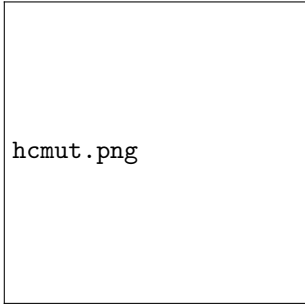


VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



hcmut.png

RESEARCH PROPOSAL FOR CE (CO4325)

Report for Research Proposal - HK211

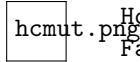
“Autonomous path planning and moving of Robot in Uncertain Environments”

Advisor: MSc. Tran Thanh Binh
Students: Tran Minh Tam - 1813926
Tran Long Vi - 1814804
Vo Phi Truong - 1814582

HO CHI MINH CITY, DECEMBER 2021

Pledge

We hereby declare that this is our research work and is under the scientific guidance of MSc. Tran Thanh Binh. The research contents and results in this topic are honest and have never been published in any form. The contents in the analysis, survey and evaluation are collected by the author himself from different sources and are referenced in the reference section. In addition, in the study, some images and models from other groups of authors were used and the sources were cited and annotated. If we detect any fraud, we promise to take full responsibility for the content of our research. Ho Chi Minh City University Of Technology has nothing to do with copyright infringement.

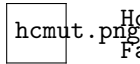


Thank you

I would like to express my sincere thanks and deep gratitude to the teachers who are lecturers of the Faculty of Computer Science and Engineering at the Ho Chi Minh City University of Technology. Special thanks to MSc. Tran Thanh Binh, Assoc. Prof. Dr. Tran Van Hoai, Assoc. Prof. Dr. Phan Thanh An have facilitated and enthusiastically guided.

In the learning process, as well as the thesis-making process, it is difficult to avoid mistakes, which I hope you will overlook. At the same time, due to the limited theoretical level as well as practical experience, the report will inevitably have shortcomings. We look forward to receiving the comments of teachers and teachers to learn more experiences.

We sincerely thank you!



Summary

This article presents the research about the subject "Autonomous path planning and moving of robot in Uncertain environments", including system components and their tasks, applied algorithms and search results. The article also refers to the technologies applied in the research, including ROS and tools supported by ROS, Turtlebot3, Gazebo, SLAM, Path planning.

Table of Contents

List of Figures

1 Research introduction

Robotics and Autonomous Systems with a long history, that have been mentioned and researched for centuries. Today, the world's robotics and automation community are very large, with a lot of scientific achievements. There are many software platforms supported by active communities, such as ROS, Turtlebot...

In the future, robotics engineering will become an industry. The research project of my group applies technologies developed by the community such as ROS, Turtlebot to build autonomous robot in Uncertain environments.

Autonomous robots are intelligent machines capable of performing tasks in the world by themselves, without explicit human control. Nowadays, there are many autonomous systems and machines such as Robots Vacuum Cleaners, Automation Guided Vehicle...

Uncertain Environments are areas where people do not yet know about the terrain and environment.

In this research, we want to build a system that robot can automatically analyse the terrain, identifies obstacles and directions, then moves to predetermined coordinates. This research is applied with technologies and algorithms that are easily applied and developed, consists of ROS, Turtlebot and Gazebo simulation, SLAM. At present, the test environment is simulated in Gazebo that is flat ground with polygonal obstructions. In the future, We will develop systems so that robots can travel in more complex and challenging environments.

2 ROS introduction

2.1 What is ROS?

Robot Operating Systems (ROS) is an open-source, meta-operating systems for Robot development. ROS prototypes originated from Stanford AI research and officially created and developed by Willow Garage starting in 2007.

ROS provides the services which an operating systems has, includes:

- Hardware abstraction.
- Low-level control.
- Implementation of commonly-used functionality.
- Message-passing between processes.
- Package management.

ROS provides tools and libraries for obtaining, building, writing, and running code across multiple computers. Therefore, it does not replace, but instead works alongside a traditional operating system.

2.2 Why ROS?

To know why ROS is commonly used in robot control projects, let's analyze advantages:

- Provides lots of infrastructure, tools and capabilities.
- Easy to try other people's work and share you own.
- Large community.

- Free, open source, BSD license.

Besides, ROS also can help to resolve a few specific issues in the development of software for robots.

- **Distributed computation:** ROS provides 2 relatively simple, seamless mechanisms (publish - subscribe pattern and services) for communication between multiple processes that may or may not live on the same computer.
- **Software reuse:** ROS has API to allows developers use good algorithms that are researched before like navigation, motion planning, mapping, etc. So developers can focus more time on experimenting with new ideas, and less time reinventing wheels.
- **Rapid testing:** ROS separate of the low-level direct control of the hardware and high-level processing and decision making into separate programs. This thing can help developers avoid time consuming and error-prone when test robot software.

Of course, ROS is not the only platform that offers these capabilities. With the level of widespread support for ROS across the robotics community, ROS will continue to evolve, expand, and improve in the future.

Finally, let's take a moment to review a few things that are not true about ROS

- Learning to use a new framework, particularly one as complex and diverse as ROS, can take quite a lot of time and mental energy.
- Approaching maturity, but still changing.
- Security and scalability are not first-class concerns.
- OSes other than Linux are not well supported.

2.3 Components of ROS

ROS consists of client library to support various programming languages, hardware interface for hardware control, communication for data transmission and reception, the Robotics Application Framework to help create various Robotics Applications, the Robotics Application which is a service application based on the Robotics Application Framework, Simulation tools which can control the robot in a virtual space, and Software Development Tools.



Figure 1: Components of ROS

2.4 Terminologies and conventions

- **Master:** The master acts as a name server for node-to-node connections and message communication. The command `roscore` is used to run the master.
- **Node:** like one executable program. Node refers to the smallest unit of processor running in ROS.
- **Package:** is the basic unit of ROS. The package contains either a configuration file to launch other packages or nodes.

- **Message:** A node sends or receives data between nodes via a message. Messages are variables such as integer, floating point, and boolean.
- **Topic:** A topic is registered by publisher node and subscriber node directly connects to the publisher node to exchange messages as a topic.
- **Publish and publisher:** Publishing is the action of transmitting relative messages corresponding to the topic. The publisher node registers its own information and topic with the master, and sends a message to connected subscriber nodes that are interested in the same topic.
- **Subscribe and subscriber:** Subscribing is the action of receiving relative messages corresponding to the topic. The subscriber node registers its own information and topic with the master, and receives publisher information that publishes relative topic from the master.

2.5 ROS version: Noetic

ROS Noetic Ninjemys is the thirteenth ROS distribution release. It was released on May 23rd, 2020. ROS Noetic Ninjemys is primarily targeted at the Ubuntu 20.04 (Focal) release, though other systems are supported to varying degrees.

In this research, we used ROS Noetic in Ubuntu 20.04 to build and implement system.

3 Communication diagram between nodes in ROS

The diagram shows the communication(send, receive message, data) between nodes in the system. The meaning of different arrow symbols in the diagram:

- **Undirected line** connecting 2 Nodes: Generally show that there is a relationship between the 2 nodes.
- **Directed dashed line** from Node A $\xrightarrow{\text{_TopicName}}$ to Node B: Node A sends data(messages) to Node B through topic _TopicName only 1 time to initialize some configure parameters in Node B.
- **Directed solid line** from Node A $\xrightarrow{\text{_TopicName}}$ to Node B: Node A sends data(messages) to Node B whenever A finished a particular task specified by user.
- **Directed solid white line** from Node A $\xRightarrow{\text{_TopicName}}$ to Node B: Node A sends some kind of data to Node B whenever B requests that data.

Topics in the diagram:

- **odom:** Provides messages containing information about:
 - **pose:** location coordinates(x, y, z)(Euler form) and orientation coordinates (x,y,z,w)(in quaternion form) of robot.
 - **twist:** moving(linear) velocity(v_x, v_y, v_z) and angular velocity($\omega_x, \omega_y, \omega_z$) of the robot.
 - **Publisher:** gazebo



Figure 2: Nodes of the implementing system

- **Subscriber: Robot Global Vision Update, Robot motion Node**
- **cmd_vel:** Publish messages(Twist) specify the desired moving(linear) velocity and angular velocity of the robot to this topic to **instruct** the robot to move accordingly to those messages.
 - In a **Twist** message, if the 3 fields of moving velocity(v_x, v_y, v_z) are set to 0, and at least 1 one the 3 fields of angular velocity($\omega_x, \omega_y, \omega_z$) is not equal to 0, Robot will be rotating.
 - In a **Twist** message, if the 3 fields of angular velocity($\omega_x, \omega_y, \omega_z$) are set to 0, and at least 1 one the 3 fields of moving velocity(v_x, v_y, v_z) is not equal to 0, Robot will be moving in a straight line trajectory, in (v_x, v_y, v_z) direction with a constant speed of $\sqrt{v_x^2 + v_y^2 + v_z^2}$.
 - In a **Twist** message, if all the 6 fields are set to 0, the robot will stay still.
- **Publisher: robot motion node.**
- **Subscriber: gazebo.**

- **cmd_moving:**

- Provides messages containing information about approximately shortest path from current position to next position. The Path between 2 points if a series of points, with the trajectory between 2 adjacent points in the series is a straight line.
- **Publisher: Robot Global Vision Update.**
- **Subscriber: robot motion node.**

- **update_vision:**

- Provides messages signaling the task of moving the robot from the current position to the next one has completed. The content of these messages is just a string "done".
- **Publisher: robot motion node.**
- **Subscriber: Robot Global Vision Update.**

Nodes in the diagram:

- **gazebo:**

- Runs the simulation program..
- Publishes **Odometry** messages to 2 Nodes Robot Global Vision Update, robot motion node only 1 time at the beginning of the program to provide pose data of the robot to initialize the stating position of the robot.
- Receives messages from **cmd_vel** topic to make the robot move accordingly to the content of received messages.

- **SLAM:**

- Runs **SLAM** algorithm to get accumulated vision that the LIDAR of the robot has scanned from the beginning of the program up to the present moment in the simulation.
- Provides **Node Robot Global Vision Update** with the accumulated vision of the robot in form of an image file.

- **Robot Global Vision Update:**

- Gets the accumulated vision image file from **Node SLAM**, identifies obstacles in the image, determines all potential moving points within the current visible local zone, ranking all potential moving points(that haven't been traversed), picks the highest score point as the next point to traverse to and calculates the approximately shortest path from the current position to the picked point.
- Sends the calculated approximately shortest path to **Node Robot motion node** through topic **cmd_moving**.
- Receives messages signaling that the robot has completed the task of moving to the next point of **Node robot motion node**. **Node Robot Global Vision Update** will redo all the tasks mentioned above whenever it receives those messages.

- **robot motion node**

- Receives messages containing an approximately shortest path from the current position to a point and move the robot to that point.
- Publish messages "done" to **Node Robot Global Vision Update** on completing moving to the next point.

4 Implementation

The system is implemented based on the communication diagram described above. In ROS, 1 node corresponds to a process (a running program). Nodes run in parallel, every node will publish messages to topics and subscribe to some topic. Suppose that Node **A** subscribes to topic **a**, when there is a message published to topic **a**. Node **A** will automatically call a pre-specified callback to execute some tasks defined in that callback (callbacks are explicitly specified functions which will be automatically executed whenever the event a message is sent to the topic that the calling Node subscribed to). Input of callbacks are messages that the calling Node receives.

The system when running will have a total of 4 nodes: **gazebo**, **SLAM**, **Robot Global Vision Update**, **robot motion node**. Node **gazebo**'s role is running the simulation, provides information about the current position of the robot through topic **./odom**. Messages sent to topic **./odom** have the type of **Odometry**, the definition of **Odometry** is shown as below: An Odometry message has 4 fields, 2 of those fields need to paid attention are:

- **geometry_msgs/PoseWithCovariance pose**: contains location coordinates information in Euler 3-dimension space(x, y, z) and orientation coordinates (in quaternion form) along with information about **covariance** of the robot in the simulation environment.
- **geometry_msgs/TwistWithCovariance twist**: contains velocity coordinates information in Euler 3-dimension space(v_x, v_y, v_z) and angular velocity($\omega_x, \omega_y, \omega_z$) along with information about **covariance** of the robot in the simulation environment.



Figure 3: Odometry message definition

A `geometry_msgs/PoseWithCovariance` message has 2 fields, in it, we only need to pay attention to field **Pose pose**:

- **Pose pose**: contains location coordinates information in Euler 3-dimension space(x, y, z) and orientation coordinates(in quaternion form) of the robot in the simulation environment.

A `geometry_msgs/Pose` message has 2 fields:

- **Point position**: contains location coordinates information in Euler 3-dimension space(x, y, z) of the robot in the simulation environment.



Figure 4: PoseWithConvariance definition

- **Quaternion orientation:** contains orientation coordinates(in quaternion form) of the robot In the simulation environment.

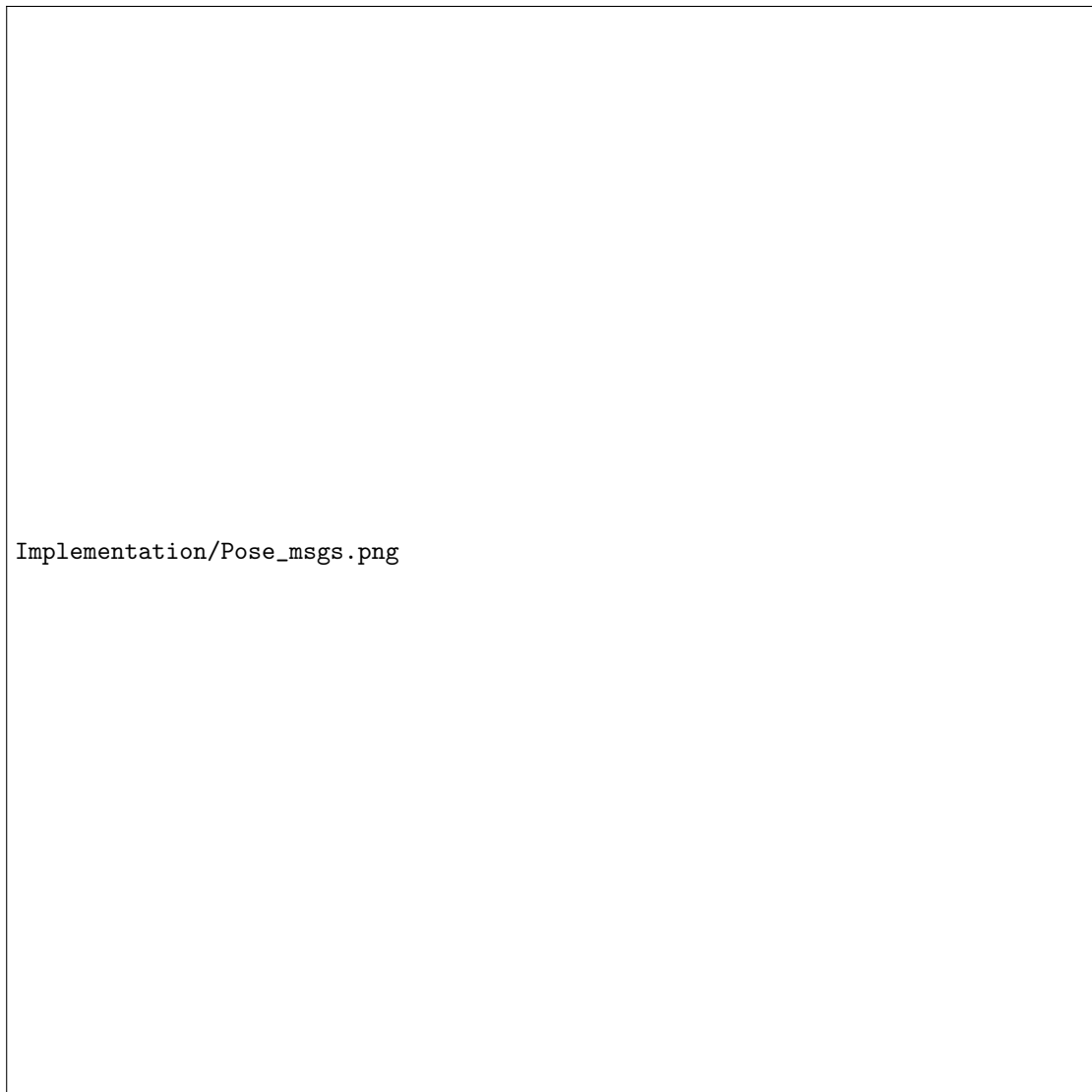


Figure 5: Pose message definition

A **geometry_msgs/Point** message has 3 fields(each field is a primitive floating point 64 bit number) show 3 x, y, z coordinates in Euler 3-dimension space.

A **geometry_msgs/Quaternion** has 4 fields(each field is a primitive floating point 64 bit number) 4 coordinates of orientation in quaternion form.

- Two Nodes **Robot Global Vision Updata, robot motion node** subscribe to topic **/odom**. Each node calls a callback that run only 1 time when each node starts to get information about the initial position of the robot in the simulation environment. This can be achieved by using a global flag(each node has its own flag), each flag in a node will be set when execute the callback(used to initialize the starting position of the robot) of calling node once, which indicating that the callback has been executed once, in the next

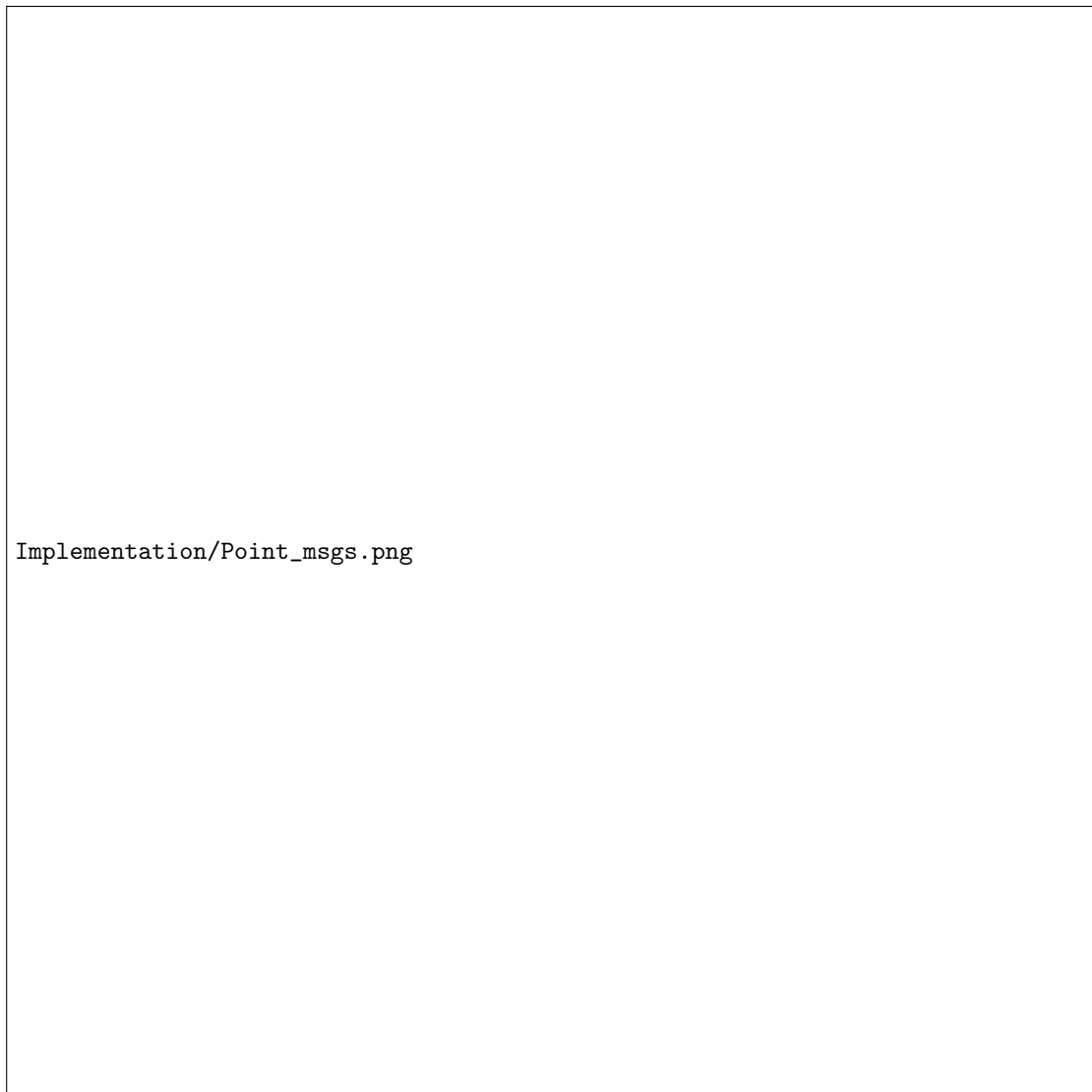


Figure 6: Point message definition

callings, the calling will just simply return.

- Node **gazebo** will be run first, the second running Node In the system is **SLAM**. **SLAM**'s job is to use LIDAR of the robot in the simulation world to scan the environment around the robot with a radius of **R**. Output of Node **SLAM** is an image file containing accumulated vision that SLAM has scanned from the beginning execution of the system to the present time. image file containing accumulated vision about the surrounding environment that the robot has "seen" will be provided to Node **Robot Global Vision Update**. Node **Robot Global Vision Update** gets the image file by issuing a command(command line), not by **Publisher/Subscriber** mechanism.
- Node **Robot Global Vision Update**, after obtaining the cumulative vision image file

Implementation/Quarternion.png

Figure 7: Quaternion message definition

of Node **SLAM**, will proceed to convert the scanned color image to a black and white image and blur the pixel matrices (constituting the image) size 3x3 to reduce the size of the image. image resolution (reduces the workload of image analysis to detect obstacles). Then, the Canny algorithm will be applied to the blurred image to detect the edges of the obstacles, and finally determine the visible obstacles (obstacles are defined as convex polygons, made up of n points). , between 2 adjacent points is a straight line). Based on the coordinates of the identified obstacles and the coordinates of the robot's current position, the Node **Robot Global Vision Update** identifies blocked areas, open areas for the current position. Next, the Node checks whether the robot can "see" the destination or not (the robot "sees" the destination when the destination coordinates fall into one of the robot's open areas).

- For each open region, the Node will choose a point within that zone as a “movable to” point. If the robot has “seen” the destination, the robot will choose the destination as the next destination. If the robot still “doesn’t see” the destination, the Node will proceed to score all the “points that have not been passed” (including the “reachable points within the current vision”, and the “points with can be reached but not visited in the past”) based on some criteria and the following self-defined scoring algorithm. Node **Robot Global Vision Update** will choose the point with the highest score as the next destination. After choosing the next destination, the Node **Robot Global Vision Update** will rely on information about the places the robot has passed, and the closed and open areas analyzed at those places and find the way to the destination. followed by a **sequence of points** where the starting point is the current position, the endpoint is the next destination, the middle points of the sequence are the points the robot has passed. The trajectory between two adjacent points is a straight line.
- Since this path is not optimal, we will use a more optimal path-finding algorithm (self-defined algorithm) based on the path we determined above. At this point, we have a more optimal path to the next destination, we will send this path to the Node **robot motion node** by publishing the **Path** message to the **cmd_moving** topic. After pulling the path, the Node **Robot Global Vision Update** will do the rest, which is to draw information about the accumulated range that the robot has up to the present time, and end the callback function.
- The **Path.msg** file defines the **Path** message. The content of **Path** contains only a “list” of “Points” (in python) or a “vector of “Points” (in C++) which is a sequence of points representing the path from the current position to the next destination.

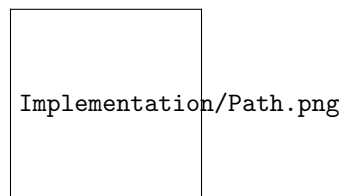


Figure 8: Path definition

- The **Point.msg** file defines a message of type **Point**. The content of the **Point** includes 2 fields x, y (coordinates of the position of 1 point, with z = 0). The type of the two fields is float32(primitive type).

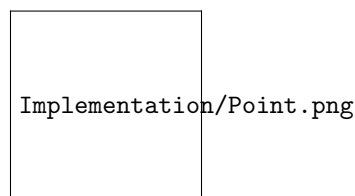


Figure 9: Point definition

- Node **robot motion node** subscribes to topic **cmd_moving**, when the robot receives a message containing information about the route to the next destination (optimized), Node will automatically execute the previously specified callback function to move the robot from current location to next destination. Node moves the robot by publishing **Twist** messages to topic **cmd_vel**(command velocity). The **gazebo** node subscribes to topic **cmd_vel**, the **gazebo** will move the robot according to the “vector” specified in the **Twist** message received by the **gazebo** on the **cmd_vel** topic. Moving between two ”adjacent” points A and B (assuming the robot is at point A) is done by turning the robot towards point B, then moving the robot from A to B by moving the robot forward. before a distance equal to the distance between two points A and B.

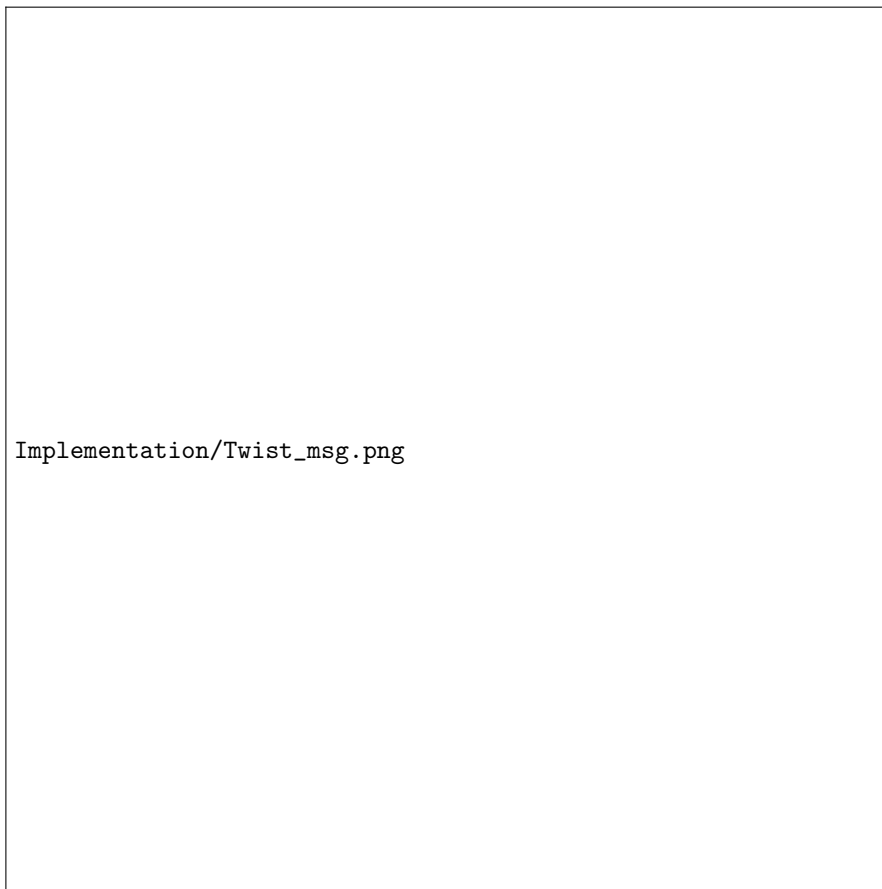


Figure 10: Twist message definition

- In **geometry_msgs/Twist** there are 2 fields: linear and angular
 - Vector3 linear: contains information about the movement vector (v_x, v_y, v_z) .
 - Vector3 angular: contains information about rotation vectors $(\omega_x, \omega_y, \omega_z)$.

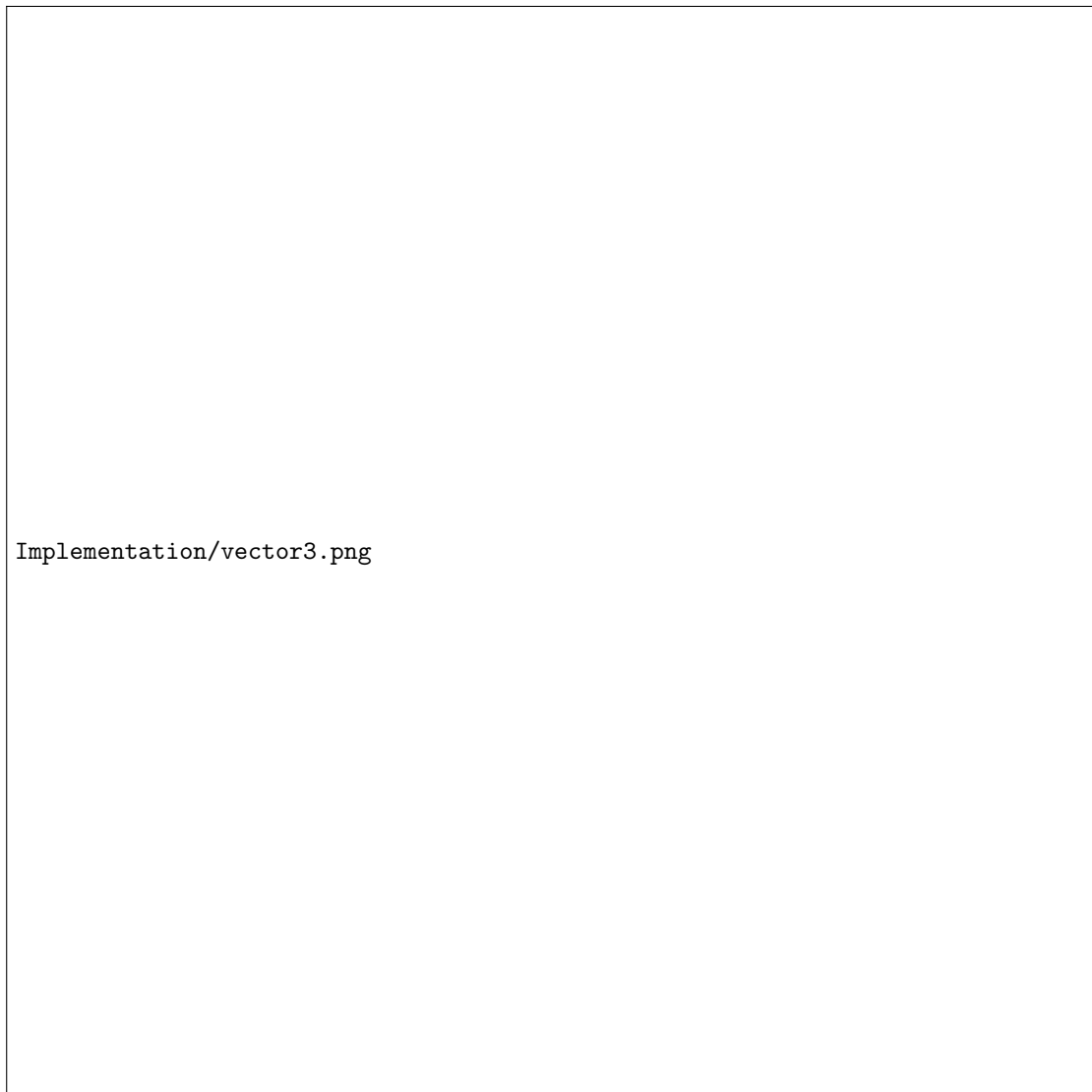


Figure 11: Vector3 message definition

- In type **geometry_msgs/Vector3** there are 3 fields: x, y, and z. The type of the 3 fields is float64(primitive).
- After moving to the next destination, the **robot motion node** publishes an **std_msgs/String** message with a string "done" to topic **update_vision** to signal the Node **Robot Global Vision Update** has completed moving to the next destination.
- In the **std_msgs/String** type, there is one field which is data. The type of field data is a string(primitive)
- Node **Robot Global Vision Update** subscribes to topic **update_vision**. When the Node **Robot Global Vision Update** receives an **std_msgs/String** message signaling



Figure 12: String message definition

the robot has completed moving to the next destination, the Node **Robot Global Vision Update** will call the pre-specified callback function to redo all tasks. mentioned above and again “computes and sends the path to the next destination” to the **robot motion node**. The loop will continue until the robot reaches its destination.

- Also for this reason, at the start of the program, to trigger the loop to happen, the **motion robot node** will send an **std_msgs/String** message to topic **update_vision** to click the Node **Robot Global Vision Update**, the loop will start from there.

5 Gazebo

Gazebo simulator is a well-designed stand-alone robot 3D simulator that can be used to quickly test algorithms, design robots, perform regression testing and train AI system using realistic panel scenarios.

Turtlebot3 provides a package `turtlebot3_gazebo` to perform simulation of a robot as well as its surroundings, the programmer can easily design simple or complex environment by creating another package and setting custom world, according to demand.

Steps to perform Gazebo simulator:

- Firstly, choose a model suitable for the robot: Turtlebot3 provides 2 models of the robot is Burger and Waffle Pi. And in our research, we use Turtlebot3 Burger.
- Then, choose the world and run simulator:



Figure 13: Turtlebot3 Burger (left) and Waffle Pi (right)

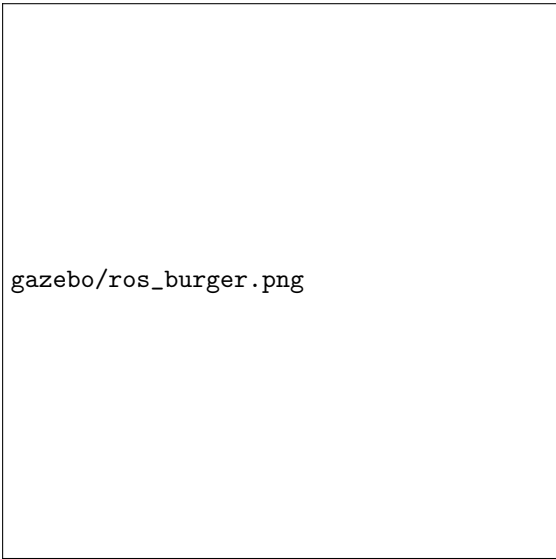


Figure 14: Turtlbot3 Burger in Gazebo simulator

Gazebo is an extremely important node in the communication model between nodes in the implementation. Since this is the main node that provides the simulation and robot world, it also allows the user to know important parameters such as the current position and angle coordinates of robot through the topic Odom.

Example: current position is (-3.0, 1.0, 0.0) and angular coordinates of the robot relative to the x axis is 0 rad.

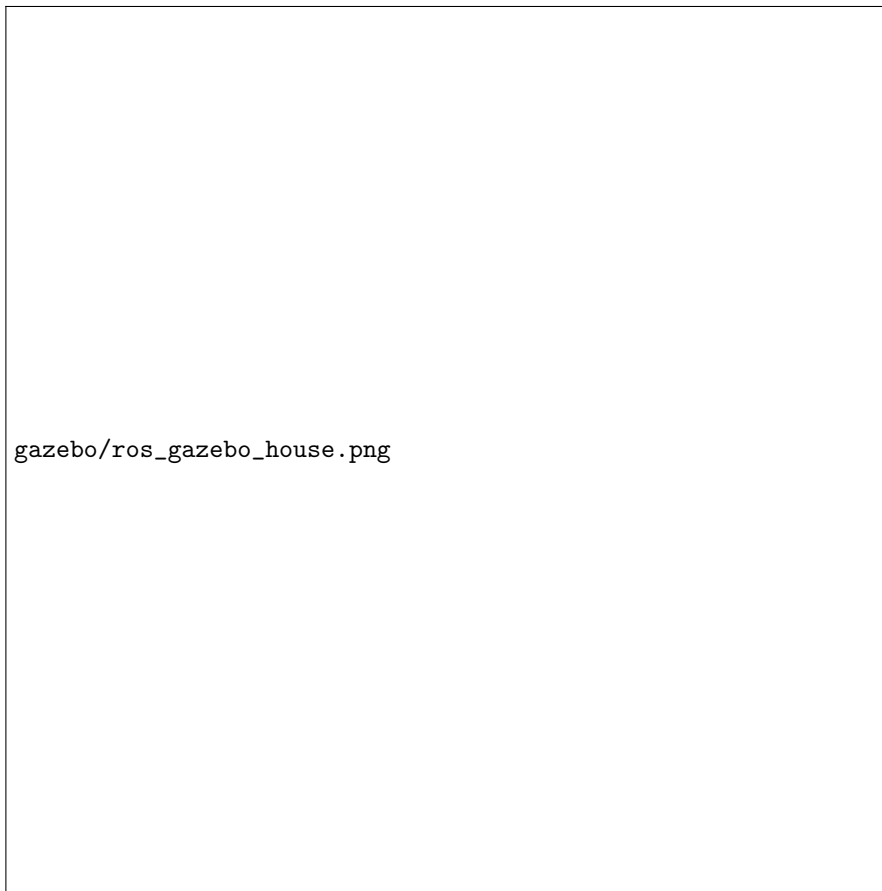


Figure 15: Gazebo simulator house



Figure 16: Current position and Angular coordinates

6 Slam

One of the most popular applications of ROS is SLAM - Simultaneous Localization And Mapping. Slam's goal in mobile robotics is to build and update a map of an uncharted environment with the help of built-in sensors that are attached to the robot (specifically in Burger robots is Lidar)

to discover.

Lidar is a survey method that measures the distance to a target by illuminating the target with a surrounding laser beam and measuring the reflected pulses with a sensor. The difference in laser time and wavelength can then be used to create a 3-dimensional (3D) digital model of the object. The name lidar, now considered an acronym for Light Detection And Ranging. Lidar is an indispensable part of geodetic, geological, and especially robotic research activities.



Figure 17: Lidar in autonomous car

Turtlebot3 provides the turtlebot3_slam package to assist in updating the map of an environment-specific environment, which is the Gazebo simulation environment provided earlier. Maps obtained from executing the turtlebot3_slam node:

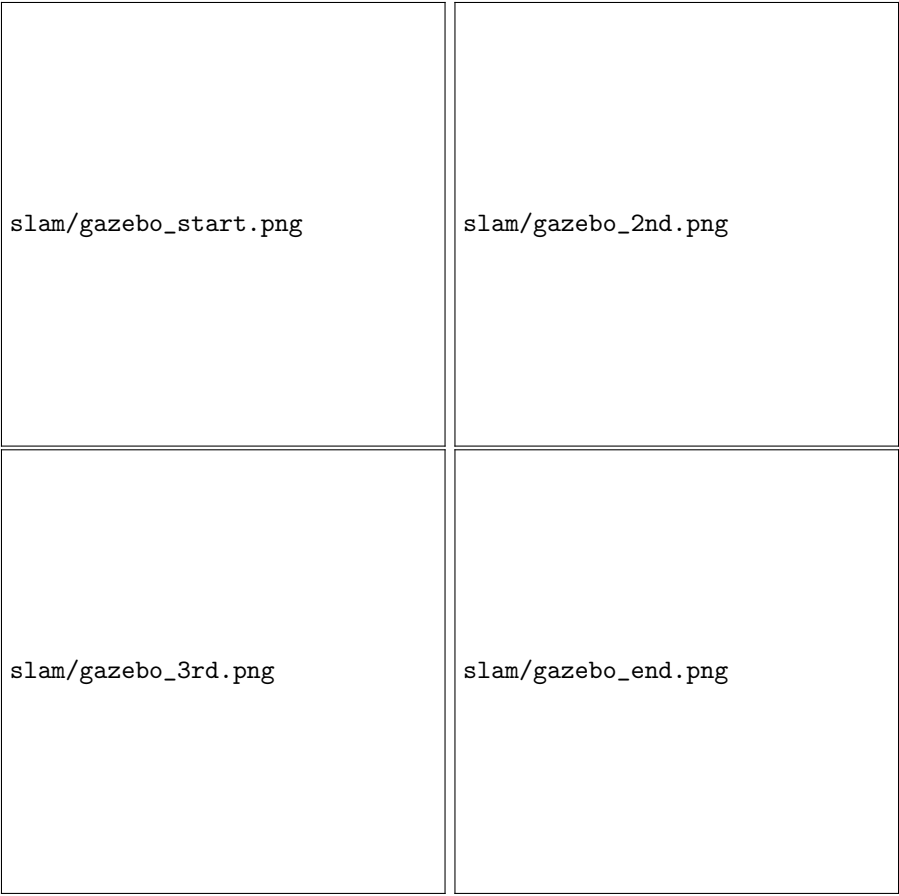


Figure 18: Move robot around world

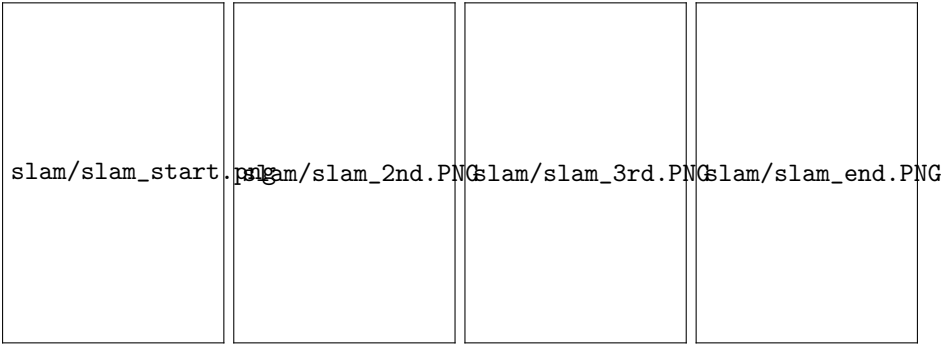


Figure 19: Map accumulated when moving the robot

7 Robot Motion Node

The robot motion node is the third node that is run after Gazebo and Slam when executing the program. We initialize the robot’s movement speed to be 0.1 (m/s) and the rotation speed to be

$\pi/36$ (rad/s). This node will send a trigger message with a data type of String with the content "done" to signal the execution of the RobotGlobalUpdateVision node. Then get the robot's angular coordinates relative to the x-axis, this function executes only once when the node is called. When receiving a message from topic cmd_moving as a set of points with coordinates of 2 values x,y with data type float from the node RobotGlobalUpdateVision. Then move from the start point to the endpoint of the message in turn.

The message received by the node is a curved line, so the robot will have to move through the points from the first element to the last element of the message. Turtlebot3 Burger is a robot that can only go straight and make the left and right turn, so let the robot move exactly from point A to point B. We must first determine the exact angle of \overrightarrow{AB} and the direction of the current robot relative to the x-axis, then perform a rotation so that the robot points in the same direction as \overrightarrow{AB} . The rotation angle is calculated as follows:

Assuming the robot is standing at position A and following the \overrightarrow{AO} , we need to move the dictionary robot A to point B, we need to rotate the robot at an angle such that, after rotating the robot's direction is vector AB. The robot will turn to the left if it receives a positive angle and vice versa. We will have 3 cases in all (where vector \overrightarrow{AO} is the current direction of the robot, vector AB is the direction of to go to the next point):

- Case 1: When O and B are on the same side of the x-axis or $\widehat{xAB} - \widehat{xAO} \in \pi$,
infer rotation angle = $\widehat{xAB} - \widehat{xAO}$



Figure 20: Calculator rotation angle (Basic cases)

- Case 2: When $\widehat{xAB} \in [0, \pi)$ and $\widehat{xAB} - \widehat{xAO} \in [-\pi, 0)$, infer rotation angle $= \widehat{xAB} - \widehat{xAO} + 2\pi$
- Case 3: When $\widehat{xAB} \in (0, \pi)$ and $\widehat{xAB} - \widehat{xAO} \in (\pi, 2\pi)$, infer rotation angle $= \widehat{xAB} - \widehat{xAO} - 2\pi$



Figure 21: Calculator rotation angle (Special case 1)

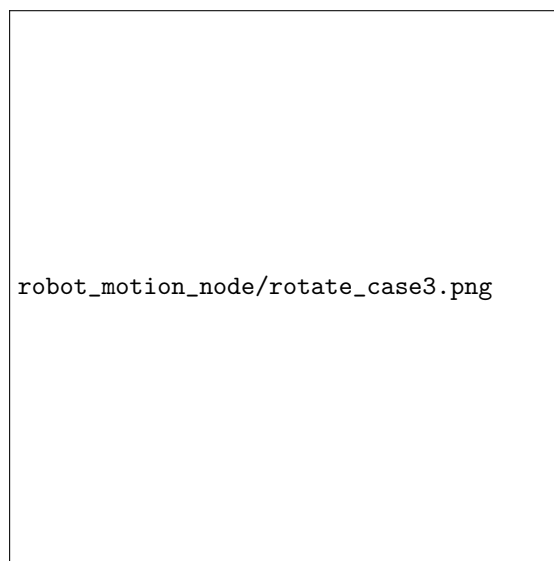


Figure 22: Calculator rotation angle (Special case 2)

After the robot has correctly oriented to the point to go, we will let the robot move to the next point by calculating the distance between the robot's current point coordinates and the next point. Then calculate the time to travel using the formula that calculates time over speed and distance: $t = v/\text{distance}$ (μ/s). Let the robot move in a straight line for time t , then we send a String message with the value "done" to the RobotGlobalUpdateVision node via topic update_vision. Then wait for a message from the RobotGlobalUpdateVision node and continue executing until the robot moves to the destination.

8 Robot Global Vision Update

A task could be a encompassed function or just lines of code that serves a particular task. Initializing tasks(tasks that running once for initialization purpose), a task could be a encompassed function or just lines of code that serves a particular task:

- `Robot_configuration_initialization`: initializes ROS **Robot Global Vision Update** node, subscribes to 2 topic **update_vision**, **odom**. Publishes to topic **cmd_moving**. Set up robot configuration parameter: robot's shape, robot vision(scanning radius of LIDAR), stable moving speed, stable rotate speed, moving accelerator, rotate accelerator, goal Call `getStartPos(msg)`.
- `getStartPos(msg)`: callback executing once to set up starting position of the robot in the simulation environment. It sets the global flag in the first execution. In the next calls to the callback, the callback simply returns(Because the flag has been set).

Repeating tasks(all belong to 1 callback) which execute automatically(in a **sequential manner**) when **Robot Global Vision Update** node receives "done" message from **robot motion node**.

```
os.system('roslaunch map_server map_saver -f ./vision'):
```

- Line of code(command line) for getting accumulated vision image file from **SLAM** node. Save that file with name "vision".
- Output of this task is the accumulated vision image file saved in the same directory as the caller.

```
def read_map_from_world(world_name):
```

- Input of this function is the accumulated vision image file has gotten.
- This function opens the image file, changes the image's color to gray and blurs the gray image by 3x3 matrices that constitute the gray image(To reduce computing effort and to be used as input of **Canny** algorithm). Then, detects edges from the blur image by apply **Canny** algorithm on the blur image. Finally, finds contours base on the output of **Canny**. Each contour comprises series of points that form a polygon representing an obstacle, each Point has 2 coordinates x, y.
- Output: All obstacles detected from the image file in polygon presentation(All obstacles the robot has seen from the beginning of the execution up to the calling present).

```
def scan_around(center, robot_vision, ob, goal):
```

- Input of this function are the current position of the robot(center), robot sight radius(circle zone that the robot can see from its current position), ob(all obstacles(in polygon form) that the robot has seen), goal(goal).
- This function calculates blind regions hidden by obstacles(arcs of the circle zone where the robot is currently standing that the robot can't see) by calling `def get_closed_sights(center, robot_vision, ob)`.

- After that, the *scan_around* calculates open sights(arcs of the circle zone where the robot is currently standing that have no obstacle in the robot way to those arcs) base on the calculated closed sights above by calling *get_open_sights*(center, radius, goal, closed_sights).
- Output of the *scan_around* function is local vision information of the current position of the robot.

def *check_goal*(center, goal, config, radius, t_sight):

- Input of this function are the current position of the robot(center), robot sight radius(circle zone that the robot can see from its current position), goal, blind regions from the previous step, configuration object containing robot configuration parameter.
- This function checks if whether the robot has reach the goal or not. If not, checks if the robot “see” the goal(e.g the goal is in the circle zone and is not in any blind region).
- The Output of this function are 2 flags **r_goal**, **s_goal** indicate the status if the robot has reached the goal or has seen the goal.

def *get_local_open_points*(open_sights):

- The Input of this function are **open sights** gotten from *scan_around*() function.
- Base on open sights information, this function extracts “open points”(Meaning that, if we have an open sight(arc), we will choose a point on that arc as the potential moving point) and reduce computing effort by take only 10 digits after the floating point.
- The Output of this function are open points(local to the current position of the robot).

def *get_active_open_points*(local_open_pts, traversal_sights, robot_vision, center, goal):

- The Input of this function are local open point(local_open_pts) gotten from the previous step, all traversal points(points the robot have gone through) and their local vision information, robot_vision, current position(center) and goal.
- This function return status(whether went through or not) of every local open points.

def *ranking*(center, pt, goal):

- The Input of this function are the current position of the robot(center), the point need to be ranked, and goal.
- Node **Robot Global Vision Update** will use this function to re-rank every potential moving points that haven’t been traversed.

def *graph_add_lOpenPts*(graph, center, lActive_OpenPts):

- The Input of this function are graph, center and local active points.
- This function updates the travel graph.

def pick_next(ao_gobal):

- The Input of this function are all open points that are still active(I.e not been traversed).
- This function will return the potential moving point that has the highest score.

def BFS_skeleton_path(graph, start, goal):

- The Input of this function are all travel graph(graph), the current position of the robot(start) and the next point to move to(goal).
- This function returns a series of points represent a path from the current position to the next point. The first point of the path is the current position of the robot, and the last one is the next point to move to. Other points are points that the robot has gone through. Since the path is not optimized, the next function will return a more optimized path.

def approximately_shortest_path(skeleton_path, traversal_sight, robot_vision):

- The Input of this function are the un-optimized path calculated in the previous task, all traversal information(traversal_sight) and robot sight radius(robot_vision).
- This function will return a more optimized Path from the current position of the Robot to the next point. The result optimized Path is still a series of points, the trajectory between 2 adjacent points is a straight line.

Finally, node **Robot Global Vision Update** will send this optimized path to node **robot motion node**. Node **robot motion node** will instruct the robot to move to the next point base on the received Path.

9 Catkin and Cmake

9.1 Catkin build system

Catkin is the build system of ROS. Catkin basically use Cmake, which had the build environment described in the "CmakeLists.txt" file in package folder.

Catkin build system makes it easy to use ROS-related builds, package management, and dependencies among packages.

9.2 CMake and CMakeList.txt

Cross-platform Make (CMake) is an open-source, cross-platform family of tools designed to build, test and package software. Catkin, which is the build system of ROS, uses CMake by default. The build environment is specified in the "CMakeLists.txt" file in each package folder. The file "CMakeLists.txt" is the input to the CMake build system for building software packages. "CMakeLists.txt" file must follow this format:

1. Required CMake Version (*cmake_minimum_required()*)
2. Package Name (*project()*)
This is the name of the package which is specified by the CMake project function.
3. Find other CMake/Catkin packages needed for build (*find_package()*)
If the project depends on other wet packages, they are automatically turned into components (in terms of CMake) of catkin, so it make easier.
4. Enable Python module support (*catkin_python_setup()*)
To make the scripts accessible in the devspace, packages need to define the installation files in a file called "setup.py" in the project root. The "setup.py" file uses Python to describe the Python content of the stack.
5. Message/Service/Action Generators (*add_message_files()*, *add_service_files()*, *add_action_files()*)
If the project uses a message/service/action, it must be add to package by use commands below, then catkin will generate programming language-specific files so that one can utilize messages, services, and actions in specific programming languages.
6. Invoke message/service/action generation (*generate_messages()*)
After generating a macro, this command will call to macro that invokes generation.
7. Specify package build info export (*catkin_package()*)
This is a catkin-provided CMake macro. This is required to specify catkin-specific information to the build system which in turn is used to generate pkg-config and CMake files.
8. Libraries/Executables to build (*add_executable()*, *target_link_libraries()*, *add_dependency()*)
For C/C++ code: *add_executable()* is used to specify an executable target that must be built. *target_link_libraries()* function to specify which libraries an executable target links against. *add_dependency()* is used to create an explicit dependency on the automatically-generated message target so that they are built in the correct order.
9. Installing Python scripts and modules (*catkin_install_python()*)
For Python code, this command is used for CMake to determine which files are targets and what type of targets they are.
10. Install rules (*install()*)
install() is often used to add "include" folder, which includes header C/C++ files and "launch" folder.

10 Demo

1. Firstly, run Master

```
$ roscore
```

2. Then, in a new terminal, export robot model and run Gazebo simulator

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

3. In a new terminal, export robot model and run Slam node

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_slam turtlebot3_slam.launch
slam_methods:=gmapping
```

4. In a new terminal, run Robot Motion node

```
$ rosrun automotive_robot robot_motion_node
```

5. In a new terminal, run Robot Global Update Vision node and input goal position.

```
$ rosrun automotive_robot RobotGlobalVisionUpdate.py
-gx goal_x -gy goal_y
```

Example: The robot runs autonomous from start position to goal have position {50.0;50.0}

```
$ rosrun automotive_robot RobotGlobalVisionUpdate.py
-gx 50.0 -gy 50.0
```

References

- [1] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, TaeHoon Lim (Dec 22, 2017), *ROS Robot Programming*, 1st ED, Chapter 2, pp. 10-15.
- [2] Jason M. O’Kane (2013), *A Gentle Introduction to ROS*, pp. 14-33.
- [3] Binh Tran-Thanh, An Phan-Thanh, Hoai Tran-Van (2020), *Autonomous Robot: A Path Planning In Environment Of Uncertainty*.
- [4] Catkin "CMakeLists.txt" tutorial: <http://wiki.ros.org/catkin/Tutorials>
Last visited on 06-12-2021 22:13.
- [5] Justin Huang, *ROS tutorial series*:
<https://www.youtube.com/playlist?list=PLJNGprAk4DF5PY0kB866fEZfz6zMLJTf8>
Last visited on 06-12-2021 22:16.
- [6] How To Import a Python Module From Another Package:
<https://roboticsbackend.com/ros-import-python-module-from-another-package/?fbclid=IwAR19YyNgLXAhgkUa36GuVv0Q5yBm9HR7nOHmhO5LCuox9W1KV-6z42q9ziw>
Last visited on 06-12-2021 22:22.