Executing RRT Paths with the AR.Drone Quadrotor

Benjamin D. Newton

***Abstract*—Quadrotors have become a popular micro aerial vehicle (MAV) for research, and entertainment. Unfortunately, they are often remote controlled by a human user. Numerous other applications would be possible if a greater level of autonomy could be reached. As a first step toward a more autonomous quadrotor, I implement a rapidly exploring random tree which computes a path through a maze. The generated path is then used to control a real quadrotor in a real environment, using a freely available state estimation framework.**

***Index Terms*—Quadrotor, AR.Drone**

I. Introduction

THE popularity of Unmanned Aerial Vehicles (UAVs) has increased significantly over the last ten years. UAVs are no longer used only for military activities, but also for many varied applications including movie production, and live entertainment. Micro UAVs (MAVs), those which have a mass of less than about 1 kilogram and a diameter of less than about 1 meter, are leading this growth. Especially popular are quadrotors(aka quadcopters), which are MAVs with 4 rotors mounted at the ends of a crossmember.

The hardware and low-level control systems for quadrotors have improved and matured, but high level software is now needed to make them more autonomous. To reach their full potential these systems must be fully autonomous, and have the ability to plan and execute paths through any environment. One path planning algorithm which works well at finding paths through difficult environments is Rapidly Exploring Random Tree (RRT) [1]. The goal of this research is to implement RRT to plan paths for a quadrotor through a simple maze, and execute those paths on a real quadrotor.

The remainder of this paper is organized as follows, Section II gives some historical information on quadrotors, and an overview of quadrotor research. Next, Section III describes the AR.Drone platform and the inputs and outputs of my implementation of RRT. Section IV then details the implementation, and Section V describes the results. Finally, Section VI describes the conclusions and potential future work topics.

II. Related Work

A. Quadrotor History

Quadrotors may seem like a new idea, but surprisingly the first quadrotor flew in the early 1920s. The huge flying machine was built for a US Army Program by Jerome-de Bothezatd, and saw several short flights before the program was canceled. The Flying Octopus, as it was nicknamed only moved forward with a favorable wind, and suffered from control difficulties and a high pilot workload. It seems not a coincidence that the quadrotor has re-emerged now that we a computer can take care of the low level control. Despite its shortcomings, the machine was quite an achievement, and predates the first traditional helicopter by more than 14 years.

B. Newton is with the Department of Computer Science,University of North Carolina at Chapel Hill, Chapel Hill, NC, 27599 USA e-mail: bn@cs.unc.edu.

B. Quadrotor Research

Numerous papers have been published recently related to quadrotors. It is outside the scope of this paper to give a full account, but some highlights include the work done at the University of Pennsylvania in trajectory generation [2], building structures as a team [3], and flying in formations [4]. Also worth mention are the juggling [5] and cooperative ball throwing and catching quadcopters [6] at ETH Zurich.

C. Research using the AR.Drone

Several systems also exist for conducting research using the AR.Drone Dijkshoorn [7] has developed a framework for more sophisticated development and simulation of the AR.Drone, but this framework requires an understanding of the Unreal game engine, and is not well documented or easy to set up.

Engel, et al. [8] from the Technische Universitt Mnchen (TUM) have recently released a state estimation and control framework they have developed for the AR.Drone. The framework is built upon Robot Operating System (ROS), and includes a module which implements Parallel Tracking and Mapping (PTAM) for state estimation.

III. Overview

A. The AR.Drone Platform

As mentioned above, quadrotors are now becoming affordable and simple enough for consumers. The AR.Drone is one of the first widely available and affordable Micro UAVs on the consumer market. The AR.Drone, developed and sold by Parrot based in France, is marketed as a platform for playing augmented reality games, but it is also a great platform for robotics research. It is especially attractive because of its reasonable price of about $300 USD.

The AR.Drone consists of four rotors attached to a cross shaped body frame, with sensors and a processor housed at the center. It is furnished with several sensors, including an accelerometer, an ultrasonic altimeter, and two cameras (one facing forward, and the other facing down). Using the sensor data and its onboard intelligence, it can automatically takeoff, land, and hover at a given distance above the ground. User input is then required to move the quadrotor from this default position. Adjusting the pitch down causes the quadrotor to move forward, and rolling to the right, causes the quadrotor to move right. In addition, a user is able to independently adjust the yaw, and the height above the ground. Communication

with the quadrotor is via WiFi, with the quadrotor acting as a wireless access point to which a laptop, smartphone, or tablet can connect. Figure 1 shows a picture of an AR.Drone.

Fig. 1. A photo of the AR.Drone.

Parrot recently introduced version 2.0 of the AR.Drone with extra sensors and higher resolution cameras, however this work was done using the original version of the AR.Drone. Parrot includes an SDK [9] and API with limited functionality enabling control of both versions of the quadrotor from a PC. Unfortunately I found this API and its examples difficult to compile and get working. I instead chose to use the framework [8] developed by Engel, et al. at the Technische Universitt Mnchen (TUM).

B. Inputs and Outputs

The framework to control the quadrotor implements a simple scripting language. A script describing the desired path of the quadrotor through the environment is generated offline, and then imported and executed at run time by the control framework. This script is the output from my RRT path planner.

The inputs to the RRT path planner include the details about the environment, the size of the quadrotor, and the start and goal locations. These inputs, and their default values are detailed in Table III-B. Their meaning will become more clear after reading the next section.

Environment witdh	500 cm
Environment height	500 cm
RRT increment	50 cm
Sub-increment	1 cm
Drone max width	64 cm
Buffer	20 cm
Objects	(see figure)
Start position	0,0
Goal position	-150, -325

TABLE I
INPUTS TO RRT PATH PLANNER

IV. METHODS

The first step was to repair my AR.Drone, which had been damaged badly by a crash. This involved dismantling the entire drone and replacing the cross member and gears. This gave me an opportunity to see the guts of the hardware, and learn how everything fits together. Next I updated the firmware and the control application for the AR.Drone and flew the drone to ensure that it was fully functional after the repairs.

A. Failed attempts

I downloaded the recently released version 2.0 of the AR.Drone SDK. The example application failed with an segmentation fault, so I had to modify it to get it working. I created a Linux application to enable control of the drone through a GUI. I was able to get the drone to take off and land, but never move in a lateral direction. There is limited documentation and example code, and many of the comments are in french. Upon further investigation I found that there were issues with the new version of the SDK.

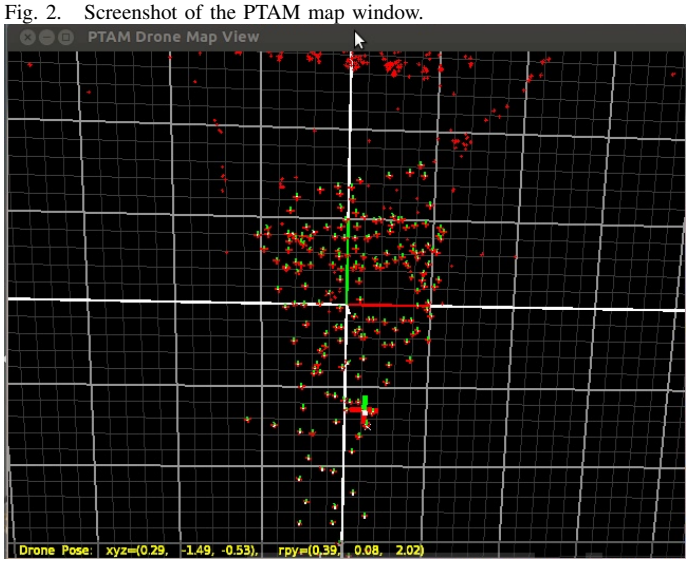
I downloaded version 1.8 of the SDK, and after solving some build issues I was able to finally get the drone to move from it's hovering position.

B. AR.Drone TUM

As mentioned above, I finally settled on using the AR.Drone TUM framework developed in Germany. It is built on the Robot Operating System (ROS). To install ROS I ended up having to install a different version of Linux. After installing ROS and setting up a workspace I then installed the AR.Drone TUM framework. After solving a few build issues, it finally compiled, and ran. It was a bit tricky to make sure the environment was set correctly and all the components were started in the correct order, so I created a script to start everything automatically. It is important to have the AR.Drone turned on, and have network connection before starting AR.Drone TUM.

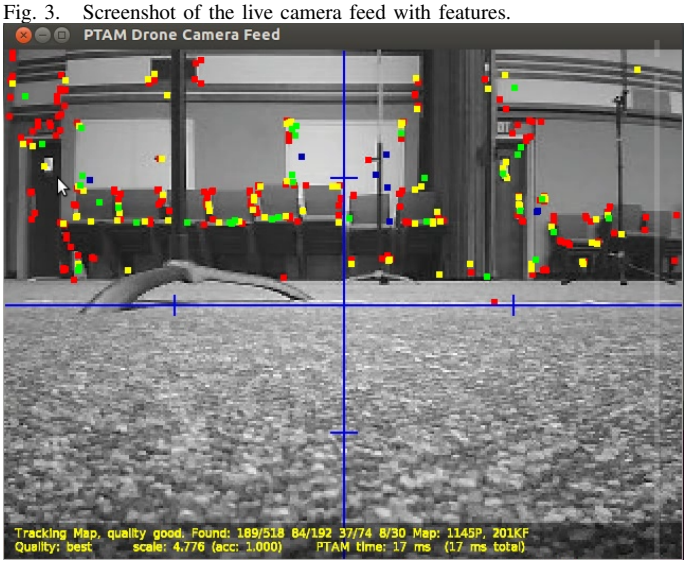
The system includes three components: the state estimation module, the controller module, and the GUI. On start-up three windows appear. The first is a 3D map showing a point cloud of the detected features in the environment and current estimated position of the quadrotor. Figure 2 shows a screenshot of this window.

Fig. 2. Screenshot of the PTAM map window.



The next window is a live view from the front facing camera on the AR.Drone. Figure 3 shows a screenshot of this window. The colored dots on the image represent the currently detected features.

Fig. 3. Screenshot of the live camera feed with features.



The final window is the GUI, which is used to control the system.

At start-up PTAM must be initialized by pressing the space bar and then moving the quadrotor vertically and pressing the space bar again. This allows the system to obtain an initial set of features. This set is then expanded as the drone sees more of the environment. Once initialized the system continually estimates and refines the position of the drone. It is important to have the front camera facing a scene with features. A large solid colored wall with no texture will not work well.

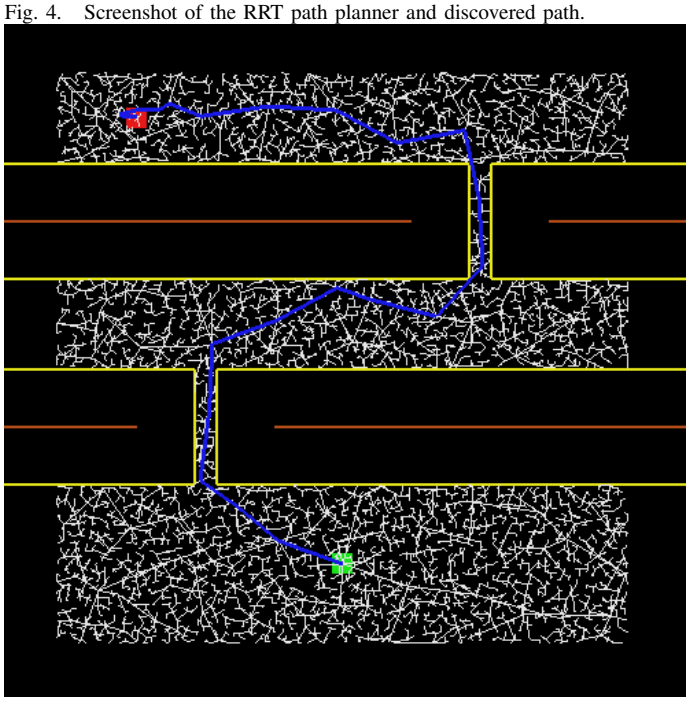
While the drone is flying, the system can be in one of two different modes. Manual mode, where the user controls the drone using the keyboard or joystick, and autopilot mode. In autopilot mode the system executes a specified script. The following is an example of a simple script which sets up some parameters, commands the quadrotor to take off, move through the environment, and then land.

```
takeoff

setReference $POSE$
setMaxControl 1
setInitialReachDist 0.2
setStayWithinDist 0.5
setStayTime 1

goto -1 0 0 0
goto -1 1.5 0 0
goto 1 1.5 0 0
goto 1 3 0 0
goto -1.5 3 0 0

land
```

C. RRT Path Planner

In order to generate a script for the TUM system which will move drone through a maze, I implemented the Rapidly Exploring Random Tree (RRT) algorithm. This algorithm creates a random tree rooted at the starting position, which randomly expands throughout the free space until the goal is reached.

Fig. 4. Screenshot of the RRT path planner and discovered path.



My implementation begins by setting the goal, start, and object positions. I assume the robot is a point robot, ignoring orientation. To ensure there are no collisions I expand the objects by half the maximum width of the quadrotor, plus a small buffer distance. A tree is initialized with a single vertex, at the start position. I use the boost graph library adjacency list container to store the tree. Each vertex in the tree has a position associated with it. I use OpenGL to display the objects and their expanded shapes.

Each time OpenGL calls the idle function I call the growTree() method which randomly picks a free position in the environment. A free position is one which is not inside any expanded object. Next I find the vertex in the current tree which is nearest the random position. The tree grows from this vertex towards the free position by a certain increment. For example, if the increment is 50, a new vertex is placed 50 units along the line from the vertex in the tree towards the random position. To ensure no obstacles are overlooked, the positions along the increment are tested at a given sub increment (for example every 1 unit), to ensure none are inside of the objects. If the path is free, an edge is added to the tree and assigned a weigh which equals the Euclidian distance between the vertices. This procedure of adding new edges and verticies to the tree continues until the tree nearly reaches the goal.

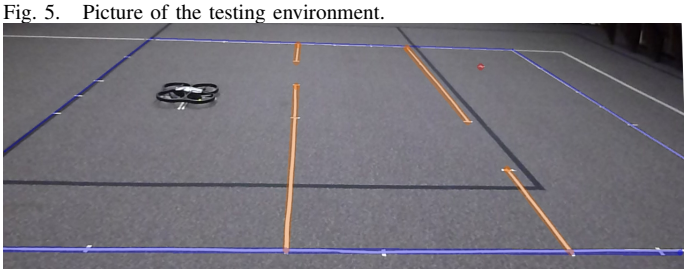
If the new position is near the goal, a final edge to the goal is added to the tree. At this point I use the boost graph library's implementation of Dijkstra's shortest path to find the shortest path through the tree from the start node to the goal

node. This path is then output to a file with the appropriate header and footer to make it match the script format for the AR.Drone TUM system. Figure 4 shows a screenshot of the path planner and the path discovered using RRT.

V. Results

To test my path planner I created a simple maze with narrow passages, once the objects are expanded. The orange lines in figure 4 represent thin walls, while the yellow lines represent the expanded objects. I then mapped the same map with tape on the floor of a gym. Figure 5 shows the map on the real floor. The blue lines are the limits of the 5m x 5m square environment map, and the orange lines are the walls. The other lines should be ignored.

Fig. 5. Picture of the testing environment.



After some initial bugs were worked out, I flew the quadrotor through the environment, but it overshot all of it's target. This was because the scale was being estimated incorrectly, which is a common issue with vision systems. I manually adjusted the input script distances by a scaling factor, and then got good results. See the video attached with my code submission.

Figure 6 shows a planned RRT path, and Figure 7 shows the execution of that path in the real world.

There is good agreement between the two paths. I also added poles at the corners of the walls to ensure the quadrotor went through the door openings. The drone would sometimes rub up against the poles or encroach on the boundaries, but in general it stayed within the limits of the environment.

I ran about 20 separate tests on one of 5 different maps. I also adjusted the parameters to make the drone fly more exact and slowly, or faster, and less accurate. The slow more accurate flights yielded fewer errors.

VI. Conclusion and Future Work

Several improvements can be made to this system. First, I could expand the RRT to 3 dimensions. This would allow for more interesting paths. I would also like to determine a way to allow the TUM system to be calibrated for accurate scale, given movements of a known distance in the real world.

In conclusion, I have developed a RRT implementation and run the paths it planned on a real quadrotor. The AR.Drone successfully moved through the maze many times with only minimal errors. This is one small step towards making a autonomous quadrotor.

Fig. 6. Trajectory following path through maze.

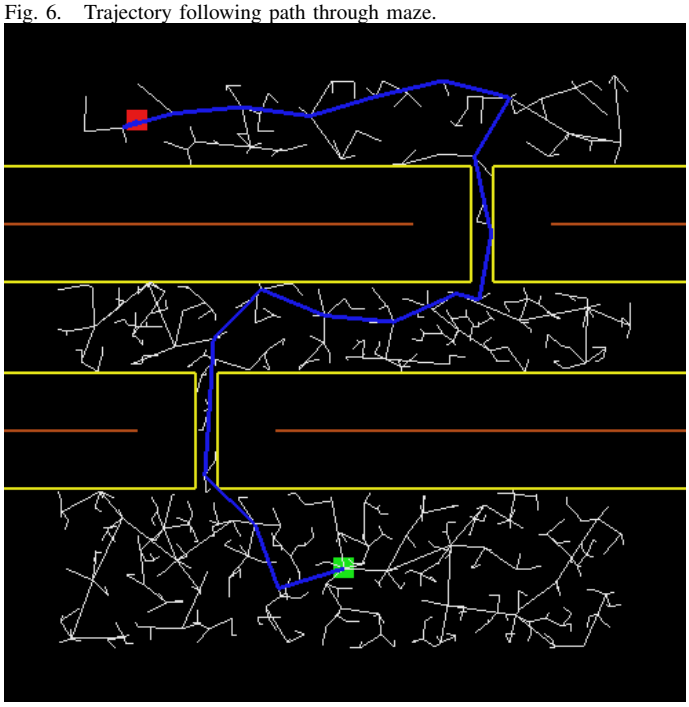
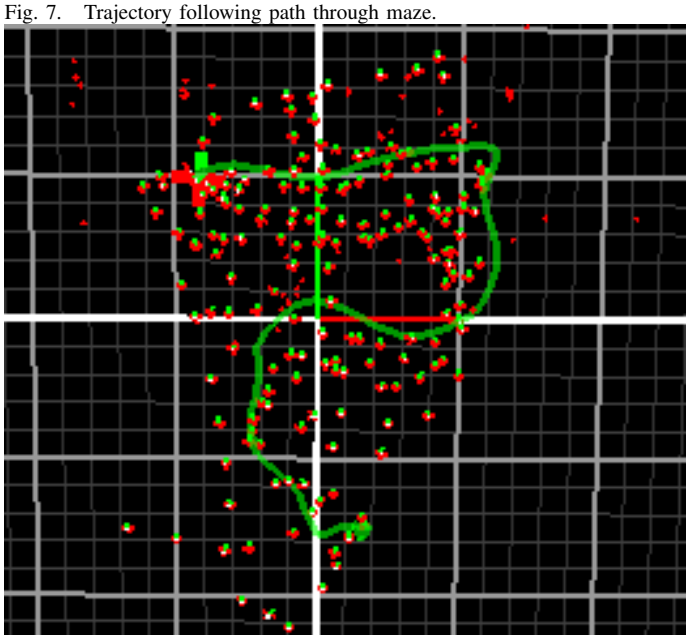


Fig. 7. Trajectory following path through maze.



Acknowledgment

The author would like to thank the researchers at TUM for making their system publicly available, the church for allowing me to use their gym for my testing, and my family for their love and patience.

References

[1] S. M. Lavalle, J. J. Kuffner, and Jr., "Rapidly-exploring random trees: Progress and prospects," in *Algorithmic and Computational Robotics: New Directions*, 2000, pp. 293–308.

[2] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *ICRA'11*, 2011, pp. 2520–2525.
[3] Q. Lindsey, D. Mellinger, and V. Kumar, "Construction of cubic structures with quadrotor teams," in *Robotics: Science and Systems'11*, 2011, pp. –1–1.
[4] A. Kushleyev, V. Kumar, and D. Mellinger, "Towards a swarm of agile micro quadrotors," in *Proceedings of Robotics: Science and Systems*, Sydney, Australia, July 2012.
[5] M. Muller, S. Lupashin, and R. D'Andrea, "Quadrocopter ball juggling." in *IROS*. IEEE, 2011, pp. 5113–5120.
[6] R. Ritz, M. Mueller, and R. D'Andrea, "Cooperative quadrocopter ball throwing and catching," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 4972–4978.
[7] N. Dijkshoorn, "Simultaneous localization and mapping with the AR.Drone," Master's thesis, Universiteit van Amsterdam, July 2012.
[8] J. Engel, J. Sturm, and D. Cremers, "Camera-based navigation of a low-cost quadrocopter," in *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.
[9] S. Piskorski, N. Brulez, and P. Eline, *AR.Drone Developer Guide*, 1st ed., Parrot, May 2011.