleware suite. Although ROS is not an operating system but a collection of software frameworks for robot software development, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor data, control, state, planning, actuator, and other messages. Despite the importance of reactivity and low latency in robot control, ROS itself is not a real-time OS (RTOS). It is possible, however, to integrate ROS with real-time code. With the rapid development and complexity of the robotics field, the need for code reusability and modularity has become stronger. The emergence of ROS is to comply with this demand and has a wide range of applications in industrial robots and intelligent robots. Also because of those characteristics like modular design, open source and multiple programming language support of ROS, we choose it to develop our projects.

3.2 Turtlebot2

Turtlebot is a low-cost, personal robot kit with open source software. Turtlebot was created at Willow Garage by Melonee Wise and Tully Foote in November 2010. It is because of the scalability and the ability to add external components of Turtlebot, which give the experiment a variety of possibilities, so we chose turtlebot. Turtlebot2 consists of an YUJIN Kobuki base, a 2200 mAh battery pack, a Kinect sensor, an Asus 1215N laptop with a dual core processor, fast charger, and a hardware mounting kit attaching everything together and adding future sensors. Assembling the kit is quick and easy using a single allen wrench (included in the kit). Additionally TurtleBot 2 comes with a fast charging dock that TurtleBot can autonomously dock with, enabling continuous operation. Our Turtlebot2 is shown in the figure 3, with Hokuyo lidar on the top in order to construct the map and a Kinect RGBD camera in the middle to detect the environment, especially mark detection.

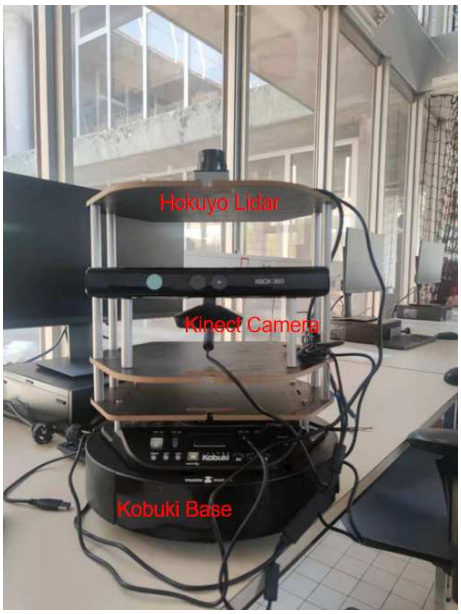

FIGURE 3: The Turtlebot2

4 SLAM(Simultaneous Localization and Mapping)

4.1 SLAM

SLAM or simultaneous localization and mapping is a challenging problem in mobile robotics under the area of artificial intelligence that has been taken widely under study for more than two decades where scientists use different techniques to improve autonomy and self-exploration of robot navigation.[3] It is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of a robot's location within it. While this initially appears to be a chicken-and-egg problem. Benefit from the extensive application of ROS and open source framework, there already exists some built-in packages to solve this problem. Table 1 comparatively shows the different

features of different packages. Considering the size of our laboratory and the hardware of our computer, gmappingSLAM is most suitable to our project.

TABLE 1: Different packages related in SLAM on ROS

	Merits	Demerits	Principle
HectorSLAM	No need for odometer,high accuracy	Need a laser scanner with high update frequency and low measurement noise	Scan-matching (Gaussian-Newton equation)
KartoSLAM	Have a better performance when drawing in a larger environment	Require a large memory to store nodes	Spare Pose Adjustment(SPA)
3D_SLAM	Build a 3D map and get more information	Need a 3D lidar and better cpu to support a lot of calculations	Point-to-Plane (Iterative Closest Point)
GmappingSLAM	high accuracy in a small environment, low requirement for the scanning frequency of the lidar	Need a lot of particle simulations to get good results	Rao-Blackwellized particle filter

4.2 Gmapping

When constructing the map, we use the *.launch* file called *hokuyo_gmapping_demo.launch*, which mainly calls this ROS package *slam_gmapping*. This package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called *slam_gmapping*. By using *slam_gmapping*, we can create a 2-D occupancy grid map (like a building floorplan) from laser and pose data collected by a mobile robot.

Slam_gmapping uses an algorithm based on RBPF (Rao-Blackwellized particle filter) to solve the SLAM problem. A complete SLAM problem is to estimate the robot pose and map at the same time with given sensor data. However,matching the robot with the map is the premise of getting an accurate pose. And If we need to get a good map, we need an accurate pose to do it. Obviously, this is a contradictory problem like Chicken-and-egg problem. Explain it in a mathematical way, the SLAM problem can be expressed as the following form:

$$p(x_{1:t}, m|u_{1:t}, z_{1:t})$$

where $z_{1:t}$ is a series of sensor measurement data from bringing up the robot to t, $u_{1:t}$ is a series of control data (in ROS typically considered as odom) , and the map information m and the robot trajectory state $x_{1:t}$ are what we want to estimate.

According to probability theory, the above formula can be simplified to:

$$p(x_{1:t}|u_{1:t}, z_{1:t})p(m|x_{1:t}, z_{1:t})$$

In this way, the SLAM problem is reduced to two problems. Among them, the map construction of the known robot pose is a simple problem, so the estimation of the robot pose is a key issue.

In *slam_gmapping*, it uses particle filtering to estimate the pose of the robot and build a map for each particle. Therefore, each particle contains the robot's trajectory and the corresponding environment map. Here we use Bayesian formula to simplify the calculation of pose. (The specific derivation is quite complicated, here only explain the results)

$$p(x_{1:t}|u_{1:t}, z_{1:t}) = \eta p(z_t|x_t)p(x_t|x_{t-1}, u_t)p(x_{1:t-1}|u_{1:t-1}, z_{1:t-1})$$

where using $p(x_{1:t-1}|u_{1:t-1}, z_{1:t-1})$ to represent the robot trajectory at the previous moment, $p(x_t|x_{t-1}, u_t)$ is the kinematic model for propagation so that we can obtain the predicted trajectory of each particle. For each particle after propagation, the observation model $p(z_t|x_t)$ is used for weight calculation and normalization, so that the robot trajectory at that moment is obtained.

But when the detected environment is quite large, we need tons of particles to estimate the pose, which is unreasonable in actual operation. In *Slam_gmapping*, it uses the maximum likelihood estimation to optimize this problem. According to the predicted distribution of the particle's pose and the matching degree with the map, the optimal pose parameter of the particle is found through scanning matching. Then this parameter is used directly as the position of the new particle posture. Through this way, the quality of sampling is highly improved. The pseudocode of Rao-Blackwellized Particle Filter Algorithm is shown in algorithm 1.

Algorithm 1 Rao-Blackwellized Particle Filter Algorithm

Require:
S_{t-1}, the sample set of the previous time step
z_t, the most recent laser scan
u_{t-1}, the most recent odometry measurment
Ensure:
S_t, the new sample set
$S_t = \{\}$
for all $s_{t-1}^{(i)} \in S_{t-1}$ **do**
$< x_{t-1}^{(i)}, w_{t-1}^{(i)}, m_{t-1}^{(i)} >= s_{t-1}^{(i)}$
//scan-matching
$x_t'^{(i)} = x_{t-1}^{(i)} \oplus u_{t-1}$

$\hat{x}_t^{(i)} = argmax_x p(x|m_{t-1}^{(i)}, z_t, x_t'^{(i)})$
if $\hat{x}_t^{(i)}$ = failure **then**
  $x_t^{(i)} \sim p(x_t|x_{t-1}^{(i)}, u_{t-1})$
  $w_t^{(i)} = w_{t-1}^{(i)} \cdot p(z_t|m_{t-1}^{(i)}, x_t^{(i)})$
else
  //sample around the mode
  for $k = 1, ..., K$ **do**
    $x_k \sim x_j || x_j - \hat{x}^{(i)}| < \Delta$
  end for
  // compute Gaussian Proposal
  $\mu_t^{(i)} = (0, 0, 0)^T$
  $\eta^{(i)} = 0$
  for all $x_j \in x_1, ..., x_K$ **do**
    $\mu_t^{(i)} = \mu_t^{(i)} + x_j \cdot p(z_t|m_{t-1}^{(i)}, x_j) \cdot p(x_t|x_{t-1}^{(i)}, u_{t-1})$
    $\eta^{(i)} = \eta^{(i)} + p(z_t|m_{t-1}^{(i)}, x_j) \cdot p(x_t|x_{t-1}^{(i)}, u_{t-1})$
  end for
  $\mu_t^{(i)} = \mu_t^{(i)} / \eta^{(i)}$
  $\sum_t^{(i)} = 0$
  for all $x_j \in x_1, ..., x_K$ **do**
    $\sum_t^{(i)} = \sum_t^{(i)} + (x_j - \mu^{(i)})(x_j - \mu^{(i)})^T \cdot p(z_t|m_{t-1}^{(i)}, x_j) \cdot p(x_t|x_{t-1}^{(i)}, u_{t-1})$
  end for
  $\sum_t^{(i)} = \sum_t^{(i)} / \eta^{(i)}$
  // sample new pose
  $x_t^{(i)} \sim \mathcal{N}(\mu_t^{(i)}, \sum_t^{(i)})$
  //update importance weights
  $w_t^{(i)} = w_{t-1}^{(i)} \cdot \eta^{(i)}$
end if
//update map
$m_t^{(i)} = integrateScan(m_{t-1}^{(i)}, x_t^{(i)}, z_t)$
//update sample set
$S_t = S_t \cup < x_t^{(i)}, w_t^{(i)}, m_t^{(i)} >$
end for
$N_{eff} = \frac{1}{\sum_{i=1}^{N} (\tilde{w}^{(i)})^2}$
if $N_{eff} < T$ **then**
  $S_t = resample(S_t)$
end if

4.3 Result

The result of *hokuyo_gmapping_demo.launch* is shown below in figure 4 and figure 5. It is obvious to notice from the result that the effect of lidar-based map construction is acceptable. It can detect the edge of the laboratory and the obstacle in the center of the laboratoy with a red circle, but we can also notice that the effect on the right side of the

constructed map is not Very ideal, because there are lots of chairs in the corresponding positions in the laboratory, which are difficult to detect properly and affect the result of this segment. Then we perform the landmark detection based on this constructed map.

FIGURE 4: The environment pf the laboratory

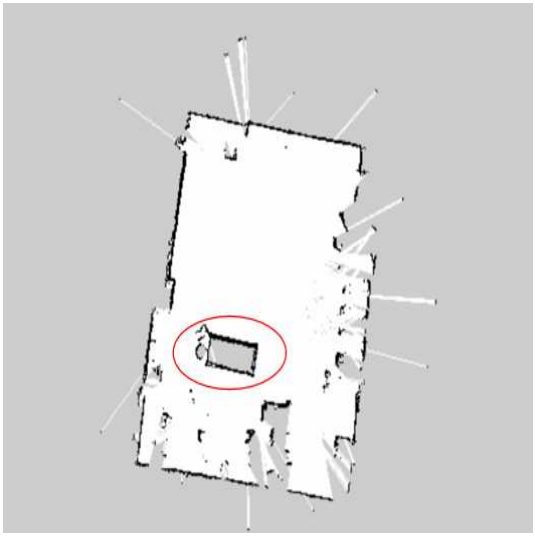

FIGURE 5: The constructed map of the laboratory

5 Landmark Detection

5.1 Aruco Marker

An ArUco marker is a synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier (id). The black border facilitates its fast detection in the image and the binary codification allows its identification and the

application of error detection and correction techniques.[6] The marker size determines the size of the the internal matrix. For instance a marker size of 5x5 is compose by 25 bits, and we use markers of this size as the landmark to be detected. Some examples of ArUco markers are shown in figure 6:

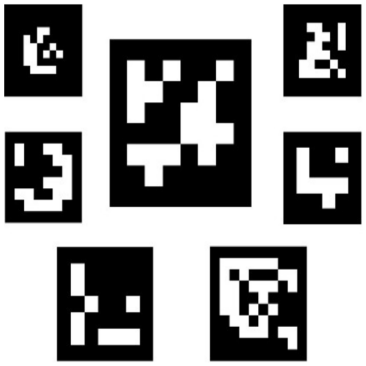

FIGURE 6: Aruco Marker

And these are the reasons why we choose this binary square fiducial marker:

- Easily detected
- Relatively robust
- Less embedded bits
- Possible for extrapolating the orientation

5.2 Marker Detection

The main goal of this part is to recognize the ArUco marker and returns its id. This process is comprise of two parts:

1. Detection of marker candidates. In this step the image is analyzed in order to find square shapes that are candidates to be markers.
2. After the candidate detection, it is necessary to determine if they are actually markers by analyzing their inner codification. This step starts by extracting the marker bits of each marker.

The ArUco marker could be recognized on the ASUS laptop, through the kinect camera located on TurtleBot, as shown in figure 7:

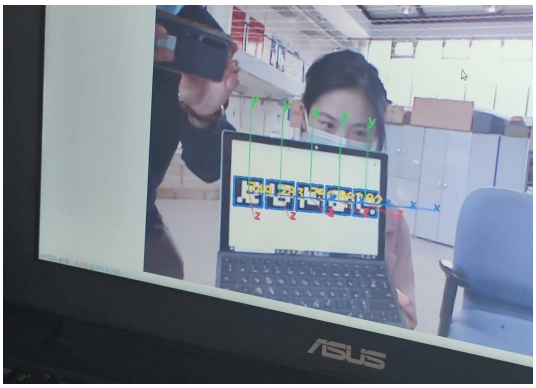

FIGURE 7: Marker detection

5.3 Pose Estimation

We know that there exists a distance between the TurtleBot and the ArUco marker, as shown in figure 8 So our problem is that the coordinates system for the TurleBot is the world coordinate system origin, whose original point is O_w. And the information captured by the camera is in the camera system, therefore, a transformation between world coordinate W and camera coordinate C should be developed.

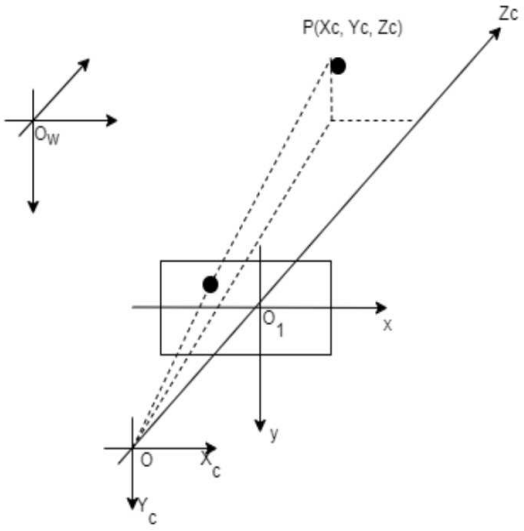

FIGURE 8: Camera coordinates and world coordinates

The basic idea of this transformation is first rotate the coordinates to the same angls.
First of all, by using CV::aruco::estimatePoseSingleMarkers() function, two very important data revc(rotation vector) and tevc(translation vector) are obtained, respectively. And from these two data, we can get the coordinates of the camera in the world coordinate system. This is where knowledge of SolvePNP is required. The R and T solved by SolvePNP are the rotation and translation required to transform points in the world coordinate system to the camera coordinate system:

- Rvec - The output rotation vector.
- Tvec - the translation vector of the output.

Rotation The first thing need to be done is turn the rotation vector into a rotation matrix using a Rodrigues' rotation.

```
double rm[9];
RoteM = cv::Mat(3, 3, CV_64FC1, rm);
Rodrigues(rvec, RoteM);
double r11 = RoteM.ptr<double>(0)[0];
double r12 = RoteM.ptr<double>(0)[1];
double r13 = RoteM.ptr<double>(0)[2];
double r21 = RoteM.ptr<double>(1)[0];
double r22 = RoteM.ptr<double>(1)[1];
double r23 = RoteM.ptr<double>(1)[2];
double r31 = RoteM.ptr<double>(2)[0];
double r32 = RoteM.ptr<double>(2)[1];
double r33 = RoteM.ptr<double>(2)[2];
```

The next step is to calculate the rotation Angle of each axis according to the rotation matrix.

```
double thetaz = atan2(r21, r11) / CV_PI * 180;
double thetay = atan2(-1 * r31, sqrt(r32 * r32 + r33 * r33)) / CV_PI
double thetax = atan2(r32, r33) / CV_PI * 180;
```

In this case, thetax, thetay, and thetaz are the rotation angles about x, y, and z respectively

Translation Here is a core function that we use to calculate the value on each axis.

```
void CodeRotateByX(double y, double z, double thetax, double &outy, doub
{
    double y1 = y;
    double z1 = z;
    double rx = thetax * CV_PI / 180;
    outy = cos(rx) * y1 - sin(rx) * z1;
    outz = cos(rx) * z1 + sin(rx) * y1;
}
```

This becomes the camera's coordinates in the world coordinate system, but one more calculation is required to translate it in the opposite direction:

```
x = x * -1;
y = y * -1;
z = z * -1 - 0.1;
```

Another thing need to be noticed is that z has to minus 0.1 due to the height to the kinect camera At this point, the obtained (x,y,z) are the coordinates of the camera in the world coordinate system with the two-dimensional code as the origin.

5.4 DataSet Construction

The pose of ArUco marker and location of TurtleBot are output as follow, shown in figure 9 and figure 10, respectively.

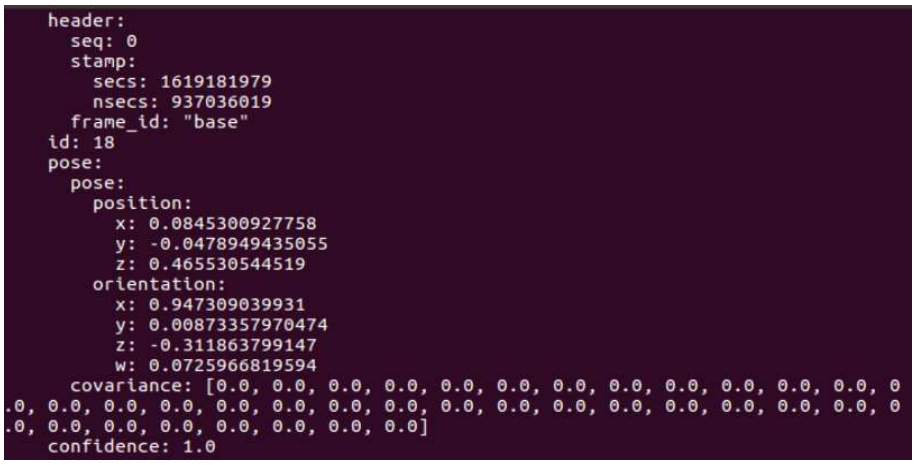

FIGURE 9: pose of ArUco marker

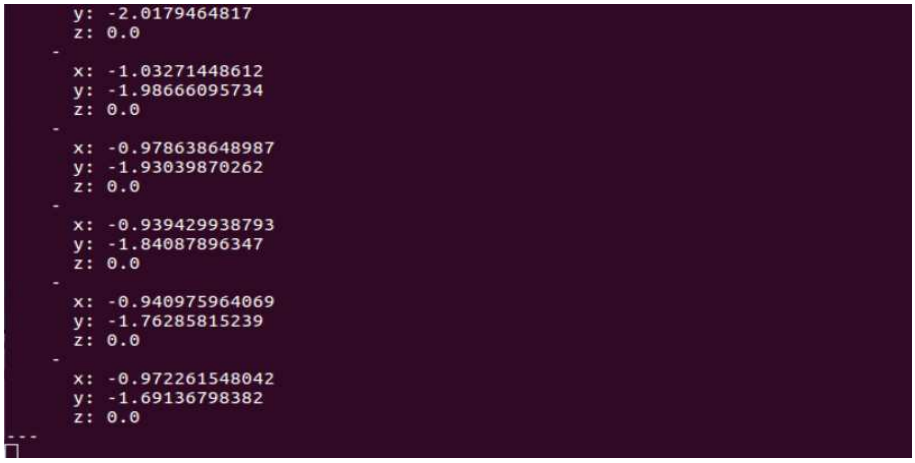

FIGURE 10: location of TurtleBot

After successfully received those data above, we save it in a two-dimension array, containing three elements:

- the first index - id of the ArUco marker
- the second index - x in the world coordinates
- the third index - y in the world coordinates

Based on these, we now have the essential data required by navigation.

6 Navigation

6.1 Navigation in ROS

In ROS, the navigation is implemented via *amcl_demo.launch* file. There are two issues relating to navigation. The first is the global navigation, navigating and locating; the second is local navigation, avoiding obstacles. The specific process of global navigation can be divided into two parts, localizing and moving. AMCL(Adaptive Monte Carlo Localization) is used for localizing and *move_base* is used for the movement. Figure 11 shows the framework of *move_base*. It can be seen that the movement plan is also based on localizing. The local navigation is determined by the boundaries of the established map.

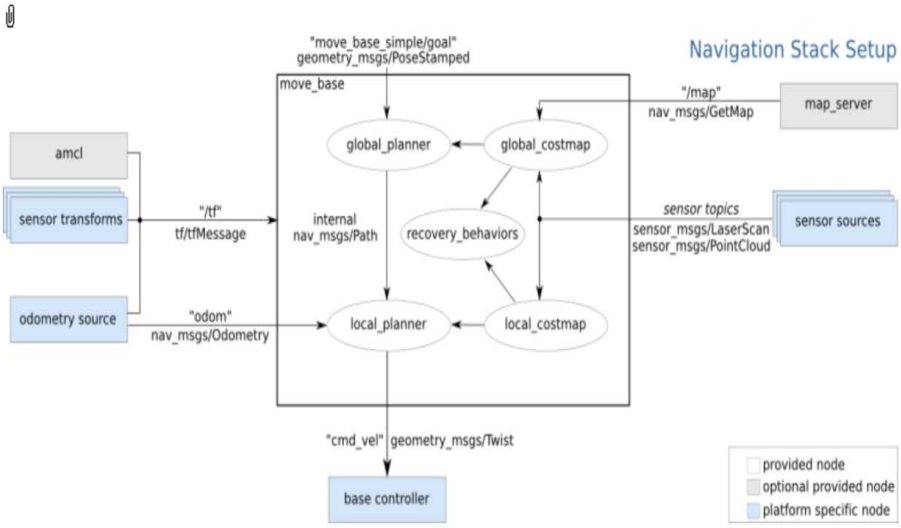

FIGURE 11: The framework of *move_base* node in ROS

6.2 AMCL(Adaptive Monte Carlo Localization)

The ability for a robot to locate itself in an environment is a common problem in mobile robots. Adaptive Monte Carlo Localization uses a probabilistic localization technique and particle filters to track the pose of the robot against a map loaded from the map server. [7] AMCL is an improved algorithm of Monte Carlo Localization, which can be simply stated as five steps:

- **Randomly generate a bunch of particles**: Particles can have position or orientation. Each has a weight (probability) indicating how likely it matches the actual state of the system. Initialize each with the same weight.
- **Predict next state of the particles**: Move the particles based on how you predict the real system is behaving
- **Update**: Update the weighting of the particles based on the measurement. Particles that closely match the measurements are weighted higher than particles which don't match the measurements very well.

- **Resample**: Discard highly improbable particle and replace them with copies of the more probable particles.
- **Compute Estimate**: Compute weighted mean and covariance of the set of particles to get a state estimate.

But there exists a problem. If in the positioning process the particles corresponding to the impossible positions gradually disappear, to a certain extent, only particles near one location can survive. And in case this location happens to be incorrect then the algorithm will fail. This problem is very serious. In fact, any random algorithm like MCL may discard particles near the correct position in the process of resampling. This problem is particularly prominent when the number of particles is small and the distribution range is wide.
To solve this problem, AMCL adds the short-term and long-term weight records. If the robot move for a long time, it will get the long-term weight. The short-term weight is for the short-term move. The long-term and short-term weights are maintained to obtain the judgment of whether there is an error resampling. After calculating the value, get the ratio of fast and slow to determine whether to add particles randomly to facilitate subsequent repositioning. The precise pseudocode of AMCL is shown in algorithm 2

Algorithm 2 AMCL(χ_{t-1}, u_t, z_t, m)

χ_{t-1}: The confidence
static $\omega_{slow}, \omega_{fast}$: Weights
$\bar{\chi}_t = \chi_t = \emptyset$
for $m = 1 to M$ **do**
  $x_t^{[m]}$ =**sample_motion_model**$(u_t, x_{t-1}^{[m]})$
  $\omega_t^{[m]}$ =**measurement_model**$(z_t, x_t^{[m]}, m)$
  $\bar{\chi}_t = \bar{\chi}_t + < x_t^{[m]}, \omega_t^{[m]} >$
  $\omega_{avg} = \omega_{avg} + \frac{1}{M}\omega_t^{[m]}$
end for
$\omega_{slow} = \omega_{slow} + \alpha_{slow}(\omega_{avg} - \omega_{slow})$
$\omega_{fast} = \omega_{fast} + \alpha_{fast}(\omega_{avg} - \omega_{fast})$
for $m = 1 to M$ **do**
  if probability $max\{0.0, 1 - \frac{\omega_{fast}}{\omega_{slow}}\}$ **then**
    add random pose to χ_t
  else
    draw $i \in \{1, ..., N\}$ with probability $\propto \omega_t^{[i]}$
    add $x_t^{[m]}$ to χ_t
  end if
end for
return χ_t

6.3 Result

The figure 12 shows the Turtlebot2 on the constructed cost-map by using the ROS built-in visualization tools, rviz. The colored square is the scope of the local path planning, and the

green clusters of arrows are particle swarm positioning based on AMCL. THe light green line indicates the path planned which the robot will navigate through.

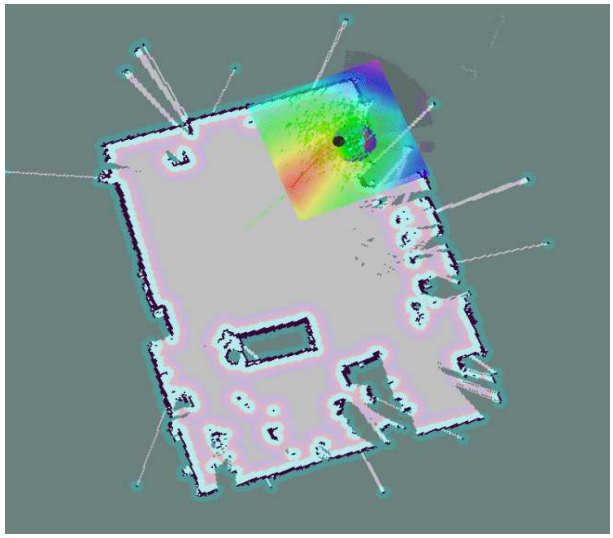

FIGURE 12: The simulation of navigation on rviz

7 Result Analysis and Conclusion

Throughout the process of this project, we can notice the following results. In the process of communication between the two hosts, because they use the wireless router communication, the process of sending back information from the turtlebot to the workstation is quite slow, so it is not advisable to control the robot to move fast and it will lose a part of the accuracy when moving fast. At the same time, it is also because of the router, when there is a large obstacle between two computers, the communication will be obstructed. When constructing the map, as shown in the figure 4, while encountering some complicated borders (a lot of messy chairs), the map effect is not ideal. In landmark detection part, Kinect camera can detect multiple markers at the same time and return relevant information with high accuracy, but it should be noted that because of the design of ArUco marker, mark detection has some certain requirements for the sharpness of the marker boundary. In the last navigation process, we noticed that there will be errors in the navigation. We noticed that by placing stickers on the target points, which means that although the coordinates of the navigation are reached, in the real laboratory, there exists a slight error from the specified target. And as the number of navigation increases, the error will also accumulate. We currently think there are two reasons, one is the error of the odometer sensor, this is the error of the hardware which is inevitable. The second reason may be that we have used some coordinate system transforms when determining the coordinates, and the coefficients of these transform matrices are all from the detection of the sensor of Turtlebot2 or our own measurement. There might be some errors. For the elimination of this error, we think it can be solved by machine learning, which will be described in the future work part.

In this project, we use the robot Turtlebot2 with Hokuyo lidar and Kinect camera to simulate a simple autonomous driving through a modular design. Our method is based on Rao-Blackwellized particle filter, OpenCV and Adaptive Monte Carlo Localization to realize map construction, marker detection and navigation respectively. Our approach computes a highly accurate coordinate of target marker based on the pose estimation with the information given back for sensors. Our approach has been implemented and evaluated on a real situation. Multiple tests with different starting points and target points have demonstrated the robustness of our design and the ability of generating a high quality map in a relatively small environment with less memory space and low hardware requirements of sensor, which has practical significance for indoor navigation and accurate location determination of targets.

8 Future Work

8.1 Auto SLAM

The auto SLAM can be interpreted as autonomous localization and mapping. In our project and most cases, the SLAM algorithm still needs to manually control the robot movement to recognize the environment. This is still a bit unsmart. We need to add motion planning to this, or called the path-finding algorithm to achieve the auto SLAM. The local path planning for the current position of the robot based on the sensor information can choose the direction of movement autonomously to avoid obstacles. Because our robor is equipped with a Kinect RGB-D camera. Using the ORB-SLAM algorithm might be considered. ORB-SLAM algorithm is an autonomous localization and mapping algorithm for mobile robot while no prior knowledge of its environment provided. A local 3D point cloud map is constructed, through the depth information acquired from RGB-D sensor and corresponding camera poses estimated from ORB-SLAM, which is then transformed to a 2D occupancy grid map using octree. Based on the 2D map, an information-theoretic exploration algorithm is used to travel through all the environment.[8]

8.2 Route Planing

At this stage, our navigation only considers entering one target point at a time. If multiple target points are input at one time, how the robot chooses the navigation order of target points is worth discussing. This question not only considers the distance between the target points and the starting point, but also the Id value of the target mark, which makes this question more practical, such as the priority of customers in the delivery process. At present, we are considering using graph theory to solve this problem. First, normalize the Id value of the target point set in order to eliminate the overall influence of the special value and make the algorithm more robust. Multiply it with the distance between the target points, and use this value as the weight of the path between the two points to construct the connectivity graph. Finally, using the Dijkstra algorithm in graph theory to solve this shortest path problem.

8.3 Machine Learning to Improve the Accuracy

As mentioned in the previous result analysis section, there is still room for improvement in the accuracy of the navigation algorithm. We can consider using the most basic algorithm in machine learning, the backpropagation algorithm [9] to reduce errors. We set the values in the transformation matrix as parameters. The distance between the final navigation position and the target position is the loss function. By iteratively adjusting the parameters with appropriate step along the gradient direction to reduce the loss. Finally we can obtain appropriate parameter. However, it should be noted that if there is an over-fitting phenomenon, this algorithm will only be applicable to the scenario of our laboratory, and is not universal. We can use the transfer learning method to solve this. First, to get a rough parameter value in our condition, and then use BP algorithm for training and learning several times in other different environments.

References

[1] Pakusch, Christina, G. Stevens, A. Boden and Paul Bossauer. “Unintended Effects of Autonomous Driving: A Study on Mobility Preferences in the Future.” Sustainability 10 (2018): 1-22.

[2] Taheri, Hamid and Zhao Chun Xia. “SLAM; definition and evolution.” Eng. Appl. Artif. Intell. 97 (2021): 104032.

[3] Grisetti, G., C. Stachniss and W. Burgard. “Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters.” IEEE Transactions on Robotics 23 (2007): 34-46.

[4] Wikipedia. ROS
https://en.wikipedia.org/wiki/Ros

[5] Wikipedia. Turtlebot2
https://en.wikipedia.org/wiki/TurtleBotTurtleBot_2

[6] OpenCV. "Detection of ArUco Markers."
https://docs.opencv.org/master/d5/dae/tutorial_aruco_detection.html

[7] Cheong, A., M. Lau, E. Foo, J. Hedley and J. Bo. “Development of a Robotic Waiter System.” IFAC-PapersOnLine 49 (2016): 681-686.

[8] Mo, Hongwei, Xiaosen Chen, K. Wang and H. Wang. “Autonomous Localization and Mapping for Mobile Robot Based on ORB-SLAM.” (2019).

[9] Fan, Fenglei, W. Cong and G. Wang. “General Backpropagation Algorithm for Training Second-order Neural Networks.” International journal for numerical methods in biomedical engineering 34 5 (2018): e2956 .

Appendix

simple_navigation.cpp

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <cstdio>

typedef actionlib::SimpleActionClient<move_base_msgs::
   MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv){
  //test-array
  //the first index - value of the ArUco marker
  //the second index - x in the coordinates
  //the third index - y in the coordinates
  int a
     [6][3]={{0,0,0},{1,1,1},{2,2,2},{3,3,3},{4,-3,-3},{5,0,0}};


  int marker=0;
  int breaknumer=10;//you can change this later
  //test-loop
  //the condition can be changed due to the actual
     circumstance
  while(marker!=breaknumber){
    printf("Please input the value of the landmark:\n");
    int temp=marker;//temp is used to save the former value
    //because we are using absolute, not relative,
       coordinates in this project
    scanf("%d",&marker);
    if(marker==breaknmber){
      break;
    }

    ros::init(argc, argv, "simple_navigation_goals");

    //tell the action client that we want to spin a thread
       by default
    MoveBaseClient ac("move_base", true);

    //wait for the action server to come up
    while(!ac.waitForServer(ros::Duration(5.0))){
```

```
      ROS_INFO("Waiting␣for␣the␣move_base␣action␣server␣to␣
          come␣up");
    }

    move_base_msgs::MoveBaseGoal goal;

    goal.target_pose.header.frame_id = "base_link";
    goal.target_pose.header.stamp = ros::Time::now();

    goal.target_pose.pose.position.x = a[marker][1] - a[temp
        ][1];
    goal.target_pose.pose.position.y = a[marker][2] - a[temp
        ][2];
    goal.target_pose.pose.orientation.w = 1.0;

    ROS_INFO("Sending␣goal");
    ac.sendGoal(goal);

    ac.waitForResult();

    if(ac.getState() == actionlib::SimpleClientGoalState::
        SUCCEEDED)
      printf("Hooray,␣the␣base␣moved␣%d␣meter␣in␣the␣
          direction␣of␣x␣and␣%d␣meter␣in␣the␣direction␣of␣y\n
          ", a[marker][1],a[marker][2]);
    else
      printf("The␣base␣failed␣to␣move␣for␣some␣reason\n");
  }
  printf("Voila!␣This␣is␣the␣end!\n");
  return 0;
}
```