Integrated Path and Trajectory Planning System for Autonomous Quadcopters Operating in Obstacle-Dense Urban Outdoor Settings

I have designed an integrated path and trajectory planning system in C++, well-suited for autonomous quadcopters operating in large outdoor obstacle-dense urban environments, that is modular, well-balanced in terms of performance and accuracy, adaptable to various configurations, and capable of reliably producing near-shortest-distance, collision-free, and smooth time-based trajectories between any two given positions on large 3D bounding-box obstacle maps.

The integrated path and trajectory planning system–which takes, as input, a CSV file specifying obstacle bounding box locations–comprises four main subsystems, which work together to produce, depending on which is desired, either a position-only waypoint sequence or a time-based sequence of waypoints.

The system's obstacle data processing subsystem processes the input obstacle geometry CSV file via the ObstacleData class, which employs methods to parse the CSV file for key features, saving these to memory, and which provides methods to check the obstacle set for collisions against query points.

The integrated path and trajectory planning system has two main modules dedicated to determining direct, collision-free, position-only waypoint sequences, one for long-distance pathfinding, that is global pathfinding, or pathfinding between positions spaced more than 100 m apart, and one for pathfinding between positions spaced closer together, also known as local pathfinding.

The global pathfinding module works by first constructing, and saving to memory, a bi-directional graph data structure to broadly and uniformly represent free space (non-obstacle space), then, using the well-established A* search algorithm, searching this construction for a shortest-distance path between two desired positions, start and goal. The module functions as a multi-query planner, with the graph of free space meant to be created only one time and the search functionality to be called repeatedly against the generated graph structure.

The module's FreeSpaceGraph class, which encapsulates both the module's graph building and the module's graph searching functionality, contains a method called generateGraph, which constructs the graph, forming it in the shape of a cubic-lattice, with nodes representing points in free space and edges representing collision-free navigable routes. Edge-length is a parameter, serving as a control on the resolution of the graph structure. The method connects each graph node to, at most, six of its cubic-lattice neighbors.

The searchGraph method of the FreeSpaceClass encapsulates an implementation of the A* algorithm and will return the shortest-distance path, to the resolution of the constructed cubic-lattice, when provided with a non-obstacle start point and a non-obstacle goal point, or it will provide an empty path otherwise.

With the strength of global pathfinding module being its ability to rapidly query the map for feasible paths between far away points, another module, called the local pathfinding module, fills in the gaps, handling the task of finding more refined paths between subsequent waypoints of the generated global

plans or handling what is called “dynamic replanning,” this being rapidly constructing paths to navigate around newfound obstacles that invalidate prior generated plans.

The local planning module is centered around a custom implementation of the rapidly exploring random tree (RRT) algorithm, introduced by computer scientist and researcher Steven M. Lavalle. The RRT algorithm works by means of a stochastic process that finds a path between nearby waypoints. The custom implementation employs a goal bias parameter to promote fast convergence to the goal position. Since the RRT output path is generally jagged and not direct, the local pathfinding module employs a path shortcutting post-process step, with the result being collision-free paths that are also as direct as possible. Furthermore, the module employs a second post-process step to interpolate additional waypoints along each segment of the output path, spaced exactly 5 m apart for seamless integration with the trajectory planning subsystem.

The local planning module’s entire functionality is encapsulated in its RRT class, which accepts as input a start and goal location as well as various RRT specific parameters, including stepSize, maxIterations, and goalBias. While the implementation provides reference values that perform reasonably well on the obstacle map provided in this repository, these parameters are meant to be tuned to achieve maximum performance, especially for maps other than this one.