Face Following and Visual ORB-SLAM in an Unmanned Aerial Vehicle

Zach Stoebner

***Abstract*—This paper presents face following and visual ORB simultaneous localization and mapping (vSLAM) implementations for an unmanned aerial vehicle (UAV) that operate in real time, out of simulation. The software was written for Ryze Tello using its hardware support toolkit in MATLAB. Building on, translating, and modifying the implementation of these systems in other scenarios, I designed a side-by-side set of functions to generalize face following and vSLAM for the Tello quadcopter and to provide tools for myself and others to integrate and improve our projects. To follow faces, the faces are first detected using an object detector and then labeled with a bounding box that is used to compute the approximate trajectory to center the Tello on the face. My vSLAM implementation modifies the ORB-SLAM paper's MATLAB example to handle real-time, imperfect image input and provide a measure of redundancy for a realistic CPS. In an ideal environment, these implementations work relatively well. As a cyberphysical system (CPS) that is heavily reliant on camera input and WiFi connectivity, working conditions can greatly impact the Tello's successful execution of the procedures. For the benefit of the community, the code is publicly available at https://github.com/zstoebs/tello_detection_SLAM.**

***Index Terms*—Face detection, following, simultaneous localization and mapping (SLAM), monocular vision, recognition, tracking, unmanned aerial vehicle (UAV)**

I. Background

The most pressing technological challenge of our time is creating systems that think and behave like humans. When machines are built that can emulate human thinking behavior, they usually remind us of particularly helpless infants, necessitating constant supervision and manual correction to even achieve their designated tasks. An even greater, seemingly insurmountable, challenge on top of just thinking and behaving is constructing systems that conduct their behavior autonomously, without endless handholding. Unfortunately, not only is inventing new software and hardware to even establish the theory of autonomy incredibly difficult but implementing intelligent system autonomy in practice is marred by unforeseen conflicts, in the sensor readings, in the operating system, etc., that often hinder the full meaning of "autonomy." But once that challenge is superseded, humanity may step into the next fold of its history. But we have so much mileage to cover and a many more checkpoints to reach.

Two of those checkpoints include: object detection & following and SLAM. At face value these terms may not seem vague, but when it comes to implementation there are many specifics that enter the fold such that implementation *all* of object detection and *all* of SLAM is not feasible. That said, we will narrow down the tasks to a digestible level for a small UAV. To help bring this autonomy to life, this project will use a Ryze Tello and its hardware support in MATLAB[1].

A. Introduction to Face Following

For us, detecting other faces is natural and reflexive. Even going a step deeper to recognize a face is also an involuntary reflex for most people. On top of that, integrating motion and vision to follow a face is also an absurdly simple task for most people to accomplish. Why are faces so straightforward for people? Notwithstanding the deeper existential answer to that question, the answer is that our faces are intrinsically human; we have dedicated cortical structures and substructures devoted to detecting, recognizing, and homing in on faces. Software and hardware lack these highly evolved structures; a rather disjunctive feat for people, discerning the mathematics and statistics behind face detection and following is necessary to bestow our capabilities to computers.

Intuitively, we can imagine human facial recognition as a probability function based on a series of glimpses from our eyes. Essentially, this statistical model is recreated using a boosted cascade that simulates attention and focus on specific high-probability details, much like human vision [1]. Beyond faces, this cascade detector can be applied to learning other objects. MATLAB provides a highly optimized cascade object detector (*vision.cascadeObjectDetector*) as part of its Computer Vision Toolbox[2] that can identify faces and specific facial attributes very quickly and reliably. You might be wondering if there is any optimized, built-in support for general objection, such as the novel YOLO algorithm which can quickly detect multiple classes of objects using a single neural network [2]. MATLAB's description is somewhat misleading because it only provides support for detecting

Paper submitted December 12, 2020. Zach Stoebner is with the Department of Electrical Engineering & Computer Science, Vanderbilt University, USA. The source is publicly available on this GitHub repository: https://github.com/zstoebs/tello_detection_SLAM.

[1] https://www.mathworks.com/hardware-support/tello-drone-matlab.html
[2] https://www.mathworks.com/products/computer-vision.html

faces and facial features, not just any object. However, if you have a GPU available to use, MATLAB has a YOLO training

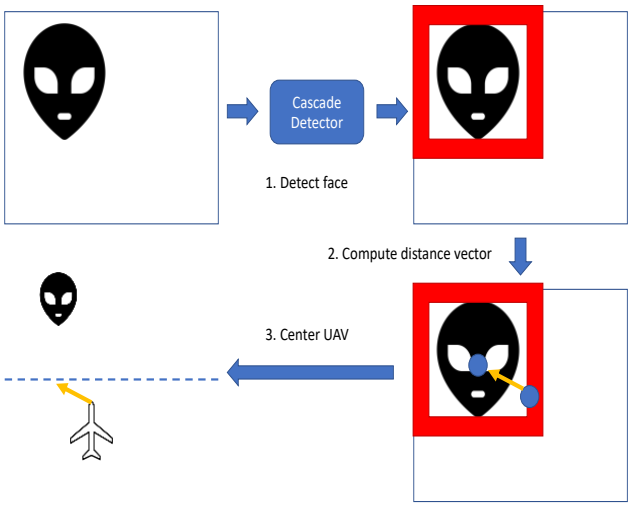

Figure 1. Face following schematic. An image is passed to the cascade object detector. The detector draws a bounding box around the face. The centering vector from the current center to the center of the bounding box is computed. The UAV moves in the direction of the centering vector while maintaining a safe, specified distance.

API that could be used to further generalize the implementation in this paper to detect and follow other objects or groups of objects.

Once a machine has detected a face using its own camera, following the face is somewhat easier. A bounding box can be drawn around the face that can be used to compute a geometric centering vector for the UAV to follow in order to center on the face. To better convey the face following pipeline, the schematic is shown in Figure 1, depicting the three primary steps after the input to center the UAV.

B. *Introduction to SLAM*

For as long as humanity has walked the plains of the world, navigation and environmental awareness has been of the utmost importance for survival and long-term viability. Similar to recognizing faces, we have optimized awareness and environmental feature detection that is ingrained in our biology that allow us to effortlessly learn, map, localize, and navigate our surroundings. Yes, sometimes navigation is a shoddy when we are introduced to new environments but, alas, we learn and cement the environments into our memories as we are repeatedly exposed to the same pattern of features.

However, machines are not as primed for this task, especially small, mobile machines that have to sacrifice robust processing and memory hardware for adequate movement. The ability to remember, store, and process onboard is one that is attuned to people, not clunky drones. Nonetheless, this problem is still partially solvable for UAVs because they can enlist the help of an immobile ground station with all the compute resources it needs to process and store the incoming data from its surroundings. Although autonomous navigation with no guides is the end-goal, the first problem that needs to be solved is simultaneous localization in and mapping of the environment. But before that we once again must relay this problem mathematically and statistically.

In order to map the environment, the UAV must detect features and store their locations as points in a 3D space. Many methods for feature detection exist, such as SIFT [3] and SURF [4] These feature detectors combine binary hypothesis testing and machine learning tools to extract pixel features from two similar images. ORB improves on feature extraction by significantly speeding up the computation and uses decorrelation and ANOVA reduce noise and remain rotation invariant [5]. At the fundamental level, the mapping problem is estimating the conditional probability distribution in Equation 1.

$$P(m \mid x_{0:k}, z_{0:k}, u_{0:k}) \quad (1)$$

m corresponds to the map of features locations. $x_{0:k}$ corresponds to the UAV's estimated location at each time point. $z_{0:k}$ corresponds to the estimated feature locations from the UAV's camera at each time point. $u_{0:k}$ corresponds to the movement sequence since the first time point.

Applying the feature detector to images from a UAV camera allows us to construct a feature map and, by comparing with previous frames, we can also determine distances and camera pose – a computation that cannot easily be done on a single frame. From these distances and poses, we can compute an estimation of where the UAV camera is relative to the features. With the mapping, distance, and pose at each point, we can also estimate a path for the UAV within the environment, which alongside the map results in a SLAM system. That said, this subproblem essentially encompasses visual odometry, which is essentially SLAM without saving the map and just tracking the camera trajectory [6]. At the fundamental level, the mapping problem is estimating the conditional probability distribution in Equation 2.

$$P(x_k \mid z_{0:k}, u_{0:k}, m) \quad (2)$$

x_k corresponds to the UAV's estimated location at the current time point. $z_{0:k}$ corresponds to the estimated feature locations from the UAV's camera at each time point. $u_{0:k}$ corresponds to the movement sequence since the first time point. m corresponds to the map of features locations.

Unsurprisingly, many SLAM variations exist since there are a lot of moving parts in this algorithm that might cause bottlenecks for some systems. Two such variations are: RGB-D SLAM [7] and ORB-SLAM [8]. RBG-D SLAM is a few years older than ORB-SLAM and passes depth images to a back-end to estimate pose and the RBG images to a front-end to map features, receive the pose estimation, and generate the point cloud and pose trajectory. ORB-SLAM

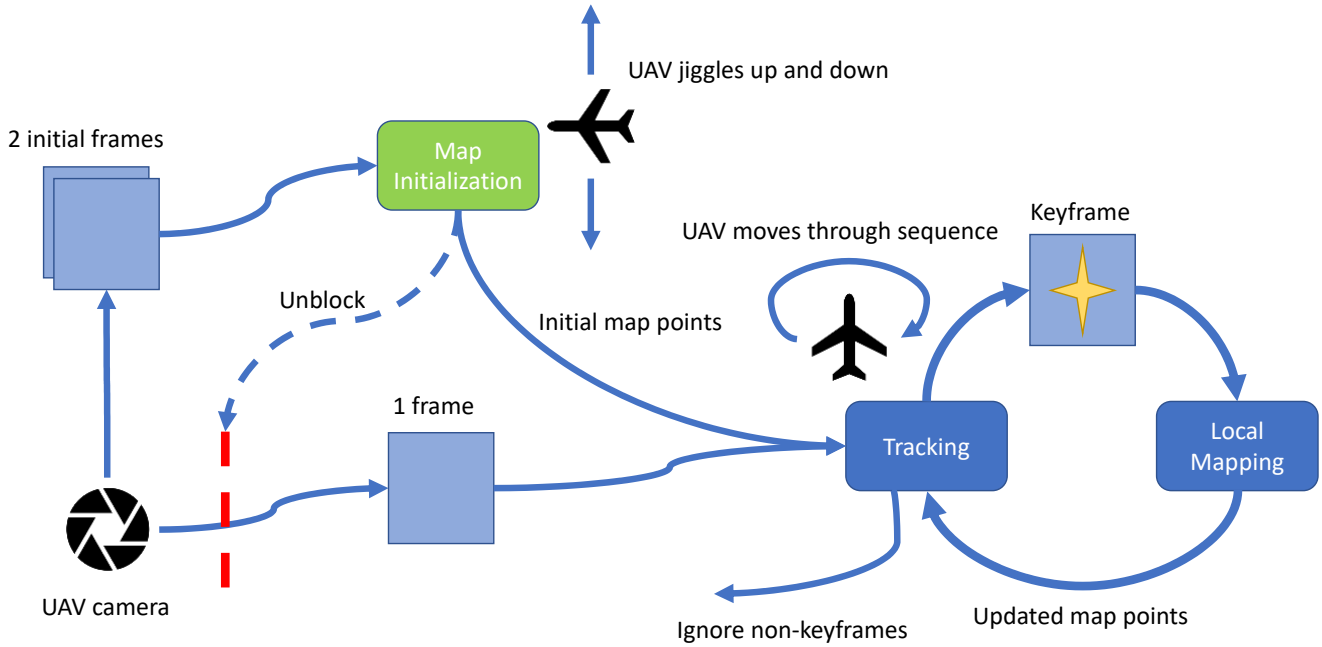

Figure 2. Visual ORB-SLAM schematic. The process starts by initializing the map with two initial frames from the camera. During the initialization the UAV jiggles up and down to snapshot slightly different pairs of images with different feature extractions but still with some matches. If the map initializes, then the program proceeds to the main loop where it first tracks the features on a new frame. If the frame is a keyframe, then the new features are updated into the map. If the frame is not a keyframe, then the loop continues. At the start of each loop iteration, the UAV executes the next move in the sequence.

specifically extracts ORB features, hence the name, initializes a map based on a pairwise feature mapping between two consecutive images, it keeps track of the local map and iterates through images until there is a "keyframe" e.g. a significant difference in the feature matches with the previous keyframe. Then, it updates the map and pose at this keyframe, saving on a significant computational expense. Traditional ORB-SLAM expects and checks for a loop closure – when an image has significantly similar features to an earlier keyframe – to terminate its main loop. Deviating away from the traditional pipeline, the ORB-SLAM schematic for this paper is displayed in Figure 2.

Other notes on SLAM: Given the background on SLAM, the alliteration of its computational expense, and some intuition, it may already be obvious to the reader that SLAM is an intractable problem. We can never solve the entire map or know precisely where we are located in it; hence the computational expense to build the map and determine the probability distribution for the location is taxing. SLAM can only ever return a best guess of the map and location but for problems that require just some partially correct point cloud to function at an adequate level, such as path planning, earn a practical benefit from the map. Despite its insolvability, SLAM is nonetheless useful.

On visual odometry, the Cadena et al paper posits that a SLAM algorithm without loop closure is essentially just visual odometry [6]. Although this claim may be entirely true, I believe that this implementation is still closer to SLAM because it maintains a persistent map whereas visual odometry just compares the features of consecutive frame, discarding the map altogether [9]. If so, the vSLAM implementation in this paper is then visual odometry that is almost ORB-SLAM, minus loop closure and the final optimization.

MATLAB provides a standard visual ORB-SLAM example on the TUM *long_office_household* dataset. This code and the associated helper functions were adapted to the project at hand. In order for this SLAM implementation to work, MATLAB's Computer Vision and Parallel Computing[3] Toolboxes are required, with the Computer Vision Toolbox's parallel processing backend toggled on.

II. Method

In order to maintain some semblance of a clean, interpretable code structure for this project. The hierarchy is delegated into three files, using MATLAB2020b:

1. *main.m*: This file is the high-level "API" for the project code. At the head, it contains all of the necessary notes for a full understanding of the code and frequent issues that were encountered during its production. Before and after the code sections, there are cleanup commands. The file then splits into three code sections: 1. connection and takeoff, 2. face following, and 3. vSLAM. The face following and vSLAM sections are intended to be run separately, not consecutively.
2. *follow.m*: This file contains the function to recognize a face and compute the centering vector and turn angle. The arguments to the function are the drone's camera

[3] https://www.mathworks.com/products/parallel-computing.html

object, the cascade object detector, and a distance parameter. It returns the distance changes along the *x*-, *y*-, and *z*-axes, the angle in radians for a subsequent turn, and the image with the bounding boxes marked in.

3. *vslam.m*: This file executes vSLAM until completion or error and returns nothing. The arguments to the function are the drone object, the drone's camera object, the movement sequence, the number of movement cycles for which to run the main loop, and the minimum number of ORB feature matches for triangulation.

On the line of clean code, since many of the utilized functions rely on the real-time status and feed of a dynamic CPS, they are prone to throw errors that are beyond software's control. In that case, I enclosed functions that were common culprits in error handling blocks to have some redundancy in case the faulty input comes back in line in a subsequent iteration.

A. *Face Following Method*

The face following function's schematic is displayed in Figure 1. The function steps are as follows:

1. Pass the frame to the object detector and retrieve a bounding box location(s) for the detected object.
2. Draw boxes around all of the detected images.
3. Use the closest bounding box's width and center coordinates to compute the relative axis change as a percentage of the max.
4. Based on some threshold percentage and some minimum movement distance, set the axes distances and return them to be used in a move command.

The drone's camera object is passed to the function to yield the image that will passed to the *vision.cascadeObjectDetector* to identify any faces to center on. The height and width of the captured frame are then gathered for later use in computing the centering vector. Any frame's center coordinates are computed as half of the width and half of the height. Afterwards, the image is passed to the object detector, a bounding box is returned, and the shape of the bounding box is inserted into the original frame to be returned by the function.

The closest bounding box is chosen based on the greatest area, assuming no giant faces or massive misclassifications. The box width and center coordinates of the bounding box are used to compute the pixel distances as a percentage of the largest bounding box size, total image width, and total image height.

$$Xdiff = (bbox\_sizes(distance) - \text{bbox\_width})/\max(\text{bbox\_sizes}) \quad (3)$$

$$Ydiff = (bbox\_x - center\_x)/width \quad (4)$$

$$Zdiff = (center\_y - bbox\_y)/height \quad (5)$$

Respectively, these three dimensions correspond to the *x*-, *y*-, and *z*-axes, whose directions relative to the Tello are displayed in Figure 3. The distance from the face to maintain was determined using the percentage resulting from Equation 3, where a list of bounding box width sizes is indexed by the face

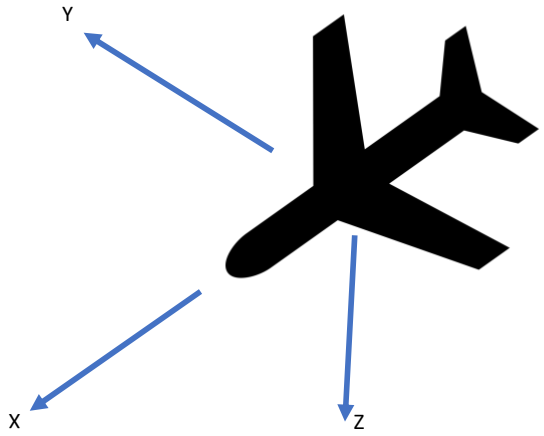

Figure 3. Axes directions for MATLAB's Tello hardware support package. Positive values are in the direction of the axes. The axes are intended to abide by the right-hand rule.

following function's distance argument and causes the drone to return a movement value that optimizes toward the desired distance. Equations 4 and 5 show a similar optimization toward the center of the bounding box, given the direction of the move command's principal axes. If the percentages supersede some manual threshold, then the distance along the corresponding axis will be returned as $\pm$ 0.2 meters depending on the sign of the percentage. $\pm$ 0.2 meters are the minimum distance that is supported by the Tello toolbox along any one axis. Additionally, a yaw angle is returned as the *y*-axis percentage of $\frac{\pi}{4}$ radians.

In the main script's loop, the return values were used to first display and save the bounding boxed images. Next, the axes distances are fed to the Tello's move function and the yaw angle called with its turn function. After 60 seconds, the loop terminates and the intermediate facial detection images can be viewed in the *./imgs/faces* directory.

Notes: Although I verified that a drone object could be passed as a function argument and commanded from within an executing function, I opted not to define the function that way. The encapsulation likely would have been better so that the face following function would not have to be run in an external loop. However, when I was working with the Tello toolbox at first, I noticed some overhead with connectivity when I was initially calling the code from within the face following function. Although, it neither stopped nor worsened when I switched to the top-level call so it may have been due to external factors such as overheating or low battery. If necessary, refactoring the code into a single function call should not be a major issue.

B. *Visual ORB-SLAM Method*

The code for this vSLAM implementation was modified from the MATLAB ORB-SLAM example[4]. In the main script, vSLAM, unlike face following, runs completely within its own function. The only reliance is the setting of the movement sequence in main. Currently the move

[4] MATLAB Monocular Visual Simulateneous Localization and Mapping

sequence has to be manually set because the Tello often disconnects randomly on extended flights so short sequences for short vSLAM test runs is best.

The vSLAM function's schematic is displayed in Figure 2. This vSLAM implementation breaks down into three key parts: map initialization, tracking and local mapping. Starting with map initialization, the steps are as follows:

1. Track the ORB features on the first image to load the pre-points, then track a second image.
2. Match the ORB feature correspondences between the two images.
3. If enough matches are made (100), compute the homography and fundamental matrices so that the correct geometric transform is applied based on which results in the least error for the relative camera pose.
4. If insufficient matches are made, then the loop restarts on a new image. Manually, the loop has a maximum of 5 iterations to find a matched image until an error is thrown. If a match is not made in 5 iterations, it may imply that the Tello has weak connection and low light and needs to be reset.
5. Triangulate the 3D locations of the matched features in the new map.

For map initialization, I modified the code to take the drone's camera images as input instead of an image data reader for a fixed dataset. That way, the images could be read the Tello's images in real-time. Recall that the homography matrix is used for planar scenes and the fundamental matrix is used for non-planar scenes. Map initialization is very important to the initial tracking problem in the main loop so it must work well in order to have a truly successful vSLAM run.

After the map initializes, the initial keyframes and map and feature points are set for the main loop to begin running. The visualizations also begin in this interim. The first part of the main loop is feature tracking, which ultimately determines whether the current frame is a keyframe such that the map should be updated with new features. A keyframe is any frame that satisfies: 1. twenty iterations have passed since the last keyframe or the current frame is tracking fewer than 80 of the current map points, and 2. current frame tracks fewer than 90% of the map points tracked by the current keyframe. Modified to work with a moving Tello, the main loop steps are as follows:

1. Move the drone according to the modulus of the current move index by the length of the move sequence.
2. If the Tello loses connection and throws an error, loop back to see if connection is regained, changing no indices except a break iteration countdown of 10. Throw the error if the break countdown expires.
3. Extract ORB features from the frame and match with the latest keyframe. If the new frame is not a keyframe, continue the loop.
4. Estimate the camera pose with Perspective-n-Point [10] in order to project the features to the current frames perspective and correct using some bundle adjustments[8]. This step, although esoteric, is important for the fast computation of that ORB-SLAM offers compared to the competition.
5. Determine if the current frame is a key frame given the criteria. If so, the process continues to local mapping. Else, the loop iterates, and the above steps are redone for the next frame. Additionally, this step also speeds up the process; instead of evaluating all of the features in every frame for mapping, only a select few that are substantially different are filtered for usage.

Now that the meat of ORB-SLAM has concluded with tracking and just figuring out whether a frame is unique enough to be mapped, the final stage in this vSLAM procedure is local mapping. Compared to tracking this procedure is much simpler and more intuitive, building the point cloud map and saving the pose. The local mapping steps are as follows:

1. Add the new keyframe to the set.
2. Compare the keyframes features against all the other keyframe features, looking for unmatched points that occur in at least 3 other keyframes.
3. Bundle adjust the pose based on the adjacent keyframes' poses.

After local mapping, ORB-SLAM typically builds and checks the loop closure dataset to determine when the camera has circled back to its starting position and then performs one last final optimization on the poses. Since this paper's vSLAM is using a predetermined move sequence instead of a predetermined dataset, I removed the loop closure since using seemed to almost intrinsically rely on the dataset having been crafted before the vSLAM run. Additionally, the final optimization often failed because it requires a continuous point cloud but since Tello's connection is weak it would sometimes black out in between iterations creating discontinuous maps. Ultimately, the optimization was not concerning because the goal of this project was just to get visual SLAM functional in a UAV.

C. Implementation Details

As indicated in the previous section, implementation of face following and vSLAM both require the Computer Vision Toolbox. It is important to then note that both of these implementations are heavily vision-based. Additionally, the Tello uses its camera to orient so mindfulness of lighting and other visual factors was extremely important to ensure that these methods worked. The Tello is additionally sensitive to drafts and gusts from the blowback of its own propellers, pushing it off course. On top of that, Tello does not have a very strong WiFi connection and often will blackout mid-flight. Of course, disturbances are inevitable and frequent, so they often did not work for me until I changed my angle, or the stars aligned; persistence is important.

III. Experiments

For testing face following and vSLAM, I experimented with both in repeat scenarios and in different settings, offering them different encounters that are not explicitly guaranteed.

A. Face Following Experiments

To examine face following, I first tested to see if MATLAB's face detector would work as expected, detecting my face adequately. Particularly, I wanted to see if it would detect my face regardless of whether or not I was looking directly at it or if my head were angled differently, such as if I were looking at my phone or not. It follows that if it can detect

my face at non-frontal angles, then it can compute a centering vector and follow my face at that angle. Figure 4

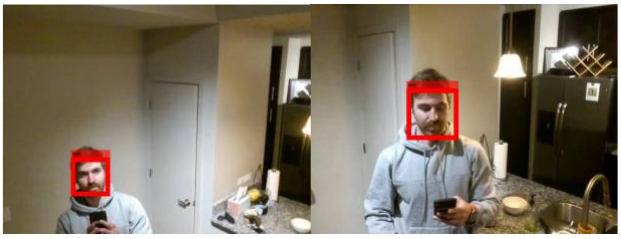

Figure 4. Examples of when my face is detected. Looking at the Tello (left) and not looking at the Tello (right). Nonetheless, it still detects my face and doesn't pick up much noise, even in low light.

Figure 5. Examples of when my face is not detected. Sliver of head off screen (left), turned away (middle), and only top of my head and eyes are visible (right). Ultimately, there has to be some frontal view of the entire bust to be detected.

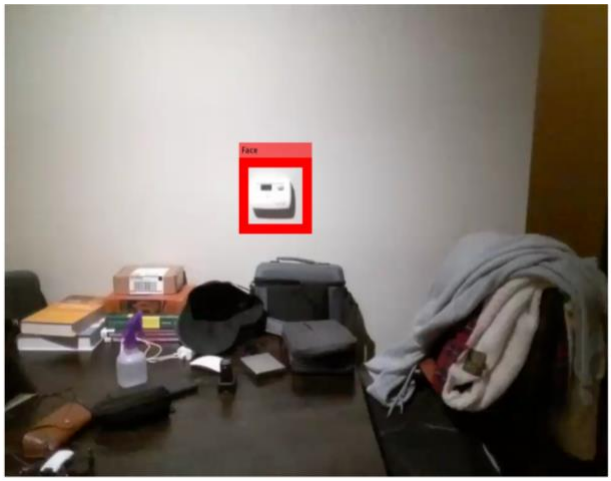

Figure 6. Examples of face misclassification. These misclassifications typically occur when there is no face in view of the camera. Otherwise, they are rare and not noticeable during a face following run.

demonstrates the side-by-side results of these experiments; my apartment has poor lighting, so all of my images are poorly lit.

Similarly, I wanted to see if it could detect parts of faces, up to the eyes and still track if some fraction of my head were cut off from its view. Tello has a somewhat wide-angle camera so I was not able to achieve this horizontally but vertically it was fairly easy to trip it up. The results of this experiment are shown in Figure 5. If even a fraction of my head were off the screen, the detector would not recognize my face.

I additionally experimented with having no face at all in view of the camera to see if the detector misclassifies non-faces. Surprisingly, it did misclassify non-faces, but only if a human face was not visible to yield a high confidence option. Tello particularly liked to focus on the thermostat, as shown in Figure 6.

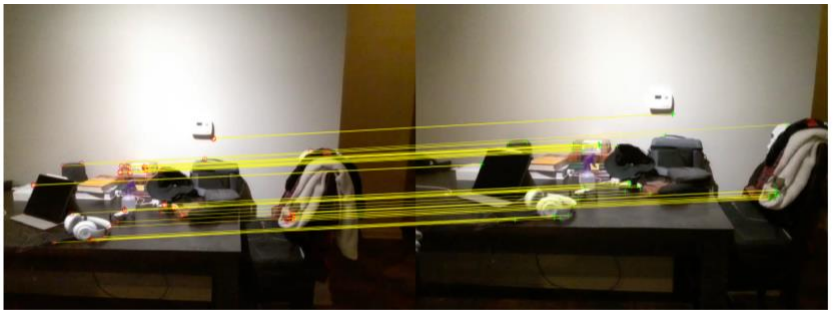

Figure 7. Example of a map initialization feature match. Typically, the map initialized and I could get a sense of where the features were.

Finally, I wanted to test the full functionality of my face following implementation. A POV video is attached in Video 1 [attached as a separate file]. Overall, the results of face following were a success. The process is fairly responsive but a little slow to update; the detector is particularly picky about vertical movement which has to be slow. Although, if you sidestep discretely and crouch and jump slowly it will follow your face well!

B. vSLAM Experiments

For vSLAM, I mainly experimented with having the Tello attempt to run it in different parts of my apartment. Different areas had different objects and more edges, which I anticipate is major factor in the feature extraction. In these experiments, I created a move sequence to cycle the drone back and forth, left and right, or in a square, both clockwise and counterclockwise. I chose these different move sequences to see if they made a different in the mapping and feature extraction. Unfortunately, since Tello cannot keep a strong connection reliably for a long enough duration to add many keyframes, I scrapped this idea because there were no obvious differences and I could not make any strong statistical claims about such findings with so few datapoints.

There were some limitations to running vSLAM on Tello, primarily the high probability of WiFi cutting for longer movement sequences and the low lighting in my apartment making feature detection that much harder. Often times, the feature extraction on the image would result in too few features due to a bad read on the image or instability. Typically, the map initialization worked well, and the matching yielded some good results. An example of a map initialization feature match is displayed in Figure 7. Examples of a good feature extraction and the average feature extraction in the main loop are shown in Figure 8. As suspected, the best feature extraction was often times in areas with nearby, crowded spaces with more edges to offer vSLAM.

In addition to the feature extraction, the point cloud map generation was also visualized for these runs. A couple of examples of the final point cloud maps and estimated trajectories are displayed in Figure 9. Since no more than 10 keyframes were acquired by Tello, the maps were small but the camera pose and estimated trajectory did a fairly good job at

estimating the Tello's travel given that there were only a few keyframe samples.

Figure 8. Examples of good (left) and average (right) feature extraction. Often times, the good initial feature extractions really set the momentum for how the rest of the main loop would turn out. Notice that the busier nearby area with more edges acquires more features.

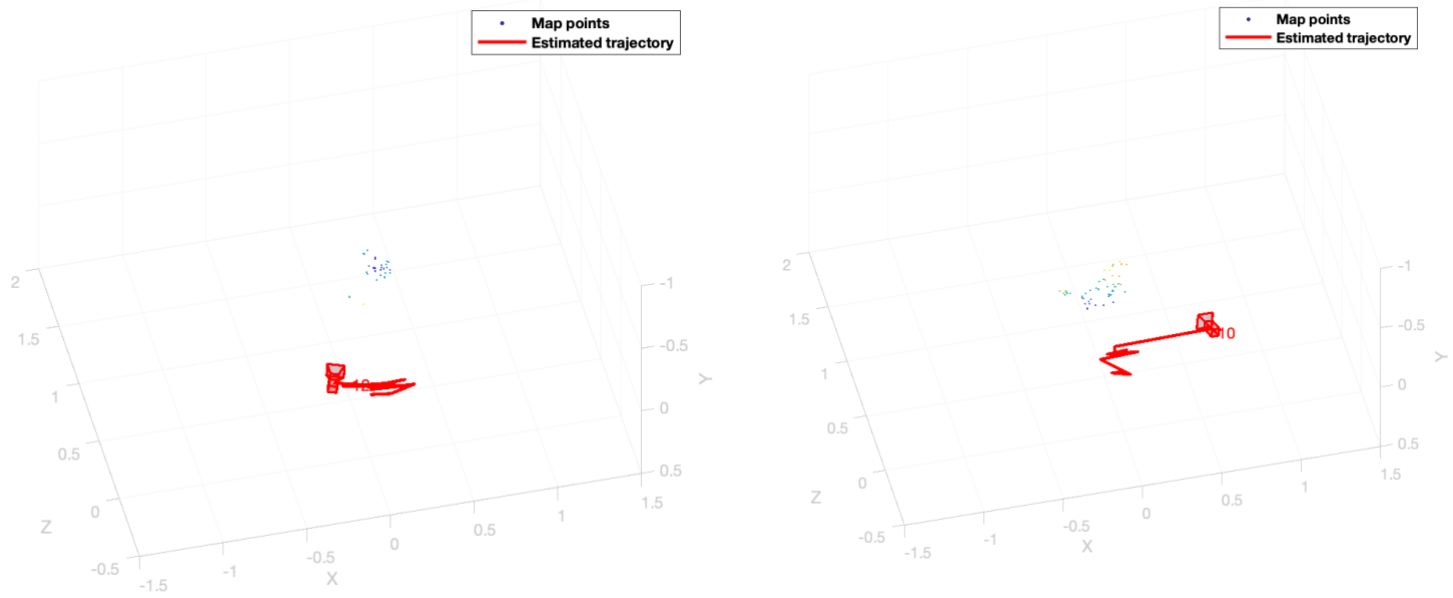

Figure 9. Examples of map plots and estimated trajectories and camera pose. Both of the movement sequences were left and right images and that the number on the camera indicates that there were 10 keyframes in this vSLAM run.

IV. DISCUSSION

Fortuitously, the face following and vSLAM implementations were at least functional. I believe that these results demonstrate. If anything, these two autonomous-leaning implementations set the walkway for building up to even more complex autonomous behavior using Tello, not considering the computational requirements and resources at my disposal currently. Of the two, face following works the best and vSLAM is slightly more trying of one's patience to get to work. As I mentioned earlier, persistence was the key here. One reason that face following might work better is that it is a simpler problem that has been studied much longer than vSLAM has and hence has more underlying optimization and definitely more documentation lying around. Overall, I am itching to develop this vSLAM implementation further; I feel as if I am just a few steps away from it working like it was meant to. If there is some way I can improve the Tello connection, improve its stabilization, and mitigate any of its other external factors that apparently cause a significant number of minor nuisances then I think that I could integrate face following and vSLAM to take quadcopter autonomy to the next level for me.

During this project, I assuredly spent more time reading code and papers to try and understand SLAM in general. I looked into using so many different SLAM approaches that it took some luck to stumble upon the MATLAB example and stare at it long enough to realize that I could repurpose some of its code to fit my needs.

Initially, I intended to use a Parrot ANAFI to implement this project in Python. At first the ANAFI seemed very promising; it had a somewhat maintained SDK in iOS and an

older version of the SDK in Python. But as I explored more and tried out the works, I realized that the Parrot SDKs were not actively being maintained, were so buggy that they were defunct, even on iOS, and that very few developers had ever even tried the SDK. At first, I thought I lucked out when I found a professor's library PyParrot library[5] but it too was not being maintained. Additionally, the library was not very flexible in accessing the camera feed and passing back up computed RC commands to the main function. I was able to get the ANAFI to fly to a predetermined, hard-coded set of commands and save images with detected faces but nothing more. By this time, I had discovered the MATLAB's Tello toolbox and decided to table ANAFI for another day. Even though I cannot recover the time that I lost to attempting to get ANAFI to work, I learned a lot about SLAM and what is available out there in terms of research

To become acquainted with the Tello toolbox, I followed along with the premier demo, spinAndFind[6]. The demo walks through a simple script utilizing a few of the key toolbox functions and actually uses an object detector to recognize Mona Lisa images around the room as it turns and builds a montage of a handful of images. I essentially used this script as a very small springboard reference since it was peripherally related to face following. Other than that, I for the most part constructed the logic of the code through trial and error and other times referring to a YouTuber's now-deprecated Python implementation[7] from before the MATLAB toolbox release that I had tested out in the summer.

Implementation Challenges

I spent many weeks and many fruitless hours trying to simply get drones to respond to my code. Obviously, I spent a substantial amount of my initial time breaking my neck with ANAFI. But the neck-breaking did not stop when I started working with Tello in MATLAB. My experience with programming quadcopters was admittedly much more limited than I thought before embarking on this project journey. I had to improvise frequently and do a lot of debugging. Early this semester, I definitely hyped myself and was far too optimistic about what I could truly accomplish in a reasonable amount of time – this was actually a theme for me this semester and a strong life lesson.

Particular to Tello, MATLAB's hardware support package does not have a substantial amount of support and there are a number of known bugs on the open forum[8], many of which I encountered while working on this project. Since the support package was only released this year, very few answers were available and even the devs seemed to be unaware as to why the bugs were occurring. Apparently, the battery, the temperature, and the lighting are all critically important for Tello to function. Additionally, indoors environments where the gusts from its own propellers can throw it off course would often push the Tello into a collision course from my wall, not to mention that spontaneous loss of connection with Tello would cause MATLAB to crash completely if the workspace was not immediately cleared.

In terms of implementing the functions, I encountered a lot of difficulty in programming the movement of the drone initially for the face following. It took a ton of manual tweaking of parameters that I had defined because the Tello was so sensitive to movement in some direction but not in others. Particularly, the support package does not operate in terms of yaw, pitch, and roll but rather x, y, z and turn. In some ways this approach was more intuitive, but in many it was quite limiting.

Implementing vSLAM was whole different beast in and of itself. As I scoured the web searching trying to find some hint of a viable SLAM implementation for a quadcopter that would not ask me to reinvent the wheel, I have never desired a Linux computer more. And a virtual machine just will not cut it for running SLAM, which invariably requires more memory and RAM than I could ever afford to give on my laptop. Nonetheless, I learned a lot reading papers and taking the time to shapeshift the ORB-SLAM example and actually understand the underpinnings of it.

V. Conclusion

In this project, I implemented and experimented with face following and ORB-SLAM on a Tello drone using the recent MATLAB support package. The road was tough but it was worth it, just like any autonomy task that is worth it. For both, the results bode well and even though vSLAM was slightly hindered by hardware it nonetheless turned out a result. With some different iron, these functions could be integrated together to take one step closer to full quadcopter autonomy. A future avenue of work that I foresee for this project is hide-n-seek; using face detect and following to hide and catch, vSLAM to learn and map the environment, and then using the generated map for path planning.

VI. References

[1] P. Viola and M. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," *2001 Comput. Vis. Pattern Recognit.*, 2001.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2016-December, pp. 779–788, 2016, doi: 10.1109/CVPR.2016.91.

[3] T. Lindeberg, "Scale Invariant Feature Transform," *Scholarpedia*, vol. 7, no. 5, p. 10491, 2012, doi: 10.4249/scholarpedia.10491.

[4] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-Up Robust Features (SURF)," *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, 2008, doi: 10.1016/j.cviu.2007.09.014.

[5] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 2564–2571,

[5] https://github.com/amymcgovern/pyparrot

[6] https://www.mathworks.com/videos/control-ryze-tello-drones-from-matlab-1595582029947.html

[7] https://github.com/Jabrils/TelloTV/blob/master/TelloTV.py

[8] MATLAB Support Package for Ryze Tello Drones (comments and ratings)

2011, doi: 10.1109/ICCV.2011.6126544.
[6] C. Cadena *et al.*, “Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age,” *IEEE Trans. Robot.*, vol. 32, no. 6, pp. 1309–1332, 2016, doi: 10.1109/TRO.2016.2624754.
[7] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard, “An Evaluation of the RBG-D SLAM System,” 2012.
[8] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, “ORB-SLAM: A Versatile and Accurate Monocular SLAM System,” *IEEE Trans. Robot.*, vol. 31, no. 5, pp. 1147–1163, 2015, doi: 10.1109/TRO.2015.2463671.
[9] B. Williams and I. Reid, “On combining visual SLAM and visual odometry,” *Proc. - IEEE Int. Conf. Robot. Autom.*, pp. 3494–3500, 2010, doi: 10.1109/ROBOT.2010.5509248.
[10] S. Li, C. Xu, and M. Xie, “A robust O(n) solution to the perspective-n-point problem,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 7, pp. 1444–1450, 2012, doi: 10.1109/TPAMI.2012.41.

VII. APPENDIX

The code and README will be attached as PDFs. Video 1 will also be uploaded separately. Code and usage documentation can be found at this GitHub repository: https://github.com/zstoebs/tello_detection_SLAM.