Section 1 - Stack

The stack used throughout the lab sessions involves four key components: Perception, Planner, Controller, and Localization each with their respective Python files. The stack described is used for our specific use case of controller the turtlebot both in lab and in simulation. The model of the robot and its hardware components will affect and change both the motion model of the robot, along with the sensor model based on the specifications and types of sensors utilized by the robot.

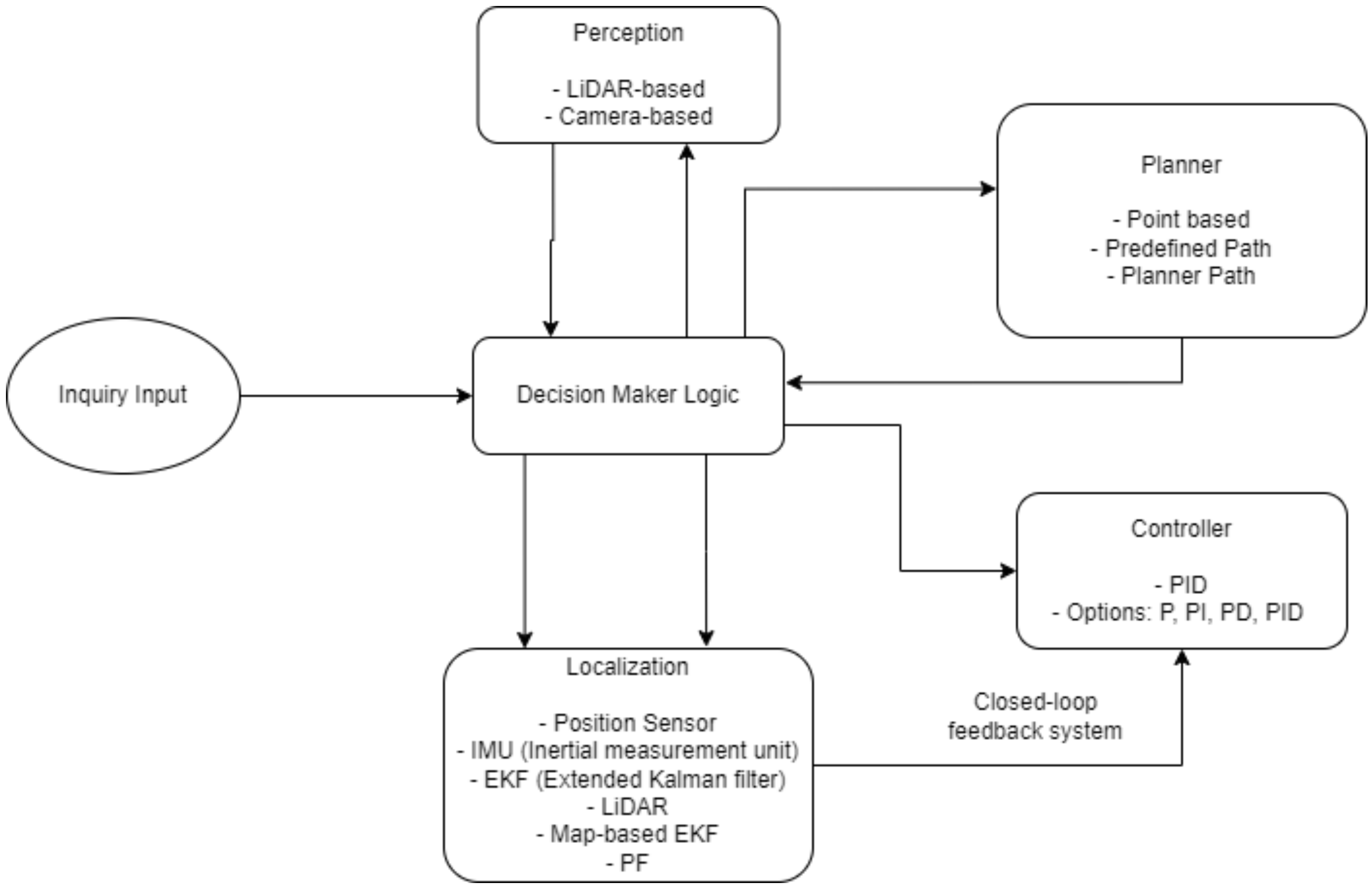
Perception is responsible for sensing and perceiving the environment surrounding the robot, using the perceived objects and environment around it to localize itself within the environment. Perception in the lab and final, can be applied to detect walls and obstacles within the traversable area for the robot. The Planner is responsible for planning the path and movement of the robot, based on the perceived environment and the desired goal position. The Planner can utilize any path finding algorithm such as A*, RRT, or RRT* to provide the robot instructions on direction in the form of x, y, and theta angle. The Controller is able to command the robot to move in space, regulating the robot's motion and controlling the motors that allow the robot to move and drive. In a closed-loop feedback controller, the controller takes in feedback from the localization module to better understand the current position of the robot compared to the goal position, correcting for errors based on information from the planner, which can be done with a PID controller. The Decisions component receives the outputs of the four key components, and combines the state context with the information gained from sensing and controls, to provide a closed-loop input back to each of the components.

The decision.py file contains the methods to design paths and control motion of the robot, along with a decision_maker class to send and receive messages for controlling motion, with instances of planner and localization classes. The designPathFor function which given a goal pose, plans a path using the selected algorithm.

The planner.py file plans the trajectory of the robot, utilizing the select path planning algorithm to determine the optimal path to follow and their coordinates . In this file, interpolation and smoothing was applied to create a smoother path with more gradual and optimal steering. The plan function calls the trajectory_planner functions, which returns a parsed and formatted path for the robot to follow to reach its desired goal pose in Gazebo.

The localization.py file performs sensor fusion between the odometry and IMU readings, along with utilizing an extended Kalman filter to estimate the robot's pose in real time. The file also subscriptions to sensor topics, updates pose estimation, and logs the sensor data into csv.

The controller.py file implements a PID controller for both linear and angular velocities, determining error from current pose versus goal pose. The trajectoryController class handles the trajectory, determining the best next goal to pursue. This file enables the robot's actual motion.



Section 2 - RRT* Implementation

RRT* works by taking in a start position, goal position, obstacle list, and metadata (expanding distance, maximum iterations, robot radius, etc.) as input to determine the optimal path of coordinates for the robot to take to reach the goal from its current position. It builds a tree of nodes data structure by randomly sampling node points within the defined area and connecting them to the tree if possible. The for loop runs through each iteration, and for each iteration, a random node is selected, and its nearest node in the tree is found. A new node is then generated by steering towards the randomly selected node from the nearest node, with a given expanding distance. The path planning algorithm is also able to check for collisions with objects in the object list, only branching the tree towards a new node if it does not collide with any object and results in a shorter path to the goal. Whenever a better path is found, the nodes are rewired so that the better path with the lower cost is prioritized. If a collision free path to the goal node is found, then the entire path from the start position to the goal position is returned as a list of coordinates.

The modifications implemented by me include the following code and actions. Choosing a random node. Creating a new node with steer. Computing costs of the nodes. Checking for collisions. Finding minimum cost nodes. Creating connection to the goal with steer. Rewiring function, including cost calculation, collision checking, and lowest cost.

The RRTStar class was integrated from the planner to the actual motions of the robot through the following process. The RRTStar planning function generates a best path from the paths explored during its iterations, which is passed into the planner.py file into the trajectory planner function. The trajectory planner function parses and sanitizes the data, reversing it to match the required input to the robot, and smoothing and interpolating the path to provide better navigation for robot motion. The trajectory planner function is called by the plan function in the same file, which is called by the designPathFor function in decisions.py. The designPathFor function is called during the initialization of the decision_maker class, which integrates the path from the planner, creating a subscriber and publisher to communicate with the robot, and calling the controller function, which in this case is the trajectory controller, to actually apply a linear and angular velocity to the robot, allowing it to move in x and y directions and rotate about angle theta.

Section 3 - Testing

Please see the appendix for plots and figures required.

The RRT* parameters were tuned through trial and error, setting different parameters for maximum iterations, goal sampling rate, and path resolution. Multiple trial runs were conducted for each combination of customizable parameter values, and each individual value was lowered and raised from the default value provided in the code, to determine its effect on the RRT* algorithm. For example, given default values of 300 max iterations, 20 goal sampling rate, and 1.0 path resolution, the two default values were held constant while the number of max iterations was raised in multiple trials, and then lowered in multiple trials. It can then be observed that a higher max iteration value results in more successful cases of reaching goal positions, especially regarding goal positions farther away, whereas lower max iterations would reach the limit before reaching the goal. Thus, the decision was made to increase the max iterations value to 500. Similar processes were applied to the other parameters in RRT*, and when tuning a parameter did not have an immediate beneficial effect on the algorithm, the value remained as the default value. Furthermore, the path was observed both through the animation, the plot, and the robot’s movement in Gazebo to determine the effectiveness and tune the optimal parameters for the plots.

The performance of the RRT* was compared against the performance of the RRT and performance of the A* algorithms and through multiple trials and iterations, it can be observed that the RRT* algorithm is able to find an optimally efficient path compared to the RRT algorithm, and perform rewiring to further optimize the tree data structure. However, from a computational perspective, RRT* is more costly due to its larger requirement on average of runtime complexity and space complexity compared to its counterpart A* and RRT algorithms.

Additionally, tuning the parameters of the RRT* algorithm as previously mentioned helped improve and optimize the performance, both in terms of effectiveness in finding a path to the goal pose, and the speed and time required to arrive at a viable path. Further performance optimizations could be made when dealing with larger maps at scale, through a distributed systems threading approach, where multiple workers handle different sections of nodes on the map and communicate with one another to simultaneously determine the optimal path.

Section 4 - Final Discussions

The integrated stack comprising the PID controller, Extended Kalman Filter (EKF), and RRT* path planner demonstrates robust performance in control, state estimation, and path planning. The PID controller provides control over the robot's motion by continuously adjusting linear and angular velocities, ensuring smooth and accurate movement towards desired waypoints. Its tuning parameters facilitate adaptable behavior, maintaining stability and responsiveness in real time. The EKF contributes significantly to state estimation by fusing sensor data, enhancing the robot's localization accuracy. This estimation enables precise path planning and navigation, for obstacle avoidance and maintaining trajectory consistency.

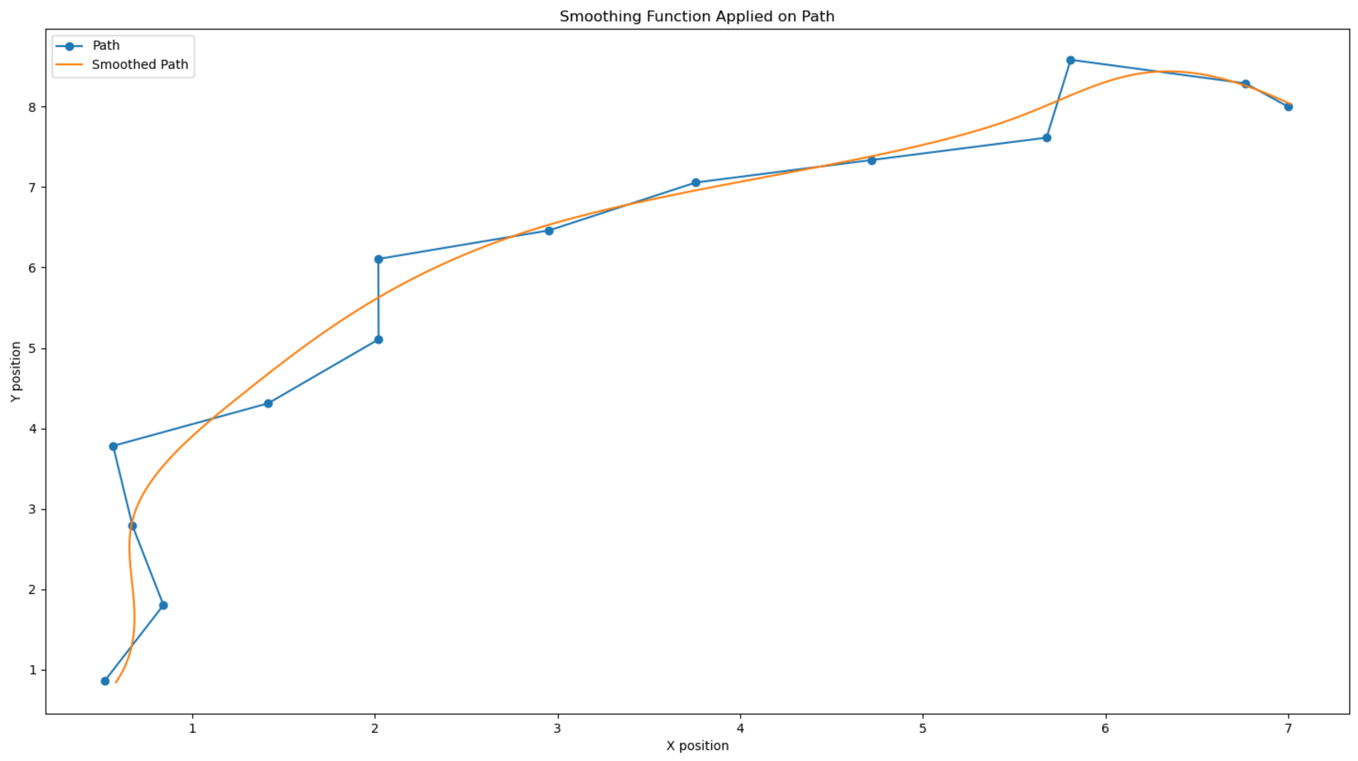
The RRT* planner, through random sampling and tree expansion, efficiently explores the state space, facilitating path planning from start to goal while circumventing obstacles. RRT* distinguishes itself by balancing optimality and computational efficiency, enabling the discovery of more cost-effective paths compared to, RRT. Its ability to perform rewiring and optimize tree structures ensures the adaptation and refinement of paths, enhancing the overall performance of navigation solutions.

The performance synergy among these components results in a effective integrated autonomous mobile robot system that excels in real-time navigation tasks. The PID controller's precise motion control, with EKF's accurate state estimation, forms a solid foundation for the RRT* planner to generate optimal and efficient paths. This allows the robot to effectively navigate through obstacles while continuously adjusting its trajectory based on real-time sensor data. Overall, the stack demonstrates reliability, adaptability, and efficiency, enabling autonomous robots to navigate safely and effectively in complex and dynamic surroundings.

Appendix

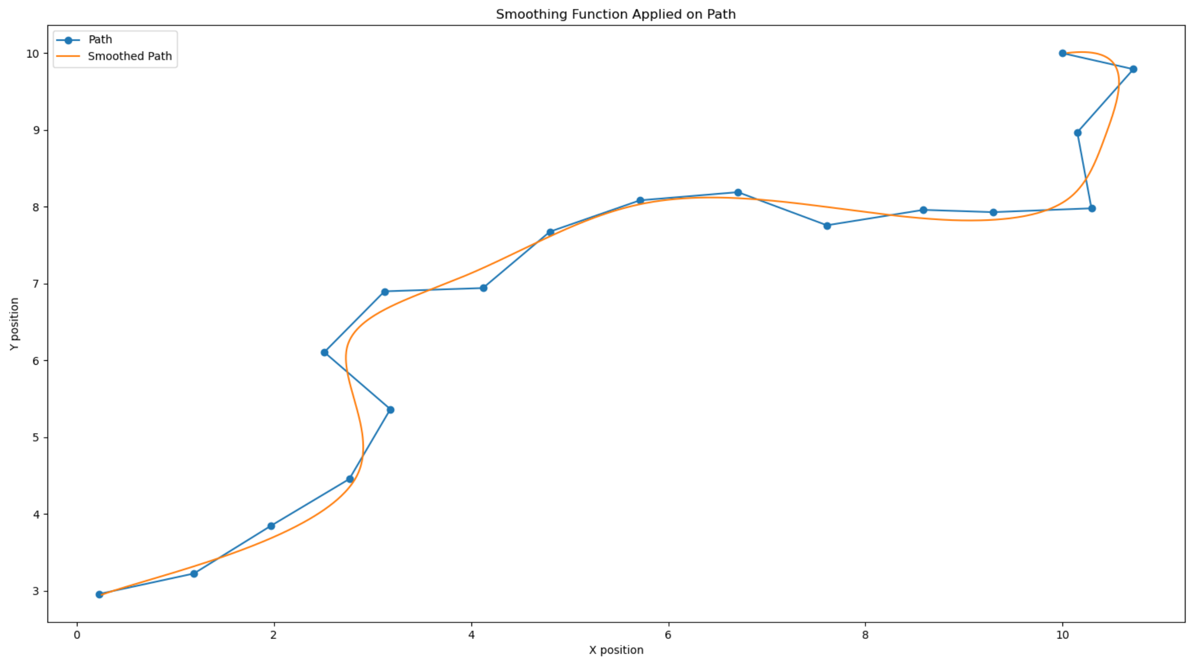
Plot of obstacle_list = [(5, 5, 1), (3, 10, 2),(7, 5, 2),(9, 5, 2), (8, 10, 1),(6, 12, 1)]

Plot of goal_pose = [7,8]

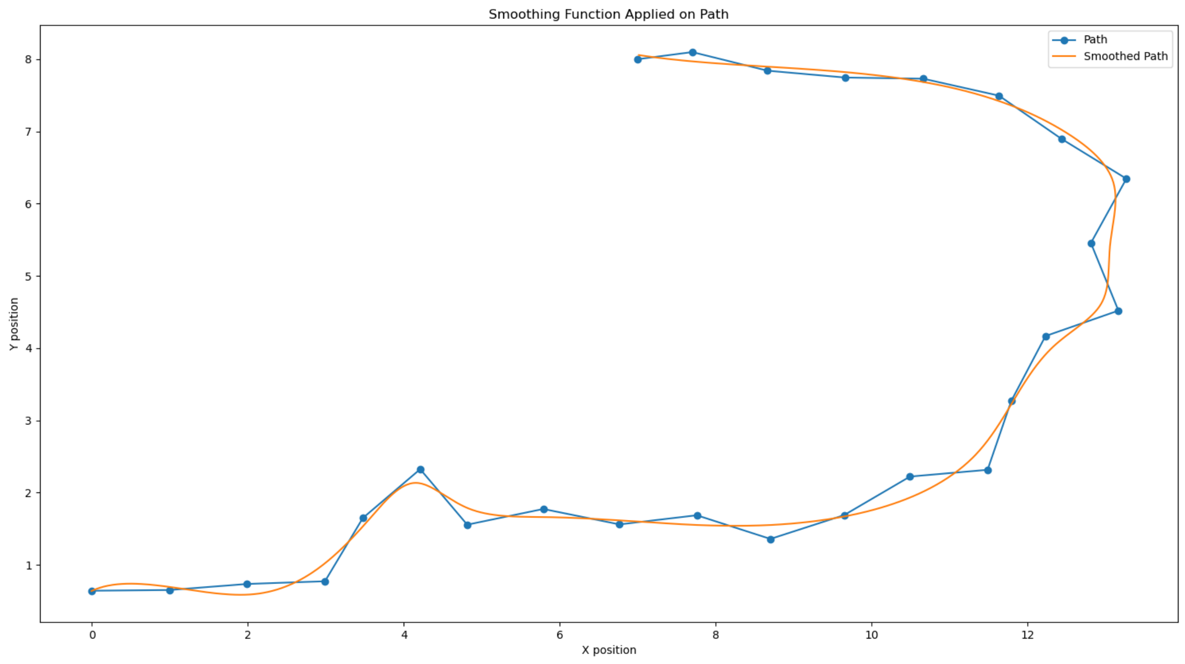


Plot of obstacle_list = [(5, 5, 1), (3, 10, 2),(7, 5, 2),(9, 5, 2), (8, 10, 1),(6, 12, 1)]

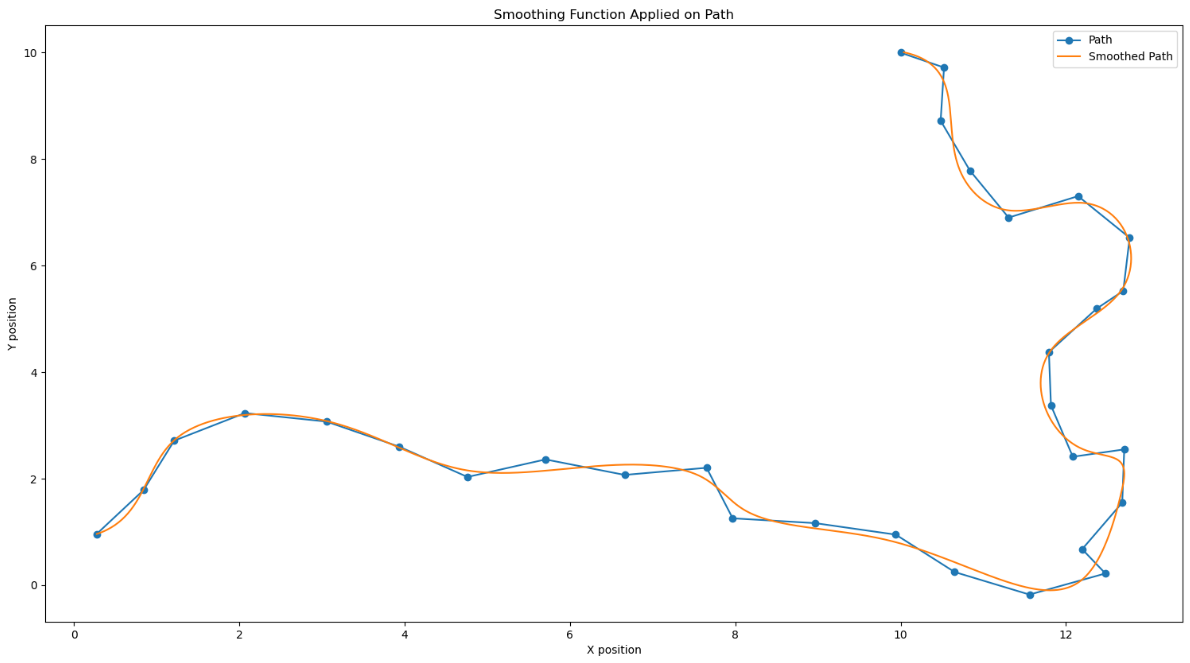
Plot of goal_pose = [10,10]



Plot of obstacle_list = [(5, 5, 1),(3, 6, 2),(3, 8, 2),(3, 10, 2),(7, 5, 2),(9, 5, 2),]
Plot of goal_pose = [7,8]



Plot of obstacle_list = [(5, 5, 1),(3, 6, 2),(3, 8, 2),(3, 10, 2),(7, 5, 2),(9, 5, 2),]
Plot of goal_pose = [10,10]


Smoothing Function Applied on Path
Path
Smoothed Path
Y position
X position
0
2
4
6
8
10
12