An Autonomous VIO-based Quadcopter

MEAM 620 Project 3

April 10, 2022

1 Introduction

The goal of this project is to integrate everything we have learned from this course so far! Let's do a quick review:

- Project 1 tasked us to plan trajectories and track them accurately, while the ground-truth states of the robot (positions and velocities) were given to us.
- Project 2 tasked us to estimate robot's state given noisy sensor measurements, but there was no robot autonomy (planning and control) involved.

Our goal for Project 3 is to integrate the state estimation from Project 2 with the planning and control from Project 1. In other words, you will use your Project 2 code to estimate the state of the robot, and use your Project 1 code to plan and track trajectories in the simulation environment.

Using visual inertial odometry (VIO) in-the-loop has proven to be a reliable approach to implement GPS-denied autonomy stack in academia [1, 2] and industry [3]. Once a quadcopter can accurately estimate its own state and use it for planning and control, the last step towards autonomy is to implement a mapping solution so the quadcopter can sense where the obstacles are using its on-board sensors.

The principle objective for Project 3 is to achieve quadcopter planning and control with on-board state estimation. However, extra credit exercises are included for students who are interested in completing the entire autonomy pipeline.

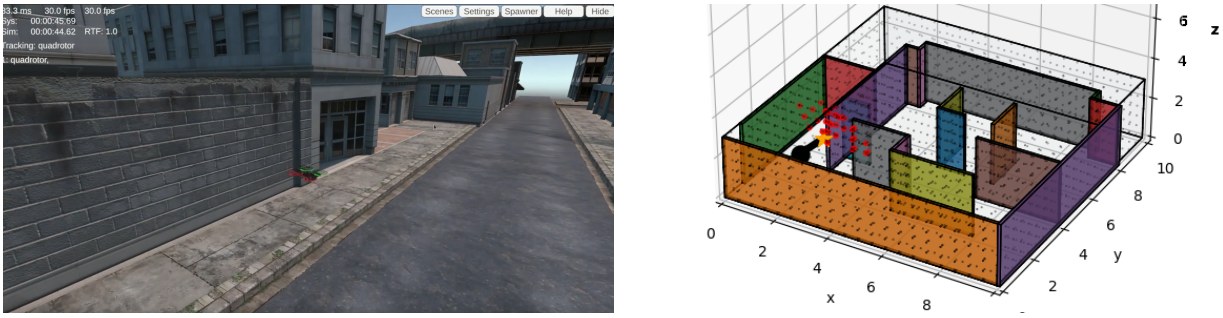
2 Simulator Implementation

A full simulator implementation of a stereo VIO-based quadcopter has to synthesize camera images from the robot position. This is the preferred approach when we want to test novel VIO algorithms, or when we want to test VIO-based autonomy. The drawback of this approach is that the simulation requires capable hardware to generate photorealistic camera images. Fig. 1a shows an example of such simulation environments.

Since these specialized hardware may not always be readily available, in this project we have generated the features beforehand in `flightsim` by attaching *markers* to the obstacles and boundaries of the simulator (Fig. 1b). The features are then projected into the camera frame of the quadcopter and this is the data that will be provided to you. This approach requires significantly less compute than providing complete frames and removes the need to use feature detection code.

3 Main Task : Trajectory Tracking with State Estimator

The task is the same as Project 1.3's, except that we do not give the controller the ground truth state. Instead, the state comes from your state estimator. It is likely you will need to modify your planner, controller, and/or even the state estimator to make the quadcopter able to track the planned trajectory.



(a) (b)

Figure 1: **Left panel:** A quadcopter in a Unity3D-based photorealistic simulation environment. **Right panel:** Example environment with sprinkled features in *flightsim* (the simulator used in the class). Features (small black dots) are sprinkled on walls, ceiling, ground of the environment. The robot center is shown as the big black dot, and the camera facing direction is shown as the ray from black dot to orange star (facing up, slightly tilted). The red dots show the features that are currently observable to the camera. Features are considered observable if (1) their distances to the camera are within a predefined threshold, (2) they lie inside the camera field of view, (3) there are no obstacles or walls exist between them and the camera.

We provide a visualization of the markers that the quadcopter is observing at a specific position. This should give you a hint on how to improve the performance of your vehicle planning and control strategies.
Hint: For you to better tune your code, you can play around with the sensor noise parameters in the file `flightsim/sensors/vio_utils.py`. Specifically, for IMU, those parameters define the noise: accelerometer_noise_density, accelerometer_random_walk, gyroscope_noise_density, gyroscope_random_walk; For stereo features, those parameters define the noise: image_u_std_dev, image_v_std_dev, image_measurement_covariance. You are recommended to **multiply those parameters with the same number** (e.g., 0.01 for lower noise level) instead of changing them individually. Note that the autograder will still use the predefined noise parameters.
Hint: An aggressive desired trajectory or PID tuning can increase the state estimation error. Start with lower gains and low velocities and be sure to reach the goal before trying aggressive trajectories.
Hint: We strongly suggest you to create new maps to test your quadcopter performance.

4 Extra credit (EC)

Extra-credit sections will require you to modify your code, but also the `flightsim` implementation. You may be required to add small modifications in the simulator to make your algorithms performant. You can attempt as many EC exercises as you want, but we strongly recommend to do one at a time.

Please note that each of the EC options below are open research problems. Hence, **TAs are not able to provide in-depth help on how to execute and implement each of the EC options below**. Nonetheless, we are always interested in discussing new ideas, so please come by to OH!

4.1 Online Mapping

Our current implementation relies on an "oracle" that is giving the position of the obstacles in the map. Usually, such information is not available for the quadcopter!

You can use the image coordinate and disparity of features that we provide to you to generate your map. You may need to sample features more densely for the mapping step (than the VIO step). Then, you can either:

- Solve mapping and planning simultaneously in a SLAM-like fashion.

- Generate a map after knowing the pose of your quadcopter (from the odometry algorithm) and the stereo features.

Goal: generate a mapping solution that allows the quad to fly without knowing the location of the obstacles beforehand.
Hint: you may need to re-plan your trajectory multiple times in shorter segments as you discover open corridors for flying. To better observe the obstacles along the trajectory, you may need to change the facing direction of the camera so that it (roughly) points towards the velocity direction of the quadcopter, and this can be done by adding a yaw alignment component in your planner.

4.2 Monocular VIO

Stereo VIO relies on ability to compute the disparity between images by placing two cameras separated by a distance d. If your quad is flying with some velocity, you can achieve this disparity by *spatial* means: two camera positions due to the robot motion will provide the disparity required.

Monocular VIO may be problematic when the quad is hovering, as there is not enough disparity to estimate the depth of the features. What would a viable strategy be when the quadcopter can only use one camera for VIO, *i.e.*, monocular VIO?
Goal: replace the current stereo-generated disparity with a monocular solution, and use it in your autonomous VIO-based quadcopter.

4.3 VIO loop-closure

Your VIO estimates the odometry based on the initial position of the quadcopter. Since the VIO is frame-to-frame tracking, as the robot flies, the error will accumulate, and the state estimation will *drift*. This behavior is better observed when the quadcopter flies a big loop: even if we return to the original position of the loop, the odometry may indicate a difference.

A traditional approach to solve this problem is to implement loop-closure: when we *detect* that we are back in a previous location, we can use this information to correct the current odometry. Furthermore, we can propagate this change to previous recorded points of the odometry, effectively reducing the impact of the drift.

Implementing VIO loop-closure in a control system can be challenging. When the loop-closure happens, the local position of the quadcopter can jump. This discontinuity in your odometry can negatively impact your controller.
Goal: Show an example of significant VIO-drift (+1 m), and implement a loop-closure mechanism to fix it. Demonstrate that the stability of your quadcopter is not affected when the loop closure happens.
Hint: You can use the spatial relationship of features around the robot to identify if it returns to a previously visited location. To better observe drift and demonstrate your loop closure, you may play around with those parameters: (1) adjust the camera orientation (i.e. change `self.R_body2sensor` in `vio_utils.py`), (2) increase the noise level of the IMU and stereo feature measurements (e.g. change `self.image_u_std_dev` `self.image_v_std_dev` in `vio_utils.py`), and (3) increase the length of your flight trajectory.

5 Code Organization

The code packet that you are being provided with as part of this assignment is organized in the usual manner. In the top level directory there is a file entitled `setup.py` which you should run in order to install all of the needed packages. The `proj3` package is divided into 2 subdirectories.

- The `util` directory contains a few maps that you can use for testing your planner, controller and estimator.

- The `code` directory contains a set of code files and sandbox files which constitute the coding assignment. You will need to copy your code from the previous two projects to this `code` directory `graph_search.py`, `occupancy_map.py`, `se3_control.py`, `world_traj.py` and `vio.py`.

6 Coding Requirements

You will be provided with a project packet containing the code you need to complete the assignment. For this phase you will need your graph search algorithm, trajectory planner and controller from Project 1, and state estimator from Project 2. In summary:

1. Copy over your Project 1 `code` directory `graph_search.py`, `occupancy_map.py`, `se3_control.py`, `world_traj.py`.
2. Copy over your Project 2 `vio.py`
3. Now that all your code resides in `proj3/code`. Make sure your `import` commands are adjusted accordingly in `proj3/code/sandbox.py` and `flightsim/sensors/vio_utils.py` (for example, `from proj1_3.code.occupancy_map` becomes `from proj3.code.occupancy_map`).
4. If necessary, improve your the implementation of `se3_control.py` and `world_traj.py`.
5. Use the provided `code/sandbox.py` to aid in tuning and analysis.
6. Test your implementation on a collection of given maps using `util/test.py`

7 Grading

7.1 Main Task

For the main task, a significant part of the grade will be determined by automated testing. You must find trajectories through six obstacle filled environments which your quadcopter must then quickly and accurately follow without collision. Performance will be measured in terms of the in-simulation flight time from start to goal. For each map, you will earn 8 points for a safe flight and up to an additional 5 points for a fast completion time for a total of 78 points. The time targets for each map will be in your Gradescope report; attaining full marks on every trajectory may be extremely challenging.

You are also required to submit a summary report, as described in Sec. 8.3. The report is worth 22 points.

7.2 Extra Credit

We will grade the EC independently. **If you attempt more than one EC, do not try to implement them at the same time**. The points for the EC are as follows:

- Mapping: +30 points, +5 extra points for (roughly) aligning camera with the flight velocity direction
- Monocular VIO: +20 points
- VIO loop closure: +20 points

For the extra-credit, your quad **has to finish in at most** $3\times$ **(for mapping) or** $2\times$ **(for monocular VIO and VIO loop closure) the time of your main-task solution**. So, if you finish in 7 seconds with your vanilla solution, you may take at most 21 seconds with online mapping, or 14 seconds with monocular VIO or VIO loop closure.

EC will be granted only if **the goal described in this document is achieved for all maps**, and **the quad flies safely to the target location**. We will test your EC in hidden test cases and your code has

to run successfully on those too. We will subtract points for small mistakes, but **we will not give partial credit for the EC items** (e.g. if you implemented a solution that is not complete or not working). If you are in doubt please write a Private piazza post.

8 Deliverables

You should submit a report document and your code.

8.1 Code Submission

When you are finished, submit your code via Gradescope which can be accessed via the course Canvas page. Your submission should contain:

1. A `readme.txt` file detailing anything we should be aware of (collaborations, references, etc.).
2. Files `se3_control.py`, `graph_search.py`, `occupancy_map.py`, `world_traj.py`, `vio.py` and any other Python files you need to run your code.

Shortly after submitting you should receive an e-mail from `no-reply@gradescope.com` stating that your submission has been received. Once the automated tests finish running the results will be available on the Gradescope submission page. There is no limit on the number of times you can submit your code.

8.2 EC Code Submission

If you attempt an EC section, you may submit this code in the corresponding entry in Gradescope. As the EC part of the assignment will be highly dependant on your implementation, we will not provide an autograder for this part.

A TA should be able to run your code with your instructions in a new python virtual environment. Please describe how to execute your code in detail. Include (if any) the modifications that are required in the simulator files.

8.3 Report

We ask you to write a 2-page report stating your findings in this project. Submit your report to the separate Gradescope assignment discussing, among other things:

1. What you have changed to your code since project 1 and 2.
2. Why did you need to make those changes.
3. Anything else you would like to discuss.

If you attempt an EC section, you can add +2 pages per EC attempt. We are interested in knowing:

1. How you implemented the EC.
2. How is the performance of your system compared to the vanilla implementation/simulator.
3. Are there failure cases in your implementation?
4. All the bibliography you used to implement the EC.

9 Academic Integrity

Please do not attempt to determine what the automated tests are or otherwise game the automated test system. This is an individual submission and must reflect your own work. You are encouraged to discuss the project with your peers in detail, but you may not share code. **Using your partners' code from Lab. 1.4 is not allowed, and it will be considered plagiarism**. Please acknowledge any assistance in detail in your `readme.txt`.

References

[1] K. Mohta, K. Sun, S. Liu, M. Watterson, B. Pfrommer, J. Svacha, Y. Mulgaonkar, C. J. Taylor, and V. Kumar, "Experiments in fast, autonomous, gps-denied quadrotor flight," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7832–7839, 2018.

[2] D. Thakur, Y. Tao, R. Li, A. Zhou, A. Kushleyev, and V. Kumar, "Swarm of inexpensive heterogeneous micro aerial vehicles," in *International Symposium on Experimental Robotics*, pp. 413–423, Springer, 2020.

[3] Skydio, "Skydio 2+." https://www.skydio.com/skydio-2-plus. Autonomous Drones for business, public safety and creative endeavors.