Quadcopter Prototype Demonstration Report

The Flying Toasters
Ashwani Aggarwal, Ashwin Srinivasan, Diego Suazo, Justin Yu
Addressed to Ariadne de Bothezat
December 9, 2020

Table of Contents

List of Figures

Executive Summary

The Flying Toasters recommends the use of this quadcopter prototype for Ariadne de Bothezat, a client who is developing and marketing quadcopter platforms and sensor-electronic kits to enthusiasts and universities. Our team's current prototype has successfully performed two challenging tasks -- autonomous maze traversal, and rudimentary spatial mapping-- demonstrating its performance and effectiveness in a variety of applications.

Our team met 15 times over 8 weeks, 8 of which were in-person to make progress on these tasks. We came to the consensus that to sufficiently demonstrate the quadcopter prototype's performance, our maze traversal algorithm should be generalizable to a maze of any shape and that our secondary task should demonstrate the feasibility of robust post hoc flight-log sensor data analysis.

We have successfully performed these aforementioned tasks. For the first task, the algorithm completes the standardized maze in <30 seconds with minimal collisions and is also applicable to other maze shapes. For the second task, we successfully perform sensor fusion to obtain and subsequently map quadcopter location, orientation, and also the surrounding environment of a 1mx1m square enclosure. Throughout the following report, we expand on our process and our results.

Based on our findings, we can confidently recommend this quadcopter platform for quadcopter enthusiasts and universities.

Introduction

Our client is planning to develop and market quadcopter platforms and sensor-electronic kits to enthusiasts and universities, and is interested in learning about our current prototypes. As a result, we've set out to test our prototype with two tasks; navigating the drone autonomously through a maze and creating a custom project that highlights the capabilities of the drone outside flight when paired with other sensors.

Maze traversal is an important problem the drone should be able to handle because it simulates obstacle avoidance in the real world. Since these drones are being marketed towards enthusiasts and universities these drones are going to be flying in conditions where they will have to progress through an area while avoiding obstacles, and a maze perfectly simulates that. If the drone can successfully and efficiently traverse a maze then it should be fit to be marketed by our client.

The drone should also be modifiable since enthusiasts and universities will want it for personal or experimental use, which goes beyond a single, pre-programmed use. As such, our custom project must prove that modifications are not only possible but encouraged in order to obtain further features from the drone and that the drone is open to the use of sensors and modifications. Therefore, our custom project was to employ the use of a LiDAR sensor to plot a rendering of the area around it. This will both show the use of an outside sensor, how effective the sensor is, and what can still be done by an enthusiast or university to further improve it, as we will discuss following our results.

Prototype Overview

In this section, we will describe the prototype system and its subsystems. A computer-controlled quadcopter is an autonomous aerial drone with four rotors. A standard quadcopter consists of a frame in the shape of an X with legs and electronic components nested on a platform at the center (Figure 1). At each end of the frame is a propeller, two of which rotate clockwise and the other two counterclockwise converting rotational motion into thrust allowing the quadcopter to achieve controlled flight. The quadcopter operates autonomously via commands given to it from a computer beforehand. The quadcopter is made up of six main parts: the frame, lithium polymer batteries, electronic speed controllers, flight controller (BeagleBone Blue), flight control software (ArduPilot and Mission Planner), and a LiDAR sensor.

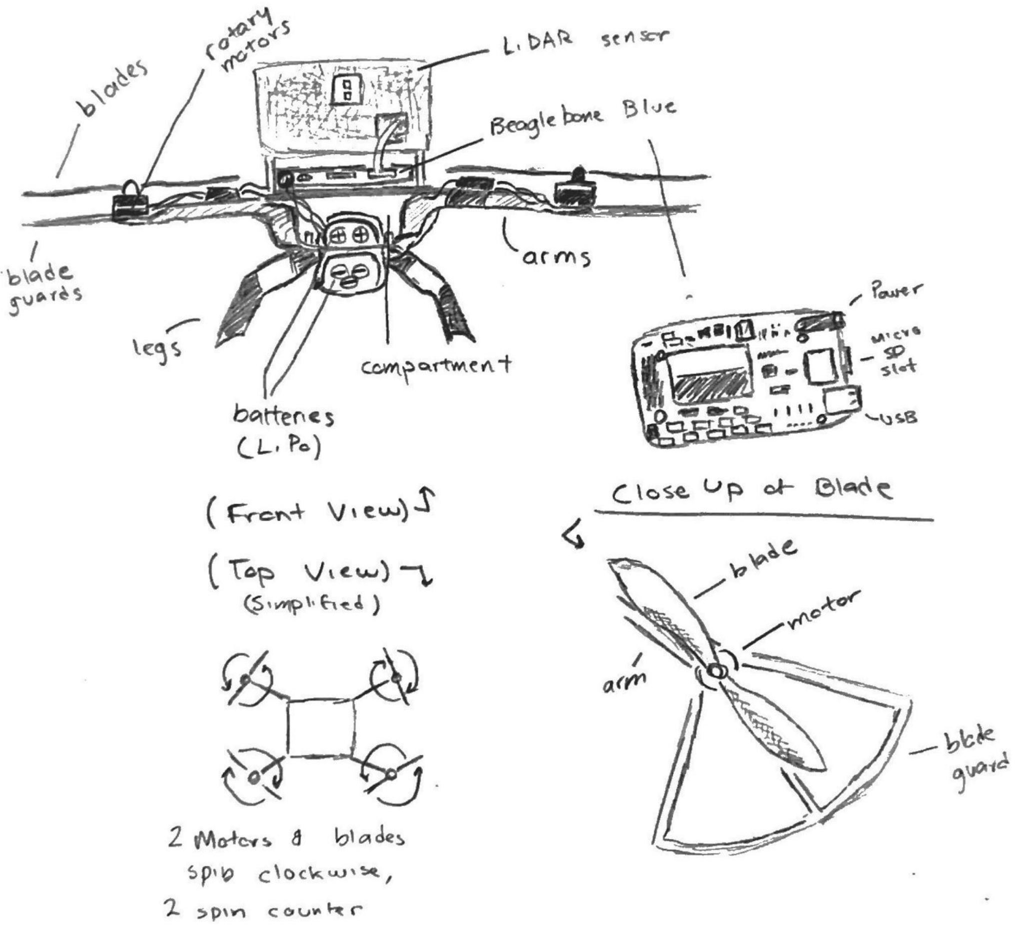


Fig 1. Drawing of Quadcopter and Parts

The Frame

The quadcopter is built upon a polymer frame that serves as the backbone for the drone and the rest of its parts. The arms of the frame form a rough X shape with a two-part rectangular platform in the center and legs underneath (Figure 1). The plates used for the platform at the center are multilayer fiberglass with conductive plating. The arms are made of fiber-reinforced engineered nylon and the legs are just engineered nylon. A motor and propeller lie on the end of each of the arms and the BeagleBone Blue circuit board lies on top of the plates while the lithium-ion batteries lie between and under them. The frame serves as the skeleton for the quadcopter, a foundation to build the rest of the parts onto as well as a protective measure so the quadcopter does not fall apart if it crashes.

Lithium Polymer Batteries

The quadcopter is powered by two lithium polymer batteries, a 3-cell battery, and a 2-cell battery. The batteries appear wrapped in blue plastic, roughly the shape of a rectangle, with two main wires protruding from them. The 3-cell battery powers the motor while the 2-cell one powers the BeagleBone Blue. Each battery has a 3.7 V nominal voltage, and each battery must stay above 3V as over-discharging the battery can damage it. When connecting the battery to the quadcopter one must strap the 3-cell battery to the bottom of the fiberglass platforms and strap the 2-cell battery in between the platforms using a velcro strap. After the batteries are strapped in, connect the 2-cell to the BeagleBone Blue, and connect the 3-cell battery to the motors. At full charge, the battery packs can maintain about fifteen minutes of flight.

Electronic Speed Controllers

The battery pack connects to a set of Electronic Speed Controllers (ESCs). The motors and speed controllers look like small cylinders located at the end of each arm. They control the propellers, which are made of black plastic with a noticeable curve to generate lift (Figure 1). The ESCs take a Pulse Width Modulated (PWM) signal from the mainboard and converts it to a speed value that is passed on to the motors. The PWM signal allows the controller to express specific percentages using a digital signal, which is then passed on to the motors to control the throttle by percentage. The motors spin the propellers at the specified speed to allow the quadcopter to fly.

BeagleBone Blue Flight Controller

The BeagleBone Blue is an ARM Cortex-A8 microprocessor with multiple sensors including a barometer, an accelerometer, a gyroscope, and a magnetometer. It runs Linux that is accessible via secure shells and can interface with additional sensors and actuators. It has the appearance of a small blue circuit board, which lies on top of the fiberglass platforms (Figure 1). The board is powered by the 2-cell lithium polymer battery, but it can also be powered and connected to through a micro-USB cable. Firmware is generally stored on an external microSD card that is inserted into the BeagleBone It takes inputs from the user through their computer and outputs PWM signals to the ESCs.

Flight Control Software

There are two pieces of flight software used to control the quadcopter, ArduPilot and Mission Planner. While ArduPilot is pre-installed on the microSD card, Mission Planner must be installed on your computer. When the microSD card is inserted into the BeagleBone, it emits a WiFi connection that a user can pair to their laptop. Once connected, users can configure and view all inputs, outputs, and parameters of the quadcopter through Mission Planner. Using the Mission Planner inputs the readings from the sensors on the quadcopter, ArduPilot directly controls the motors and flight patterns.

LiDAR Sensor

The LiDAR sensor is one of many sensors that the quadcopter employs to better understand its surroundings and motions. A LiDAR sensor, in a generic term, is any sensor that measures the distance away from any object in front of it using pulses of laser light. The LiDAR used on this specific quadcopter, as seen in Figure 1 on page 1, is mounted above the Beaglebone Blue microcontroller and has four separate "eyes", which are two separate components that combine to give it vision. The first component projects a laser light, and the second component measures the reflection of it, taking into account the return time to calculate the distance the sensor is away from any object. Through the use of four of these laser projection and reflection-detector pairs, the quadcopter has a 360-degree vision of the objects around it, which it can then use to either inform the manual pilot of an incoming collision or can allow it to make an informed evasion if it is flying autonomously. Below in Figure 2, a picture of a LiDar sensor without the box covering it can be seen, though it is missing a reprogrammable Arduino microcontroller. The LiDAR and its microcontroller connect to the quadcopter through a USB cable.

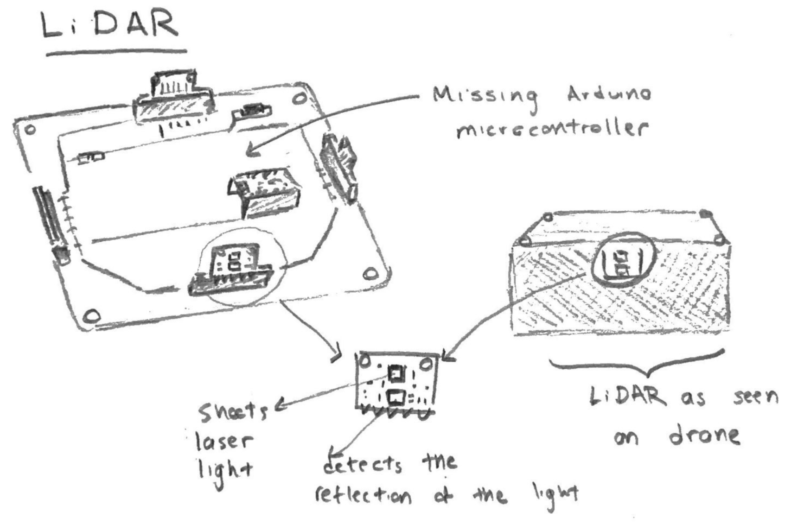

LiDAR
Missing Arduino microcontroller
Shoots laser light
detects the reflection of the light
LiDAR as seen on drone

Fig 2. Drawing of LiDAR Box and Parts

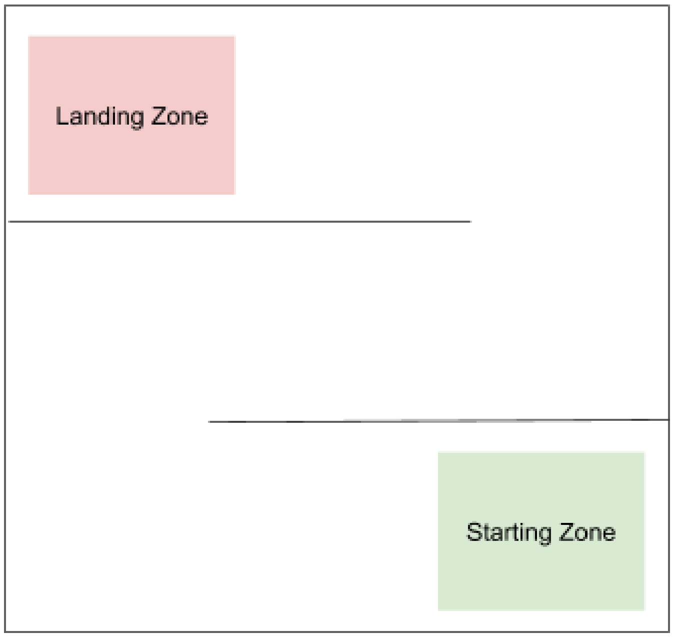
Technical Specifications

In this section, we will discuss the two projects we worked on with our prototype quadcopter - maze traversal and spatial visualization. At a high level, the former involved the quadcopter autonomously traversing a maze via an algorithm we coded. The latter involved extracting and processing sensor data from the quadcopter to construct a 3D map of the ambient space. We were able to complete both projects.

Maze Traversal

Objectives

The primary objective of the maze traversal project was for the quadcopter to successfully autonomously navigate through a maze constructed out of PVC tubes and netting and land in the landing zone. For simplicity, we only worked with a single maze design (Fig 3) without dead ends.



Furthermore, our optimization objectives were to minimize collisions and achieve a smooth rapid traversal. In particular, we wanted to tune PID parameters and various other parameters within the code to accomplish this.

Finally, we wanted to ensure the reliability of the traversal algorithm under varying conditions ranging from atmospheric conditions to the calibration of the quadcopters sensors on a given day.

Fig 3. Diagram of Maze

Traversal Algorithm

Our approach to the algorithm for the quadcopter's flight is rooted in the idea that since the maze has no dead ends, we should be able to tell from the data from the four LiDAR sensors and the current heading of the quadcopter if it should turn, continue in the same direction, or land. Assuming this holds, the algorithm just needs to start the quadcopter in the correct direction and then adjust it or end flight based on the LiDAR data.

Therefore, we can reduce the problem of determining an algorithm for these state changes based on the current state and the LiDAR data (Fig 4). The first point to realize here is that we generally want the quadcopter to move in the direction where there is the most room for it to advance in the maze. But, at the same time, we do not want it to turn around if it has just come from a long corridor. Therefore, we restrict our attention to the forward, left, and right directions relative to the quadcopter's current heading.

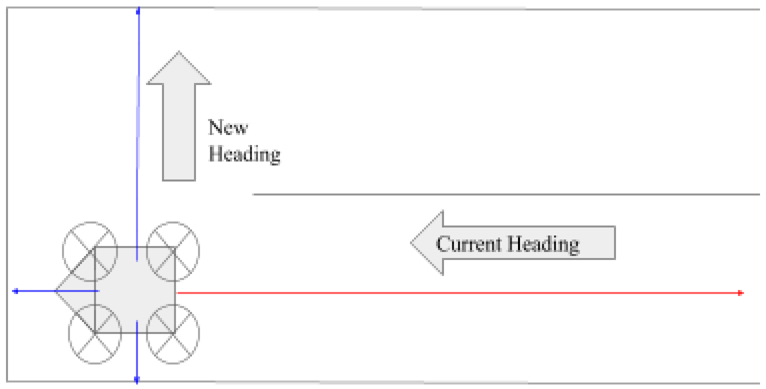


Fig 4. Algorithm Turning Quadcopter Right

It turns out, however, that this is not sufficient to prevent the quadcopter from potentially reversing. If the quadcopter repeats this logic in the next instant, it will conclude that it needs to turn right again resulting in it ultimately going backward (Fig 5).

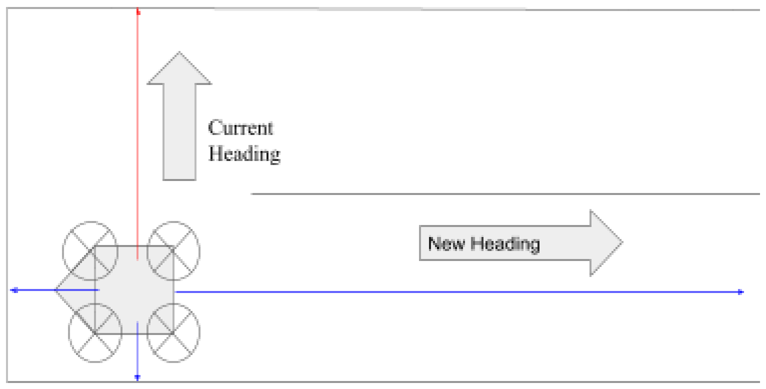


Fig 5. Algorithm Causing Quadcopter to Reverse

To solve this problem, our approach was to introduce a delay between decisions to allow the quadcopter to implement the direction it has chosen and clear corner scenarios like this. After experimenting thoroughly, we concluded that a three-second delay was the optimal value to keep the algorithm reliable, while not slowing down traversal.

The aforementioned logic allows the quadcopter to decide whether to turn or continue in its current heading, but we also need to be able to decide when to land. It turns out this can be solved simply by checking if each of the forward, left, and right LiDAR distances are sufficiently small (about 40 cm) when updating the heading. If so, this means the quadcopter has nowhere left to go and has reached the end of the maze.

To summarize this algorithm (Fig 6), first, the quadcopter sets its heading as "forward" (relative to its nose) and takes off. Then it moves in the direction of the heading for three seconds (the delay). After that, it completes its check of whether or not to land and whether or not to turn. If it decides to land, the algorithm is complete. Otherwise, it returns to moving in the direction of the new heading for three seconds and loops.

In the background, we have a collision avoidance protocol that checks if the quadcopter is getting too close to any walls and if so, acts to move it away possibly overriding its movement in the heading direction. This algorithm is influenced by the cutoff distance for it to activate and several parameters in the Mission Planner software including the PID control, all of which had to be tuned. See Appendix I for the full maze traversal algorithm code.

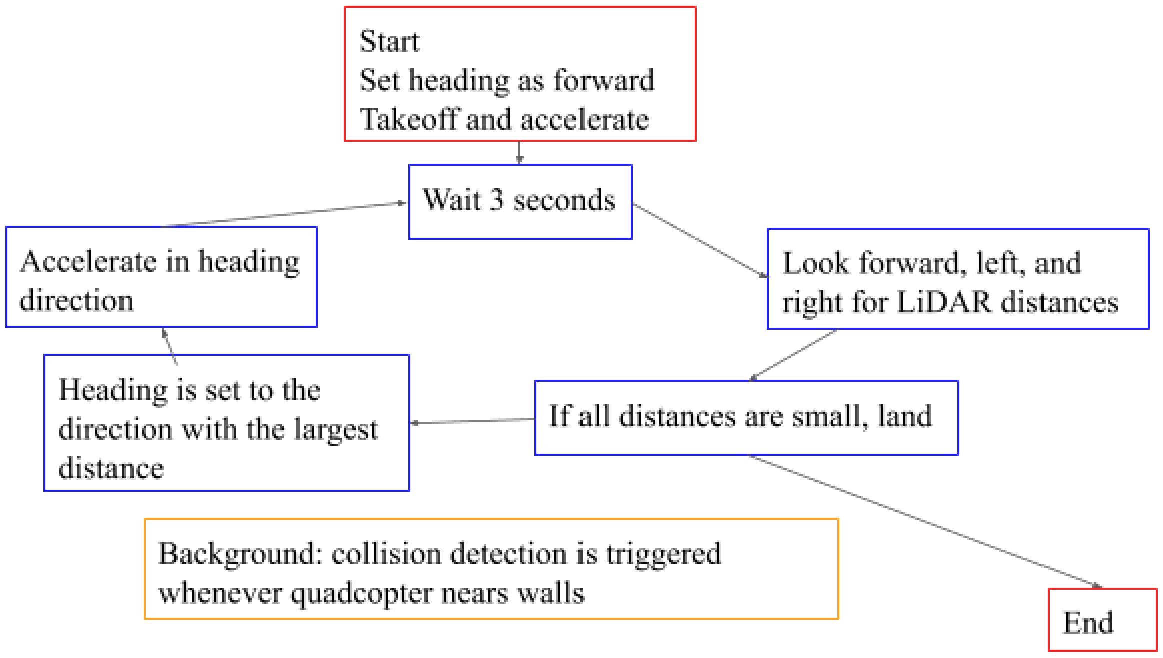

Start
Set heading as forward
Takeoff and accelerate
Wait 3 seconds
Look forward, left, and right for LiDAR distances
If all distances are small, land
Heading is set to the direction with the largest distance
Accelerate in heading direction
Background: collision detection is triggered whenever quadcopter nears walls
End

Figure 6. Maze Algorithm Chart

Discussion of Results and Recommendations

Overall, we feel that our results on the maze project met the objectives that we set out to accomplish. Our best flight had only one minor collision and was completed in just under thirty seconds. As such, we believe that we met the primary objective of navigating the maze autonomously and the optimization objectives of reducing collisions and increasing speed.

Additionally, the algorithm in its current state works, theoretically, on any maze as long as it has no dead-ends, similar starting and ending conditions, and similar corridor width. Albeit, we were not able to test this due to time restrictions.

The primary place for improvement that we noted was in reliability, especially in consistently minimizing collisions. For the most part, we were able to complete the maze reliably, however, we often had in the range of three to five collisions.

If given more time and resources, we would recommend finer tuning of collision avoidance and PID parameters and perhaps investing in better LiDAR sensors to get a more accurate idea of the space around the quadcopter for decision making.

Spatial Visualization

Objectives

For our custom project, we set out to construct a 3-dimensional visualization of the space our quadcopter is flying in, based on post hoc analysis of onboard sensor data obtained from the quadcopter's exploration flight-logs. Several high-level objectives include extracting position, orientation, and LiDAR data from the quadcopter flight logs. Furthermore, an algorithm to fly the quadcopter such that reliable data can be obtained is necessary. Finally, we wanted to develop a correct and reliable algorithm for rendering the space using the extracted data.

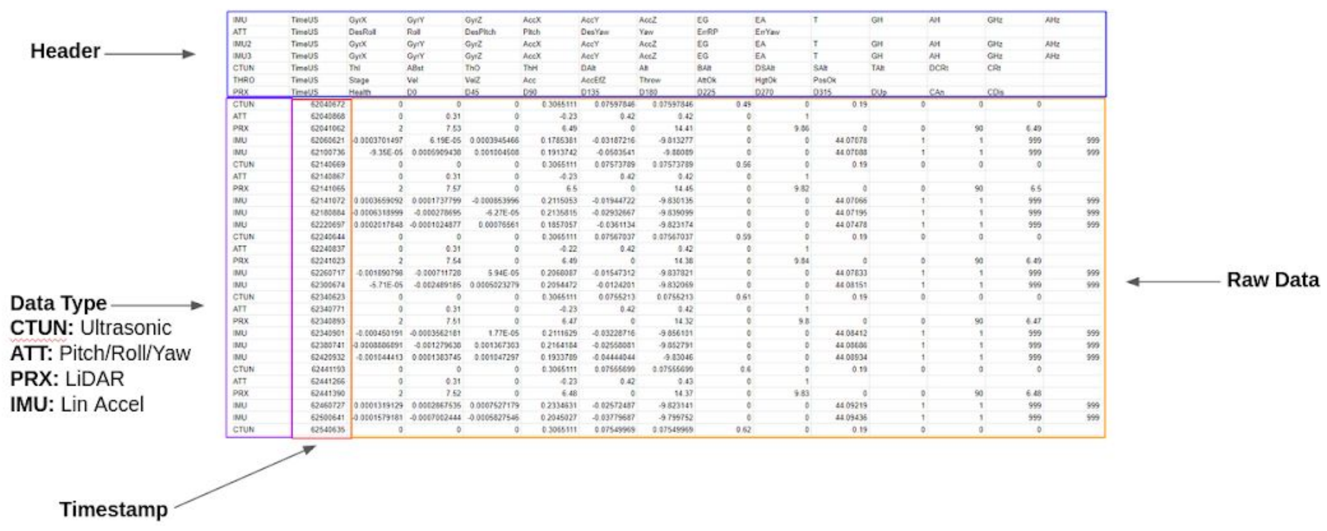
Log Extraction

To render the space of the quadcopter by analyzing past flight data, we needed to successfully extract useful sensor information from the Mission Planner logs. To define the quadcopter position and orientation, referred to from this point as the quadcopter pose, we required linear acceleration, altitude, and gyroscope orientation data. To estimate the distance of the ambient environment to build a spatial map, we required distance data.

These data values are made available to the quadcopter by the downward-facing ultrasonic sensor, the LiDAR box situated at the top of the quadcopter, and the 9 DoF IMU system integrated with the BeagleBone Blue microcontroller. This data is subsequently logged throughout a flight, along with many other sensor readings. We take interest in only the aforementioned data types, and thus as a part of our log extraction process, we run our log through a Windows Powershell command that filters for rows containing the keywords "IMU", "PRX", "CTUN", or "ATT" (Fig 7). The result is a dataset (Fig 8) that includes only the data types that we are interested in manipulating and plotting to our map with our visualization algorithm.

```
select-string IMU,PRX,CTUN,ATT '53 10-13-2020 11-31-05 AM.bin.log' | select-object -expandproperty line > outfile.csv
```

Fig 7. Windows Powershell Keyword Filter Command


Header
Raw Data
Data Type
CTUN: Ultrasonic
ATT: Pitch/Roll/Yaw
PRX: LiDAR
IMU: Lin Accel
Timestamp

IMU	TimeUS	GyrX	GyrY	GyrZ	AccX	AccY	AccZ	EG	EA	T	GH	AH	GHz	AHz
ATT	TimeUS	DesRoll	Roll	DesPitch	Pitch	DesYaw	Yaw	ErrRP	ErrYaw					
IMU2	TimeUS	GyrX	GyrY	GyrZ	AccX	AccY	AccZ	EG	EA	T	GH	AH	GHz	AHz
IMU3	TimeUS	GyrX	GyrY	GyrZ	AccX	AccY	AccZ	EG	EA	T	GH	AH	GHz	AHz
CTUN	TimeUS	ThI	ABst	ThO	ThH	DAlt	Alt	BAlt	DSAlt	SAlt	TAlt	DCRt	CRt	
THRO	TimeUS	Stage	Vel	VelZ	Acc	AccEfZ	Throw	AttOk	HgtOk	PosOk				
PRX	TimeUS	Health	D0	D45	D90	D135	D180	D225	D270	D315	DUp	CAn	CDis	
CTUN	62040672	0	0	0	0.3065111	0.07597846	0.07597846	0.49	0	0.19	0	0	0	
ATT	62040868	0	0.31	0	-0.23	0.42	0.42	0	1					
PRX	62041062	2	7.53	0	6.49	0	14.41	0	9.86	0	0	90	6.49	
IMU	62060621	-0.0003701497	6.19E-05	0.0003945466	0.1785381	-0.03187216	-9.813277	0	0	44.07078	1	1	999	999
IMU	62100736	-9.35E-05	0.0006909438	0.001004508	0.1913742	-0.0503541	-9.88089	0	0	44.07088	1	1	999	999
CTUN	62140669	0	0	0	0.3065111	0.07573789	0.07573789	0.56	0	0.19	0	0	0	
ATT	62140867	0	0.31	0	-0.23	0.42	0.42	0	1					
PRX	62141065	2	7.57	0	6.5	0	14.45	0	9.82	0	0	90	6.5	
IMU	62141072	0.0003659092	0.0001737799	-0.000863996	0.2115053	-0.01944722	-9.830135	0	0	44.07066	1	1	999	999
IMU	62180884	-0.0006318999	-0.000278695	-6.27E-05	0.2135815	-0.02932667	-9.839099	0	0	44.07195	1	1	999	999
IMU	62220697	0.0002017848	-0.0001024877	0.00076561	0.1857057	-0.0361134	-9.823174	0	0	44.07478	1	1	999	999
CTUN	62240644	0	0	0	0.3065111	0.07567037	0.07567037	0.59	0	0.19	0	0	0	
ATT	62240837	0	0.31	0	-0.22	0.42	0.42	0	1					
PRX	62241023	2	7.54	0	6.49	0	14.38	0	9.84	0	0	90	6.49	
IMU	62260717	-0.001890798	-0.000711728	5.94E-05	0.2068087	-0.01547312	-9.837821	0	0	44.07833	1	1	999	999
IMU	62300674	-5.71E-05	-0.002489186	0.0005023279	0.2094472	-0.0124201	-9.832069	0	0	44.08151	1	1	999	999
CTUN	62340623	0	0	0	0.3065111	0.0756213	0.0756213	0.61	0	0.19	0	0	0	
ATT	62340771	0	0.31	0	-0.23	0.42	0.42	0	1					
PRX	62340893	2	7.51	0	6.47	0	14.32	0	9.8	0	0	90	6.47	
IMU	62340901	-0.000450191	-0.0003562181	1.77E-05	0.2111629	-0.03228716	-9.856101	0	0	44.08412	1	1	999	999
IMU	62380741	-0.0008806891	-0.001279638	0.001367303	0.2164184	-0.02558081	-9.852791	0	0	44.08686	1	1	999	999
IMU	62420932	-0.001044413	0.0001383745	0.001047297	0.1933789	-0.04444044	-9.83046	0	0	44.08934	1	1	999	999
CTUN	62441193	0	0	0	0.3065111	0.07555699	0.07555699	0.6	0	0.19	0	0	0	
ATT	62441266	0	0.31	0	-0.23	0.42	0.43	0	1					
PRX	62441390	2	7.52	0	6.48	0	14.37	0	9.83	0	0	90	6.48	
IMU	62460727	0.0001319129	0.0002867535	0.0007527179	0.2334631	-0.02572487	-9.823141	0	0	44.09219	1	1	999	999
IMU	62500641	-0.0001579181	-0.0007002444	-0.0005827546	0.2045027	-0.03779687	-9.799752	0	0	44.09436	1	1	999	999
CTUN	62540635	0	0	0	0.3065111	0.07549969	0.07549969	0.62	0	0.19	0	0	0	

Fig 8. First 35 Rows of a Filtered Log

Flight & Spatial Visualization Algorithm

One of our objectives is for the logged data to be reliable, so we needed to design our flight algorithm with this in mind. And to do this, we modified the template flight algorithm so that our quadcopter ascended, spun around 3 times, and then landed. Our quadcopter PID and collision avoidance layers were left unchanged from the maze traversal task.

For the spatial visualization, the algorithm runs a core loop iterating over the rows of data (Fig 10). Taking a log such as the one shown in the previous section, our objective is to manipulate the raw data to produce a visualization of the flight path and an environment map that is intuitive to us.

At the start of the algorithm's core loop, we are only concerned with the very first row of data, ignoring the headers. We read in the data type and based on what that data type is, we manipulate the raw data in a certain way with some math, and we update a buffer to hold this information temporarily. For example, if the data row was from the IMU, we want to double-integrate the linear acceleration values with respect to time to obtain absolute displacement information, which we subsequently store in a dx and dy buffer.

Once a buffer is updated, we take these things -- the pitch-roll-yaw, altitude, dx dy, and define the quadcopters pose. The quadcopter pose is represented in our algorithm by a 4x4 homogeneous transformation matrix and contains information about the quadcopter's location in 3d space as well as its orientation.

The math to update the pose estimation involves a series of affine transformations on a 4x4 matrix, which we'll briefly touch on (Fig 9). Precisely, the transformation includes a translation, rotation, followed by another translation. To define an updated pose, we take a 4x4 matrix containing the dx and dy displacement information from the buffer. This effectively translates a point along the x-y plane according to the environment frame, but not the quadcopter frame. Next, the resultant matrix is dotted with a rotation matrix containing the roll-pitch-yaw information from the buffer. This effectively makes our first dx & dy displacement matrix a translation from the origin in relation to the quadcopter reference frame. Finally, this result is dotted with another translation matrix which translates the point using the x-y-z cartesian coordinates from the previous pose. Finally, the altitude buffer updates the z coordinate in the pose matrix. This produces a 4x4 matrix containing the quadcopter's updated location in 3d space as well as its new orientation.

$$P = T_2 \cdot R \cdot T_1$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{21} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & b_{14} \\ 0 & 1 & 0 & b_{24} \\ 0 & 0 & 1 & b_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_y \cdot c_p & c_y \cdot s_p \cdot s_r - s_y \cdot c_r & c_y \cdot s_p \cdot c_r + s_y \cdot s_r & 0 \\ s_y \cdot c_p & s_y \cdot s_p \cdot s_r + c_y \cdot c_r & s_y \cdot s_p \cdot c_r - c_y \cdot s_r & 0 \\ -s_p & c_p \cdot s_r & c_p \cdot c_r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where P is the final pose matrix,
dx dy are from the displacement buffers,
c_x or s_x denote $cos(x)$ or $sin(x)$, respectively,
p, r, or y are from the attitude buffer,
and b_{14}, b_{24}, and b_{34}, are the respective x y z Cartesian coordinates from the previous pose

Fig 9. Affine Transformation Series to Determine the Updated Pose Estimate Matrix P

Through a similar process, we manipulate the LiDAR readings and add these to a LiDAR scatter point list where they all get added to our mapped visualization.

Finally, the loop moves on to the next row where the same process is repeated for a new data type.

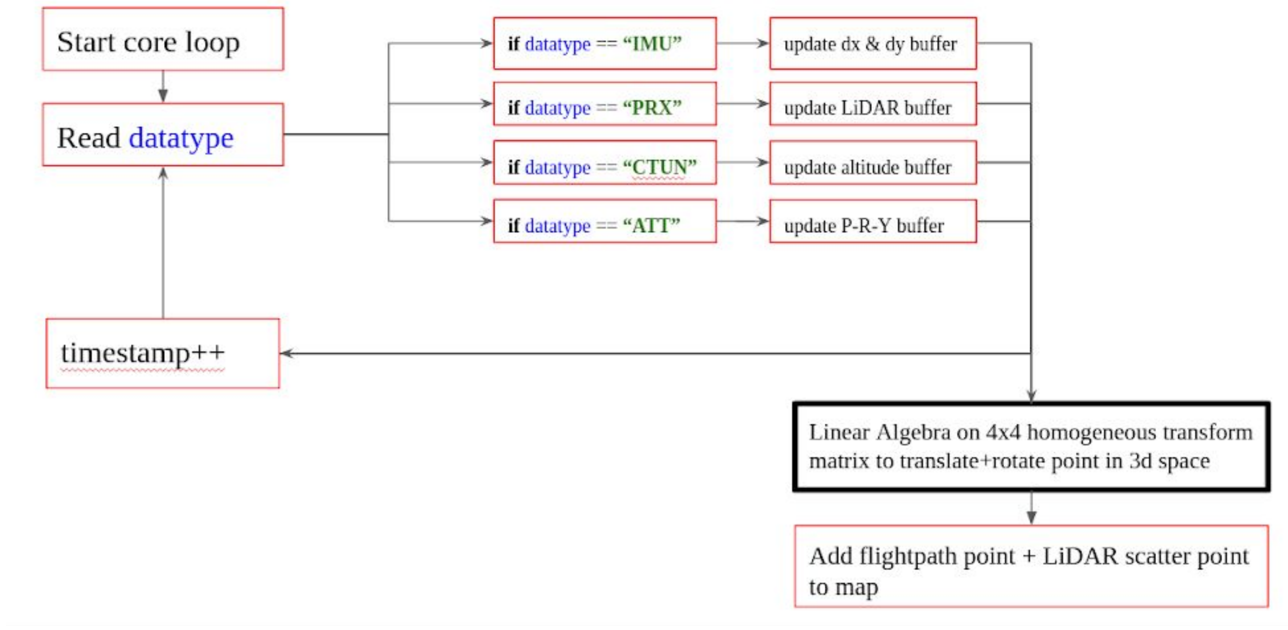


Fig 10. Visualization Algorithm Core Loop

Discussion of Results and Recommendations

We feel that the results of the code, data extraction, and modeling were satisfactory. We were able to achieve a model which showed a structure that did resemble our enclosed space (Fig 11), and we did so while extracting the data we needed from the quadcopter, and being able to manipulate it. It also appeared that the orientation given by the gyroscope was fairly accurate.

On top of our results, which prove the concepts of modifying the quadcopter with sensors and analyzing the data that comes from the IMU, we also found a significant room for improvement which the client and their customer can take advantage of.

Due to the unreliable nature of the IMU, the positional readings experienced significant amounts of drift, which could be improved upon given better IMUs, or an improved filter or technique, such as a Kalman filter. For the mapping portion, better LiDAR sensors could be implemented, and other technology and techniques like SLAM or cameras can improve the visual aspect of this project.

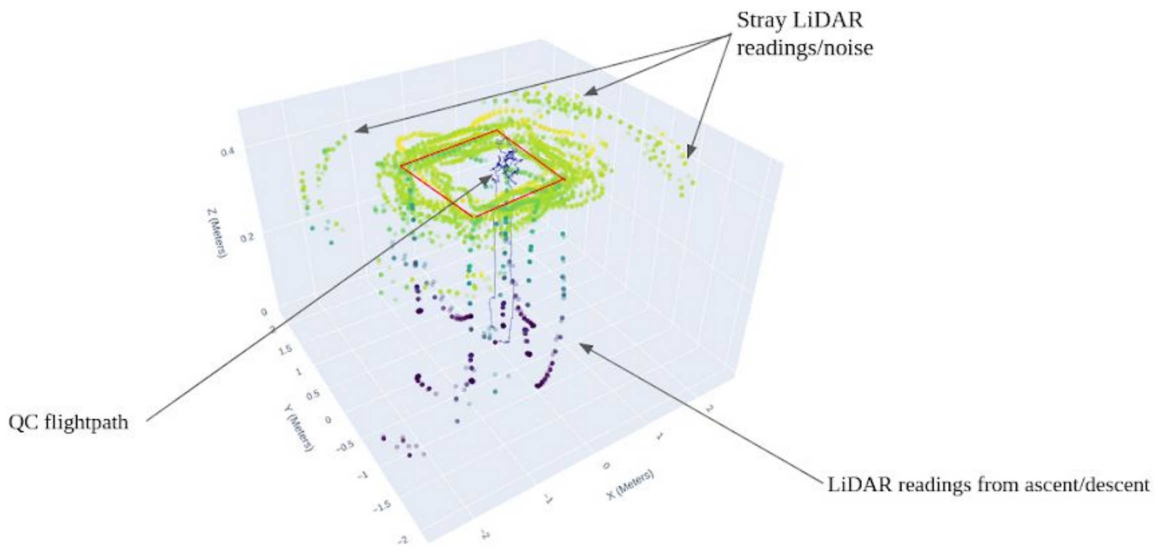


Fig 11. Resultant environment map

Conclusion

Our successful maze flight following the implementation of our algorithm and parameters was achieved in less than thirty seconds, with one minimal collision. The drone had countless occasions where it could have hit a wall or edge of a wall, but successfully avoided it, and showed that it met all the needs of our customer and their audience. The drone can successfully fly through a maze, by itself, in an efficient manner. Beyond that, the algorithm also works for any arbitrary maze, meaning a layout like the one we tested isn't necessary, and any university or enthusiast can use it for personal use.

To further minimize collisions from one to zero, further time could be spent from us, or from the universities and enthusiasts to improve the flight parameters, or modifications could be made to improve sensors and actuators on the prototype. What is presented by the prototype is sufficient for a successful and efficient flight, but the client's customers can explore physical and parameter improvements.

On a similar subject of physical implementations of sensors, our spatial visualization project showed that sensors can be implemented successfully onto the quadcopter. For our spatial visualization project, we were able to successfully, albeit roughly, map a square enclosure. Proving both the effectiveness and possible use of the LiDAR sensor on a quadcopter. LiDAR data, along with flight data from the quadcopter in general, can be extracted successfully with timestamps, which implies that countless analytical procedures, such as our modeling of the positional data, can be achieved. The "box" plotted using the LiDAR also proves the quadcopter can be modified successfully beyond its standard use of flying and sensors are compatible with it, making it of great and versatile use to universities and enthusiasts.

Improvements can be made to this part of the drone. The IMU positional tracking suffered a great deal of drift, so better IMU's and filtering techniques, such as a Kalman filter, can be employed by the user for tasks and projects that require a greater degree of accuracy. Other improvements can also include using cameras and SLAM techniques instead of LiDAR for better mapping.

Appendix I: Maze Traversal Code

```
bool Copter::autonomous_controller(float &target_climb_rate, float &target_roll, float &target_pitch, float
&target_yaw_rate)
{
    static int timer=0;
    static int current=0;
    float rangefinder_alt = (float)rangefinder_state.alt_cm / 100.0f;
    const float dist_meter=0.5f;
    float dist_forward, dist_right, dist_backward, dist_left;
    g2.proximity.get_horizontal_distance(0, dist_forward);
    g2.proximity.get_horizontal_distance(90, dist_right);
    g2.proximity.get_horizontal_distance(180, dist_backward);
    g2.proximity.get_horizontal_distance(270, dist_left);
    array<float,4>dist={dist_forward,dist_left,dist_backward,dist_right};
    target_pitch=0.0;
    target_yaw_rate = 0.0f;
    target_roll=0.0;
    target_climb_rate = 0.0f;
    //collision avoidance code
    float angle_wall=150.0f;
    if(dist_forward<10*(1.0f*dist_meter)){
        g.pid_pitch.set_input_filter_all((10*(1.0f*dist_meter))-dist_forward);
        target_pitch+=angle_wall*g.pid_pitch.get_pid();
    }
    if(dist_backward<10*(1.0f*dist_meter)){
        g.pid_pitch.set_input_filter_all(-10*(1.0f*dist_meter)+dist_backward);
        target_pitch+=angle_wall*g.pid_pitch.get_pid();
    }
    if(dist_right<(10*1.0f*dist_meter)){
        g.pid_roll.set_input_filter_all(-10*(1.0f*dist_meter)+dist_right);
        target_roll+=angle_wall*g.pid_roll.get_pid();
    }
    if(dist_left<(10*1.0f*dist_meter)){
        g.pid_roll.set_input_filter_all(10*(1.0f*dist_meter)-dist_left);
        target_roll+=angle_wall*g.pid_roll.get_pid();
    }
    if(timer>=1200){    //3 second delay
        //update heading
        int max=current;
        if(dist[(current-1)%4]>dist[max]){
            max=(current-1+8)%4;
        }
        if(dist[(current+1)%4]>dist[max]){
            max=(current+1+8)%4;
        }
        current=max;
        timer=0;
        //check landing condition
        const float dist_stop=0.5;
        if(dist[current]<dist_stop*10.0f){
            return false;
        }
        gcs_send_text_fmt(MAV_SEVERITY_INFO,"%i",current);
    }
    const float dist_meter2=0.5;
    const float del_angle=275;
    //move forward
    if(dist[current]>10.0f*dist_meter2)
        current%2==0 ? target_pitch=pow(-1,1+current/2)*del_angle:target_roll=pow(-1,1+current/2)*del_angle;

    timer++;
    return true;
}
```

Appendix II: Spatial Map Visualization Code

```
#import matplotlib.pyplot as plt
import plotly.graph_objects as go
import plotly.express as px
import numpy as np
#from mpl_toolkits.mplot3d import Axes3D
from FIR import fir
import time

filename = 'log100sel.csv'
data = np.genfromtxt(filename, delimiter=",", dtype="|U10", usecols=(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10))

class QC:
         def __init__(self, x, y, z, roll, pitch, yaw, data):
         self.x = x
         self.y = y
         self.z = z
         self.dx = x
         self.dy = y
         self.roll = roll
         self.yaw = yaw
         self.pitch = pitch
         self.pointset = [[x,y,z]]
         self.LiDARptset = []
         self.pose = [[1,0,0,x],
                  [0,1,0,y],
                  [0,0,1,z],
                  [0,0,0,1]]
         self.LiDAR = [[1,0,0,x],
                  [0,1,0,y],
                  [0,0,1,z],
                  [0,0,0,1]]
         self.prx= [0,0,0,0]
         self.data = data
         self.I = [[1,0,0,0],
                  [0,1,0,0],
                  [0,0,1,0],
                  [0,0,0,1]]

         def translate(self, dx, dy):
         return [[1,0,0,dx],
                  [0,1,0,dy],
                  [0,0,1,0],
                  [0,0,0,1]]
         def rotate(self, yaw, pitch, roll):
         return [[np.cos(yaw)*np.cos(pitch),
         np.cos(yaw)*np.sin(pitch)*np.sin(roll)-np.sin(yaw)*np.cos(roll),
         np.cos(yaw)*np.sin(pitch)*np.cos(roll)+np.sin(yaw)*np.sin(roll),0],


         [np.sin(yaw)*np.cos(pitch),
         np.sin(yaw)*np.sin(pitch)*np.sin(roll)+np.cos(yaw)*np.cos(roll),
         np.sin(yaw)*np.sin(pitch)*np.cos(roll)-np.cos(yaw)*np.sin(roll),0],


         [-np.sin(pitch),
         np.cos(pitch)*np.sin(roll),
         np.cos(pitch)*np.cos(roll),0],
         [0,0,0,1]]

         def addpoint(self):
         self.pose = np.dot(self.translate(self.pose[0][3], self.pose[1][3]),
         np.dot(self.rotate(self.yaw, self.pitch, self.roll),
         np.dot(self.translate(self.dx, self.dy),
          self.I)))

          self.x = self.pose[0][3]
          self.y = self.pose[1][3]
          for i in range(0,4):
          self.LiDAR = np.dot(self.translate(self.pose[0][3], self.pose[1][3]),
          np.dot(self.rotate(self.yaw+(i*np.pi/2), self.pitch, self.roll),
          np.dot(self.translate(self.prx[i], 0),
          self.I)))
          self.LiDARptset.append([self.LiDAR[0][3]/10,self.LiDAR[1][3]/10,self.z])
          self.pointset.append([self.x, self.y, self.z])

          def update_line(self, hl, new_data):
           xdata, ydata, zdata = hl._verts3d
           hl.set_xdata(list(np.append(xdata, new_data[0])))
           hl.set_ydata(list(np.append(ydata, new_data[1])))
           hl.set_3d_properties(list(np.append(zdata, new_data[2])))
           plt.draw()

          def core(self):
          a=0
          del_time = 0.000001
          time = float(self.data[0][1])
          ATT = []
          filter = fir(5)
          for i in range(1,self.data.shape[0]):
          if self.data[i][0] == 'PRX':
                   a+=1
                   self.prx = [float(self.data[i][3]),
                   float(self.data[i][5]),
                   float(self.data[i][7]),
                   float(self.data[i][9])]
          if self.data[i][0] == 'IMU':
                   a+=1
                   newTime = float(self.data[i][1])
                   velx= float(self.data[i][5]) * del_time
                   vely= float(self.data[i][6]) * del_time
                   self.dx = (float(self.data[i][5])*0.5*del_time*del_time+velx*del_time)
                   self.dy = (float(self.data[i][6])*0.5*del_time*del_time+vely*del_time)
                   del_time = (newTime-time) * 0.000001
                   time=newTime
          if self.data[i][0] == 'CTUN':
                   a+=1
                   self.z=filter.apply_sma(float(data[i][10]) - 0.19)
          if self.data[i][0] == 'ATT':
                   a+=1
                   self.roll = ((float(self.data[i][3])*np.pi/180))
                   self.pitch = ((float(self.data[i][5])*np.pi/180))
                   self.yaw = ((float(self.data[i][7])*np.pi/180))
          if a>=1:
                   self.addpoint()
                   a=0

qc = QC(0,0,0,0,0,0, data)
qc.core()

x= [row[0] for row in qc.LiDARptset]
y= [row[1] for row in qc.LiDARptset]
z= [row[2] for row in qc.LiDARptset]
fig = go.Figure()
fig.add_trace(go.Scatter3d(
          x=x,
          y=y,
          z=z,
          mode='markers',
          marker=dict(
          size=2,
         color=z,                    # set color to an array/list of desired values
         colorscale='Viridis',   # choose a colorscale
         opacity=0.3
         )
))
xp = [row[0] for row in qc.pointset]
yp = [row[1] for row in qc.pointset]
zp = [row[2] for row in qc.pointset]
fig.add_trace(go.Scatter3d(
         x=xp, y=yp, z=zp,
         marker=dict(
         size=0.1,
         colorscale='Viridis',
         ),
         line=dict(
         color='darkblue',
         width=3
         )
))

fig.update_layout(
         width=1800,
         height=1000,
         autosize=False,

         scene=dict(
         xaxis_title='X (Meters)',
         yaxis_title='Y (Meters)',
         zaxis_title='Z (Meters)',
         camera=dict(
         up=dict(
                  x=0,
                  y=0,
                  z=1
         ),
         eye=dict(
                  x=0,
                  y=1.0707,
                  z=1,
         )
         ),
         aspectratio = dict( x=1, y=1, z=0.7 ),
         aspectmode = 'manual'
         ),
)

fig.show()
```