CLEMSON UNIVERSITY INTERNATIONAL CENTER FOR AUTOMOTIVE RESEARCH

Table of Contents

Table of figures

Abstract:

In a future intelligent factory, a robotic manipulator must work efficiently and safely in a Human–Robot collaborative and dynamic unstructured environment. Autonomous path planning is the most important issue which must be resolved first in the process of improving robotic manipulator intelligence. Among the path-planning methods, the Rapidly Exploring Random Tree (RRT) [1] algorithm based on random sampling has been widely applied in dynamic path planning for a high-dimensional robotic manipulator, especially in a complex environment because of its probability completeness, perfect expansion, and fast exploring speed over other planning methods. Therefore, an autonomous obstacle avoidance dynamic path-planning method for a robotic manipulator based, called RRT, is proposed. A path optimization strategy based on the maximum curvature constraint is presented to generate a smooth and curved continuous executable path for a robotic manipulator. The proposed method not only provides great practical engineering significance for a robotic manipulator's obstacle avoidance in an intelligent factory, but also theoretical reference value for other type of robots' path planning [2].

Introduction:

Robot Operating System (ROS) [3] is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. Gazebo is a 3D simulator, while ROS serves as the interface for the robot. Combining both results in a powerful robot simulator. Gazebo enables to create a 3D scenario on the computer with robots, obstacles and many other objects. Gazebo also uses a physical engine for illumination, gravity, inertia, etc. In general, a robotic manipulator must also avoid dynamic obstacles autonomously at the same time, such as a sudden entry from a human, which requires the manipulator to accomplish dynamic path planning. Path planning is defined as that a non-collision continuous path for a robotic manipulator can be found from an initial pose to a target pose in a configuration space in which the manipulator's constraints must be satisfied. Among the path-planning methods, the Rapidly Exploring Random Tree (RRT) algorithm based on random sampling has been widely applied in dynamic path planning for a high-dimensional robotic manipulator in a complex environment. RRT has probability completeness, perfect expansion, and a fast exploring speed, so there is no need to map the obstacle from the task space to the configuration space. For the above advantages of using the Rapidly Exploring Random Trees algorithm for path planning, we went for developing a local path planner in the gazebo environment.

Methodology:

The RRT algorithm with the Turtlebot 3 burger model simulation in the gazebo world was planned to have its motion control with the RRT algorithm for expanding the nodes for the path and a path search algorithm like A star algorithm for finding the optimal and the shortest path to reach the nodes

and a low-level controller like a Proportional Integral Derivative (PID) [4] controller to help in achieving the motion control of the robot. To achieve this logic, we figured two ways and the methodologies are as follows:

Approach 1 (Preliminary):

The turtlebot3 burger model was exported to the gazebo environment and a sample world called the turtlebot3_world was taken for the test purpose. The gazebo environment was taken for visualization and a screenshot image was taken from the top view for reference. Then the image was taken for detecting the lines and edges using the edge detection feature in MATLAB. When the edges were detected, the image was saved in a binary format with the value '1' representing the obstacle zone which would be liable to collision when the bot moves into the location. The binary image's intensity values which represented the 0 and 1 values for the C_free and C_obstacle spaces was stored in the csv format and the file is accessed for getting the collision in the image coordinate. We also found the center point of the image coordinate and used it as a reference to convert the image coordinate into the gazebo environment coordinate. This center point location will indeed represent the offset value that we need to set for getting the gazebo coordinate values. Moreover, from the image we found out the radius of the cylindrical objects as to be 15 units and the original value of 0.15 units in gazebo environment urdf file. Hence the conversion unit was found to be 0.01 to get the image coordinate to the gazebo coordinate. Then the values were read, and the nodes were plotted on the image to check for collision for the values that can be reciprocated to the gazebo coordinate later. Finally, the nodes were added to the collision free zone and if found any collision will set the collision flag to true.

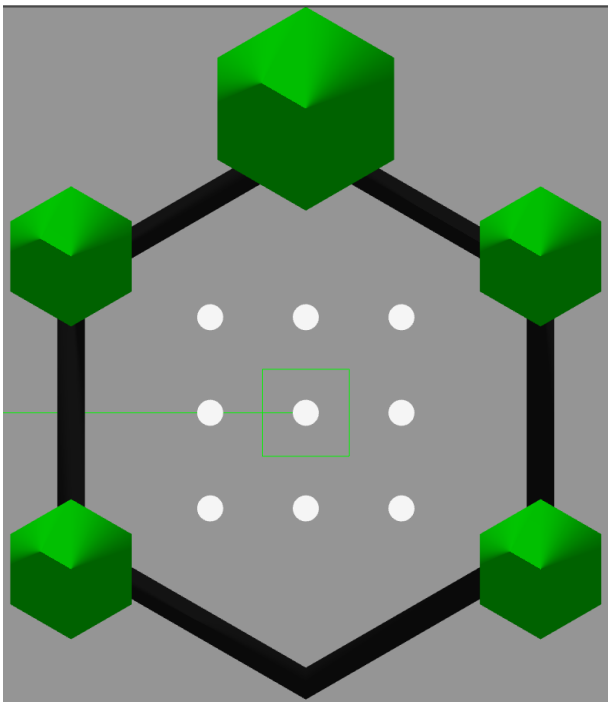

Figure 1 Image coordinate from top view

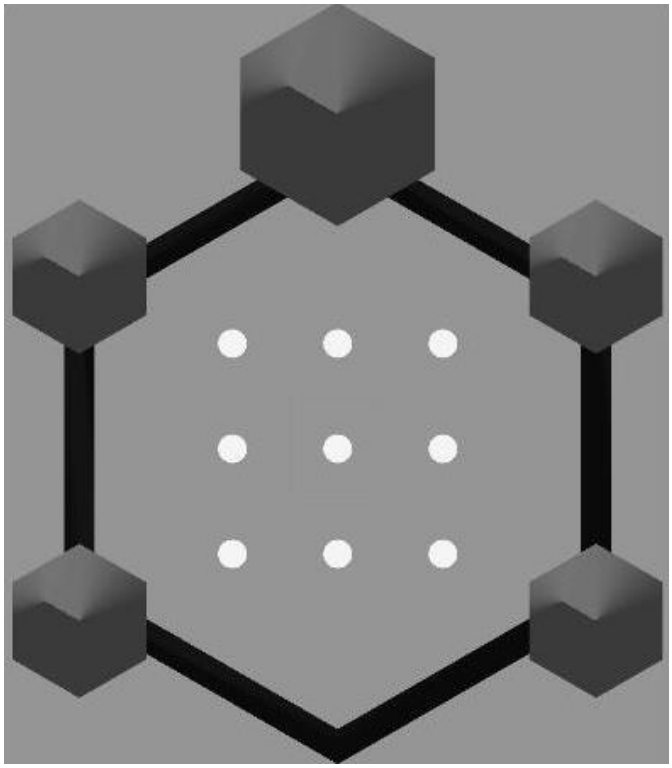

Figure 2 Binary Image

The circles are detected which represented the radius of the cylindrical blocks using circle detection as below.

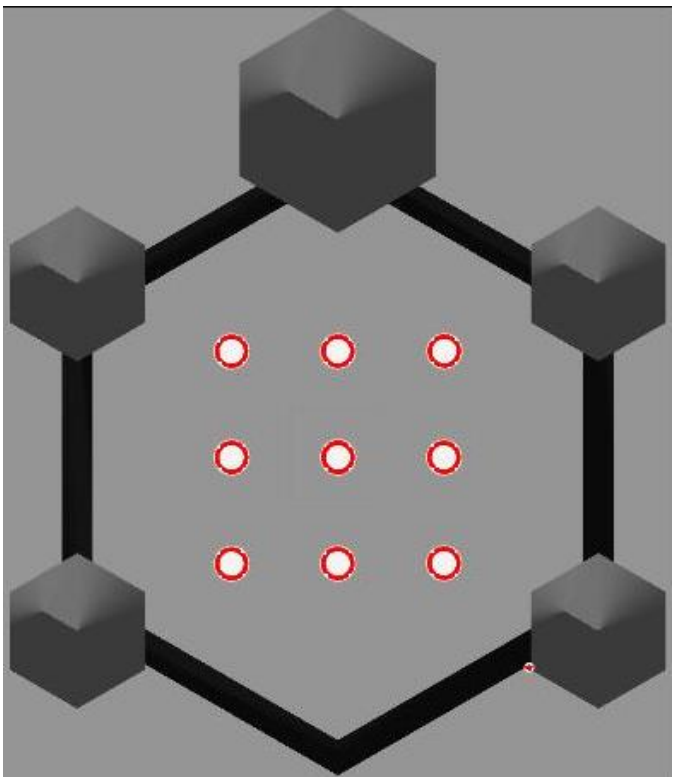

Figure 3 Circle detection for finding the conversion unit

The conversion script was written to take the image coordinate to the gazebo coordinate with the offset value to the center of the gazebo where the robot usually spawns. This also established the collision checking part for the edges and nodes which cannot pass through the collision frame of the cylindrical objects in the gazebo environment.

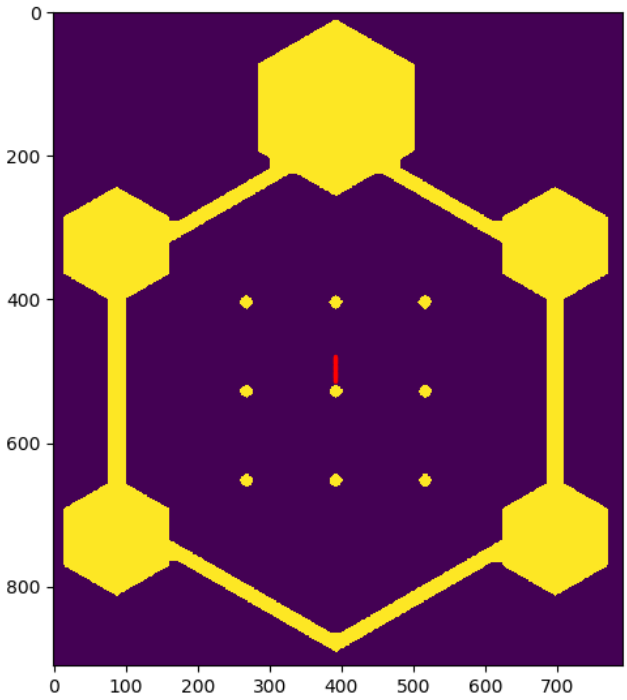

Figure 4 Nodes placed in the C_free space in Image with gazebo coordinate

Disadvantages of this approach:

The coordinate conversion from the image coordinate to the gazebo coordinate was unreliable because of the offset values. The nodes established in the image coordinate was set at a threshold of 0.5 units in the gazebo environment which would have the collision factor when the RRT algorithm is done. Hence the reliability of low-level controller will be more which should be quite opposite with the maximum reliability on RRT algorithm. Thus, the next approach with the static obstacle placement in the gazebo world and the list of the locations for those obstacles were used as a final method for the project.

Final Approach for RRT algorithm with static obstacle in the gazebo world:

RRT algorithm [5]:

Rapidly exploring Random Trees (RRT) is a path planning algorithm expanding which expands nodes randomly and finds the path from start to end point which may not be optimal path from start to end point. RRT works by randomly generating points in a specific space. These points are connected to the nearest pre-existing nodes. The points need to be checked whether they are in the free space or not before being connected to the nearest nodes. The path joining the random point and the node should also be checked in order to make sure that it does not pass through an obstacle

space. The random points are generated until a node is formed within a certain distance from the goal [6]. Once a node is formed within a certain distance from the goal, then the RRT algorithm tracks back the path form goal to start point.

$G = (V, E)$, where $V = \{q_I\}, E = \emptyset$

for i = 1 to k **do**

    $q_r \leftarrow$RANDOM_CONFIG()

    EXTEND (G, q_r)

Figure 5 RRT algorithm

RRT pseudocode: [7]

X = [0, 10] // x coordinates of the map
Y = [0, 10] // y coordinates of the map
While a node is formed with a certain distance from the goal:
[X_rand, Y_rand] = [random.uniform(0, 10), random.uniform(0, 10)] // selecting a random point
[X_near, Y_near] // Finding node which is nearest to the random point
[X_new, Y_new] // selecting a point which lies on the line joining random point and the nearest node and also within a certain distance from the nearest node.
If [X_new, Y_new] not in collision space:Join [X_near, Y_near] and [X_new, Y_new]
While not popped node == start point: Track back to the previous node

Results:

The RRT path planning algorithm is applied on a prebuilt gazebo world where the turtlebot 3 burger is used to navigate from a pre-defined start to end points. In the current project the robot starts form [0, 0] as its starting position and [5, 3.5] as its goal. The blue lines represent the total nodes expanded and the green line represents the path from start to finish. The red colored circles are the obstacles.

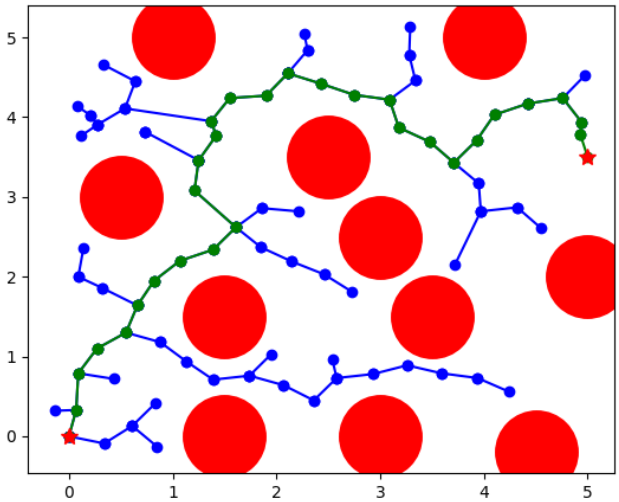

Figure 6 Plot implementation of RRT algorithm

A* path search algorithm: [8]

A* is a search algorithm that is widely used in path finding and graph traversal, the process of plotting an efficiently traversable path between points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. As A* traverses the graph, it follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. The process continues until the goal is reached. A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). It uses a distance-plus-cost heuristic function (usually denoted by f(x)) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions:

- The path-cost function, which is the cost from the starting node to the current node (usually denoted by g(x))
- An admissible "heuristic estimate" of the distance to the goal (usually denoted by h(x)).

Below is the pseudocode of the A* algorithm:

```
START ←a node with STATE==problem.INITIAL-STATE, PATH-COST=0, H-VALUE=Heuristic(node.STATE)
OPEN.insert(START) // Open is a priority queue
CLOSED = []
while !OPEN.empty()
  node = OPEN.pop()
  if node.STATE==GOAL return path(node)
  CLOSED.insert(node.STATE)
  for each child of node:
     if child.STATE not in OPEN or CLOSED
        OPEN.insert(child)
      else if child.STATE in OPEN with higher PATH-COST
          update that OPEN node with child
```

Figure 7 A algorithm Psuedocode*

Source: Dr. Ioannis Karamouzas, Motion Planning, Lecture notes

We have used A* algorithm for finding the shortest path between the start node and the goal node once the RRT algorithm expands and creates all the nodes. Once the A* algorithm finds the shortest path, this path will be realized by the TurtleBot using a local planning algorithm which will be explained in detail in the next section.

Low-level controller: (Proportional-Integral-Derivative controller) [9]

A PID controller is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID

controller continuously calculates an error value as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral and derivative terms. In practical terms it automatically applies accurate and responsive correction to a control function. The most common example is cruise control where the measured speed to the desired speed will be restored with minimal delay and overshoot by increasing the power output of the engine.

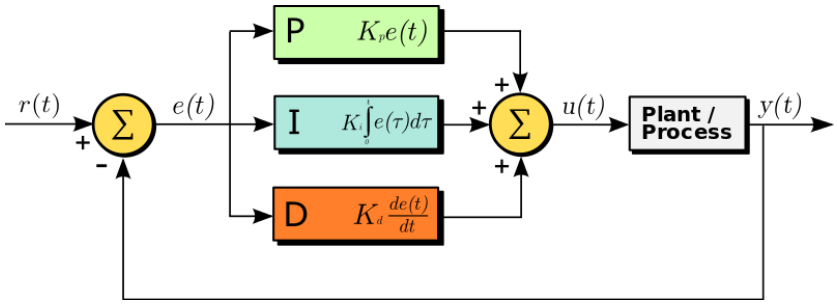

Figure 8 PID controller block

Logic of PID controller in the project:

From the PID controller, we adopted the P controller which helped in making the motion of bot smooth during the turns. First, we subscribed the '/odom' topic to get the odometry readings of the bot. The publisher that we used to publish the velocity to the robot's movement was '/cmd_vel'. This gave us the position along the x and y direction and the yaw of the bot was needed which was taken from the orientation class of the bot. Then a simple Euler conversion to get the quaternion values in rad/s which by default was deg/s. Then the theta and the distance values were computed by using the slope formula and the Euclidean distance formula, respectively. The bot will have the linear_x and angular_z velocities corrected with the product of P controller value and the distance and theta change values to get the low-level controller working. Typically, the bot will reach the nodes with this velocity and makes the motion smooth and regains its pose to move to the next node with respect to the RRT algorithm.

Future scope of the project:

Although RRT helps in expanding the nodes, we have come across few algorithms like RRT* and Bi-RRT algorithms for path planning. Sometimes, the RRT algorithm in general as issues like inability to resolve well a path planning approach when the robotic manipulator is faced with a dynamic unstructured environment in which lots of obstacles are distributed randomly and non-uniformly. For instance, the slow node extension speed of the RRT algorithm reduces the rate of convergence, which fails to meet the real-time requirement of dynamic path planning for a robotic manipulator. Furthermore, the obstacle constraint makes the path generated by the RRT algorithm's random sampling contain many unnecessary breakpoints, resulting in unsmooth and discontinuous curvature paths. Consequently, the motion tracked by the manipulator is often unstable. But, when implied with Bi-RRT algorithm, we can improve the node extension efficiency. Moreover, the various algorithms like RRT, RRT* and Bi-RRT have been planned to be implemented in the real time robots like turtlebot3 models and amazon AWS.

References

[1] [Online]. Available: https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree.

[2] [Online]. Available: https://www.sciencedirect.com/topics/engineering/path-planning.

[3] [Online]. Available: https://en.wikipedia.org/wiki/Robot_Operating_System.

[4] [Online]. Available: https://en.wikipedia.org/wiki/PID_controller.

[5] [Online]. Available: http://paper.ijcsns.org/07_book/201610/20161004.pdf.

[6] [Online]. Available: http://roboticsproceedings.org/rss06/p34.pdf.

[7] [Online]. Available: https://medium.com/@theclassytim/robotic-path-planning-rrt-and-rrt-212319121378.

[8] [Online]. Available: https://fb7024eb-a-1e6e9713-s-sites.googlegroups.com/a/g.clemson.edu/cpsc-motion/schedule/lec10.pdf

[9] [Online]. Available: http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID.

[10] [Online]. Available: http://paper.ijcsns.org/07_book/201610/20161004.pdf.