Drone Collision Avoidance with NED Vectors

Ben Merrick

Section 01 - Introduction and EARS Requirements

For this assignment, we were tasked with implementing a simple drone collision avoidance program. Essentially, we have two drones that are flying their specific routes. If they come within range, we must ensure that the drones do not collide if they are bound too. To begin, I started the basics of my project by outlining the requirements for the code and main collision algorithm in the EARS format. Below are the three main requirements that I decided were necessary to create a successful program.

1. While in flight each drone shall fly to their designated target and check for imminent collisions throughout the flight path
2. When a collision is imminent the ground control class shall tell one drone to stop and the other to continue its flight path
3. When the drone not chosen to stop has moved past the calculated point of collision, the stopped drone shall resume its initial flight path

These requirements mainly outline the specifics of the algorithm to detect and avoid collisions. However, as the collision avoidance algorithm makes up most of the code, then these requirements are perfect for the scope and development of my project.

Section 02: Collision Avoidance Algorithm Explained

The idea behind my collision avoidance algorithm was fairly simple. The first step of the process is to check the proximity of the two drones in the air. If they are flying within a provided range, then the two drones may experience a possible collision. This is the initial phase of the algorithm which simply checks for the possibility of a collision as outlined below in Figure 1.

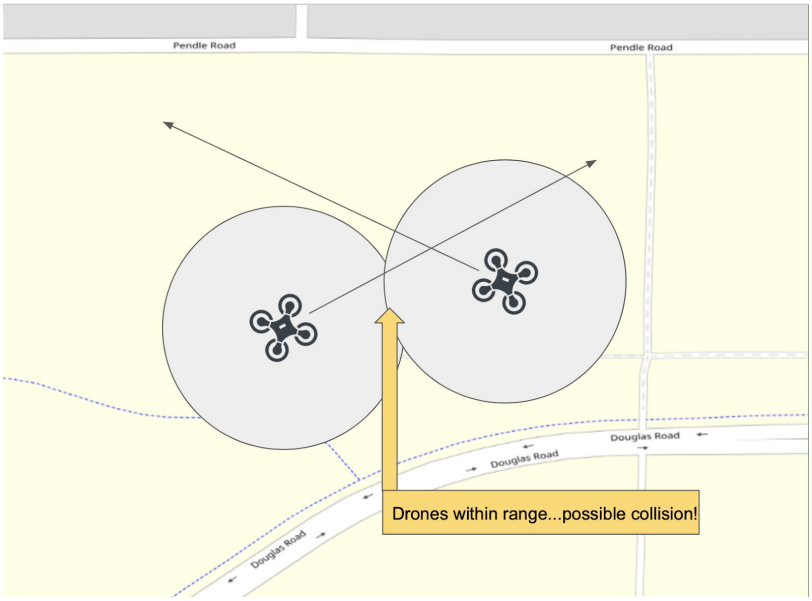

Figure #1: Drones within tolerance range indicating possible collision

If the drones are within a calculated tolerance range, we must then calculate their current geometric trajectory in order to determine if they will collide. We enter this new phase to rid the algorithm of edge cases such as drones flying in parallel which will never collide. The first step checks for a possibility of a collision and this step calculates if the drones are on a path to collide. By using a provided Python library, I will be able to determine if the vectors provided from the start and end coordinates of each drone will intersect. By finding this point, we will have determined that the drones are flying on paths that will intersect. This idea is outlined below in Figure 2.

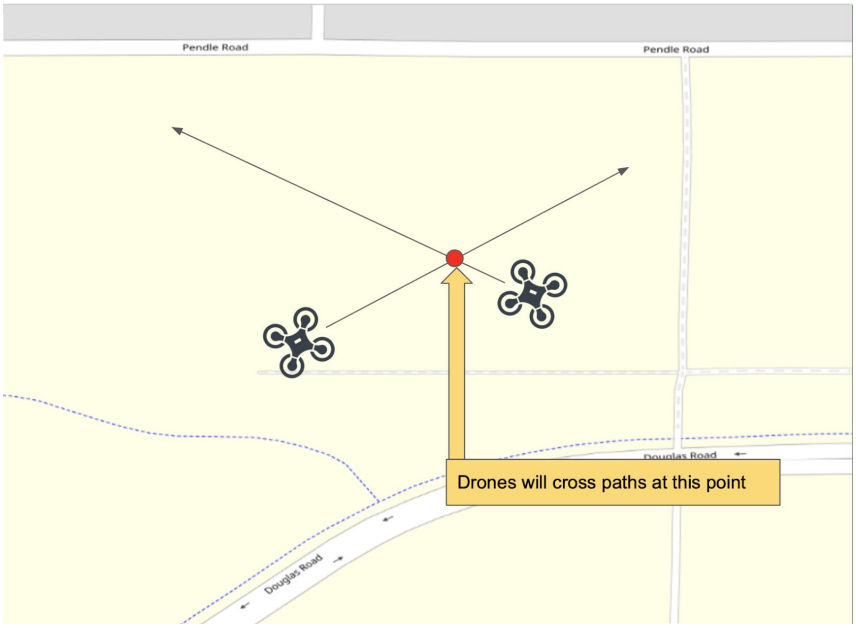

Figure #2: Calculated the point at which drone paths will cross

Once we have calculated the point of collision, we must then determine how long until the drones intercept that point. This point only tells us that the drones will intersect at some point during the total flight path. One drone could cross that point far before the other drone does which would not indicate a collision. As a result, we can then use the simple physics equation $V = m/s$ to determine the time until each drone reaches that point. We can calculate the distance to the point using a help function and the velocity is given to us as the drone's groundspeed. From there, we must check to see if they will arrive at that location within a time range. If they do, then we have ourselves a drone collision! This is outline in Figure #3 below.

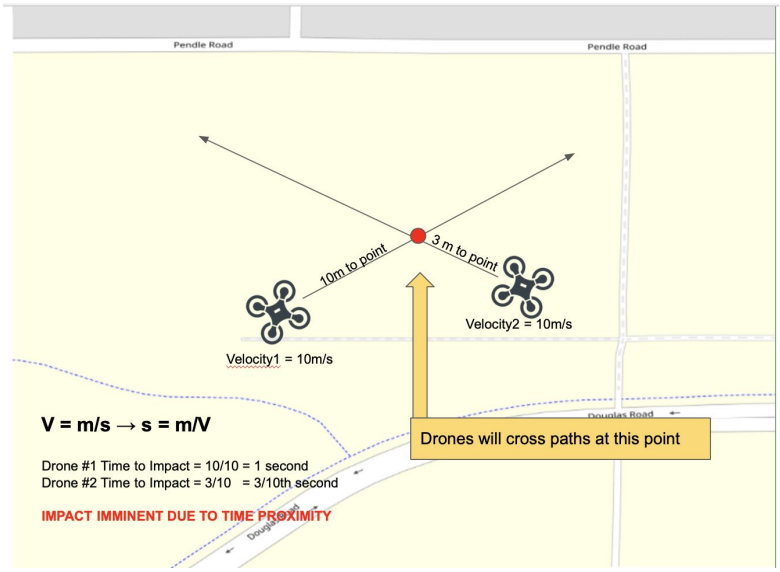

Figure #3: Time until collision calculated

Once we have determined that the drones are indeed going to collide, we must engage in the last step of the process. This final phase is telling the slowest drone to stop until the other drone has passed the calculated collision point. There are much better methods for actual drone collision instead of stopping one drone and waiting, however, I am in a team of one and I believed this would be right within the scope of my project. Nevertheless, in this method, the slowest drone will stop and wait until the other drone has passed the point of collision until it can resume it's normal flight path. This is outlined below in Figure #4.

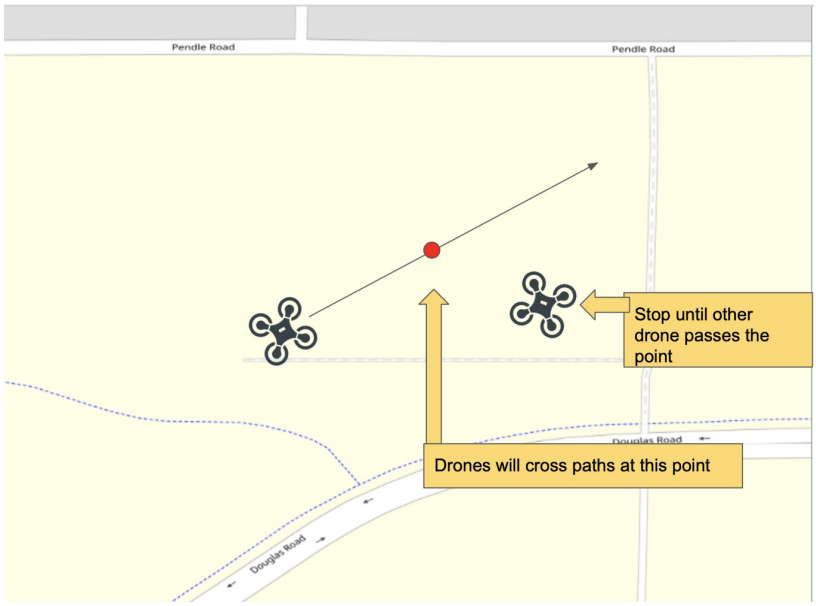

Figure #4: Slowest drone stopping until other drone has passed

After all of these calculations and actions, we have then successfully avoided a collision and the drones can continue on their normal path. The reason for this waterfall type function calling is to prevent an overload of calculations each time we calculate a NED vector. Instead,

we can check one piece at a time until we get to the final desired outcome of detecting and avoiding a collision

Section 03: Unit and Acceptance Tests Review

After building my final algorithm and testing it along the way, I built several unit and acceptance tests to test the accuracy of both the specific collision function and the entire program. First, the unit tests only tested the accuracy of the collision detection algorithm. Using a special flag, the unit test returned the proper information regarding a possible collision. In order to successfully implement the test, I had to build drone objects, a ground control class, and take off the drones. However, I did not fly the drones during testing. I only tested the collision program based upon their initial takeoff location. Below are the results for the two unit tests I created.

Test ID #	Desired Result	Actual Result
0	No Collision	No Collision
1	Collision	Collision

Table #1: Unit Test Results

As you can see from the table above, the unit tests demonstrated that the algorithm worked in basic practice. In Test #0, the drones were given two different starting locations with end goals that did not cross that were far apart. Therefore, a collision was not possible and the algorithm accurately determined this. For Test #1, the drones were given the same starting locations with end goals that did cross. Therefore, a collision was possible and the algorithm once again, accurately determined this. After I completed my unit tests, I moved onto a full implementation of the code which involved the actual “stop and wait” principle as demonstrated in my acceptance tests.

In my acceptance tests, I tested four different scenarios. Drones with an angled intersection path, drones intersecting at a cross, drones flying head on towards each other, and drones flying in parallel. They were also given varying speeds as detailed by a NED scalar integer. In each scenario, the algorithm appeared to act as it should and avoid collisions. The graphs of the flight paths of each acceptance test are shown below, but it is difficult to mark the exact point where the slowest drone was told to stop and wait. So, I have done my best to indicate through markings where this point could have been.

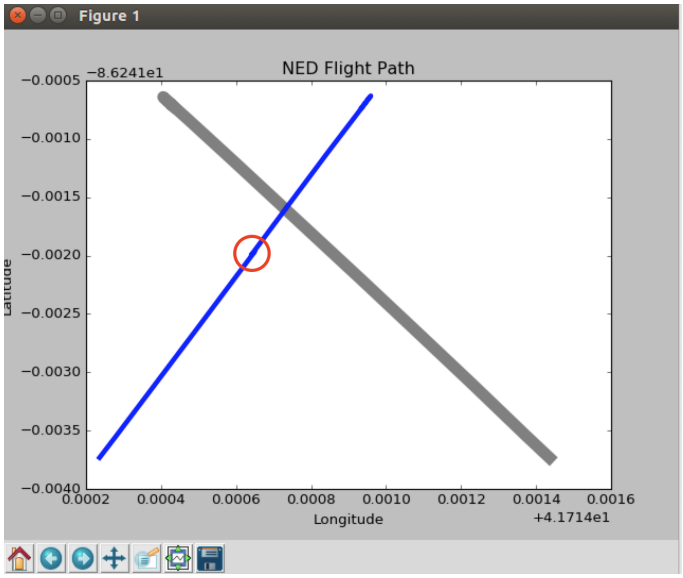

Figure #5: Angle intersection of flight paths

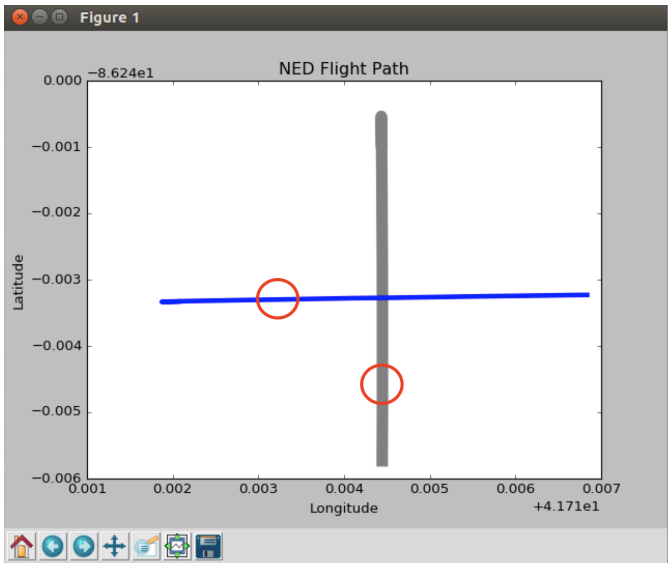

Figure #6: Cross Intersection of Flight Paths

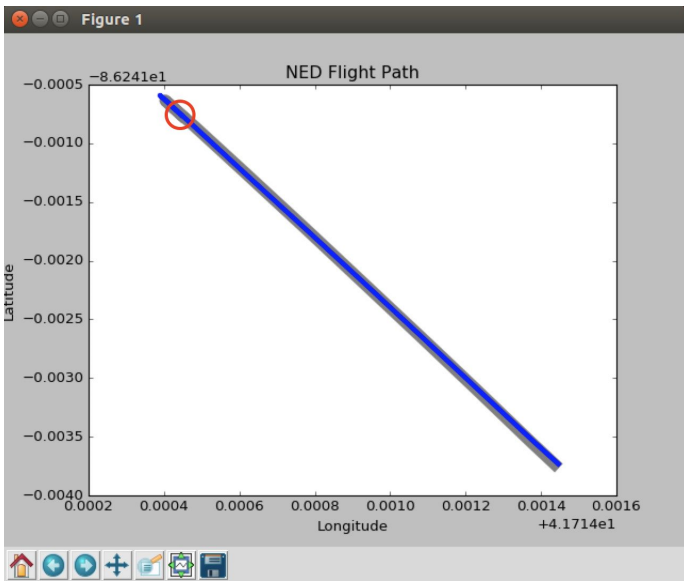

Figure #7: Head on Collision Flight Path

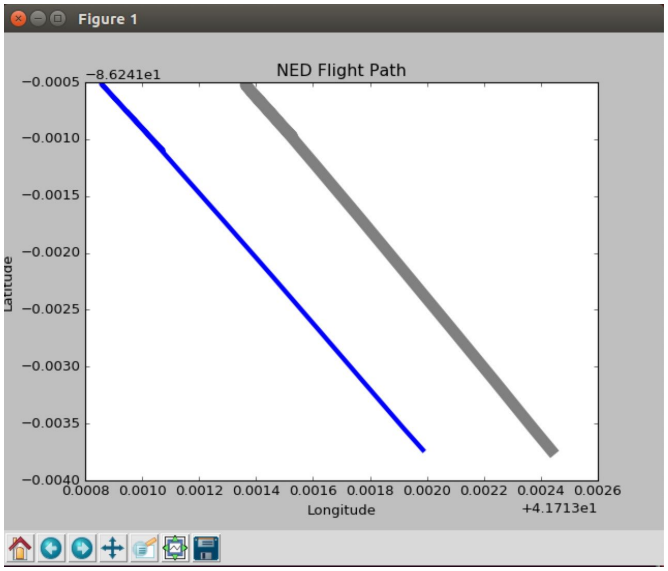

Figure #8: Parallel Flight Path

As you can see in the first three figures, the believed spots of one of the drones stopping is circled by the red marks. In examining the code output, you can actually view the indicators of whether or not a collision is imminent, the exact location the two drones cross paths, the time until the drones cross paths, and the stop and wait actions of the drones. These graphs mainly represent the flight paths of the drones. Nevertheless, the algorithm still worked as designed and a successful implementation of a collision avoidance program was created.

Section 04: Concluding Thoughts

In conclusion, I was able to create a simple variation of a collision avoidance program. There are indeed bugs within my program that I noticed through final testing and development that if I had more time, would fix and test. One such example is when the drones are flying to the exact same, but opposite end points. Figure #6 does its best to mimic this, however the two start/end points of the drones are not exact opposites. This is because the Python function I use does not account for a slope of 0 and results in undefined behavior when attempting to determine the point of intersection. Another bug within my code is the provided tolerances. Right now, I have the tolerances for checking the proximity and time to collision as relatively high. This is because I struggle with the NED vectors and their speed and the drones are often going past the desired point of contact before I can calculate it. As with any programmer reviewing their code, they will always find bugs, but with my current implementation, I satisfy the requirement of a collision avoidance program.

If I had more time, I would have worked on a more reactive model for collision avoidance. Instead of the stop and wait approach, I would have used a model where drones alter their current paths to avoid flying drones. This would have been very complicated and out of scope for my project as I am a team of one. Nevertheless, this would still be a great challenge to take on if I had more time.

All in all, this was a really interesting and fun project. I really enjoyed building this code and hope you enjoy my basic drone collision avoidance program. Please refer to the code print/output statements for exact details regarding collision avoidance for specific acceptance tests