Stuart Campbell
SN:7796403
COMP: 4560

Graph-Based navigational model for Crazy-flie 2.0: Report

Forward: due to the nature of this project, a repository has been linked here. Testing results will be included in the report directly however the raw data, code and larger data types and records will be stored in the repository and will be referenced here in.

Github: https://github.com/StuartLiam/DroneNavigationOnboard

Abstract: The original idea of the project is to take some number of drones (one or more) and create a vector of sensor data from them. We then feed that data into a computational model which processes that data and returns us some graph based navigational model. We then use that model to define the actions the drones are to take to smartly navigate obstructed and non-obstructed spaces and successfully preform simple tasks. However, the sensors that where to be recorded demonstrated a high variance between exact runs. This uncertainty leads to a poor compatibility with a purely data driven and dependent model such as a perceptron. Ultimately a space decomposition-based approach was taken to produce a similar graph-based navigation.

II Introduction to the topic: This project will focus on a graph based navigational model that emphasises heuristics to be applied to a wide range of navigational problems instead of being purpose built for a specific task.

IV Related work:

The closest lab to the project scope at the university is the Autonomous Agents Lab whose work surrounds autonomous bipedal agents whereas our scope is more related to quadcopters. Recently an SAE student group was formed around the idea of drone development, they are related more so to tuning drones and less so implementing specific models onto the drones.

V Problem Statement:

Currently, the methods and solutions to either abstract or complicated drone tasks are relegated to human control or rigid solutions that preform poorly on problems that they are specifically designed to tackle. Inspecting railway bridge stresses and conditions in the remote areas of Canada are routinely checked with drones. How does one check thousands of different bridges in arbitrary locations to make sure that they comply with a set of standards. It’s a combination problem of correct navigation and pattern matching. While the expert knowledge can be offloaded to engineers, our systems aims to implement an abstract navigational ability which is adaptable to its surroundings and task factors.

VI Methodology:

As a proof of concept this project only considers a 2D plane as its problem space. This reason is 2-fold. The drones are expensive and would be difficult to replace, any unnatural height gain and subsequent loss of control would put an unnecessary amount of impact force on the drone. In testing the drone showed no inbuild process to land safely when an error occurred. Upon a python script error, packet loss, unseen obstacle; the drone would quickly death spiral while maxing thrust. As this particular behaviour was observed early in testing any height gain above 1m was quickly ruled out. Secondly, the drone compatible sensors where not robust. The drone has 4 simple lidar sensors in each cardinal direction and one directly upwards. These sensors have little to no spread. This makes complex scanning of 3D space prohibitive as the drone would have to be directly perpendicular to a surface and close to recognise the obstacle.

At a high level the drone is initialized into a 2-dimensional plane of some predetermined size. Though this can be set arbitrarily high if exact dimensions of the area in question is not known. The drone acts in this known space and tries to reach a preselect goal. Numerous goals can be declared, and the drone will navigate them in the order that they are declared.

As the drone moves about the space it will detect obstacles. Once detected, the drone will attempt to scan a length of the surface and create a simple geometry obstacle that it knows to avoid. Once created the drone will decompose the known space based on the FBSP space decomposition algorithm except it does not statically decompose the world every time it records a new obstacle, instead it dynamically decomposes existing blocks that intersect with the new geometry. This is done deliberately in in effort to reduce error cumulated over time and leverage the drones pre-existing knowledge of its space. Since the drone's error is cumulative, the uncertainty with which the algorithm places every new obstacle will increase as it gets less sure of its coordinates. Thus, decomposition based on older splits with older found objects will have a lower error and have a higher overall accuracy reflecting their real-world locations. This is especially important in cases with more than one goal, where the drone is forced to back track to previous known nodes. Though the heuristic does not consider the age of the nodes as it assigns weight, the algorithm is designed to keeps old nodes for as long as they are useful.

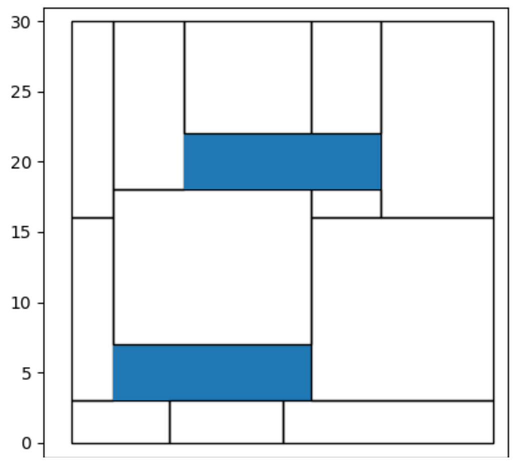

Figure 1: An example of how the drone sees and stores the world. For a world of size 30 the white blocks represent empty space the drone assumes it can fly through. The blue space represents the known edges that the drone cannot cross over.

In the drone's understanding of the world, it knows blocks of traversable space considered as **flyable**, blocks that it cannot navigate called **obstacles**, it knows its position and the position of its current goal. While the drone considers itself and the goal to be in the same block, it will path directly to the goal. If, during this path it detects an obstacle in its way, it scans the surface detected and creates an obstacle around the surface and performs decomposition. A video of this behavior has been uploaded to the Git-hub for reference. The surface is given an area even thought the scanned area is just a line after a successful implementation of this concept by intel labs[1].

Stuart Campbell
SN:7796403
COMP: 4560

The reasoning is that; the drone in the heuristic and algorithm is considered a 1D point, while in reality the drone is actually about 10cm by 10cm. So, the area surrounding the line object is a safety net for the area of the drone. This way, we do not have to dynamically check intersections between a drone area traveling along a path and 1D obstacles but instead we can check a drone line path against object geometry which is computationally less intense.

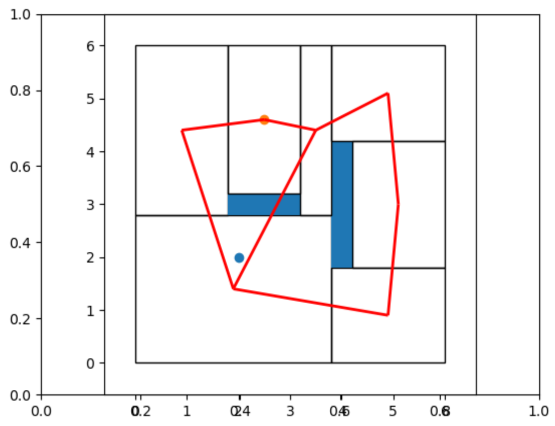

Figure 2: A simple space graph with the shown navigational graph overlay. The starting point is the orange dot at 2.5,4.5 and the goal is the blue dot at 2,2. No processing has been done on the navigational graph yet. Though the traversable edges are not efficient, they do stay away from the obstacles safely.

If the drone is not in the same block as its goal it will look to see if it is at the center point of its current block. Each flyable block in the space has a node at its center that the navigational graph keeps track of. These are the nodes that the drone uses to take advantage of the space decomposition. Flyable blocks with adjacent areas are considered having an edge and are traversable. The nodes are placed thusly to minimize the chance of a drone getting close to an obstacle and colliding with it. In essence it is a safety measure that, while it will extend travel time and path length of the drone, ultimately creates reliable paths for the drone to traverse and helps maximize the chance of a successful path.

On top of decomposition the algorithm contains a few helper functions that improve the safety of traversal. A particular issue is when 2 traversable nodes are adjacent but the vector between their center-points traverses an obstacle (Fig. 2). In this case the algorithm looks at the 2 blocks or flyable space in question and splits the larger of the 2 in half so that after split, the resulting center-point edges no longer traverse an object.

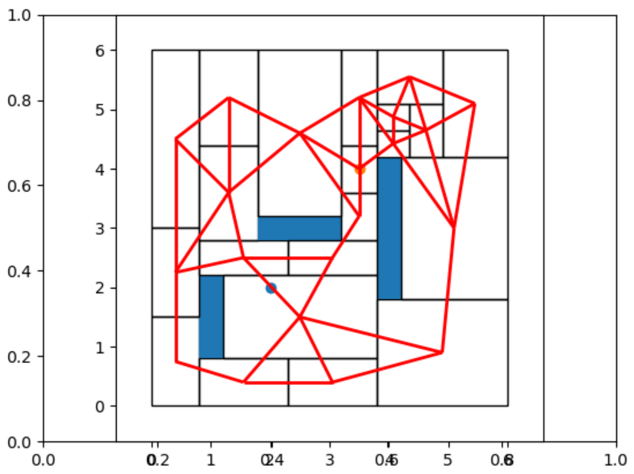

Figure 3: A final navigational graph. All edges shown are safely traversable and would be used to traverse and remember a space.

In particular cases of continued decomposition some flyable spaces are decomposed to be too small to be traversed reliably. This can best be observed in irregular and small obstacles. In this case a combination function is called to recombine some blocks until the traversable area only contains usefully large blocks.

Once at the center-point of its current block, the drone will look at the available edges connecting its block and adjacent blocks by their center-point nodes. The drone will then set a path for the connected block node with the best heuristic in a greedy fashion. Similar to all other times while the drone is in the air, if a new object is found, it scans the obstacle and decomposes the affected blocks. The drone does not implement a* namely due to the dynamic nature of its space. Since obstacles will be dynamically found, the path to get to the current location from the origin is dynamic which makes the G(x) arbitrarily less important and so it really isn't considered a quality for a best path.

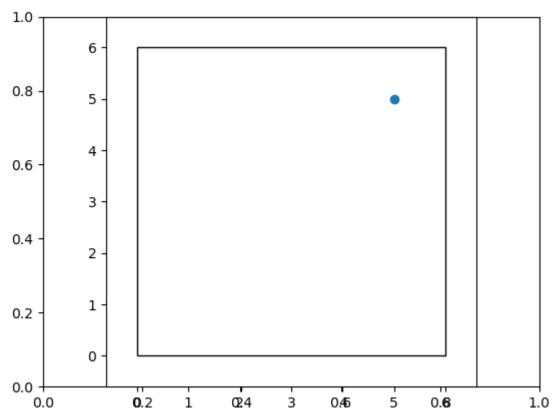

Figure 4: Step 1 of the dynamic space decomposition. The world is supposed blank, path towards goal. Assume the drone is pathing from 0,0 to displayed goal

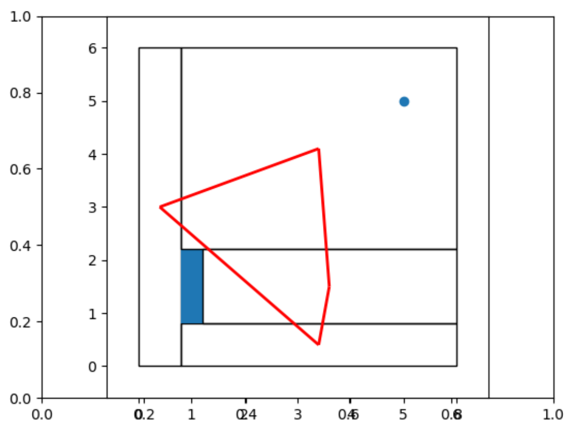

Figure 4.2: Step 2 of the dynamic space decomposition. An obstacle is found, the world is decomposed for that one block. Paths are primitive.

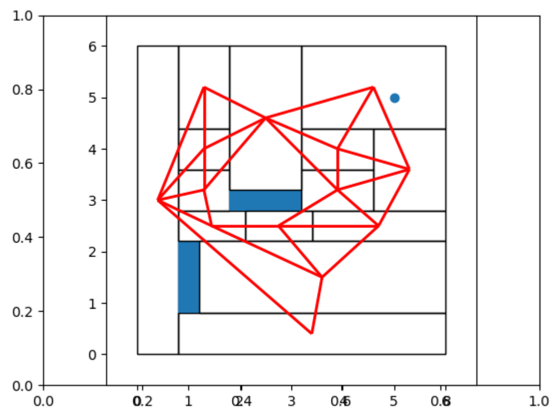

Figure 4.3 Step 3 of the dynamic space decomposition. Space is decomposed around a second block. While the edges start to appear closer to the obstacles, the obstacles include a large forgiveness zone so the physical drone will be rather far from the actual obstacle.

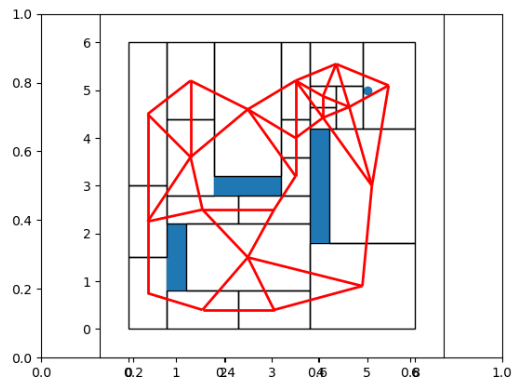

Figure 4.4 Step 4 of the dynamic space decomposition around the 3rd block. The complexity of the navigational graph becomes more apparent. Though the new blocks will only be dependent on 3 degrees of separation of accuracy from the starting coordinates.

A note on error. Much of this report talks about error. In essence what this project aims to provide is an outlook on a method to navigate a drone without external sensors. The main shortcoming with this approach is that the drone has no way of double checking its position. Unlike standard approaches, it has no external anchors or lighthouses to tell it where it is in its space. Thus, it must rely on its onboard sensors to calculate the current position. However, the onboard Kalman estimator computes the current position based of the previous position. So, if the previous position had an error of E, the current position will have an error of E + C where C is the error of the computation. Thus, this chaining of error extends backwards to the start position. Effectively meaning that with every computation of a new position, the drone gets less sure of its exact position. The navigational graph implemented is a method of reducing this cumulative error allowing the drone to make safe and efficient paths in the space.

The heuristic. The heuristic is applied to every node in the navigation graph anytime the graph is updated. It starts by giving each node a weight of an arbitrarily large 99999. Since this heuristic is looking for the lowest score, all the nodes default to unreachable. It then looks for the block that contains the goal. This block is given a weight of 0 since you could consider this the "goal block". Then the heuristic looks at all of the blocks that share borders with that block or in other words looks at nodes that share an edge. The new node from the goal node then compares its current weight to the weight of the node its coming from plus the distance between the 2 center points. If that new value is smaller than the current weight replaces it. This spirals out, and this operation is performed on every edge. This leaves us with a navigation graph whose nodes have a weight value of how close there fastest walk is. From a node, a drone chose from the available edges, the node who's weights plus its distance is the smallest.

However, as the heuristic is, it does not prioritize the safety of the walk and may chose a path that is arbitrarily difficult to navigate by means of small decomposition blocks. So, what the heuristic does is for every block that has a center-point node (they all do) it normalizes the weight based on its percent area of the total world area. So, if some computationally best node N1 is adjacent to the node but it is a small block the heuristic will be inclined to chose N2 a slower but much larger and thus much safer block. It does this to maximize the chances of a successful walk. Since realistically what we are trying to maximize with this algorithm is not speed but repeatability and reliability.

We also need to acknowledge that the drone is an imperfect flyer and with its limited sensor capability we cannot reasonably rely or the drone's ability to react dynamically to obstacles. Thus, some inbuilt safety preference is preferable. Making many small adjustments will unnecessarily increase the uncertainty with numerous yaws and small movements at awkward angles. When weighing the graph nodes for traversal, consideration is given to the size of the decomposed blocks. The weight of the nodes is normalized based on the size of the block they represent. This way the heuristic tends towards choosing larger areas of open air to fly through.

Stuart Campbell
SN:7796403
COMP: 4560

VII Implementation:

Drone:

The drone used is the crazyflie 2.0 made by bitcraze a small 10cm by 10 cm quadcopter with an opensource firmware. The drone is equipped with 2 extension boards. The multirange which adds the cardinal direction ranging functionality used to detect obstacles and the flow v2 board used to estimate current positions and stabilize flight.

An additional "dongle" used for communication was also used with the base station(laptop).

Main issues raised and accompanying solution:

While many minor issues accompanied the drone, only one major set back occurred. A superior crazyflie 2.1 was provided in addition to the crazyflie 2.0 used. When building the newer drone, it was discovered that a part had been installed incorrectly. The drone successfully flashed the firmware and flew manually perfectly fine. But when both the flow deck and the multi-ranger deck where installed the crazyflie failed its Startup checks and would not fly. The drone works with only one of either deck but not both. The start up sequence debug detailed that it was a physical issue with how the board was constructed and thus could not be solved with conventional means.

Solution:

I do not have the tools or the knowledge to perform the manual re-soldering some online solutions described thus the parts where retrofitted to the crazyflie 2.0 for use.

Algorithm:

the algorithm went through a few iterations and was under a few considerations as described below.

The initial consideration was to use a perceptron-based approach, where every step of some length the drone would scan its surroundings and mark a node for it to return to. This was mainly considered for its ability to attack the drone's weak points directly. Everytime it entered a node it would be designed to re-align itself effectively negating the innate tendency for cumulative error. However, this method was quickly disregarded. This system firstly was over reliant on the sensors' precision. The sensors had to reliably replicate a series of data points every-time it entered a node at precise angles that relied on the onboard gyroscope to yaw accurately and precisely. This would allow the perceptron to use this data like a key signature and match the data to a known nodes data signature. This method however relies as well on the uniqueness of every nodes data. Or otherwise, each node would have to have a different subset of surrounding polygons to create unique keys. If 2 nodes had unnecessarily similar positions the system could not tell them apart. In a broader case this solution would perform poorly on sparse spaces. Since every node created in space space would look the same you couldn't tell them apart. This would extend to requiring past node states to be added to the input vector of the perceptron. This way we could have some N history of nodes as an input vector to hopefully make a unique signature that allows the system to determine the current node. At which point the cost in development time and computational time exceeded the benefit of a reduced cumulative error and the algorithm was dropped for a decomposition-based algorithm.

Quad-tree decomposition and FBSP decomposition:

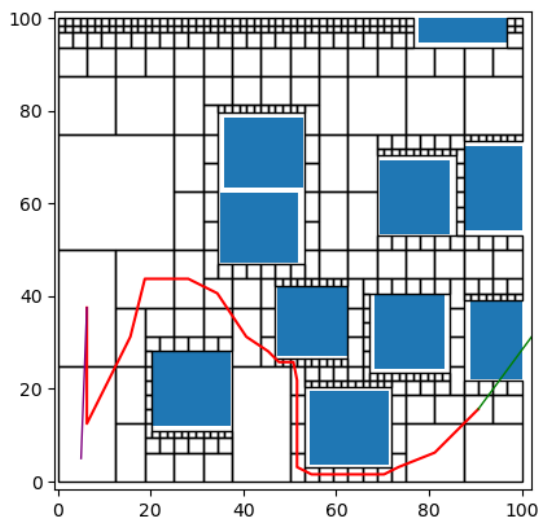

Figure 5: An example of a quadtree decomposition. Graph taken from a joint project by me and Anton Sikorsky whose permission I have to include this graph

The space decomposition algorithms are a subset of pathfinding algorithms that given a space with some obstacles, split the existing space into smaller subsections until a distinction between flyable and non-flyable space has been made. Quad-tree decomposition splits any block that intersects an obstacle in half until either a no blocks overlap, or the block is too small to split. FBSP decomposition finds all the vertices of the obstacles in the world and splits the existing space along the X and Y axis of those vertices. While the result of these algorithm more closely resembles our objective navigational graph, they have numerous flaws I found during testing, which after consideration I built the final decomposition algorithm to minimize.

They do not avoid creating small or thin blocks. As you can see in Fig.8 numerous impassible blocks are created too small to be useful. FBSP creates similarly small traversable blocks.

They are slow. The quad tree decomposition algorithm that produced this graph took around 3 minutes to create this graph.

Dynamic decomposition:

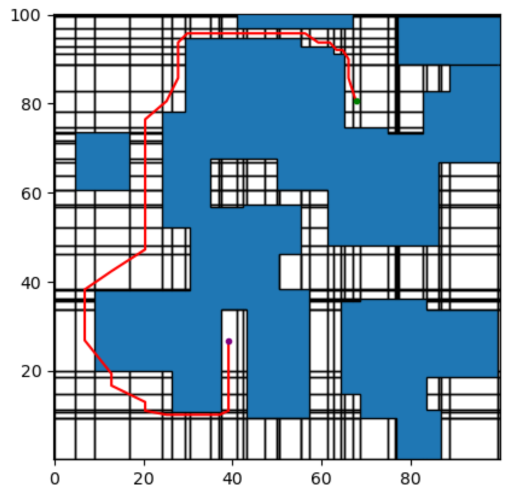

Figure 6: This graph is also from the same project and has also been allowed to be shown here with Anton's permission. This is an example of FBSP decomposition. Admittedly with more blocks, but this clearly shows numerous thin blocks.

This is the current working version of the project's decomposition algorithm. It splits the world dynamically, as more objects are added it only splits the needed blocks and leaves existing nodes alone as much as it can. The algorithm starts with a blank, flyable world. As obstacles are found by the drone the algorithm only looks at blocks that intersect the new obstacle. These blocks would be split along their intersection point with the obstacle in the same direction (vertical or horizontal) as their line of intersection. This is different from the above static decompositions, as unlike FBSP it does not split every block on the axis, but only the directly affected blocks. Unlike quadtree it splits the block directly on the point of intersection reducing the need for re splitting and further decomposition (See the Fig. 4 breakdown). Unlike both algorithms, this method leverages the lesser error of previously created center point nodes by not overwriting the new area with a new subset of blocks, saving accuracy and time.

VIII Results:

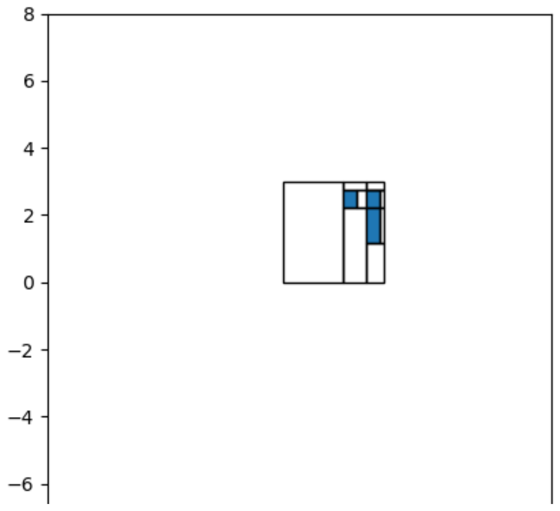

Figure 7: The drone scanned a box in the middle and some stairs on the right

First and foremost, at a high level, the project does not work. This is mainly an integration issue between the drone firmware to API to my python model. It makes wild decisions, maximizes and cuts thrust at inconvenient times and generally does not like to fly and take measurements at the same time or fly for more than 30 seconds before a 2 hour recharge. By Monday, the git-hub will likely have some functional example that implements the expected behaviour however this will likely require an extra 2-3 days of intensive work that this report cannot wait for. Thus, I will be going over what the drone and algorithm have accomplished.

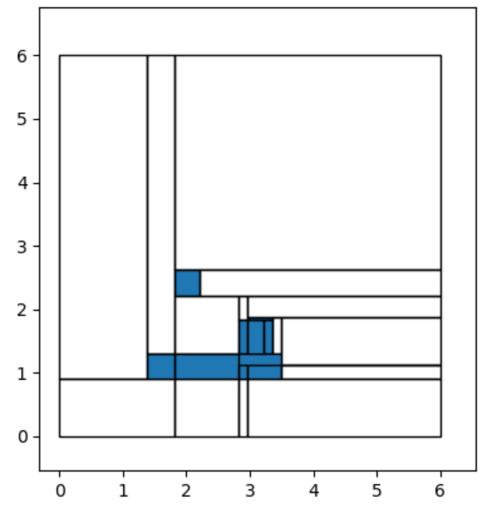

Figure 8: the drones scan of a corner of my kitchen

The drone succeeds in totally uncluttered environments, while the graph is not pictured as it would just be blank, recordings of the flight can be found on the git-hub.

In cluttered environments, the drone successfully starts up the sequence, finds the intended goal, turns towards that goal and proceeds towards it. If, during that path, the drone finds an obstacle, it will successfully scan that obstacle into its memory and will successfully avoid that obstacle from there on. It will then move to the next node correctly and find when it reached that node. It will then successfully apply the heuristic and move to the next node and if it finds another obstacle it will scan and update the navigational graph exactly as the Fig 4 example describes however after about 15 seconds or on average 3 node traversals, odd and unintended behaviour occurs however after about 10 seconds. Thus, I have been unable to fully complete a cluttered test run. Numerous video examples will be provided of this behaviour on git-hub.

The navigational graph without the drone works perfectly well. Numerous examples will be provided on git-hub to display and document how the navigational graph handles the environments under different situations and the effects on the best path. Though this work will only be theoretical proof not a physical drone implementation.

IX Discussion:

With the project coming to a close, in retrospective the development cycle was a net positive experience. There where many hiccups implementing the drone, especially trying to translate the theoretical model to a real working physical model. While the development of the algorithm was iterative and long it was rather straightforward. The drone and its capabilities produced a unique set of real-world circumstances that I was not accustomed to and could not deal with. This led to compromising on planned features, especially machine leaning models which in a testing environment worked well, but the real-world environment was not as hospitable. If by some account this course was extended by an extra month or so to allow for an additional iteration, I would first like to get the 2D model solidly built and flexible and then I would like to increase to 3D and thoroughly test the original problem statement of a varied 3D space. However, this would likely require a review of the hardware components of this project would have to be undertaken as the current implementation leaves the drone vulnerable in 3D dynamic movement and is generally not a platform I would consider for future on-board projects.

Working on a project of this feature scale and time scale was daunting. While I was prepared for the theoretical implementation, the hardware integration to a real-life environment unexpectedly took most of the time to develop. The crazy-flie Firmware and python API had many quirks which where a combination of interesting and frustrating to work around, especially with undocumented behaviour. Especially in cases related to the drone's self preservation similar to why 3D was not pursued, the drone simply did not try and reduce damage and seemed to actively try to break itself. As an example, when an error occurs in the python code, the drone maximizes all 4 motors' thrust and does not stop for about 5 seconds before it kills the thrust entirely, leading to some rather large dive bombs initially. Thankfully only the propellers where damaged and replaced.

---

1. L. Campos-Macías, D. Gómez-Gutiérrez, R. Aldana-López, R. de la Guardia and J. I. Parra-Vilchis, "A Hybrid Method for Online Trajectory Planning of Mobile Robots in Cluttered Environments," in *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 935-942, April 2017.