OPTICALLY-GUIDED MULTIROTOR AUTONOMOUS DESCENT AND LANDING ON A MOVING TARGET

Matthew Dupree, Yingchao Zhu, and Yogananda Isukapalli
Department of Electrical and Compute Engineering
University of California, Santa Barbara
Santa Barbara, CA
mdupree@ucsb.edu, yingchaozhu@ucsb.edu, yoga@ucsb.edu

ABSTRACT

We demonstrate the use of the AprilTag visual fiducial system for the precision landing of a multirotor vehicle on a moving target with no GPS use after target acquisition. Existing IR-LOCK precision landing code in a PixHawk flight controller with ArduCopter firmware is repurposed for passive ground target tracking using position information from a Raspberry Pi and PiCam, configured to identify and track a paper AprilTag. Debugging telemetry during development was exported by MAVLINK over 802.11ac Wi-Fi. The Apriltag removes the need for an active, IR- or RF-emitting ground beacon, allowing for precise aid delivery to unpowered disaster sites without necessitating human multicopter pilots be nearby for short-range work, nor cluttering longer-range cellular and radio bands. AprilTags allow for a sufficiently low false-positive rate to be used in debris-strewn environments and sufficient positional accuracy to land on small dinghies in flood zones or in the bed of an aid pickup truck.

INTRODUCTION

The University of California, Santa Barbara (UCSB) provides for senior undergraduate student teams year-long capstone projects. Our capstone team was tasked by the United States Naval Sea Systems Command (NAVSEA) to develop an autonomous multirotor capable of landing on a seafaring vessel without remote piloting assistance nor any radio or light beacon coordination. We were also directed to work with as minimal a reliance on GPS as we could manage.

The core task involved identifying a landing site that may be in motion without reliance on radio transmissions or active beacons. This immunizes the vehicle against interference by third parties, because a device not listening for external input cannot be hacked. To replace active external coordination, a passive AprilTag3 visual fiducial [1] is placed on the target landing site for low energy, low false-positive identification and relative pose extraction. This relative pose measured over time then provides information about relative velocity necessary to match velocities and perform a landing. This information derivative is taken by the vehicle's flight control unit during landing.

Removing the use of radio frequencies during flight has knock-on benefits for use in disaster zones

and aid scenarios. Doing so leaves WiFi, cellular, and aircraft telemetry bands uncluttered while still allowing for precise targeting of drone landings. The lack of active ground beacon also means that, in an emergency situation, a landing site may be prepared with a ruler, a typical black marker, and a bright, flat surface.

This paper focuses primarily on our landing system and the tools we developed to extract telemetry and observations from our test vehicle to fix issues arising in it during development.

VEHICLE DESIGN

A. HARDWARE

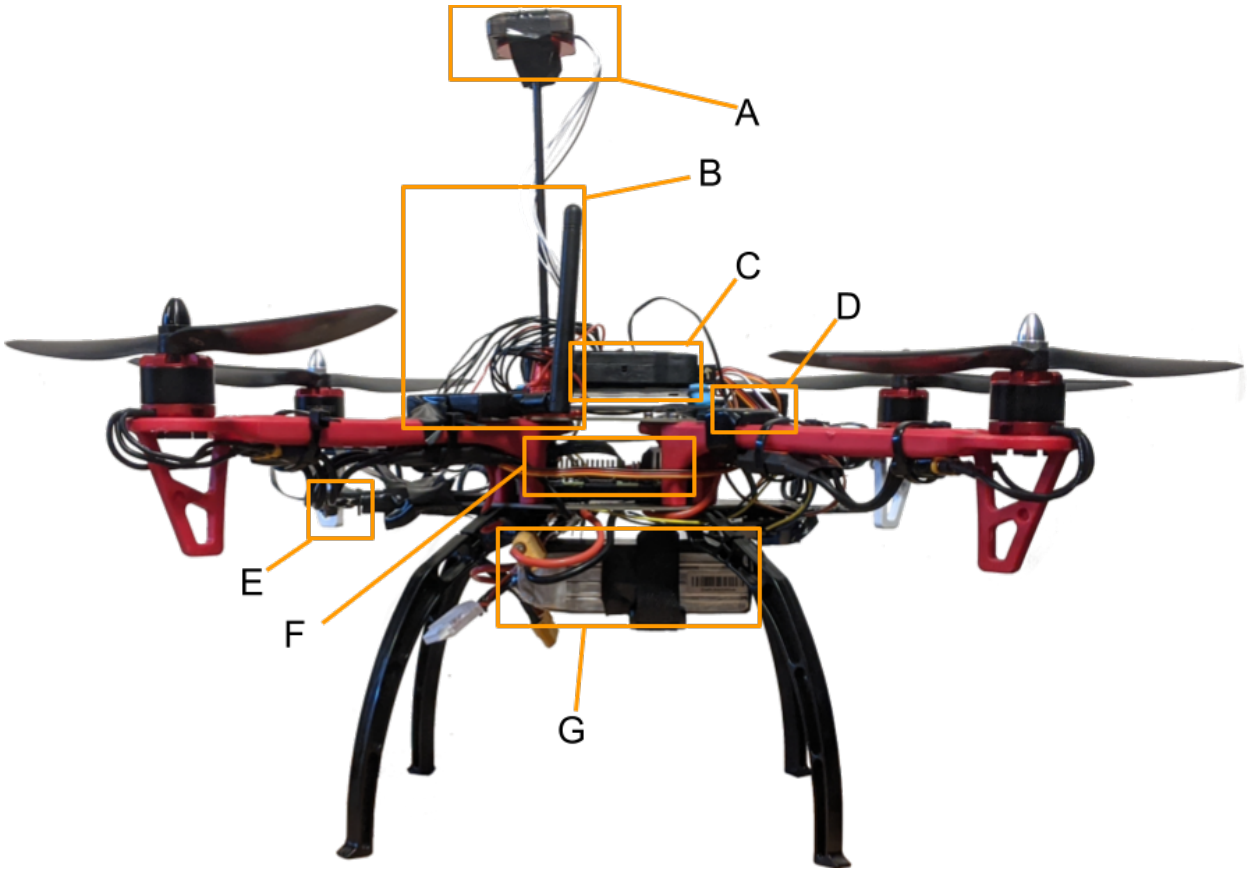

Figure 1: Test vehicle hardware. (a) uBlox M8N Micro GPS unit. (b) Holybro 915MHz telemetry radio. (c) Pixhawk v2.4.8 Flight Control Unit. (d) Remote control receiver. (e) Raspberry Pi Camera v2 (mounted facing down.) (f) Raspberry Pi 3B+ companion computer (mounted interior.) (g) Battery Pack.

Our test platform was a custom 1.2kg quad-rotor autonomous flying drone constructed from consumer-off-the-shelf (COTS) parts. The frame is a YoungRC F450 Drone Frame Kit with landing legs. Onto each arm we mount a LHI 920KV Brushless Motor with a SimonK 30A Electronic Speed Controller (ESC) for converting from flight controller pulse width modulation (PWM) to three-

phase motor drive. Onto each motor, we mount one black plastic 10x4.5 propeller. A Tattu 14.8V 2300mAh 4S 45C LiPo battery pack provides DC electric power storage for all system components. These test vehicle flight components were selected to be extremely forgiving in weight margins as we had little prior experience with quad-rotor vehicle design.

Flight control hardware is visible in Figure 1. Centrally, a Pixhawk 2.4.8 Flight Control Unit (FCU) provides IMU and magnetometer readings, collects position estimates from other hardware, checks mission objectives, then produces pulse-width modulation (PWM) motor control commands to drive the vehicle ESCs. A uBlox M8N Micro GPS unit provides global position updates to allow the vehicle to navigate to its programmed landing zone without internal position estimate drift beyond the order of meters. A Holybro 915MHz telemetry radio and FlySky FS-iA6B 2.4GHz remote control receiver allow remote mission alteration and direct control in case of unexpected behaviour during a test flight. Finally, a Raspberry Pi 3B+ companion computer and Raspberry Pi camera v2 module provide visual position updates of the landing target during our landing phase. Testing of our Raspberry Pi 3B+ independent from the test vehicle indicated its camera's 10 frames-per-second (FPS) 640 pixel horizontal, 480 pixel vertical mode was the best possible resolution and framerate without failing to process frames during AprilTag detection and pose extraction. This resolution and speed setting also provided the camera's full field of view which was ideal for our use case.

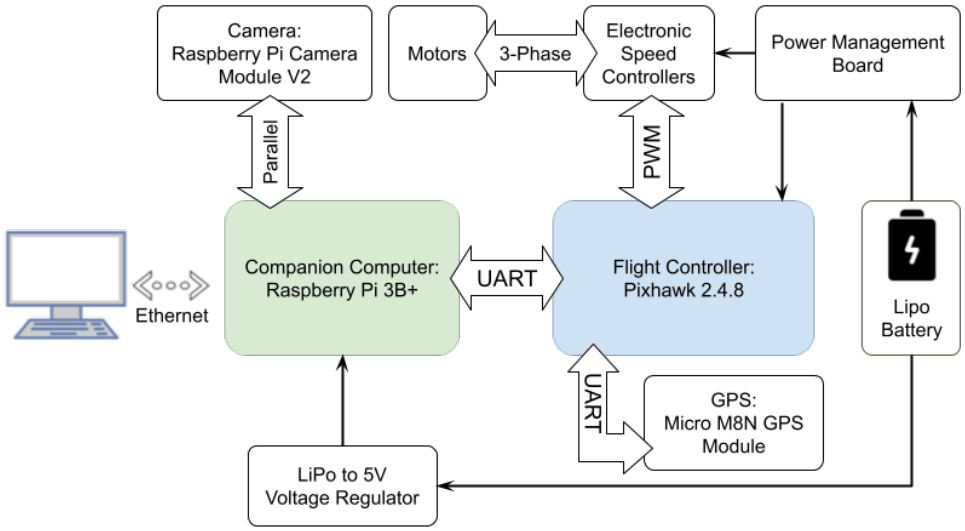

Figure 2: Test vehicle hardware block diagram. The WiFi module was not drawn as it is an internal subcomponent of the companion computer. The 915MHz telemetry radio and 2.4GHz controller receiver were left off as noncritical to the test vehicle's intended mission functionality.

Communication between all hardware components is accomplished with standard consumer protocols, as seen in Figure 2. The GPS is driven by the flight controller over UART with a baud rate of 96000, using drivers common to most autonomous consumer drone operating systems. Our companion computer communicates with the flight controller over UART, again at 96000 baud, using the Micro Air Vehicle Link (MAVLink) autonomous vehicle communication protocol. For testing

purposes, all MAVLink communications between the companion computer and flight controller are also echoed through our 915MHz telemetry radio for receipt by our ground station computer.

Vision data is supplied by our Raspberry Pi Camera Module v2, transmitted over parallel port cable to the Raspberry Pi 3B+ companion computer, then processed digitally for AprilTag detection and relative pose extraction.

Finally, as neither the FCU nor companion computer accept raw lithium polymer battery voltages, we had two regulator systems to manage power distribution to these compute devices. The FCU received power from a Hobbypower Power Module v1.0 power management board, which incorporates power for the FCU and voltage sensing to track battery drain. To power the companion computer, we designed a custom 2-layer printed circuit board (PCB) visible in schematic and fabricated form in Figure 3. The PCB was designed to accept 14 volts to 22 volts of input and produce a stable 5 volt output to up to 1.5A of load. The arrays of via holes in the design are for thermal dissipation. The two largest PCB components placed are an inductor, center top, and an input inversion protection diode, center bottom.

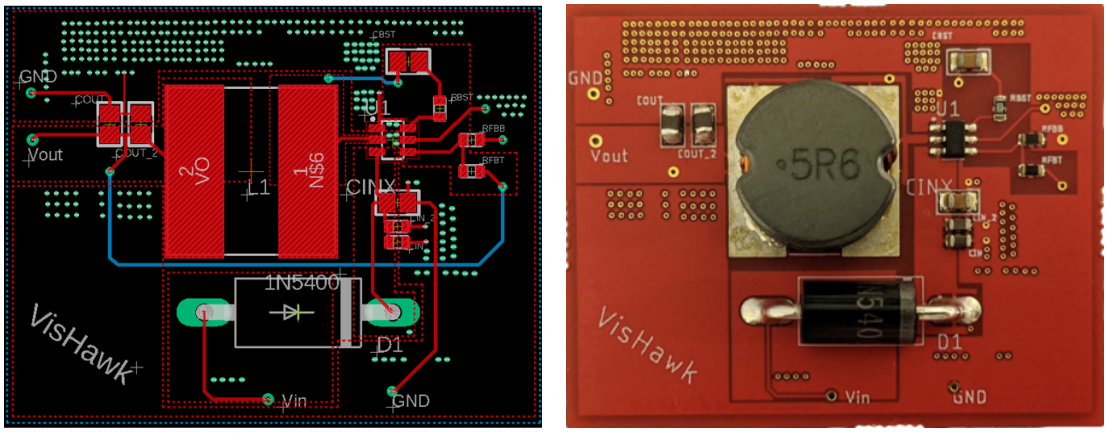

Figure 3: Custom 2-layer 5 volt regulator PCB designed to power companion computer. Left: Schematic of the PCB. Right: The fabricated board.

B. DIGITAL SYSTEMS

B..1 FLIGHT CONTROL UNIT

Our consumer PixHawk 2.4.8 FCU came pre-loaded with a copy of the open source PX4 autopilot firmware in its flash memory. At the recommendation of many sources in the drone enthusiast community, we moved to a more recent edition of the same PX4 firmware [2] before testing. We encountered issues doing so as the particular run of PixHawk 2.4.8 from which we acquired our test unit suffered from a hardware error that halved their externally writeable flash storage. To work around this, we manually recompiled the firmware locally, stripping out features unnecessary to our project. Being able to read the source code and recompile the firmware was beneficial in the long run of our project as direct analysis of the source code revealed to us that the PX4 firmware

Kalman filter was not programmed to be able to use the `landing_target` MAVLink messages from our companion computer.

Rather than attempt to alter the PX4 Kalman filter, we transitioned our test vehicle FCU to the older ArduPilot firmware [3]. This firmware contained a "precision landing" mode originally designed to be used with the IRLock active-infrared precision landing beacon. By adapting our project to produce messages acceptable to this precision landing system, we were able to successfully engage the precision landing code that already existed within the ArduPilot firmware and add our estimates of the tag position to the landing operation. As this precision landing code replaces ArduPilot's horizontal velocity estimates with that derived from the tag position changes, this causes the drone to match speeds with the target and land successfully.

B..2 COMPANION COMPUTER

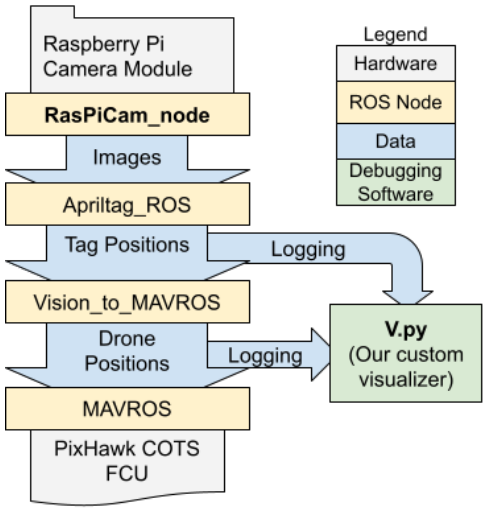

Figure 4: Apriltag visual processing stack.

Onto our Raspberry Pi 3B+, we installed a Ubiquity Robotics distribution of Ubuntu 16.04 preloaded with Robot Operating System (ROS) version Kinetic Kame. We chose to use the ROS software platform to simplify loading of images from the Raspberry Pi camera module and transmission of extracted poses over MAVLink. Use of ROS has the added benefit of separating concerns into indepentent processes, allowing for parallel completion of independent hardware accesses, parallel computation of independent processing steps, and greatly simplified logging. Our operational ROS node flow is visible in Figure 4. The RasPiCam_node [4] interfaces with the camera module and Raspberry Pi 3B+ graphics hardware and produces bitmaps in memory. ROS then transfers these bitmaps to the Apriltag_ROS [5] node for tag identification. The Apriltag_ROS node finds all valid Apriltags within view of the camera, then checks the tags' IDs against that configured for landing. If the landing tag is detected, its pose is extracted and sent to the Vision_to_MAVROS [6] module. Vision_to_MAVROS performs a simple coordinate transform to turn the tag's position relative to the camera into a test vehicle position relative to the tag. Fi-

nally, the position of the test vehicle relative to the tag is published to the flight control unit as a `landing_target` MAVLink message over UART using the MAVROS [7] module.

We had to make corrections to our copies of two of these modules. First, Apriltag_ROS used a ROS object to receive camera images that assumed the image and corresponding camera property matrices would be published in perfect lockstep. However, Raspicam_node produced its camera images and their camera properties as separate ROS calls, as a necessity to provide additional features we did not use like motion maps from hardware H.264 encoding. To force the version of Raspicam_node available to us at the time to produce lockstep images and property matrices, we needed to rewrite a significant portion of the module.

Second, an enumeration within MAVROS called `MAV_FRAME`, used for describing the coordinate frame in which a position is given, had an off-by-one error for all nonzero enumeration values. This included the `LOCAL_NED` frame in which we were reporting our `landing_target` and `visual_odometry` MAVLink messages to our FCU. (The "`LOCAL`" means the coordinates have an origin within a few miles at most as they're measured in meters. The "`NED`" means the coordinates describe a position X units of distance to the north, Y units of distance to the east and Z units below the takeoff site, assuming down is always parallel to the radial axis of the earth and north is always true north.) Investigation of extracted uORB message streams from the PX4 firmware had indicated it was rejecting our `landing_target` messages despite receiving them successfully over the UART channel. This enumeration error was the reason for the rejection, as the PX4 firmware refused to perform coordinate conversion to use a `landing_target` message published with any frame except the correctly specified `LOCAL_NED`. Fixing the enumeration in our copy of MAVROS was a one-line change.

TELEMETRY EXTRACTION

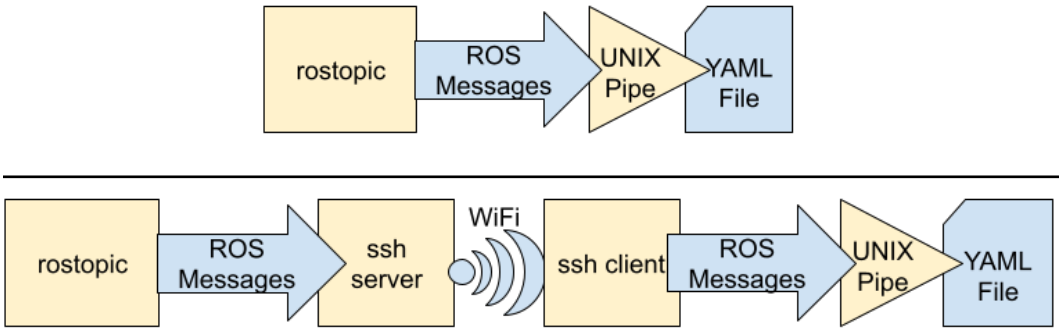

Figure 5: ROS message logging. Top: logging on the companion computer directly. Bottom: logging on the ground control station via SSH bridge. The YAML file in the latter case may be replaced with a direct UNIX pipe to `v_sub.py`.

Using the ROS program `rostopic`, we logged copies of the tag position and landing target messages, produced by Apriltag_ROS and Vision_to_MAVROS respectively, into our companion computer's storage, as shown in Figure 5, top. As ROS messages are identical to multi-document

YAML files, trivial python code can load the stream of messages and process them for offline analysis. We extended the design of our visualization tool, `v_sub.py`, to use the POSIX standard input as its loaded stream, allowing us to stream in offline files or pipe live data from a Secure SHell (SSH) session over WiFi or Ethernet. This streaming was also convenient to reduce manual copying of log files, allowing us to directly save them on our local machine as visible in Figure 5, bottom. Finally, using the UNIX program `tee` with additional UNIX pipes allowed us to simultaneously save the data to a local file and stream over SSH during some tests to have both live visualization *and* records for playback.

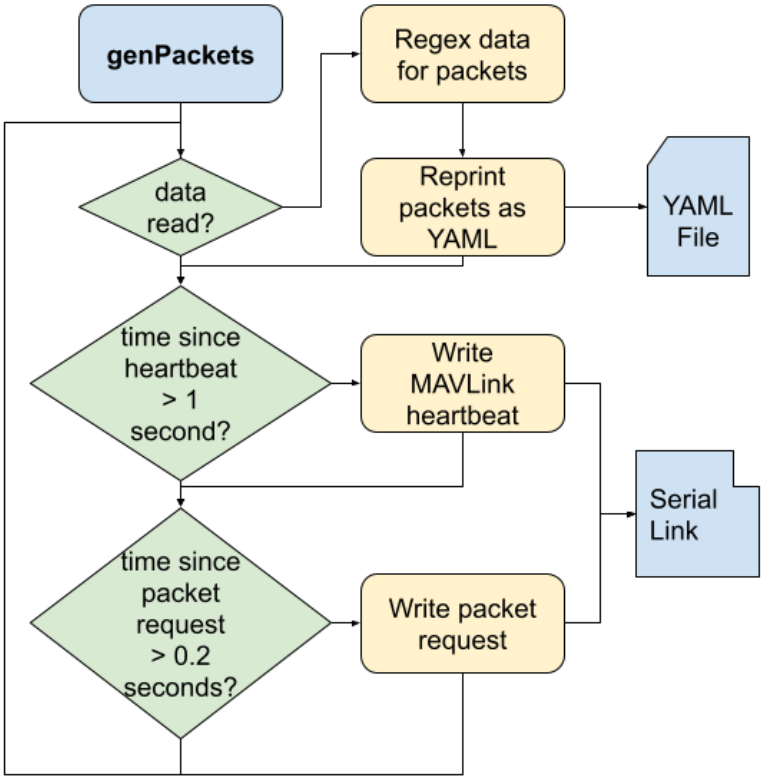

Figure 6: Extraction of data from our FCU's operating system via the MAVLink Serial control. Data is read from pyserial, a regex is applied to locate complete uORB messages, then the packets are reprinted as YAML. If necessary, a MAVLink heartbeat is echoed to prevent disconnection. Then a request for another single uORB packet is written.

During motor-disarmed ground experimentation, we also had need to log telemetry as received by our FCU's firmware's internals. As the MAVLink mirroring from the companion computer to the telemetry radio occurred before decoding of the messages by the FCU, it did not accurately represent the rejection of our landing target updates as observed in the test vehicle's behaviour. Luckily, the MAVLink standard comes with a `SERIAL_CONTROL` message type allowing for passthrough access to FCU serial ports, including an FCU shell if one exists. One existed in the firmware we initially attempted to use, PX4, but this feature did not exist in our second firmware, ArduPilot. Using an adaptation of a tool provided by the PX4 repository for bringing up the MAVLink serial shell (which itself was built on PyMAVLink and PySerial) we were able to produce a script

generating a continuous stream of tag position sightings as understood by the FCU's internals. A diagram of our tool's operational processes is visible in Figure 6. This made obvious the enumeration mistake in MAVROS as well as some remaining mistakes with our configuration of the Vision_to_MAVROS coordinate transformation.

TELEMETRY VISUALIZATION

Visualization of our telemetry was performed using Python3.8.3, MatPlotLib 3.3.2, and PyYAML to produce a tool we called `v_sub.py`. ("`v`" for visualizer and "`sub`" for subsampled.) The operational process of this tool is visible below in Figure 7. Using MatPlotLib animations, we repeatedly render 3D plots of position and orientation for streamed-in values. As MatPlotLib is not a high performance plotting library, we were limited to 15 to 25 of the most recent samples to meet a rendering speed target of 10 FPS – the rate at which apriltag_ROS could process vision images on our Raspberry Pi 3B+. Extensive trials manually lifting the test vehicle over our landing tag while observing readings in `v_sub.py` allowed us to determine some inconsistencies in our coding of quaternions representing the vehicle orientation at different points in our vehicle stack, among other pose transformation snafus.

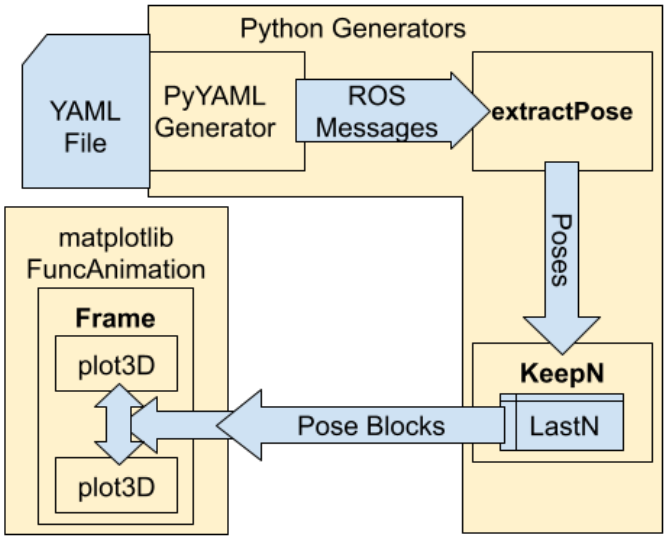

Figure 7: `v_sub.py` processing pipeline. The YAML file may be replaced with a live YAML stream from the drone over SSH. Python generators are functions repeatedly yielding new elements to the next stage until the source produces a StopIteration exception. KeepN is a generator that provides arrays of the last N elements generated by the preceding generator, labeled here as "Pose Blocks". A MatPlotLib `FuncAnimation` accepts any iterable and a function to render its elements, then repeatedly calls the rendering operation in the MatPlotLib windowing context. Bold blocks were written directly by us.

CONCLUSIONS

As only a few days remained in the project before final presentations, only one flight with a moving target acquisition and landing was ever performed with the completed test vehicle software and hardware. One landing on a static tag was also performed. The moving target landing demonstrated overshoot and correction during matched-velocity descent and a significant traversal from the initially-programmed GPS coordinates due to following the tag (over 8 meters.) A diagram of the flight path as recorded by onboard GPS and barometer alone is visible in Figure 8. Video of the flight demo is available at `https://youtu.be/OROEFxYX0Mw`.

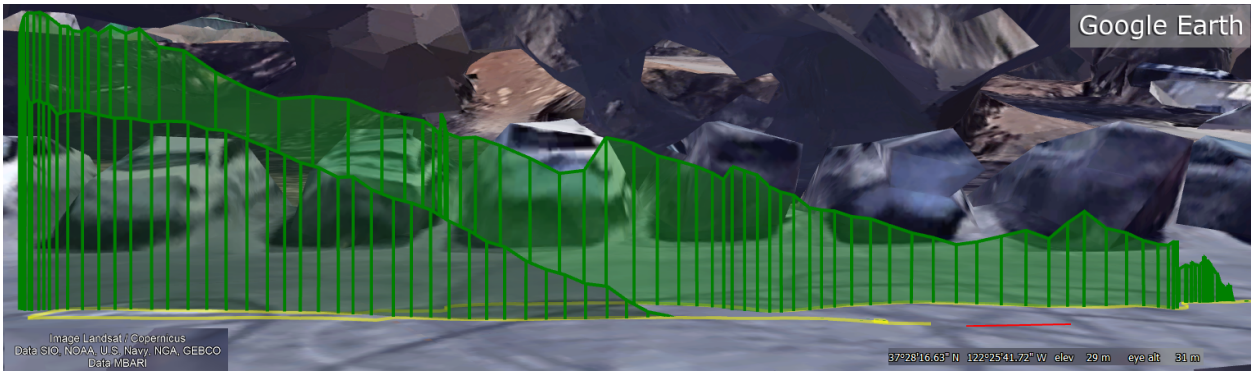

Figure 8: Path of our successful test flight with tag following, side view, rendered by Google Earth from data extracted by ArduPilot Mission Planner from offline logs. The initial ascent is the closer track and is afflicted by altitude mis-estimation, but shows the stable ascent profile visible in the demonstration video. The jump in altitude at the center of the image during descent matches the moment when the drone corrects its overshoot of the tag during flight. Altitude estimations again begin to fail at the end of the descent due to ground effect on the test vehicle barometer. The red line near the flight start point is a 1 meter scale measure.

Our paper demonstrates that simple consumer parts may be used to produce a quad-rotor vehicle capable of landing precisely on passive, visual targets. We also demonstrate telemetry export from ROS by SSH bridge, telemetry export from PX4 by MAVLink serial control, and one visualization strategy for both types of extracted online or offline data in Python MatPlotLib.

ACKNOWLEDGEMENTS

We would like to thank the University of California, Santa Barbara for offering the capstone program that gave us the opportunity to design and build this drone. We would also like to thank our third project partner, Xihan Liu, for her excellent work in bookeeping project resources, soldering components, and other critical tasks. Finally, we would like to thank the United States Naval Air and Sea Command (NAVSEA) for guidance in this project.

REFERENCES

[1] M. Krogius, A. Haggenmiller, and E. Olson, "Flexible layouts for fiducial tags," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2019.

[2] PX4 Drone Autopilot, "Px4-autopilot." `https://github.com/PX4/PX4-Autopilot`, 2020.

[3] ArduPilot, "ardupilot." `https://github.com/ArduPilot/ardupilot`, 2020.

[4] UbiquityRobotics, "raspicam node." `https://github.com/UbiquityRobotics/raspicam_node`, 2020.

[5] AprilRobotics, "Apriltag ros." `https://github.com/AprilRobotics/apriltag_ros`, 2019.

[6] T. Nguyen, "Vision to mavros." `https://github.com/thien94/vision_to_mavros`, 2019.

[7] MAVLink, "Mavros." `https://github.com/mavlink/mavros`, 2020.