CSE 571 Winter 2019 Project Report

Local Collision Avoidance for a Nano-drone

Melanie Anderson Chung-Yi Weng Bindita Chaudhuri

University of Washington

1 Introduction

Navigation in the real world is one of the most important characteristics of robots, particularly Unmanned Aerial Vehicles (UAVs). Automatic navigation involves careful motion planning to avoid collision with all obstacles in the navigation environment [1], and the problem becomes more challenging when there are dynamic obstacles in addition to static obstacles along the path of motion. In this project, we design an effective local collision avoidance system for a nano-drone that allows it to navigate safely through static and dynamic objects in a custom indoor environment. For static objects, we pre-compute the environment grid map using sensor data from the drone lasers and use a shortest path algorithm to get the optimal motion plan. For dynamic objects, we track the laser data input during motion and accordingly update the map and the plan when obstacles are encountered.

2 Background

Generally, the robot and obstacle geometry are described in a 2D or 3D workspace and the motion is represented as a path in configuration space. Low-dimensional problems can be solved with algorithms like sampling-based algorithms that overlay a grid or a graph on top of the configuration space [2]. Similar motion planning and obstacle avoidance tasks have been implemented on small drone platforms. In [3], a nano-drone uses a stereo camera for static obstacle avoidance. In [4], a neural network is used to estimate depth from a monocular camera on a drone platform while performing SLAM. However, these systems require a large amount of computational power to implement their tasks. For our project, we chose to constrain the problem to a horizontal 2D plane and implement motion planning and obstacle avoidance on a nano-drone platform equipped with multi-directional laser range sensors only.

3 Methodology and Results

3.1 Robot Platform

We are going to use CrazyFlie 2.0 (`https://www.bitcraze.io/crazyflie-2/`) equipped with the Bitcraze Flow Deck 2.0 and the Multi-Ranger Deck for our experiments. The Crazyflie 2.0 is a small, lightweight, open-source, nano-drone platform. When equipped with the Multi-Ranger and Flow Decks as shown in Fig.1, this platform has 6 laser sensors (left, right, front, back, up and down) which can measure distances up to 4 meters and a downward facing optic flow camera. We have chosen the Crazyflie because of its ease of use and the ability to easily modify its programming to give motion commands from and log sensor data to a base computer. The Flow 2.0 Deck and Multi-Ranger Deck provide laser ranging and optic flow sensing capability already fully integrated into the platform. To send commands and collect sensor data, we will be using rospy_crazyflie (`https://github.com/JGSUW/rospy_crazyflie`), a ROS package developed for controlling crazyflies using python, running on a Linux based PC.

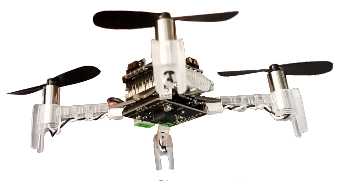


Figure 1: CrazyFlie 2.0

3.2 Environment Setup and Data Collection

We decided to create a simple maze as the environment for the drone to navigate in a conference room in one of our teammates' office building. We used multiple rectangular cardboard boxes to create a zigzag path such that the drone is required to enter from one end and reach the other end (goal). Fig. 2 shows a picture of the environment. Fig. 3a shows the drone's trajectory in blue when the drone is manually navigated to collect data from the environment, and the sensor measurements marked in red. Measurements taken at closer distances (< 1.5m) are more accurate and are marked with darker markers, and measurements taken at further distances (1.5-4m) are marked in lighter markers. We use these measurement confidence values to weigh the sensor measurements during mapping to create a more accurate map. We also observed that the drone's estimate of position (from the optic flow camera) is much better when the drone moves in the X and Y directions rather than in diagonal directions. Accordingly, we restricted our motion planning to positive and negative X and Y direction movements only.

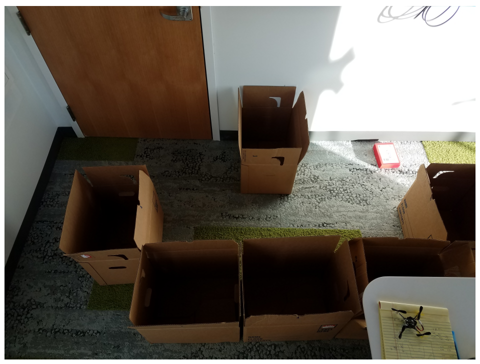


Figure 2: Environment setup for mapping. The middle box is removed during the initial mapping and replaced for obstacle avoidance.

3.3 Mapping

The map representation we used is occupancy grid map. We first partition the 2D space into $W \times H$ grid cells, and then compute the occupancy probability of each cell. We assume the cell probabilities are time independent and also do not depend on their neighbors. Hence the giant joint probability problem $p(m|z_{1:T}, x_{1:T})$ can be broken down into small ones $\prod_t \prod_i p(m_i|z_t, x_t)$. The trick of log odds is used to avoid numerical issues. We used four-beams sensor model since Crazyflie has laser measurements in four directions (front, back, left, right). For each time step and grid cell, we first identify the nearest beam, and then determine its probability as p_{free}, $p_{obstacle}$ or $p_{unknown}$, with fine-tuned parameters α (the opening angle of of the beam) and β (the width of obstacle region). In Fig. 3 we show the raw data we collected, the grid map we built, and the final grid map we get by applying thresholding.

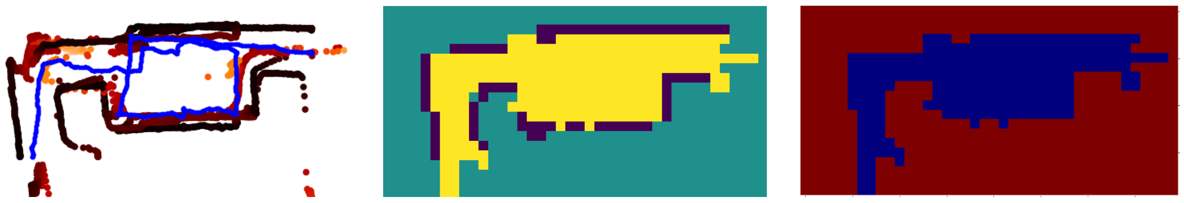

(a) Raw laser data (b) Occupancy grid map (c) Final map after thresholding

Figure 3: Raw laser data and rebuilt occupancy grid maps.

3.4 Motion Planning

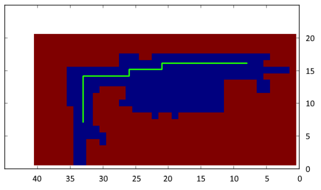
Given a 2D occupancy grid map, motion planning can be done using search-based or sampling-based algorithms. Since our map is relatively less complex and has relatively narrow passages, we planned to use weighted A* search to plan the optimal path. In simple words, the algorithm expands the neighbors of the current state (beginning from the start) at each step and updates the state to the neighbor with smallest cost $f(s) = g(s) + w * h(s)$ until it reaches the goal. We consider the (x, y) co-ordinates of each grid cell as the states and manually select the start and goal states in the map according to the desired start (top right) and end (bottom left) points of the drone in the world environment (maze). We use the Manhattan distance between states as the path cost function ($g(s)$) and the heuristic function ($h(s)$) with weight w empirically chosen to be 1 in the implementation for a smoother path.

(a) Start with no obstacles.

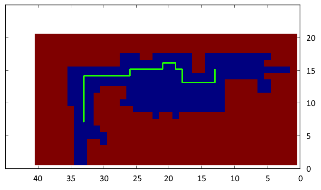
(b) Obstacle box placed.

(c) 2nd encounter with box.

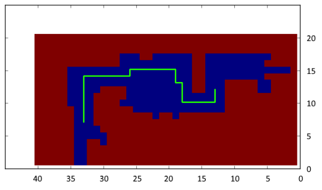
(d) Final encounter.



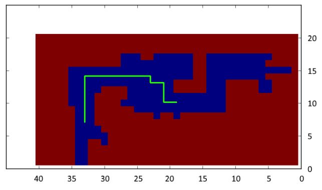
(e) A* with unedited map.



(f) 1st encounter with box.



(g) 2nd encounter with box.



(h) Final encounter.

Figure 4: Dynamic obstacle avoidance scenario: the drone updates the map and the motion path.

3.4.1 Static Objects

Since we only allow X and Y direction movements, we consider 4 neighbors ([+1,0],[-1,0],[0,+1],[0,-1]) for each state in the map. The waypoints along the estimated path are then fed to the drone to change its position at every time instant. A* performed on the static occupancy grid is shown in Fig. 4e.

3.4.2 Dynamic Objects

During the automatic motion, the drone keeps track of its sensor readings so that it can detect dynamic obstacles along its normal motion path which were not included in the pre-computed map. When detected, it stops, adds the obstacle to its map and reruns A* starting from the current position to determine a revised obstacle-free path to the goal. However, the drone cannot estimate the depth or horizontal extent of the detected obstacle. To test the dynamic obstacle avoidance scenario, we used the arrangement of the boxes shown in Fig. 4e as the environment with only static obstacles and the arrangement shown in Fig. 2 as the environment including dynamic obstacles (middle box is placed as a dynamic obstacle after the drone starts its motion). Fig. 4 shows snapshots of the drone flight, and the recalculated map and A* estimated plan to get around the obstacles. First, the drone tries to fly straight but stops when it sees an obstacle, and adds the obstacle to the map and reruns A* (Fig. 4b,4f). Since the box is a large obstacle, the new A* plan still directs the drone to fly into a space where the box is, so the drone once again adds more of the obstacle to the map, and reruns A* (Fig. 4c,4g). This happens once more when the drone sees the side of the box (Fig. 4d,4h). At last the drone is free from obstacles and is able to reach the goal.

4 Conclusion and Future Work

We have successfully created a grid map using the sensor data from the drone and used this map for motion planning to have the drone autonomously navigate through the environment while avoiding collision with static and dynamic obstacles. This work could be used to navigate incompletely mapped spaces or in disaster scenarios, such as finding a way out of a tunnel where some of the passages have collapsed. In future work, we would like to integrate the sensor data with data from a mounted RGB camera during navigation in order to navigate more efficiently in terms of stability, path optimality etc.

Contributions

We worked together to plan our project, decide and set up the environment, combine individual implementations into an end-to-end system and finally run the experiments. Melanie had previous experience with the drone and helped the others get familiar with the platform. In terms of implementation: **Bindita:** motion planning; **Chung-Yi:** environment mapping; **Melanie:** data collection and system integration.

References

[1] LaValle, Steven M., *Planning algorithms,* 2006, Cambridge university press.

[2] `https://en.wikipedia.org/wiki/MotionPlanning`

[3] McGuire, K. et al.,*Efficient Optical Flow and Stereo Vision for Velocity Estimation and Obstacle Avoidance on an Autonomous Pocket Drone,* IEEE Robotics and Automation Letters, vol. 2, no. 2, 2017

[4] Yang, X. et al., *Reactive obstacle avoidance of monocular quadrotors with online adapted depth prediction network,* NeuroComputing, vol. 325, pp. 142-158, 2019