

Multi-Agent Planning to Detect Obstacles and Navigate in an Unknown Environment

Agarwal, Shantnav Deshmukh, Shlok Singh, Manupriya Theocharous, Alexandros
5939933 5928516 6050425 5930901

Abstract—Rescue operations for individuals lost in a forest require drones to quickly scan a large unknown environment and quickly provide relief. We propose to work on this problem by breaking it into 2 stages. 1) multiple drones cooperatively scan the entire area with unknown obstacles communicating with each other to create a single map. 2) Once a person is found, another drone can be sent from the base to their location to provide relief by finding an optimal path.

I. INTRODUCTION

Objective:

- 1) Multiple drones plan and scan the unknown environment. The unknown environment is broken down into multiple grids and the task of the drones is to visit each grid in a cooperative manner.
- 2) The drones detect obstacles around them which are then transformed to the initial frame and updated on a common map.
- 3) When a person is found, a different drone using the common map calculates the optimal path to reach their location and provide relief.

Environment:

- 1) We will use PyBulletDrone environment for this project where cylinders are used to represent trees.
- 2) The environment spawns these trees at random locations throughout the map which is unknown to the planning algorithm. These obstacles are sensed by the robot as they come within the sensing range of onboard sensors.
- 3) A person is said to be detected when the drone is close enough to their location.

Procedure:

- 1) Our workspace is R^3 . Configuration space of the quadcopter is $R^3 \times SO(3)$. We will

plan in the workspace R^3 itself. **Should we search for the optimal path in the 3D World space or the 6D C-Space of the robot or 12D Position + Velocity space? We can use kinodynamics of the quadrotor to simplify the problem.**

- 2) We use a high level planner that directs the drones to scan the environment in a Breadth first search manner. The robots communicate with a central node that directs the search such that 2 drones do not explore the same region.
- 3) The drone gets its next target location from the high level planner. We use the RRT^* algorithm on the common map to plan the path for the drone from its current location to the target location.
- 4) We use the RRT^* algorithm over A^* because the entire map is not known. As RRT^* builds the tree incrementally, we can update the path with new information without recomputing the map entirely.
- 5) Then make the quadcopters follow this path using an off the shelf control algorithm.
- 6) The algorithm will be evaluated on the time taken to find the lost person and the time taken to provide aid to the person once found.

II. ROBOT MODEL

We have decided to use Quadrotor as our robot.

A. Working of Quadrotors

The quadrotor is a highly non-linear, six degree-of-freedom and under-actuated system. A quadrotor has two sets of counter-rotating propellers, therefore neutralizing the effective aerodynamic drag. It has four principal modes of operation: Vertical movement is controlled

by simultaneously increasing or decreasing the thrust of all rotors. Yaw moment is created by proportionally varying the speeds of counter-rotating parts to have movement with respect to quadrotor's z-axis. Roll can be controlled by applying differential thrust forces on opposite rotors of the quadrotor to have movement with respect to quadrotor's x-axis. Pitch can be controlled by applying differential thrust forces on opposite rotors of the quadrotor to have movement with respect to quadrotor's y-axis.

B. Quadrotor Model

1) *Model of a rotor*: Each rotor rotates with angular velocity ω and generates a lift force F and moment M . Moment is acting opposite to the directing of rotation.

The lift Force F and moment M of i th rotor can be calculated by:

$$F_i = k_f * \omega_i^2, \quad k_f = k_T * \rho * D^4$$

$$M_i = k_m * \omega_i^2, \quad k_m = k_Q * \rho * D^5$$

where:

k_T is thrust coefficient

k_Q is torque

ρ is fluid density

D is diameter of propeller

2) *Equations of Motion*: Total thrust and moment is the sum of individual ones in each of the 4 rotors.

Thrust: $F = \sum F_i - m g a_3$

Here, F_x are individual lift forces by the propellers and $m * g$ is the one by gravity.

Moment: $M = \sum r_i * F_i + \sum M_i$

Here, $r_i * F_i$ are the moments created by forces in quadrotor's centre of gravity and M_x are the individual moments created by the propellers.

3) *Newton-Euler Equations for Quadrotor*:

Linear Dynamics Applying Newton's Second Law for system of particles, we get (in inertial frame);

$$F = mass * acceleration$$

$$acceleration(\ddot{r}) = d\dot{r}/dt, \text{ where } \dot{r} = [u, v, w]^T \quad (3.3)$$

Since, w is the yaw-axis in which we calculate thrust, we get;

$$mass * \ddot{r} = \begin{bmatrix} 0 \\ 0 \\ -m * g \end{bmatrix} + R_{\psi} \phi \theta \begin{bmatrix} 0 \\ 0 \\ \sum F_i \end{bmatrix}$$

Rotational Dynamics Applying Euler's rotation equations, we get (in body frame);

$$M_c = {}^A dH_c^B / dt = {}^B dH_c^B / dt + {}^A \omega^B \times H_c^B$$

where, H_c is the angular momentum and ${}^A \omega^B$ is angular velocity of body B in frame A which is given by $p.b_1 + q.b_2 + r.b_3$

General vector form of Euler's equation is;

$$M_c = I\dot{\omega} + \omega \times (I\omega)$$

For Quadrotor, after rearranging the general vector form;

$$I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Let $\gamma = k_M / k_F$, $M_i = \gamma F_i$, we get;

$$I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} 0 & L & 0 & -L \\ -L & 0 & L & 0 \\ \gamma & -\gamma & \gamma & -\gamma \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Final equations using Linear and Rotational dynamics equations, we get;

$$\begin{bmatrix} T \\ \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} = \begin{bmatrix} k_F & k_F & k_F & k_F \\ 0 & Lk_F & 0 & -Lk_F \\ -Lk_F & 0 & Lk_F & 0 \\ k_M & -k_M & k_M & -k_M \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix}$$

III. MOTION PLANNING

1) *RRT Star*: We use the RRT Star algorithm first described by Karaman et al.[2] for planning an error free path between the start and goal positions. The drones maintain a constant and unique altitude at all times ensuring they don't collide with each other. The drones are constrained to maintain the same yaw angle as this does not have any influence on the simulation in our chosen scenario. Therefore, planning is done in the X-Y 2 Dimensional space to reduce search space and improve performance. The start position is the drone's current position, goal position is received from the global planner. The occupancy map is sliced to contain the start and goal locations with sufficient padding. This reduces search space for RRT^* algorithm and improves performance. We have implemented the RRT^* algorithm in Python. The RRT^* takes start, goal position, occupancy map and radius as input parameters. Occupancy map contains all the valid positions in XY space that the drone

can visit. All vertices in RRT* including start and goal are stored as Node(s). The Node object stores their position, cost, parent nodes and child nodes. $\text{cost} = \text{parent's cost} + \text{Euclidean distance from parent}$. All vertices and edges in the RRT* are stored in a networkx[1] undirected graph G. Algorithm:

- 1: Node: x_{rand} = A random point sampled from the free space (uniform distribution)
- 2: Node: $x_{nearest}$ = Point nearest to x_{rand}
- 3: Node: x_{new} = Point in free space along line (and farthest to) $x_{nearest}$ to x_{rand} with distance $\leq \text{radius}$
- 4: Node(s): x_{arr} = List of nodes with Euclidean distance from $x_{new} \leq \text{radius}$
- 5: Find node $x_{min} \in x_{arr}$ such that x_{min} cost + Euclidean distance between x_{min} and x_{new} is minimum.
- 6: Add node x_{new} and edge (x_{min}, x_{arr}) to G. Set x_{min} as the parent of x_{arr}
- 7: **for** x_{near} in x_{arr} **do**
- 8: Bool: collision = True if line between x_{near} and x_{arr} goes through an obstacle else False
- 9: Float: $new_cost = x_{near}$ cost + Euclidean distance between x_{near} and x_{arr}
- 10: **if**(collision == False & $new_cost < x_{near}$ cost)
- 11: Remove edge (parent x_{near} , x_{near})
- 12: Add edge (x_{new}, x_{near}) and set x_{new} as parent of x_{near}
- 13: **end if**
- 14: **end for**

IV. RESULTS

1 page

V. DISCUSSION

1/2-1 page

REFERENCES

- [1] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.
- [2] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.