ILR11: Progress Review 12

Maitreya Naik

Team B: Arcus

Teammates: Logan Wan, Clare Cui, Angad Sidhu

ILR10

Apr. 18, 2016

1. Individual Progress

Since the last progress review, I was responsible for developing and testing a Rapidly-exploring Random Tree [1] Planner with straight line trajectories for autonomous navigation.

1.1. High-level RRT Planner placement

The first step to programming a planner in an existing code base was to figure out what types of inputs to send, what the backbone structure would look like and how it would interface with the system. For our case, this high-level structure is shown in Figure 1. For reference, our system functional architecture is shown in Figure 2. The RRT Planner sits in the "Find Trajectory" part of the Autonomy block.

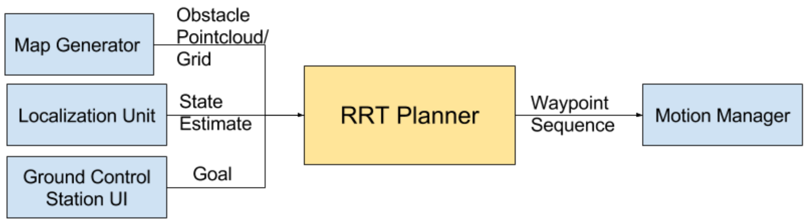

Figure 1: High-level RRT planning architecture

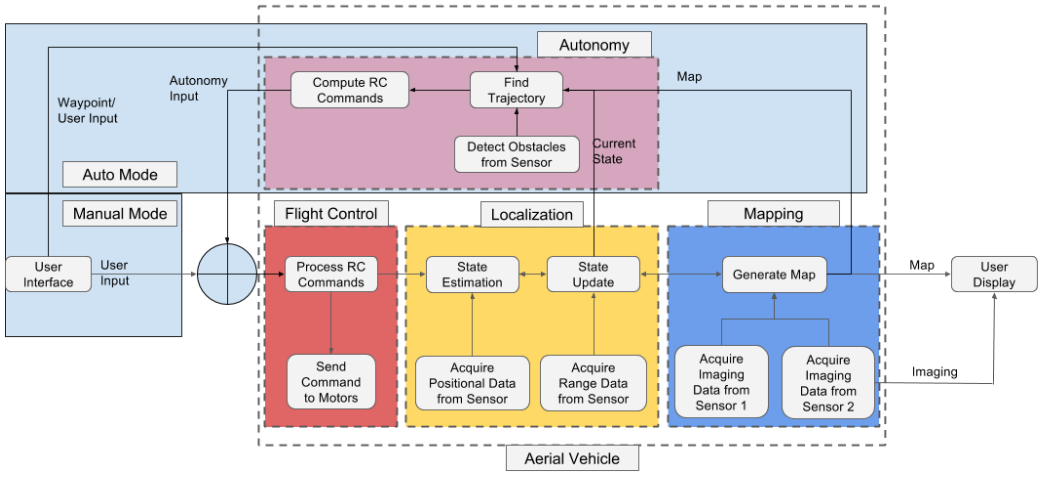

Figure 2: Arcus Functional Architecture

As displayed, the planner should take the current state of the drone as the starting point, get the obstacle point cloud or occupancy grid from the map generator and the goal to plan toward from the Ground Control Station UI. The importance of this was to understand how to interface it in the 40+ package code base from RASL. Having this high-level architecture in place, I could move on to the backbone development for RRT Planning.

1.2. RRT Backbone

The RRT backbone in C++ comprises of using a structure to define the RRT Tree – consisting of the state of the robot, a KD tree of states – each representing a vertex, a dynamic array of vertice, and a dictionary of edges. The actual planning then expands this RRT Tree starting from the start robot state to the goal state, while avoiding obstacles by referring to the obstacle point cloud or occupancy grid.

The state of the robot is defined by the x, y, z, and yaw of the drone, and 2 orders of differentiations of each of these. However, I only plan for the first 3 dimensions and keep the other values zero by default. I used a state structure from one of the existing packages, 'control_arch_utils' in the RASL [2] code base, developed by Vishnu Desaraju [3]. I forked and appended the package to enable initialization with zero (through a constructor), and basic arithmetic and comparison operations with objects of the same structure type or with a standard C++ data type like int, float or double. The structure of the RRT Planner and RRT Tree is shown in Figure 3.

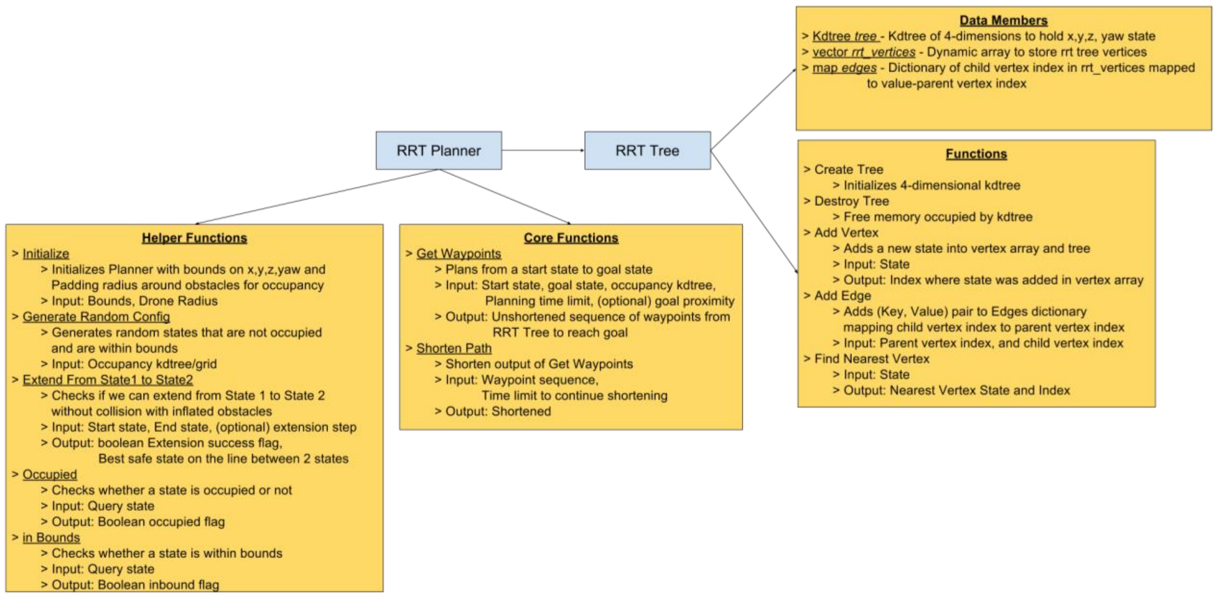
RRT Planner
RRT Tree
Data Members
> Kdtree tree - Kdtree of 4-dimensions to hold x,y,z, yaw state
> vector rrt_vertices - Dynamic array to store rrt tree vertices
> map edges - Dictionary of child vertex index in rrt_vertices mapped to value-parent vertex index
Functions
> Create Tree
> Initializes 4-dimensional kdtree
> Destroy Tree
> Free memory occupied by kdtree
> Add Vertex
> Adds a new state into vertex array and tree
> Input: State
> Output: Index where state was added in vertex array
> Add Edge
> Adds (Key, Value) pair to Edges dictionary mapping child vertex index to parent vertex index
> Input: Parent vertex index, and child vertex index
> Find Nearest Vertex
> Input: State
> Output: Nearest Vertex State and Index
Helper Functions
> Initialize
> Initializes Planner with bounds on x,y,z,yaw and Padding radius around obstacles for occupancy
> Input: Bounds, Drone Radius
> Generate Random Config
> Generates random states that are not occupied and are within bounds
> Input: Occupancy kdtree/grid
> Extend From State1 to State2
> Checks if we can extend from State 1 to State 2 without collision with inflated obstacles
> Input: Start state, End state, (optional) extension step
> Output: boolean Extension success flag, Best safe state on the line between 2 states
> Occupied
> Checks whether a state is occupied or not
> Input: Query state
> Output: Boolean occupied flag
> in Bounds
> Checks whether a state is within bounds
> Input: Query state
> Output: Boolean inbound flag
Core Functions
> Get Waypoints
> Plans from a start state to goal state
> Input: Start state, goal state, occupancy kdtree, Planning time limit, (optional) goal proximity
> Output: Unshortened sequence of waypoints from RRT Tree to reach goal
> Shorten Path
> Shorten output of Get Waypoints
> Input: Waypoint sequence, Time limit to continue shortening
> Output: Shortened

Figure 3: RRT Planner Description

Once the backbone was in place, it was simple to apply the RRT algorithm using the RRT Planner class and its member functions. We use the RRTPlanner::GetWaypoints function to do a sanity check on the start and goal state provided before beginning to plan. If the plan succeeds, we receive a vector of waypoints to reach the goal from the start. If the plan fails, we just get a one-element vector with the start state as a member.

If the plan succeeds, we can reduce the generated vector of states by using the RRTPlanner::ShortenPath function. Once the path is shortened, we visualize the trajectory in simulation and send the waypoint one after the other to the flight controller. We send a new waypoint if the drone is in hover mode and hasn't reached the goal state or if a new goal is

assigned while the drone is in tracking or hover mode. A video of the planner working in simulation can be found at: https://goo.gl/J6tH3D
Some images of the simulation are attached in Figure 4.

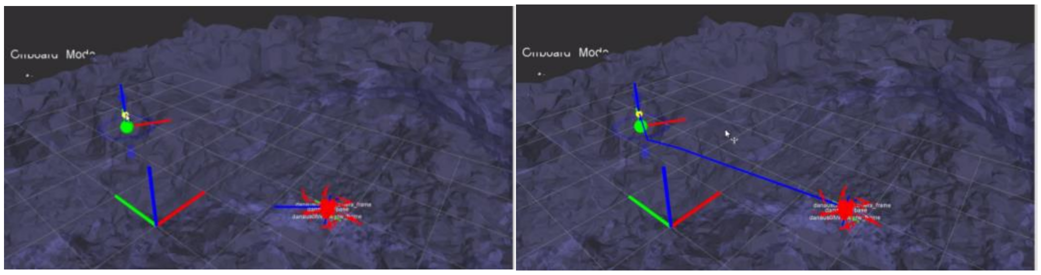

Figure 4: Snapshots of the simulation with the goal defined by the green sphere, and the drone in red

2. Challenges

The biggest challenge was scouring the RASL codebase for an existing structure to define the state of the robot, followed by making decisions on using existing tools for purposes like searching for the nearest vertex. As is noticeable, the RRT Tree and the occupancy data structure are both K-D trees. This is because K-D trees are best for searching algorithms with time complexity O(log n) rather than sequential sampling search or binary search which is O(n). This enables RRT planning, waypoint generation and shortening in less than a second (depending on planning and shortening time limits) for a volume of 15x15x2m.

3. Teamwork

Clare and Logan finished the colorization of the occupancy grid with assistance from Angad. Logan added virtual obstacles to the map (currently cylinders) and continued valuable project management. Angad took care of camera intrinsic calibration and made comparisons between BLAM mapping and Vicon state estimate integrated mapping.

4. Plan

I will be focusing on testing the navigation pipe on the small quadrotor, followed by testing basic line tracking, and finally tracking RRT waypoints. Upon accomplishing these, I will be interfacing the big hexrotor with the working trajectory generation code and finish additions to the autonomy stack.

References

[1] S. Lavalle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," Department of Computer Science, Iowa State University, Ames, 1998.

[2] C. Robust Adaptive Systems Lab, "Robust Adaptive Systems Lab, CMU," [Online]. Available: http://rasl.ri.cmu.edu/.

[3] V. Desaraju. [Online]. Available: http://www.cs.cmu.edu/~vdesaraj/.

[4] "Moses Bangura," [Online]. Available: http://www.ri.cmu.edu/person.html?person_id=4504.

[5] H. C. Ltd, "Hard Kernel ODroid," [Online]. Available: http://www.hardkernel.com/main/main.php.