AmadorUAVs 2024-25 Software Entrance Assignment

Introduction

This is the entrance assignment for the Software division of AmadorUAVs. If you want to apply for the Software division, this is the right place.

The Software division of AmadorUAVs is tasked with developing reliable, usable, and efficient software for both the drone and the team. Throughout the competition season, we will work on developing a navigation system and creating a software stack that can successfully navigate around the competition field as well as creating an accurate object detection stack. Over the year, you will gain hands-on experience with multiple software technologies and work as a team to create a working stack. Software is an extremely important part of the club, and the stress is on us to make the drone fly to where it's supposed to go. It is up to us to make sure the drone completes the mission properly and succeeds at its designed tasks.

This assignment will evaluate your prior knowledge and ability at programming. While prior knowledge is encouraged, we are looking for programmers who are **determined** and able to **problem-solve.** Googling is encouraged; our club runs at its best when our members can teach themselves. Good luck!

Please do not share code or collaborate/receive help from others. This is supposed to be a test of your problem-solving—**AI might be able to solve the problems. Still, the synchronous interview is meant to examine YOUR knowledge and process, so using**

it will hugely affect your application. In addition, plagiarism will rescind both the person offering help and the person receiving it.

Part 1 - Can You Code?

This part of the assignment will evaluate whether you can effectively code a solution to a problem. Solve the following problems. You may use C, C++, or Python; if you have another language in mind, email prathampmehta@gmail.com. You must keep a *readable* commit history using GitHub to ensure impartiality and document your problem-solving process. **Google is allowed; please use it :)**

For each problem, you will be required to use your chosen language's IO libraries to read input from a file and write output to another file, with the names of each file given in the problem statement. Please follow the given format for input/output files, including all spaces and new lines.

Each problem will be stress tested against cases not provided here aside from just the example test case. Ensure your program can reliably pass test cases with a runtime of no more than a couple of seconds–efficiency is a huge factor in the code we write for our UAVs.

0. References

Resource for GPS distance: https://www.calculator.net/distance-calculator.html

ODLC - ODLC is an abbreviation of Object Detection, Classification, and Localization which represents the specific task we must perform at SUAS, however, ODLC is also

interchangeably used with the standard objects, which must be classified in this task, on the ground which defines the delivery location for each payload (water bottle). These standard objects are a colored shape cutout 8.5x11 inches in size with an alphanumeric character on it.

Standard object (left; white A on blue triangle) and Emergent object (right; manikin dressed in clothes)

To learn more about the specific tasks that must be performed at SUAS, I recommend looking at the 2024 rulebook.

YOLOv8 - YOLOv8 is an object detection, segmentation, and classification model that be can trained and accessible through Python's `ultralytics` as well as a command-line interface (CLI). YOLOv8 is used frequently as it features impressive speed and accuracy with just a single forward pass (You Only Look Once). In UAVs, we use YOLOv8 for color, shape, and pinpointing of ODLCs. To learn more about YOLOv8, look at the Github repository.

MAVSDK - MAVSDK allows us to communicate with MAVLink, a messaging protocol for supported systems, such as drones. MAVSDK is available in a few programming languages, but we only use its Python wrapper (MAVSDK-Python) to programmatically access our telemetry, mission progress, and other onboard operations. To learn more

about MAVSDK, look at the introduction page and MAVSDK-Python documentation.

NED - NED coordinates are defined with the x-axis pointing north, the y-axis pointing east, and the z-axis pointing down hence the name North-East-Down. MAVLink uses NED coordinates compared to GPS coordinates because they are local where the reference frame is described by the vehicle's current position and orientation rather than global which can lead to imprecision in small-scale navigation. To learn more about NED, I recommend looking at MATLAB's comparison of 3D coordinate systems.

1. Find the Center

Overview:

One of the software division's primary tasks is localizing the GPS positions of ODLCs, which are ground markers, to indicate where payload drops should occur. As part of the competition, we autonomously identify and localize these objects on the ground using a downward-facing camera and a YOLOv8 image segmentation model. We take and process pictures every 50 milliseconds, giving us hundreds of ODLC position estimates that must be condensed into 5 unique collections. Lastly, in the competition, there are emergent objects that may be accidentally identified as an ODLC in the airdrop zone; these outliers greatly skew the center of the collections away from the "true" position.

Your Task:

- Find the centroid of every collection, representing the "best" possible estimate of the ODLC's position given N-ordered pairs describing (latitude, longitude) points.
- Filter out outliers as best as possible (*note: It does not be the same as mine for every test case*) (hint: Google!!).

Restrictions:

Don't use non-standard libraries except `pandas`, `sci-kit-learn`, `numpy`, `matplotlib`, and `geopandas`; if you want to use any other libraries, email prathampmehta@gmail.com.

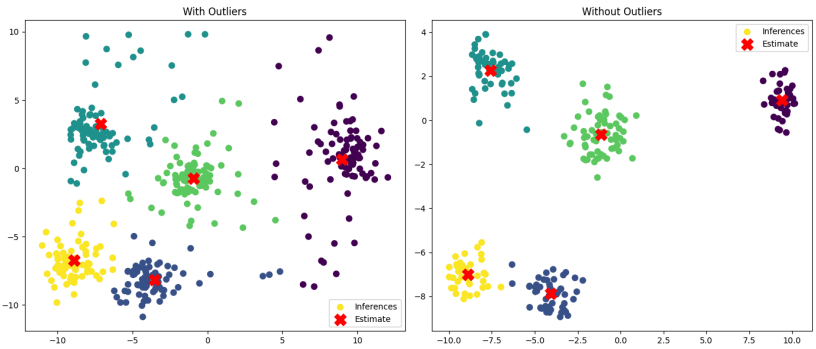
File Input Format:

The first line of input contains N followed by N lines with $latitude$, $longitude$ separated by spaces of each point.

File Output Format:

Output 5 lines, each with the identified center $latitude$, $longitude$ rounded to 5 decimal places separated by spaces of each point. The coordinates do not have to match the ones provided exactly, but they should be within 20ft or 0.0038mi away.

Demonstration:



Note: The coordinate axes are scaled for simplicity, but GPS coordinates often only differ $0.000x$ *for small distances.*

P.S: This demonstration assumes a flat world, but remember that we live on a sphere. How would you need to change your solution to account for this?

Example:

In (inferences.txt):	Out (centers.out):
no outlier.txt	37.68160 -121.89258 37.68179 -121.89246 37.68201 -121.89267 37.68221 -121.89293 37.68222 -121.89222
with outliers.txt	37.68147 -121.89240 37.68209 -121.89235 37.68227 -121.89234 37.68227 -121.89287 37.68230 -121.89261

2. Navigate

Overview:

One of the software division's primary tasks is creating an autonomous flight plan. At SUAS, we are given a list of 10 GPS coordinates as waypoints our UAV must fly to, and a geofence, essentially a polygon (convex or concave) defining the UAVs bounds. We use an edge smoothing algorithm known as centripetal catrom-mull spline (if you want to learn more, I would recommend using this Desmos sandbox), but for this question, we can assume that we just connect the waypoints, forming an n-gon. However, the polygon

sometimes intersects with the geofence, resulting in an invalid flight plan, which must be corrected. Lastly, we use autopilot software known as QGroundControl (QGC) where plans can be imported using a .plan file.

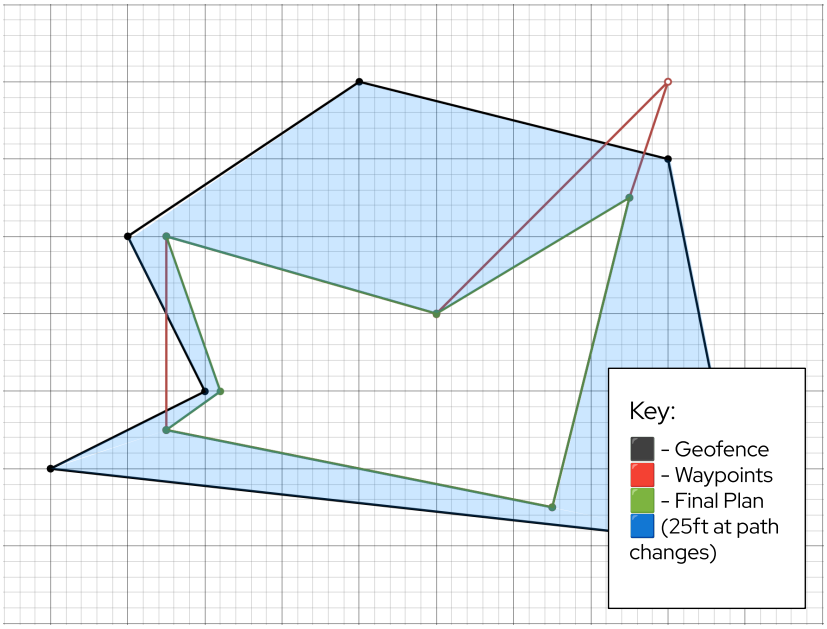
Your Task:

- Fix the flight plan by traversing within the geofence with a 25ft distance from the edge instead of exceeding its bounds (see demonstration image).
- Generate a .plan file (hint: use `JSON`) which can be imported in QGC. Learn how these files are formatted by installing QGC on your personal computer (QGC Install) (if you are using a Chromebook, email prathampmehta@gmail.com, and we'll try to find a compromise)

Restrictions:

Don't use non-standard libraries except `shapely` and `numpy`.

Demonstration:



File Input Format:

The first line of input contains N, M followed by N lines with $latitude$, $longitude$ for the geofence coordinates separated by spaces of each point, followed by M lines with $latitude$, $longitude$ for the waypoints separated by spaces of each point

File Output Format:

Output the `navigate.plan` JSON file (to view, I recommend using a JSON editor) which can be directly imported into QGroundControl (assume the altitude for all coordinates is 100ft and waypoint speed is 30mph). The waypoint coordinates do not have to match the ones provided exactly, but they should be accurate to 4 decimal places.