Aerial LiDAR: Path Planning

Otto Jursch
CSE @ UC San Diego

Jason Ma
CSE @ UC San Diego

Phuong Nguyen
Math @ UC San Diego

Owen Wu
ECE @ UC San Diego

Abstract

Large-scale aerial drone surveying is becoming increasingly common in geological and archaeological applications, but the challenges associated with avoiding terrain and obstacles is prohibitive. Conventional plane surveying can be very expensive due to the overhead costs of contracting planes equipped with advanced sensing systems to fly over areas of interest. We present an autonomous path planner utilizing existing LiDAR scans of an area for low-altitude drones to minimize costs for aerial surveying. By flying closer to the ground, the Aerial LiDAR: Path Planning project is able to generate an aerial path for drones that not only avoids obstacles from the terrain, but also remains within close enough proximity of the ground that it can perform LiDAR scans.

Introduction

One of the most exciting developments in archaeological technology in recent years is aerial LiDAR technology. LiDAR uses a pulse laser to measure the distance between the sensor and the closest object at that angle. Aerial LiDAR platforms involve putting a LiDAR sensor in a plane or other aerial vehicle, and then measuring the distance between the plane and the ground. Over the entire flight, this generates a point cloud, and different processing algorithms separate the ground, buildings or other features rom the rest of the point cloud. While these techniques may seem simple, they have lead to huge archaeological finds. In this year alone, there have been two major discoveries using aerial LiDAR technology. In Guatemala, archaeologists were able to reveal a massive, 60000 structure “Mayan Megapolis.” Based on these findings, archaeologists believe they greatly underestimated the population density of the Mayan civilization in this region[1]. Even LiDAR scans not originally taken for archaeological purposes have been immensely useful to the archaeology community; in Britain, an archaeologist found a 2000 year old Roman road by analyzing data originally taken to analyze flood patterns[2].

With the advent of aerial LiDAR technology, the archaeological landscape will never be the same. Unfortunately, this technology is not cheap, and is likely out of the price range of many archaeologists. However, researchers do believe that replacing the expensive manned aircraft with smaller, cheaper, and autonomous flight platforms can greatly reduce the cost of aerial LiDAR technologies. Making this switch is complicated, as these aerial LiDAR systems have a much smaller margin of error. Since they carry less powerful LiDAR scanners, they have to fly much closer to the ground and forest canopy and have an increased risk of crash. This requires more involved path planning algorithms, since the vehicles now have to both avoid obstacles and stay within a certain range of the ground for the LiDAR to operate correctly.

The Aerial LiDAR: Path Planning project provides a framework for generating and evaluating flight paths, which can aid surveying teams in quickly deploying a safe, cheap solution for mapping areas of interest. This comes with the ability to generate test cases, partition LAS files into bare earth/surface map raster models (tagged image file or .tif format), generate paths associated with the appropriate terrain, and visualize paths. The generated waypoints can then be uploaded to any aerial vehicle supporting ArduCopter, which will fly the path. The flight log (actual flown path), any generated paths, and the rasterized maps of the area can be loaded into the visualizer as well, which can overlay them to give a good idea of how the paths look compared to each other in 2D or 3D. This enables the development of future path planning algorithms, providing a comprehensive suite for simulating, flying, and analyzing paths.

Technical Material

System Design

DEM Creation

A Digital Elevation Model (DEM) is a 3D representation of the surface of a Geographical Model as if looking at a low resolution image of the terrain from a satellite view. It gets generated from LAS files which get processed by a GIS tool that generates a Raster or 3D polygonal mesh out of its data points which then takes the highest point based on its X and Y coordinates, to then generate a Lookup Table or Image with each pixel being an altitude value instead of an RGB value. This can still be viewed as an image, but it will be be in grayscale since it only has one channel of information instead of the usual three with RGB. The specific tool we are using to generate this DEM is ArcGIS. This requires a license to use and there may be intricacies with using this tool as we found unexpected behavior in trying to process our DEMs. This tool can also filter out which type of classification in the LAS data to use for generating the DEM, such as only utilizing data points which can be considered the ground. This allows us to create DEM's that only include the highest points in the tree canopy, or only the highest points that are on the ground, for example.. We generate and make use of both of these types in our path planning algorithm.

Our path planning algorithm makes use of this map to set a keepaway distance and to avoid the tallest obstacles in its way. This is being done trivially for now by only going over the obstacle and not going around even if it is more optimal and less time consuming to do so. This in return simplifies the path planning algorithm and facilitates full LiDAR mapping of the intended areas.

Path Planning Algorithms

An effective LiDAR surveying path for small autonomous vehicles has two major requirements. The first is that the quadcopter must fly at an altitude that keeps the bare earth within range of the LiDAR sensor as much as possible. Essentially, this means that a good LiDAR surveying path must follow the relief of the canopy/highest terrain obstacles in order to generate a useful LiDAR point cloud. The second requirement is that the path must not intersect with the bare earth or the canopy in order to avoid crashing. To put this in more mathematical terms, an ideal aerial LiDAR surveying path must maintain the following invariants:

Given that X is a safe distance away from obstacles, Y is the range of the LiDAR sensor, and $X < Y$

1. The path is always at least X meters away from the maximum altitude of the canopy and bare earth models
2. The path is always within Y meters of the bare earth

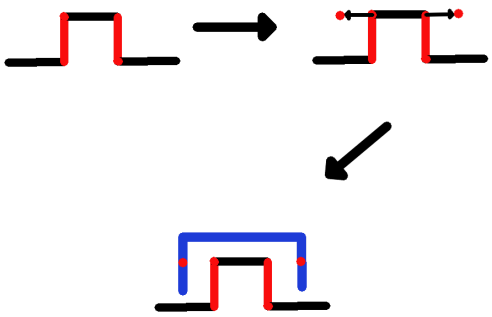
We developed two approaches to this problem: a slower, provably correct approach as well as a faster, less safe approach.

Provably Correct Approach

What do we mean when we say a "provably correct" approach? When we say that we can "prove" that our approach is correct, we hope to show that it generates a path that satisfies both of our invariants in all situations. We intend to do this by performing a series of transformations on the bare earth. Consider the following steps.

1. Find every single point along our path in which the altitude of the bare earth and canopy changes
2. Project each of these points along our path in the direction of the lower altitude
3. Using the x and y position of these points, we add two new points; one with an altitude that corresponds to the lower altitudes

Visually the transformation looks like this:



In this image, imagine that the step function looking graph is a cross section of the terrain. In particular, this terrain starts at one altitude, increases in altitude, and then decreases back to the original altitude. The first step of our algorithm is to find the altitude changes, symbolized by highlighting the regions where the altitude changed in red. Next, we find new waypoints by moving over the lower altitude region a certain distance, symbolized by the arrows and the red points. Finally, at each of these red points, we generate two new points, one with an altitude that lies at our LiDAR sensor range above the lower altitude and another that lies at our LiDAR sensor range above the higher altitude region. These points are the corners of the blue rectangle, which in itself represents the path that we hope our flight platform will fly.

The only real challenge in implementing this algorithm is finding every point in which the altitude changes. To do this efficiently, we transform the geotiff file so that is represented by a list of polygons, where each polygon represents a region that has one altitude. Once we have these representations, we can efficiently find the polygons that intersect a path using space partitioning data structures from Shapely, a python library for performing geo

spatial and geometric operations, and then find the points in which the path intersect these polygons to find every single altitude changing point. In order to avoid having to determine these polygons every time we run our algorithm, we write them to a file

While this approach has the advantage of being provably correct, it has a significant speed and memory cost that potentially limit its ability to adjust the path of a UAV while the UAV is in flight. Converting the geotiff file into a vectorized format takes a while; for example, generating the vectorized shapelist for the entirety of a digital elevation map of UCSD takes between 2.5-5 minutes. This makes setting up our system a bit annoying if no preprocessing was done prior to going out into the field, but theoretically we can still use this algorithm in real time as we could create these data structures once and then keep them in memory for fast access. However, part of the reason generating these data structures is slow is because of their size. Combined, the shape-altitude and shapefile files are around 200 MB for just the area over UCSD. We were concerned that we may run into memory issues eventually. For instance, if we want to run the path planning algorithm on a somewhat memory constrained computer on a drone while also flying over an area much larger than UCSD, we may eventually run into problems.

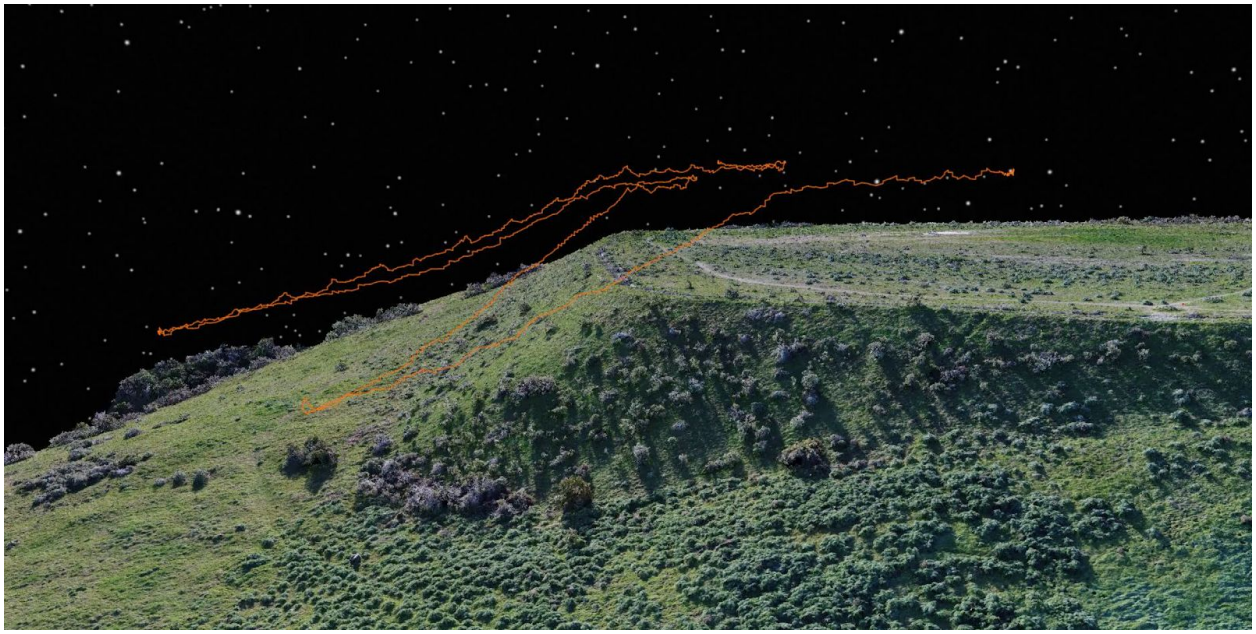
Fast Approach

As noted in the provably correct approach, some steps associated with the provably correct approach require a substantial amount of time and memory, which we cannot afford to use flight time for should the path change in flight. Our solution to this problem was to create an alternative algorithm that approximates correctness very closely while increasing the speed of the algorithm. Rather than finding every single point in which there is an altitude change, we create a list of points at a certain spacing and place them a fixed height above the surface map model. This eliminates the need to create the large shapefile and shape-altitude map, while retaining the relevant information in memory only for the duration of the program's execution. Once the list of points has been generated, the same smoothing processes and conversion of waypoints into GPS coordinates can be applied. This is no longer provably correct; for instance there can be a large peak in between two points generated along the line due to the way points interpolate between two pixels on the rasterized image. However, reducing the distance between waypoints and through the application of smoothing algorithms minimizes this issue.

Smoothing Algorithms

Rationale

The paths that the algorithms mentioned in the last section do satisfy our two invariants, but they ignore the crucial fact that our quadcopter exists in the real world and must follow the laws of physics. Since the mentioned paths generate a couple of waypoints for every single altitude change (or enough waypoints to closely approximate every altitude change), they generate a massive amount of waypoints that are very tightly packed together. Theoretically, this is fine, but in reality our quadcopter's controller and control systems are not precise enough to hit every single waypoint, and as a result the quad copter flies erratically. For instance, take a look at this 3 dimensional rendering of the flown path during a test flight:



Notice how there are several sudden jumps along the path. This indicates that the quad copter failed to closely follow the path and had to plateau suddenly to reach some waypoints.

To fix this problem, we hope to create algorithms that "smooth" the path. By smoothing, we refer to removing waypoints from the path while maintaining the somewhat rough shape of the individual path, essentially converting the series of very minute waypoint updates into a smooth line between waypoints. This allows the user to choose a point in the tradeoff between a very simple path with collinear waypoints and a path that more closely follows the surface of the terrain. A simpler path is much easier for the vehicle to fly through, potentially sacrifices some LiDAR coverage.

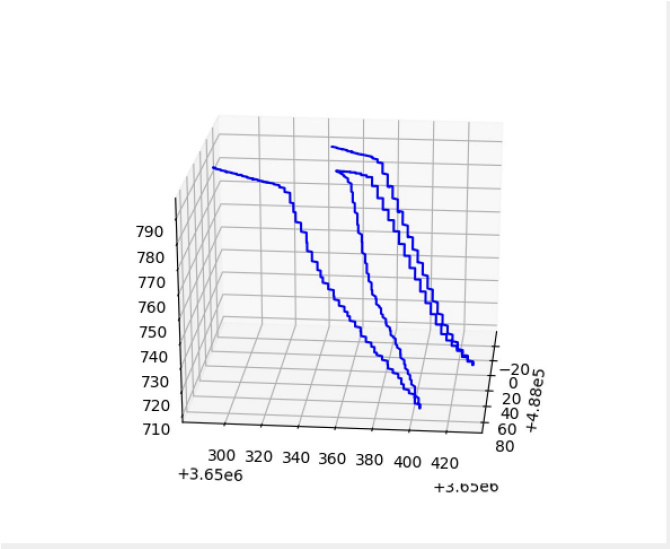
Simple Smoothing

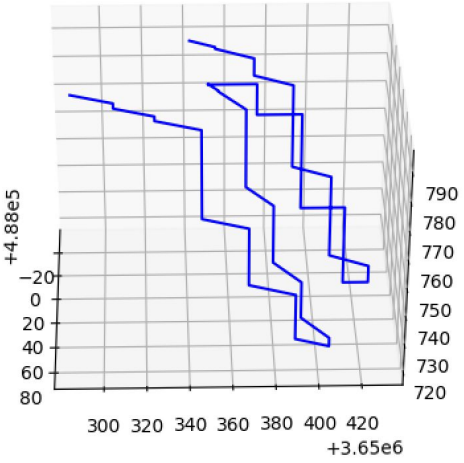
The simplest smoothing algorithm that we developed was to simply take the limit the number of waypoints to a certain number per interval. Essentially, the algorithm works as follows:

Given an interval distance X and a path generated using the provably correct algorithm discussed earlier

1. Create an empty path new_path
2. For each X meter long interval along the path
    a. Find the maximum altitude over the altitude
    b. Add a new point to new_path where x and y are the x and y position of the start of the interval and z is the maximum altitude over the 20 meter interval
    c. Add a new point located to new_path where x and y are the x and y position of the end of the interval and z is the maximum altitude over the interval
3. Return the new path

Visually we can see that this does a very good job of removing waypoints. On the left, we have our unsmoothed path that many minute altitude adjustments, and on the right we have our smoothed path which we can clearly see a couple of larger altitude changes.





Because we always choose the maximum altitude over the interval, the algorithm guarantees that the smoothed path always flies above the original path. Therefore, as long as the original path is above all of the obstacles, the smoothed path is guaranteed to be as well.

A downside of this smoothing algorithm is that it still demonstrates somewhat of a step function-esque pattern, i.e the path will alternate between being purely horizontal and purely vertical. If we're thinking of the path in terms of total distance, this is the slowest way to achieve the correct altitude; we can greatly decrease the total distance of the path if we ascend and descend at an along the hypotenuse of this triangle.

Peak Algorithm

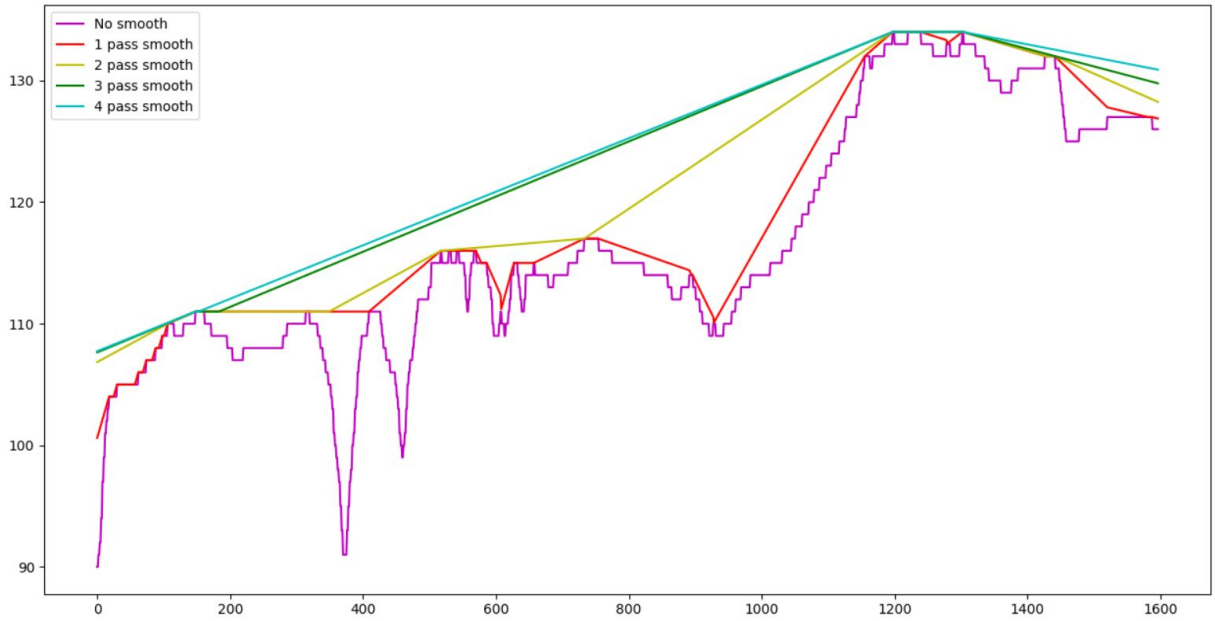
Another one of our smoothing algorithms achieves simplification by travelling as directly as possible between all local maxima (peaks) in the path, with a maximum climb/descend rate to ease rapid altitude changes. The algorithm is roughly as follows:

Given a list of waypoints Waypoints, and the set of waypoints in Waypoints that are Peaks

1. For each sequential pair of peaks (peak1, peak2)
    a. For every waypoint on the section of the path between peak1 and peak2
        i. If the waypoint lies above the line directly connecting peak1 and peak2
            1. Keep the waypoint's altitude the same
        ii. Otherwise
            1. Adjust this waypoints altitude so that it lies directly on the line between peak1 and peak2

From these peak waypoints, the algorithm computes the slopes between the peaks and travels downwards when possible in both directions following the computed slopes, greatly reducing the amount of slope changes and spikes in the path. This is also done in a way such that the smoothed path only changes the height of any individual waypoint if it is at or above the original waypoint. Furthermore, a maximum climb/fall rate can be imposed such that the path never makes a sharp climb or fall, forcing it to smooth out climbing to or descending from a cliff over a long set of waypoints if possible. This was implemented by setting the slope to the maximum rate if it exceeded the

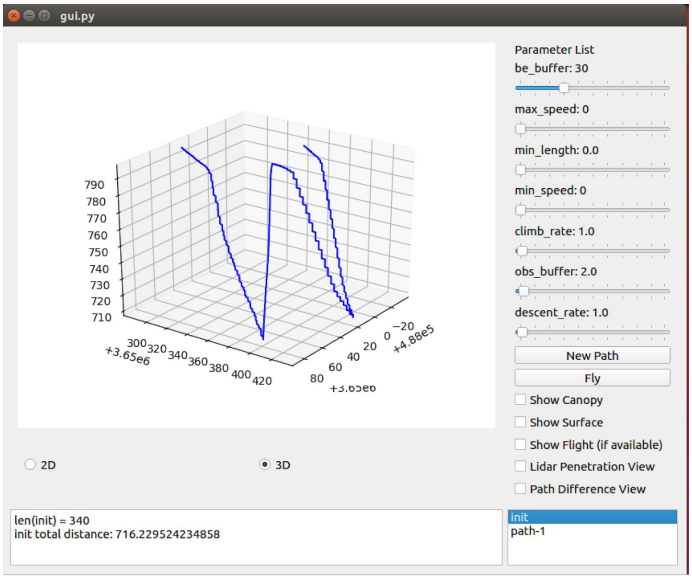
maximum rate. This smoothing can be applied iteratively based on the frequency of obstacles and noise in the data to smooth to the desired amount. Additionally, this algorithm can be run very quickly (O(n) where n is the number of waypoints in the original path) and always produces paths that are at least at the same height as or above the original path, guaranteeing that as long as the original path avoids obstacles correctly, the smoothed one does as well. The key is to find the balance in amount of iterations used in the smoothing, as a low amount could still have some spikes, while a large amount could lead to the quad flying too high over the ground in many areas. An illustration of this algorithm in action is shown below:



The path without any smoothing very closely follows the terrain, along with all of its bumps and spikes. As the amount of iterations in the smoothed path increases, the path begins to reduce the amount of waypoints and peaks gradually, up to the point where the path becomes too simple and the vehicle flies too high off the ground. It appears that visually, an ideal balance between simplicity and height off ground appeared at the second iteration of smoothing in this case. However, if height off the ground was a more important consideration than flight time (path complexity), a single pass of smoothing may lead to a better path than the two iteration smoothing path.

Analysis Methods

To facilitate the development of new smoothing algorithms, we created an application to quickly prototype and analyze algorithms. Features include simple menus to adjust the parameters of our algorithms, a graph to display 2D and 3D representations of a paths, and automated metrics about paths.



Empirical Analysis Methods

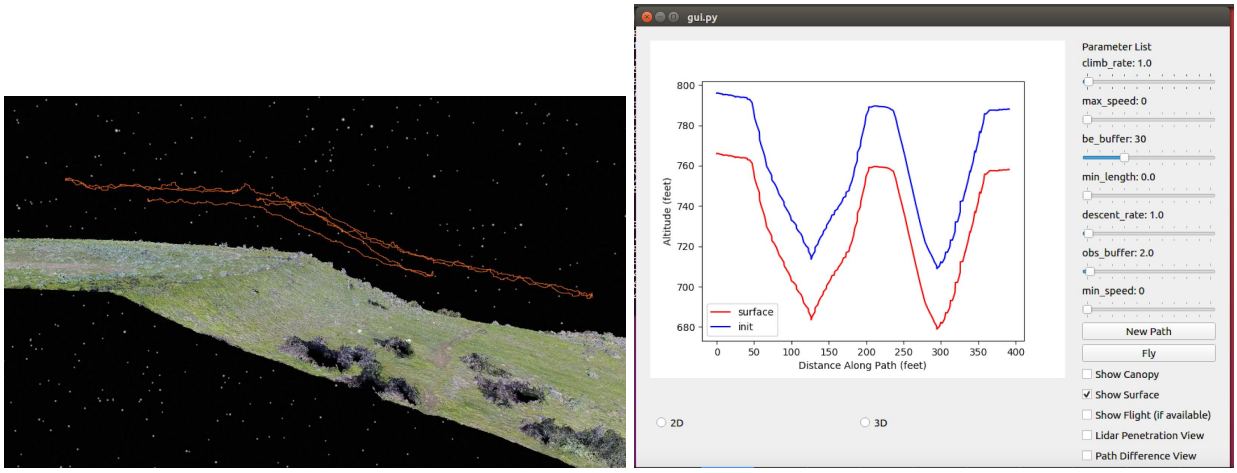
The main goal of smoothing algorithms is to reduce the complexity of the path, which can be measured in a couple of ways. The simplest way to measure the complexity of a path is to simply count the number of waypoints, and this is actually one of the most important measures to consider as the principal problem with the unsmoothed path is the surplus of waypoints. In addition, distance of the path is another measure of complexity and is important to keep in mind due to constraints like the battery life of our platforms. Our application calculates and then displays both of these metrics for every single path that is selected.

While the ability to analyze the complexity of a path in isolation is valuable, metrics to compare two different paths are perhaps even more valuable. These metrics will allow us to better understand the tradeoffs between two different soothing algorithms, or how well the quadcopter is actually able to fly a generated path. To perform this comparison, our application calculates the Mean Squared Difference between two paths as well as the area between two paths. These two metrics give us a good way to compare exactly how much two paths differ.

Visualization Tools

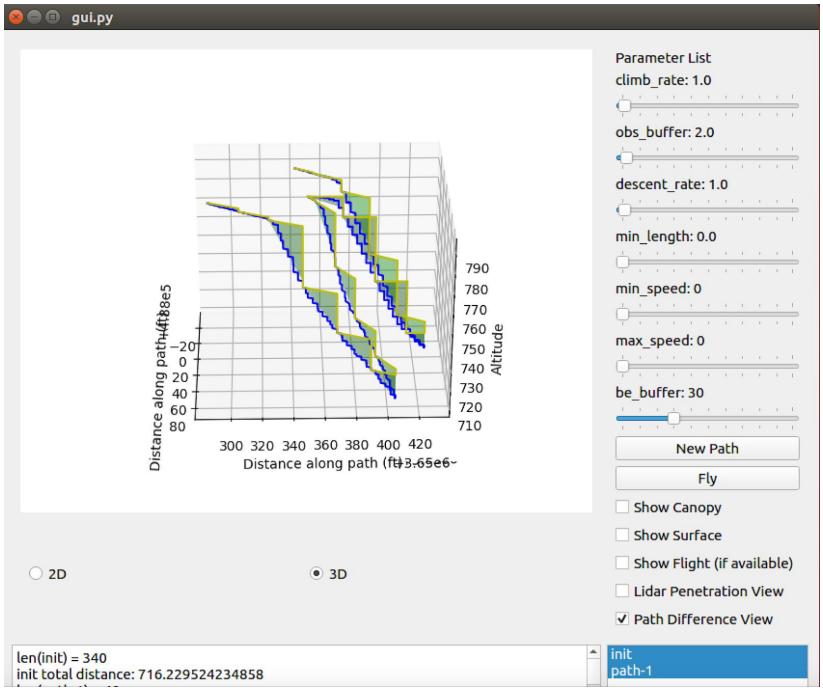
While empirical methods can give us a useful idea of how are paths are performing, it is also important to have a way to visualize the path's performance so that we can pinpoint the situations our algorithm performs well in and the situations in which our algorithm performs poorly.

We developed tool different ways to view our path. The first is a simple 3 dimensional graph of the path, where the x, y, and z axes correspond to the the position of the path on a Universal Transverse Mercator map projection (essentially, this is just a map projection that uses a Cartesian grid). Our second graph view takes the path and projects it into 2 dimensions; the x axis corresponds to the distance along the path, while the y axis represents the altitude of the path. An easy way to think of it is the path and the surface viewed in profile. Take these two images:



On the left you can see a path rendered in 3D allowing us to see the profile of the path and how it follows the relief of the hill. This is essentially the same view we are recreating in our 2D plots.

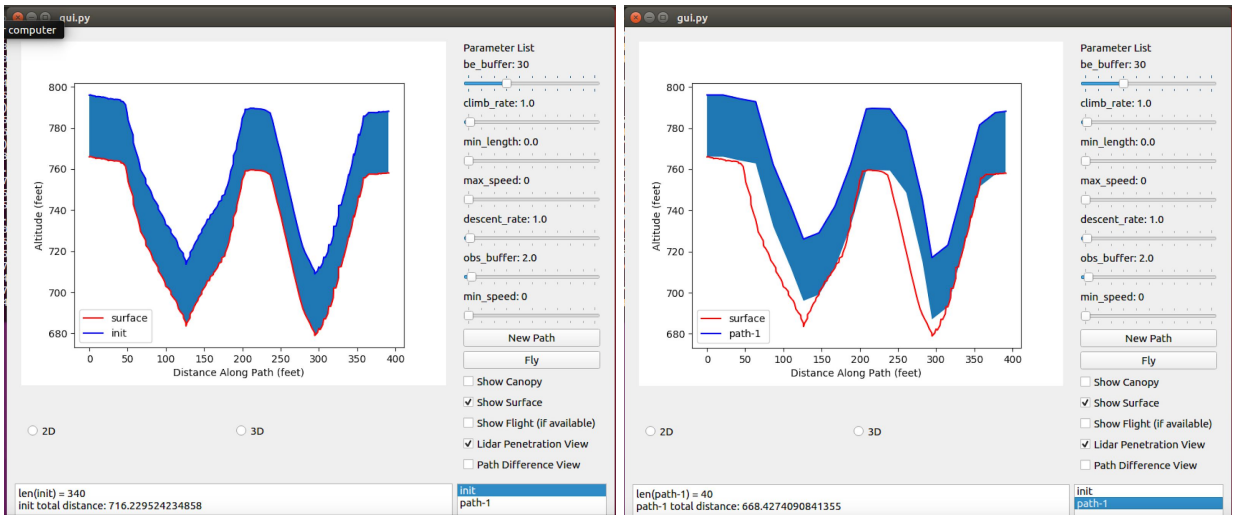
In order to visualize the distance between two paths in 3-D space, we created a tool that automatically shades the surface between any two paths. This is useful because it allows us to locate the differences between two paths that our empirical metrics calculate quickly on the graph. Take this example:



Here we have the original provably correct plot on the graph in blue, plotted against one of our smoothed paths in green. The green surface makes it easy to determine how exactly these two paths differ.

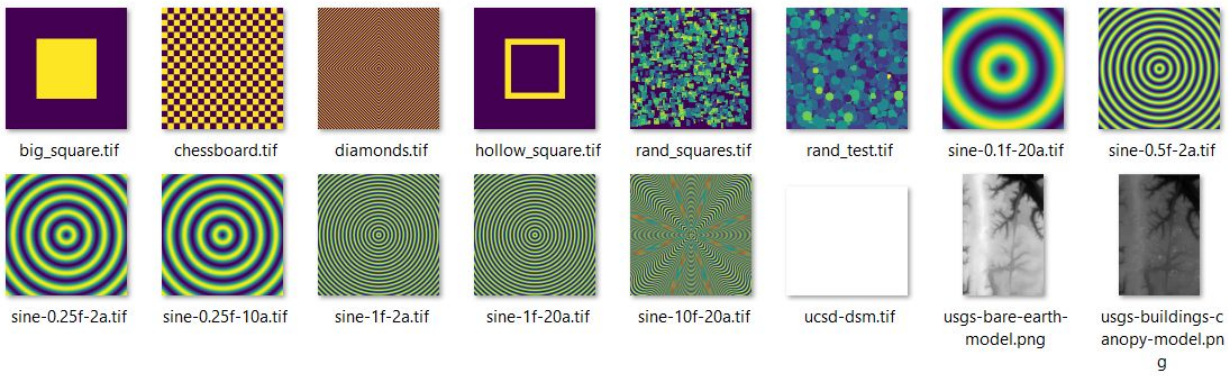
On our 2-D projection plot, we wanted to be able to get an idea of what areas of the surface are in LiDAR range to help us understand how smoothing our paths affects our LiDAR coverage. To visualize this, we simply shade the region under the path that we expect our LiDAR sensor to be able to reach. In these images, the region we can expect the LiDAR to reach is shaded in blue. We can see how this is useful; the plot on the left shows the shaded region of our original provably correct path, and we can see how it reaches every point of the surface of the earth. In

comparison, the plot on the right shows LiDAR range of our smoothed path, and it reveals the regions of the surface that we do not expect to be able to hit with this path.



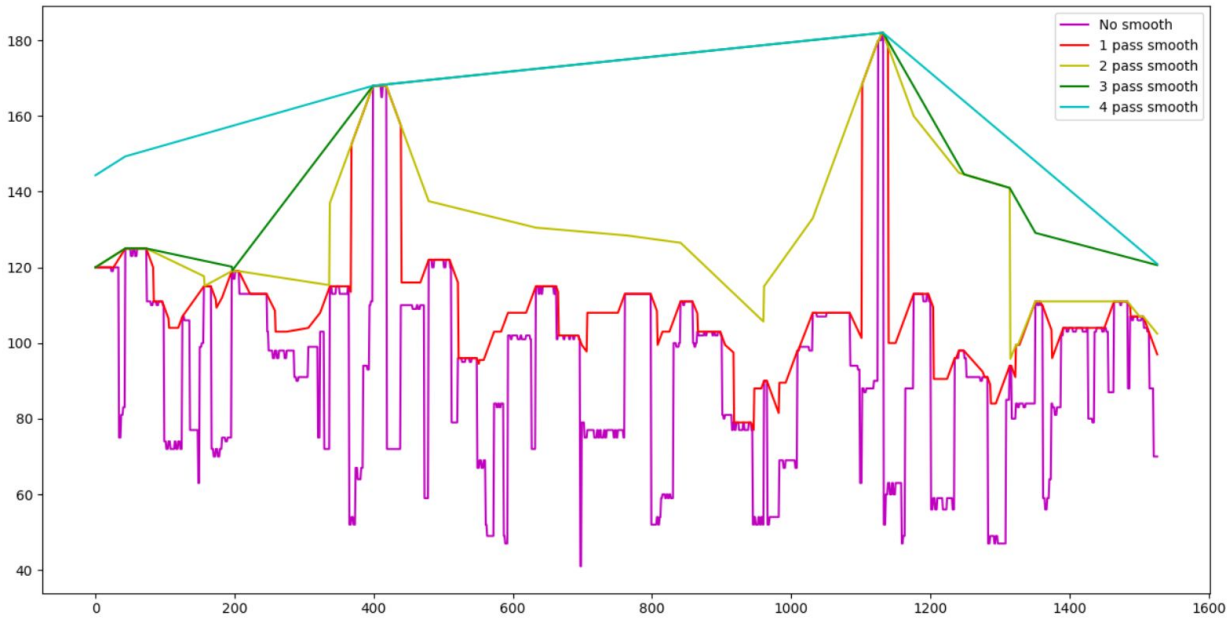
Test Case Generation

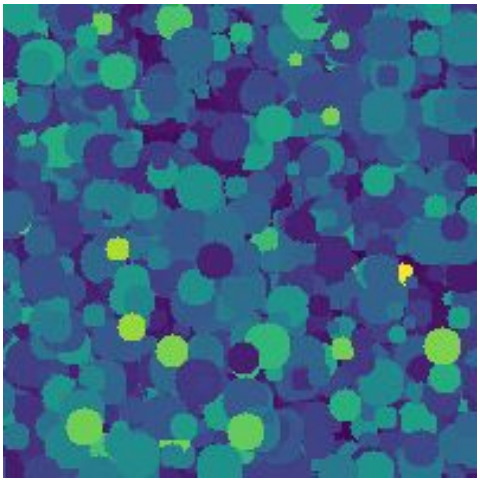
We also wrote a utility to generate test terrain which is useful in evaluating how our base path planning and smoothing algorithms perform in both realistic and crazy scenarios. This allows us to vary the frequency of obstacles and shape of the terrain. The case is ultimately saved in a rasterized format, which can be loaded in to the path planning generation algorithm as a surface map. The test generation tool includes options for generating buildings, sine waves, diamond patterns, walls, and fake trees/terrain curves. Cases such as buildings, walls, and fake trees/terrain curves can help the user understand how their algorithm reacts to realistic scenarios, while sine waves, chessboard, and diamond patterns are good for evaluating how an algorithm deals with varying frequencies of obstacles. More cases can easily be added through the associated source code file in the GitHub repo. Some examples of the generated cases can be seen below:

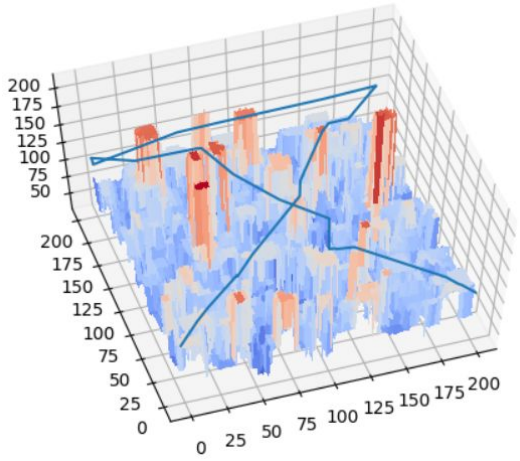


The above images were displayed using a color scale where dark blue was lowest elevation and yellow was highest elevation. As is the case with normal .tif files, these can also be visualized in 2D or 3D along with the paths to see how the path performed. Below is an example of us testing and visualizing the

iterative smoothing algorithm on a test case where multiple fake trees were generated in a simulation of a dense canopy.







The top diagram depicts the original path (follows exact shape of terrain), as well as the smoothed paths using 1-4 iterations of smoothing on the dense canopy simulation case. The bottom left diagram is the top-down view of the case on a blue (low altitude) to yellow (high altitude) color scale, and is essentially the DEM image read in by the path planning program. The bottom right diagram is the 3D visualization of the 2 iteration smoothed path overlaid over the DEM, which is displayed on a blue (low altitude) to red (high altitude) color scale.

Milestones

Initial Planned Milestones

Milestone	Status
Set up ROS environment/get code from Eric	Not needed
Determine the constraints/dynamics of the drone so that we can account for them in the path planner	Complete
Test flight with initial algorithm generate path offline	Complete
Simulation platform using ArduPilot SITL + Gazebo constraints of the plane for testing	Complete
Algorithm to process LiDAR data into aerial surface map. Could involve researching and finding tools or processing ourselves	Complete
Algorithm that takes in an aerial surface map and then returns a path	Complete
Integrate Gazebo simulations into framework with algorithms	Not Needed

Revised & Added Milestones

Milestone	Status
Path Planning Smoothing/Point Elimination ● We created two path smoothing and Point elimination algorithms, which greatly simplify the path while roughly maintaining the desired characteristics of the original path.	Complete
Path Planning Test Case Generation ● Generated several test tifs usable with the path planning algorithms, including cases for sine waves, blocks, walls, cylinders, with conditions like varying terrain/obstacle heights.	Complete
Benchmarking - Generate Constrained Paths - Take some quad flight parameters into account when generating path, so the path does not create movements that are very annoying for the quad to perform. - Decided that we could get a good enough comparison just by running simulated flights	Not Needed
Benchmarking - Measure Distance or Error Between Two paths ● Defined a metric for which we can say how close two paths are. ● Several of the metrics we used were Mean Squared Error or the area between two paths.	Complete

Benchmarking - Visualization ● Extend our current visualization setup to display multiple paths and highlight the area that difference between the two paths. ● This can be used to visualize whether a path was on target or not in a 3D plot. ● Allow us to see whether some generated paths were meeting requirements or not such as being above a certain keepaway distance.	Complete
Benchmarking - Generate Actual Flight Paths - We were able to use the Ardupilot Software in the Loop plane to fly - We were also able to parse the generated .BIN file into one of our path formats, which we were able to graph and compare to our	Complete
Integrate more closely with Mission Planner - We created a scripts to convert our paths into the QGC Waypoint format which allows Mission Planner to read in our paths and then upload them to the quad copter - However we did not get around to integrating our software with the Mission Planner GUI	Partial/WIP
Travis CI Integration - On every push to master, Travis CI will build a version of our application into a docker container and then push it onto Dockerhub - This will allow new users to set up our application with minimal set up constraints	Complete
AirSim ● Initial environment setup and generating a script to fly over pre-planned waypoints are done. ● Still need to load in model of UCSD/BlackMountain and convert GPS bounded coordinates to AirSim World coordinates.	Partial / WIP
Test Platform - Through generating running simulated flights, we were able to ensure that the quadcopter could successfully run through our mission - We then did an actual test flight using our algorithm, proving that our algorithms and system were in fact flight ready	Complete
Real-time Path Planning ● Quad is inherently capable of receiving new waypoints in flight through mission planner, so we chose to dedicate our time to ensuring that the path planning algorithm was runnable in real-time. This enables everything to be set up and ready to go quickly. ● As for the real-time path planning algorithm implementation details, we implemented an alternative path planning algorithm using numpy and mostly just the tif that can quickly generate a rough path that follows the terrain without the preprocessing. This always runs in seconds or in fractions of a second on our computers for a tif of size 500x500 (compared to the other one	Partial/WIP

which takes several minutes to preprocess the tif, then minimal run time in subsequent runs).	
Photogrammetry ● Pipeline for generating models already accomplished by previous built infrastructure on Eric's side. ● Still need to improve pathplan algorithm to capture better data for this pipeline.	Not needed

Conclusion

Over the last 10 weeks, we developed a path planning algorithm for cheap autonomous aerial LiDAR surveying platforms. In addition, we have created a platform to ease analysis and development of new and improved path planning algorithms in the future. We believe that with these things combined

Future work on this project might include (but not limited to) improving or adding additional path smoothing algorithms, better AirSim support for viewing the planned path in almost a real life environment, more data processing for newer sensors, extend to more visualization capabilities where necessary and to improve our algorithm to be more efficient and optimal when it comes to avoiding obstacles.

References

[1] T. Clynes Exclusive: Laser Scans Reveal Maya "Megalopolis" Below Guatemalan Jungle. In National Geographic, 2018.

[2] M. Gannon 'Lost' Roads of Ancient Rome Discovered with 3D Laser Scanners. In Live Science 2018.