AEROSP 740: Fiducial Landing

Daniel Angkiat, William Cohen, Dushyanth Ganesan
Department of Aerospace Engineering
University of Michigan
Ann Arbor, United States of America

***Abstract*—This project seeks to extend and augment Quadlab to fly autonomously without the use of the motion capture system. The paper explains a methodology by which the quadrotor can achieve localization with fiducial markers or "AprilTags" to find its current state. Additionally, the project aims to guide and land the quadrotor at a predesignated landing zone by optimizing a smooth feasible trajectory on-board. The project uses ROS to implement most of the core functionality and algorithms necessary on a Raspberry Pi 4. A serial interface was established between the Pi and the BeagleBone Blue running `rc_pilot` to control the quadrotor. Ultimately, the quadrotor was able to localize successfully with delays due to the multi-device infrastructure, rendering autonomous control only semi-successful. The project required significant systems and software integration that took more time than initially foreseen. With more time, however, the team is confident that the project can be successful.**

I. Introduction

The Fiducial Landing project works to extend the functionality implemented during the quadlab into a motion capture-free localization, guidance and landing system based on landing-zone fiducials. Fiducial markers, specifically AprilTags, are conceptually similar to QR Codes, in that they are a type of two-dimensional bar code. However, they are designed to encode far smaller data payloads (between 4 and 12 bits), allowing them to be detected more robustly and from longer ranges. Further, they are designed for high localization accuracy— you can compute the precise 3D position of the AprilTag with respect to the camera [1]. The goal is to create a system that can generate global positioning and setpoints based on the known poses of grounded fiducials, in this case, AprilTags shown in Figure 1.

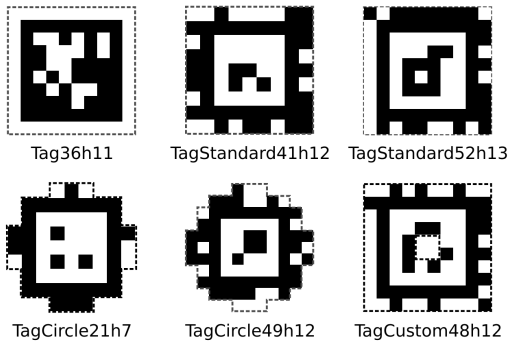

Fig. 1: AprilTag fiducial markers, 41h12 was used in this project.

Once the pose localization in the global inertial frame is achieved, a smooth trajectory for the quadrotor can be calculated and divided into reference setpoints for the controller to follow to allow for autonomous flight conditions for station-keeping and landing.

A. Hardware

The existing Quadlab infrastructure uses `rc_pilot` developed at the University of Michigan on a BeagleBone Blue (BBB) on-board the M330 quadcopter. This project requires the augmentation of the existing drone with additional hardware and software. Localization with AprilTags requires image processing and depth information, so an Intel Realsense L515 RGB-D camera was selected and added to the quadcopter facing down. Since the BBB has limited computational power and storage capacity, the team opted to use a Raspberry Pi 4 to interface with the camera and run the AprilTag detection, localization and trajectory generation algorithms.

B. Software Infrastructure Decisions

The team chose to install, learn, and use Robot Operating System (ROS) Noetic on the Raspberry Pi running Ubuntu Mate. ROS is middleware that consists of a set of software libraries and tools that help build robot applications. What makes ROS useful is its ability to run multiple threads called "nodes" and allow cross communication between threads (even on multiple devices) by the nodes subscribing and publishing "messages" to "topics". The ROS Master runs on one device that is responsible for connecting and keeping track of the different nodes and their connections through a TCP/IP server. The ROS Master removes the necessity of hardcoded IP addresses to nodes making system and software integration simpler. This was especially important for communication between the Raspberry Pi and the BBB which is discussed in this paper.

An additional advantage of using ROS is the vast amount of open-source resources and libraries available. This allowed us to interface with APIs for state-of-the-art algorithms for concepts like AprilTag pose detection and trajectory optimization without having to implement them from scratch; thus, improving the precision and accuracy of the software while keeping our goals achievable in the short time span apportioned. Figure 2 shows the overall Quadrotor software/hardware infrastructure design. Each component is discussed in detail in subsequent sections,

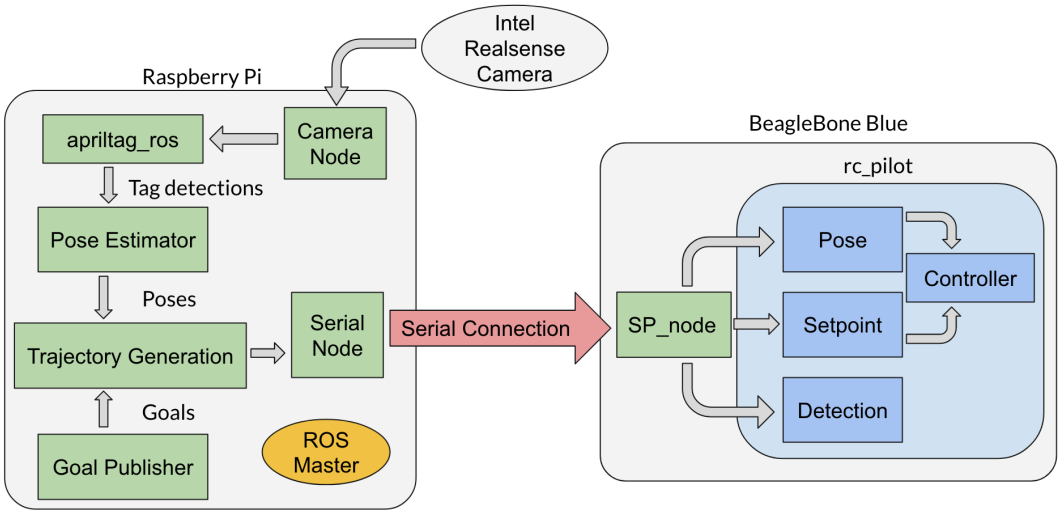

Fig. 2: Hardware and Software Infrastructure for Drone

II. Methods

In this section we will discuss the methods and code written to perform localization, guidance, and landing using fiducial tags.

A. Realsense Camera Node

The Intel Realsense L515 camera was connected to the Raspberry Pi 4 by USB. The installation of drivers and APK necessary to interface with the camera were tricky for ARM-based processors like the Pi; however, there exists documentation to manually build the binaries for Linux devices. This build and installation took some time but was successful and the performance was reasonable upon testing. Realsense cameras have an open-source camera ROS-wrapper node [2] that publishes camera data and RGB-D image frame info to topics on the ROS server. The node was installed on the Raspberry Pi for use in the project.

B. AprilTags and AprilTagROS

The AprilTag 3 library was used as the fiducial markers due its out-of-the-box functionality with ROS. The library subscribes to the data published from the Intel Realsense camera node, providing position and orientation data in the camera's reference frame. We used tags 1 through 4 of the Standard41h12 library with an 8cm size with various positions in order to simulate a well-marked landing environment. The AprilTag ROS node subscribes to the camera image and intrinsic matrix topics from the Realsense camera node, run an algorithm to detect the AprilTags in the image and convert them into a pose (position and orientation) in the camera frame. These poses are then published to a `/tag_detections` topic. Figure shows the subscriber/publisher infrastructure of the node.

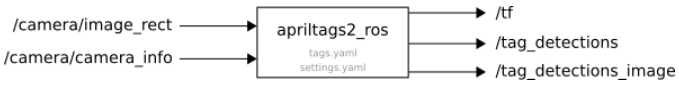

Fig. 3: Node apriltag_ros infrastructure

The Pose Estimator node then subscribes to these detection poses discussed in Section II-C. The positions in the camera frame are relative to that particular tag, requiring the pose estimator to convert the pose into a global position and orientation in the inertial global frame.

C. Pose Estimator

The pose estimator was designed to subscribe to the AprilTag node topic with messages containing their corresponding poses, and convert them into a single global position of the quadrotor in the inertial global frame for each time step. The two frames can be seen in Figure 4.

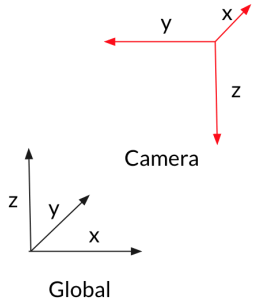

Fig. 4: Camera and Global Frame

The pose estimator is set up as a ROS subscriber to the detection array from the AprilTagROS node, which provides up to four detections per observation. For each observation, the pose is correlated to a global position of the tag ID taken from a config file. The poses of the individual AprilTags

in the global inertial frame are known and are placed on the ground by the user, therefore each tage ID is associated with a known global pose. Once the global tag location and the relative position are determined for each detection, the poses are averaged to a final global pose utilizing quaternions for orientation averages. The quaternion representation of orientations can be converted into rotation matrices with the `Eigen` C++ library and the pose can be represented in the form of a transformation matrix. T_A^C is the transformation matrix of the AprilTag (A) in the camera frame (C), similarly R_A^C is the rotation matrix and t_A^C is the position column vector.

$$T_A^C = \begin{bmatrix} R_A^C & t_A^C \\ 0 & 1 \end{bmatrix} \tag{1}$$

T_C^A is then the transformation of the camera from the AprilTag's body reference frame.

$$T_C^A = (T_A^C)^{-1} = \begin{bmatrix} (R_A^C)^T & -(R_A^C)^T t_A^C \\ 0_3^T & 1 \end{bmatrix} \tag{2}$$

If T_A^G is the known transformation matrix of the AprilTag in the Global inertial reference frame (G), then the transformation matrix of the camera, i.e. the pose of the quadrotor T_C^G can be calculated.

$$T_C^G = T_A^G T_C^A = T_A^G (T_A^C)^{-1} \tag{3}$$

Once the transformation matrix of the camera/quadrotor body with respect to the global inertial frame is calculated, the rotation matrix and position vector can be isolated. The rotation matrix can then be converted back into quaternion form and a pose message can be built. These are then published to the `/pose_estimator/pose` topic to be read downstream in the trajectory generation node, discussed in Section II-D, and directly to rc_pilot on the BBB through the serial connection discussed in Section II-E.

D. Trajectory Generation

Generating a feasible trajectory is imperative to successfully navigating to and landing at the target. Calculation of a trajectory was done on the Raspberry Pi with a ROS node using an open-source trajectory optimization library called `mav_trajectory_generation`. The node generates a trajectory which is smooth and navigable by the quadrotor. The library uses a "vertex which defines the properties of a fixed point on a polynomial trajectory path. A pair of vertices are connected with an optimal polynomial to form a "segment" [3]. Each vertex contains information about the position and derivative constraints. The start and end vertices have default derivatives of position set to zero. Figure 5 shows the polynomial trajectory with vertices.

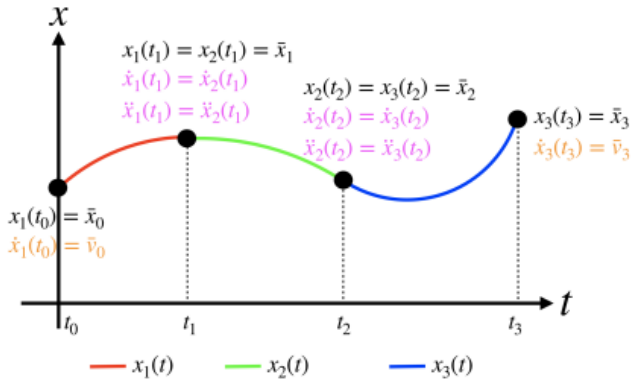

Fig. 5: Trajectory generation polynomial optimization

The vertices formed when the trajectory planning node spins are the current position of the quadrotor as the start vertex, a position with the same X-Y position at a hover height of 0.75 meters, a position 0.75 meters above the landing point, and an end vertex that is the landing zone. The node uses `mav_trajectory_generation` to define these vertices and a dimension of 10 for the polynomial to optimize through the vertices. A smooth trajectory can be obtained by minimizing the snap of the quadrotor's motion. Snap is the derivative of the quadrotor's acceleration.

The time that the quadrotor takes to traverse each segment of the trajectory can be adjusted by changing the maximum velocity and acceleration defined in the node. These parameters had to be qualitatively tuned to achieve the desired performance. Once a trajectory and its associated time for each trajectory point is calculated, the desired pose message containing the position, velocity, and orientation is published to the ROS network under the topic `/desired_state` at the correct time. These published messages can then be read by `rc_pilot` and set the desired setpoints for the controller to move through the trajectory successfully.

In addition, trajectory testing node was made for running tests during development. This node was essentially built as a state machine that allows for user input to select goal points. Three states were developed: hover, guide, and land. The hover state publishes a goal point above the current state of the quadrotor, guide published a goal point above the landing zone, and land publishes the landing pose as the goal. These goals are published to the `/goal` topic which the trajectory generation node subscribes to and calculates the optimal trajectory to the goal. The user input to set the state can be provided through the command line with SSH on the Raspberry Pi.

E. ROS Serial

ROS provides the library `rosserial` which serves as a bridge between the Raspberry Pi and the BBB through a serial USB connection. A major consideration when using ROS is the overhead that it requires to install and run. Smaller embedded devices and microcontrollers cannot install

ROS with the limited storage capacity. The ROS package `rosserial_python` implements a standard ROS node `serial_node.py` on the Raspberry Pi that sends and receives topic messages to and from the BBB running a node with `rosserial_embedded_linux`, a lightweight headers-only ROS library in C++ on the BBB. This allowed us to minimize memory and compute requirements on the BBB itself, as it should be primarily running the `rc_pilot` executable.

The BBB ROS node was implemented on the lightweight C++ server called `SP_node`. This server listens for published topics over a TCP server hosted from the Raspberry Pi serial node. Once the pose and trajectory information was received, it was processed as discussed in Section II-F. The overarching infrastructure can be found in Figure 2, whereas the serial-specific structure can be found in Figure 6.

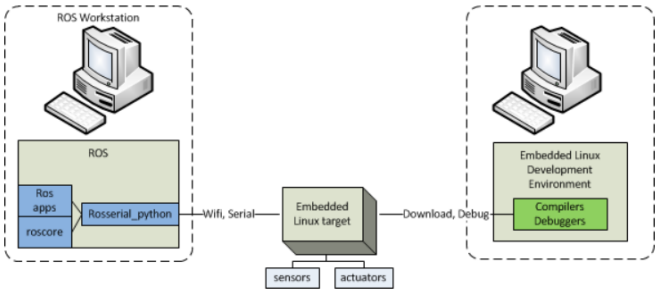

Fig. 6: Rosserial Connection Diagram

F. Controller Integration

The connections discussed in II-E allow ROS information to be transferred over serial from the Raspberry Pi to the BBB. On the BBB, there is a ROS server, `SP_node` in Figure 2, running with a set of functions for writing the pose, setpoint, and detection information. These are declared in a shared header file that is used inside the controller.

When information is transmitted over the serial connection, a series of subscribers are triggered with callbacks that write the data to memory. These memory locations are then dereferenced in shared functions between the C++ server and the C controller controlled via the header file. These are compiled into a single executable so that the C++ server is running every time `rc_pilot` is initialized so that the information is always available.

Once the server is communicating with the Raspberry Pi, `rc_pilot` calls the functions when the information is required by the controller. A new mode was implemented called `APRILTAG` that handles position estimation and setpoint generation using the transmitted data. The positions are called on every IMU interrupt in the `state_estimator` code. In the `APRILTAG` mode, the global position and orientation are written from the data supplied by the pose estimator instead of the global XBee positions. These serve as our ground truth positions for station keeping and landing. The setpoints are also called on IMU interrupts and are implemented using the `APRILTAG` mode in the `setpoint_manager` code. The setpoints are set from the trajectory generation information and all feed-forward terms are zeroed out. From here, the controller runs the standard PID controller from the quadlab [4]. The augmented controller is shown in Figure 7.

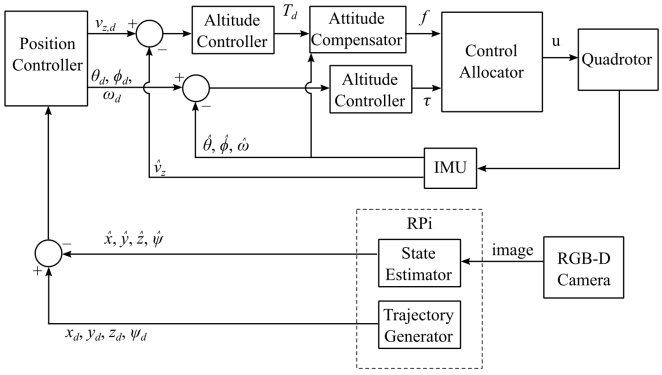

Fig. 7: Augmented control diagram with camera state estimation

III. Results

Here, we will discuss the results of our implementation and flight testing.

A. Global Position Estimation

To initially test our controllers efficacy, we ran the motion capture system in parallel with the AprilTag detections. In Figure 8, we can see that the tracking follows the rough shape of the motion capture data. This demonstrates that the AprilTag pose estimation method is functioning as a rough global positioning system for our drone.

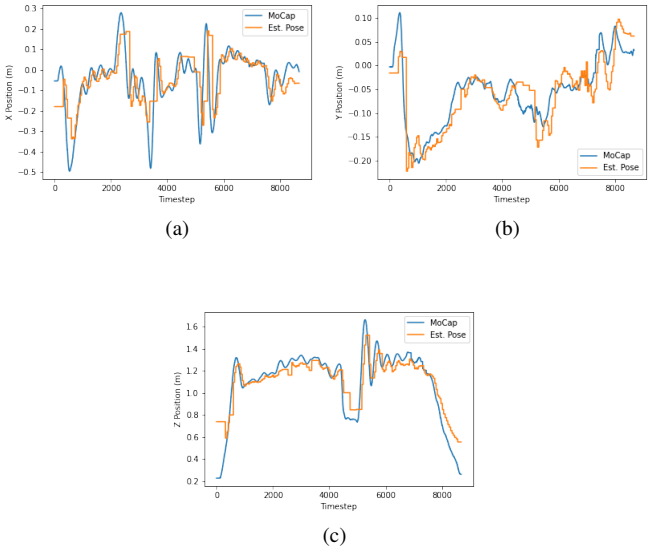

Fig. 8: Mocap vs. Pose Estimator Comparison

However, when running these flight tests we found that our average error from the global position was around 15cm. This,

as well as the root mean squared errors (RMSE) of each of our three axes, can be found below in Table I.

RMSE	X	Y	Z	Total
	0.109	0.034	0.095	0.149

TABLE I: RMSE of Flight Test Data

Given the fact that the error is within 15cm and that the shapes of our trajectories are mirroring their counterparts from the motion capture system, we can consider the integration of the Raspberry Pi and ROS into the controller was a success. However, there is further work to be done to minimize these errors that will be discussed in Section IV.

When running the controller with our station keeping and landing commands, we found that the drone required significant assistance to stay aloft. This is due to key limitations with our method of implementation. On closer inspection of Figure 9, which is a zoomed in view of 8a, we can see that the position estimator is refreshing at a rate of approximately 5 Hz. This would likely impact the stability of our controller as we expect motor refreshes at a rate of 200 Hz. This, when coupled with the lack of feed-forward terms, resulted in erratic flight behavior that often destabilized the quadrotor or obscured the view of the fiducial.

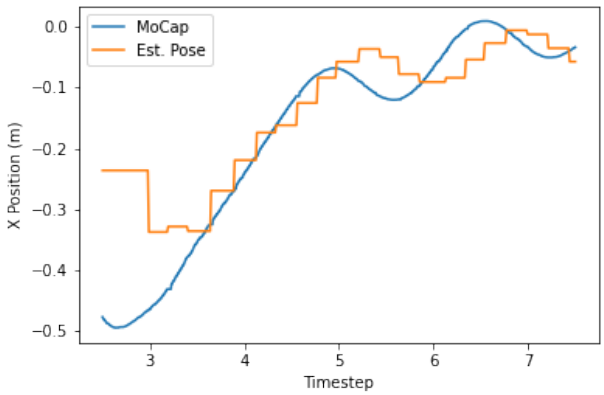

Fig. 9: Signal Delay in Zoomed Position

IV. Future Work

The limitations of this implementation stemmed from the limited time and compute power during the course. Many of the issues we saw stemmed from incomplete or inadequate implementations for the problem scope. In this section, we will discuss possible extensions of this work to finish the proposed motion capture-free localization, guidance and landing.

A. Compute Limitations

In our implementation, we utilized a 4GB RAM Raspberry Pi as the primary compute for the image processing, pose estimation, and trajectory generation algorithm. This unit was instantiated with a full GUI and was therefore running with limited memory and compute. We saw in multiple cases when running the algorithm or making new executables that the CPU had a high low and ran low on memory. As such, we believe that this impacted our ability to provide the controller with on-time and accurate estimates. The controller at times would be operating on positions that were up to 200ms out of sync with the IMU data. Given more time, this could have been mediated by the addition of position extrapolation using a filter, but would have also required the implementation of a velocity and angle rate estimation mechanism onboard the Raspberry Pi. This would have once again increased our compute requirements. To mediate some of our issues, the GUI could have been removed after testing the components that required it, such as the camera integration. This would have freed up both CPU and memory and could have allowed us to add further estimation components that could have increased stability.

B. When-Ready Implementation

Another aspect of our implementation that could have introduced the instabilities that we saw in flight was our use of when-ready data. Instead of a set refresh rate for the pose and setpoints, we wrote the data whenever it was available. This meant that the controller, which was running at the 200Hz refresh of the IMU, would often see large jumps in position between single time steps. This contrasts with the relatively smooth data of the motion capture, as seen in Figure 9, and could have caused the jerky flights and motor commands that we saw. To remediate this issue, we could have implemented a set refresh rate with smoothing based on prior poses and twists. This would give the controller reliable data to provide motor commands from and damped the large accelerations we saw in our test flights.

C. Augmented Quadrotor Dynamics and Controller Gains

The quadrotor had significant changes to its dynamics due to the addition of the camera, Raspberry Pi, mount for the Pi, and battery pack to power the Pi. Sub-optimal design of the mounting system and large weight disparity lead to different and unstable dynamics. The center of gravity was misaligned with the center of rotation causing an induced pitch to the system during hover. The gains from quadlab had to be re-tuned; however, more time was necessary to do so.

With an improved mounting design, concentrating the weight at the center of rotation and a better tuned controller will significantly improve the system performance. Since the AprilTag localization does not estimate pitch or roll states very well, an extended Kalman filter to fuse the MPU data with the AprilTag localization will help correct any induced pitch/roll from uneven mounting of the extra hardware.

The additional weight also had the disadvantage of lower battery life available for tests. The team was able to complete only 2 or 3 flight tests at most per full charge.

V. Conclusion

We consider this project to be a success with the caveat of our limited flight capabilities. The implementation of a

pose estimator and trajectory generation from the fiducials produced measurements that were approximately in line with the global position, but they were inadequate for performing fully autonomous flight. Our software was successfully integrated and communicated data between the Raspberry Pi and BBB, and that data was accessible in rc_pilot for the controller to actuate upon. However we would require further extensions in velocity and rate estimation, as well as position estimation and smoothing onboard the BBB, to achieve a stable flight mode. As a result of these caveats, we cannot consider this project a full success, but we achieved the software integration that we initially set out to do.

REFERENCES

[1] April Robotics Laboratory, "Apriltag 3." [Online]. Available: https://april.eecs.umich.edu/software/apriltag

[2] "ROS wrapper for Intel® RealSense™ devices." [Online]. Available: https://github.com/IntelRealSense/realsense-ros

[3] M. Achtelik, M. Burri, H. Oleynikova, R. Bähnemann, and M. Popović, "`mav_trajectory_generation`: Polynomial trajectory generation and optimization, especially for rotary-wing mavs." [Online]. Available: https://github.com/ethz-asl/mav_trajectory_generation

[4] P. Gaskell, "Controls, lecture notes for aerosp 740," 2022.

[5] Q. Quan, *Introduction to Multicopter Design and Control*. Springer Singapore, 2018.