PATH PLANNING

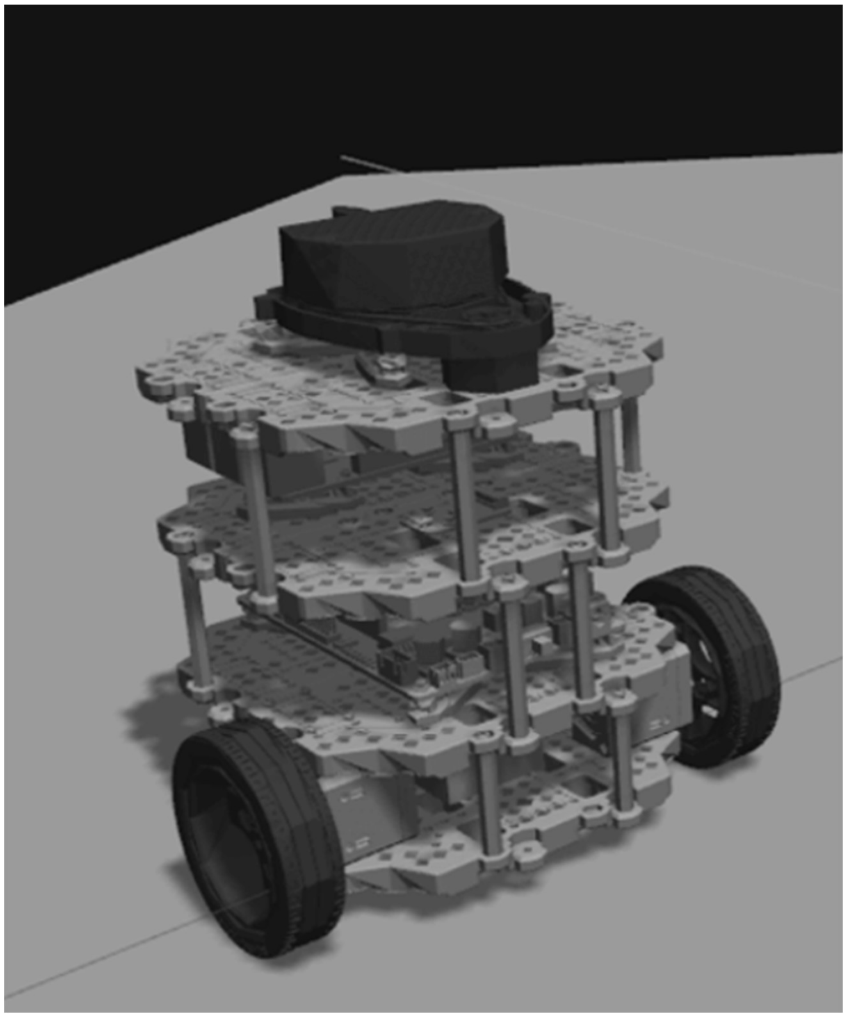
Report

A0266505A

ARUN GANDHI SHYAM KRISHNA

SUMMARY

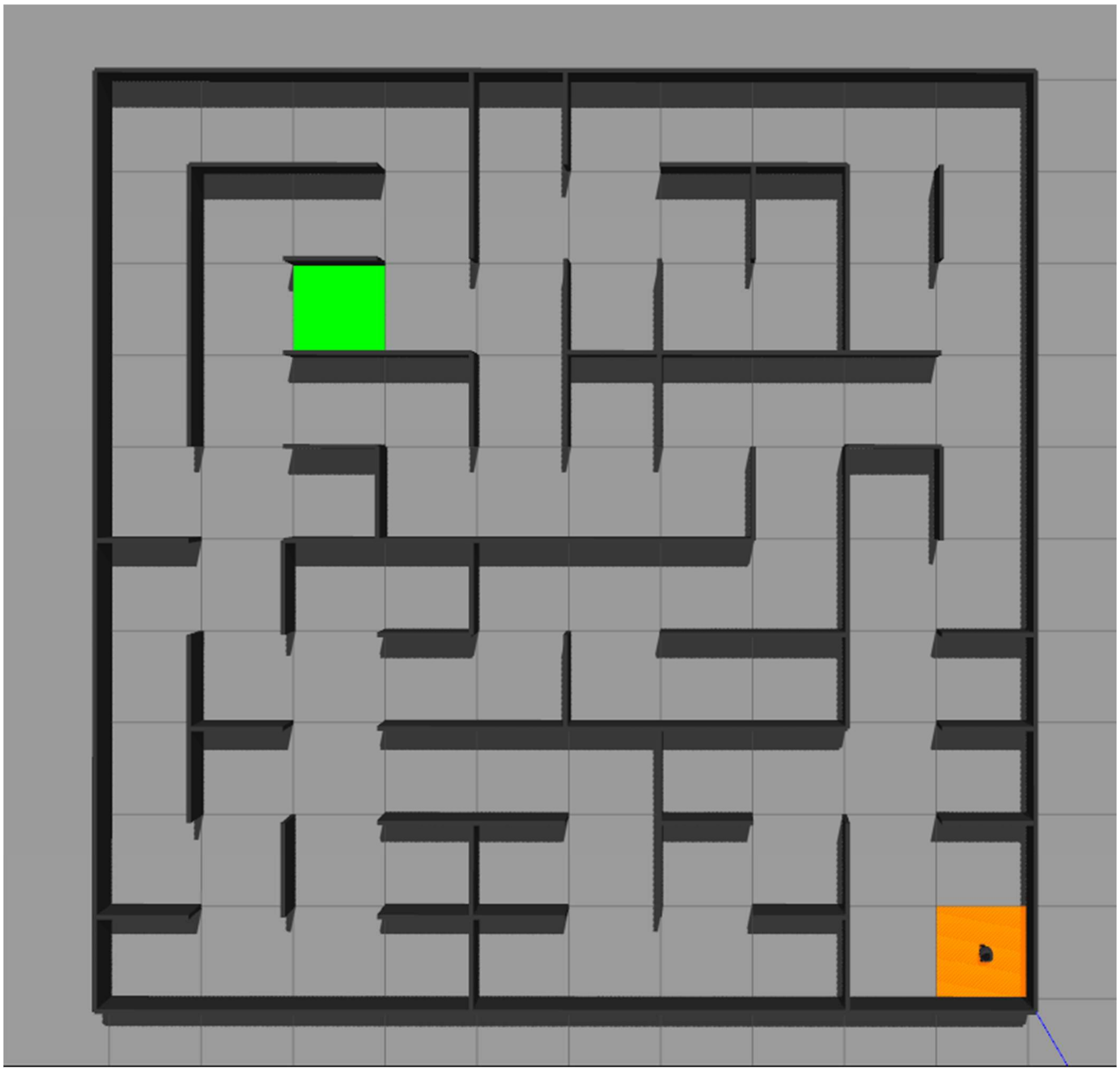
The project's goal is to develop an algorithm for the Turtlebot robot that will allow it to navigate on its own through a structured map that it is unfamiliar with. The Turtlebot knows that the maze has 10 by 10 cells, and each cell has a bigger area than the robot itself. The maze has numerous walls that are taller than the robot's sensor module, preventing it from seeing past the wall. While navigating the maze, the robot will sense the barriers and update and reroute itself to the desired location. The Turtlebot3 must move autonomously from the initial cell to the destination cell without hitting obstacles.



INITIAL CONDITIONS

SIZE OF MAZE: 10 x 10 **STARTING CELL:** (0.5 , 0.5)

DESTINATION CELL: (7.5 , 7.5)



DESIGN OF PATH PLANNING

The Turtlebot plans the shortest path at the start of the simulation. When it encounters the obstacle, it plans an alternate path and reroutes itself. There is a separate node in ROS to plan the path. Breadth First Search (BFS) technique is implanted in this model.

When the Turtlebot finds two or more possible paths, it breaks ties by following the priorities given in the code. The priority list is as North, East, South and West. The Turtlebot recognizes a particular cell with the coordinates of the midpoint of the cell.

At its designated cell site, the robot can identify the presence of barriers in all directions. As the robot learns about walls, the path distance value is updated. The location of the wall is noted and considered when determining the new optimal course of action. After scanning its surroundings, it travels to the next target place that is queued. The robot follows the computed optimal course until it learns more about the locations of the walls, at which point it recalculates the path until it reaches its destination.

BREADTH FIRST SEARCH ALGORITHM: (BFS)

Breadth-first search is a graph traversal algorithm in which all the adjacent vertices are traversed until there is no unvisited adjacent vertex. Then, we should start traversing from one of the adjacent vertices based on priority conditions. This cycle continues until we reach the desired vertex.

PSEUDO CODE:

```
Node *cur_node = queue.front();
queue.pop_front();
int cur_x = cur_node->x;
int cur_y = cur_node->y;
int cur_cost = cur_node->cost;
```

- The queue is continuously updated, and the value of the current node is stored in the queue.
- Pop removes the latest element from the queue.

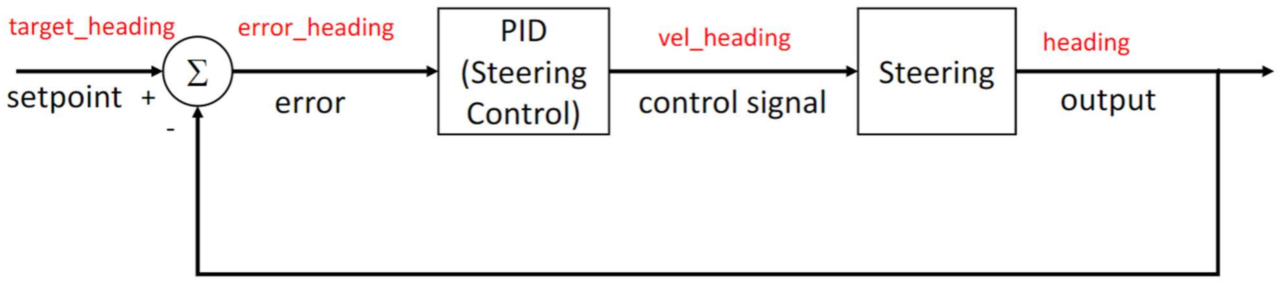
```
if (cur_x == goal_x && cur_y == goal_y)
 {
   Node *node = cur_node;
   do
   {
   path.push_back({node->x, node->y});
   node = node->parent;
    } while (node != nullptr);
    break;
  }
```

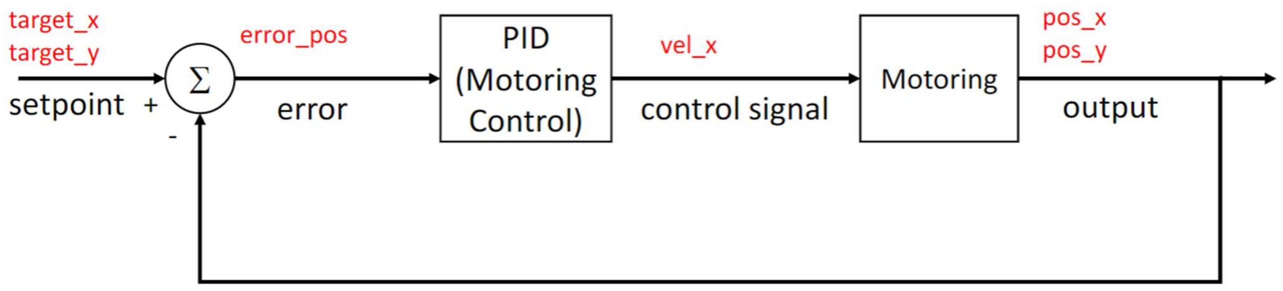
- The above loop is executed only if the current coordinates is same as of the destination coordinates.
- The loop is to push back the current node into the generated path queue, update the parent node and store it.

The `path_plan_node` publishes the coordinates of the cell to which the turtlebot must move. Thereafter, the `bot_control_node` is responsible for the navigation.

DESIGN OF NAVIGATION

Navigating the robot to the specific cell published by the path-planning node is done using PID control. The path-planning node publishes the center of the cell to identify it. When the robot is within the particular cell, the path-planning node considers that the robot has reached the cell and publishes the next target. The main task in navigating through the cells published by the path-planning node is to -- control the linear and steering mechanism of the Turtlebot using two independent PID feedback controls, whose setpoint keeps on changing. So, it is important to make the control less vulnerable to sudden shift in setpoints.





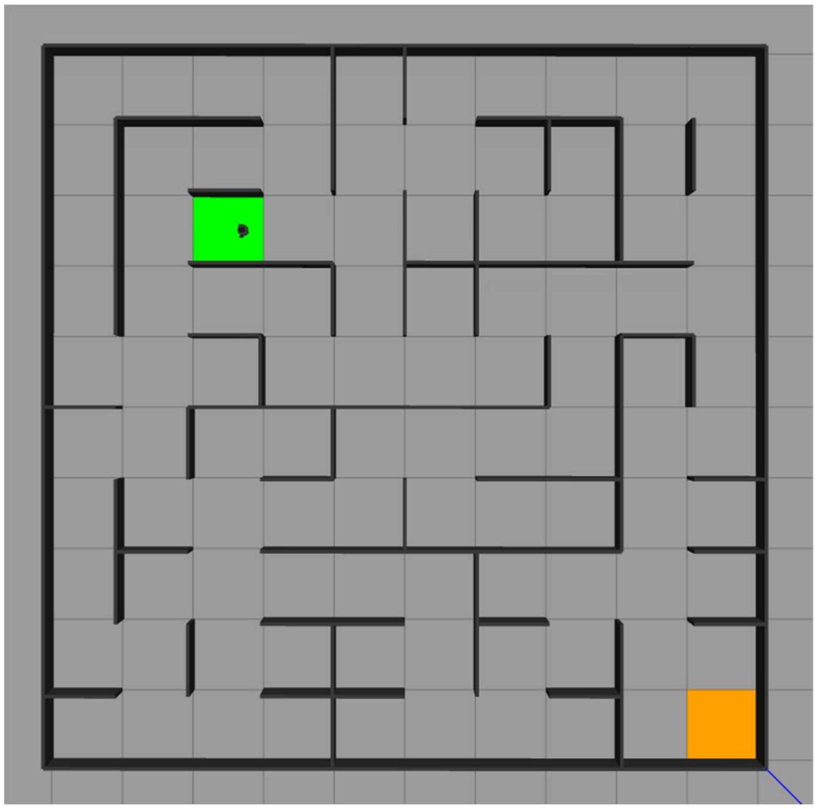
The above figure briefs the structure and design of PID contols used in navigating the Turtlebot to the published cell with the parameters as mentioned above.

HOW DID I ENHANCE THE SIMULATION?

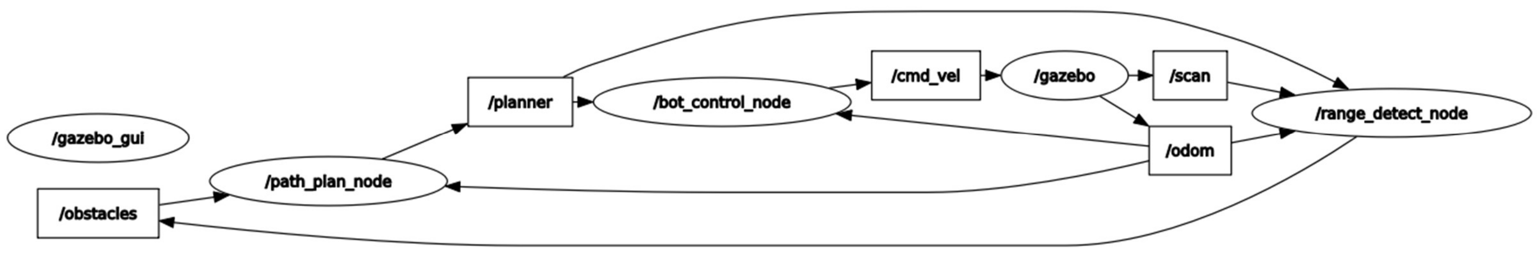
When I tuned the PID controller to a descent level, I made a few observations. During path planning there are sudden shift in the setpoints which can lead the Turtlebot to an obstacle. The Turtlebot can sense the obstacles surrounding a cell only by reaching its edge. So, I made the Turtlebot to slowdown explicitly when it is within a certain range from the setpoint (the center of the cell). This definitely improved the simulation and decreased the probability of collision of the Turtlebot with an obstacle.

PERFORMANCE OF THE SIMULATION

The Turtlebot can reach the destination cell comfortably with a good speed and time. Initially the Turtlebot did not pass all the tries. The tuning was done   better and better which eventually resulted in a better version of my simulation.



ROS NODES AND TOPICS



`path_plan_node`

This node computes the path to the destination cell. As the map keeps on updating, this node reroutes path and publish the cells to be navigated.

`bot_control_node`

This node defines the PID control for the linear and steering mechanisms of the Turtlebot. In other words, this node determines the speed and direction of the Turtlebot.

`range_detect_node`

This node detects the surrounding obstacles in the current cell.

`cmd_vel`	:	The control command to move the Turtlebot.
`scan`	:	Laser scanning data.
`odom`	:	Odometry data of the Turtlebot.
`obstacles`	:	The walls in the boundary of a cell.
`planner`	:	The route that the Turtlebot is supposed to take.