UNIVERSITY OF BATH

FACULTY OF ENGINEERING AND DESIGN

IMEE FINAL YEAR MEng PROJECT REPORT

Collaborative Autonomous Ground-Air Vehicle

Finley Hewson

20/05/2020

"I certify that I have read and understood the entry on Plagiarism, duplication of one's own work and Cheating on the Quality Assurance Code of Practice QA53 and in my Departmental Student Handbook. All material in this assignment is my own work, except where I have indicated with appropriate references. I agree that, in line with Regulation 15.3(e), if requested I will submit an electronic copy of this work for submission to a Plagiarism Detection Service for quality assurance purposes."

Author's signature: Finley Hewson

Supervisor: *Dr Pejman Iravani*

Assessor: *Dr Kang Ma*

Summary

A project was initiated at the University of Bath to develop an entry to the European Robotics League Emergency Competition. This required the development of a collaborative team of air-ground vehicles, capable of autonomous navigation in a static, complex environment. In the scope of the project, there was a need to design and build the unmanned ground vehicle component of the team. This report documents the process of selection and integration of the hardware components required for the Unmanned Ground Vehicle and the design and creation of the autonomous guidance, navigation, and control system code. This system is responsible for the robot's localisation, mapping, local path planning, and actuation functionality.

A review of pertinent literature, combined with a previous review carried out by the author, aided in the selection of the methods of collaboration with the Unmanned Aerial Vehicle, path planning, mapping and localisation and the sensors which would be used for each of these functions. This led to the decision to use the hybrid A* path planner. For the mapping method an occupancy grid map was selected due to a need for a metric map representation and the its simplicity to implement. The map was populated with data from the Intel D435 Depth camera. Localisation was carried out using Visual Inertial Odometry SLAM on the Intel T265 Tracking camera. The method of collaboration with the UAV was defined.

Following this, hardware was selected and assembled for a full-scale implementation of the UGV. The software of the UGV's Guidance, Navigation, and Control system was developed, and the methodology and choice of implementation was justified. Simulations were carried out over a range of test scenarios, to prove the design's efficacy. It was shown that compared to the A* path planner the Hybrid A* path planner would produce smoother more efficient routes. A real-world test was carried out and it was proven that the design could calculate an efficient route and avoid static obstacles in an online manner. If newly detected obstacles were within the initial planned loop, the path planner could create a new path plan.

Some issues were identified with the current design's implementation such as the path planner only considering the current target waypoint, leading to suboptimal routes. However, the UGV and its autonomous guidance, navigation and control system are deemed a success

due to being able to operate in a real-world scenario. Though future work could be carried out to fully integrate it into the cooperative team and to improve the system's reliability.

Acknowledgements:

The author would like to acknowledge the following people for their support through this project:

Dr P. Iravani for his expertise, advice, and guidance.

Dr K. Ma for his advice and positive feedback.

Mr F. Sherratt for his coding support, expertise, and general knowledge of Mobile Autonomous Robots, which was invaluable.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Background

The use of autonomous mobile robots is becoming more prevalent in society with the development of the technology, and increasing demand for the automation of dangerous or undesirable tasks. The market size for mobile autonomous robots is expected to reach $58.9 billion by 2026 [1]. However, the developments in technology that occur from this field could have a huge impact on much larger markets, such as how the development of sensors and path planning algorithms could be implemented in the automotive industry for autonomous navigation.

There are many sectors which use autonomous mobile robots in their respective operations. Mobile autonomous robots are used by the warehouse and logistics industry, where they can be beneficial by significantly reducing operational costs. Another sector is the defence and security industry where they can be used to supply ammunition to soldiers during combat situations or to carry out search and rescue tasks in hostile environments to humans or during disasters. It has been estimated that the autonomous mobile robot market will have a huge compound annual growth rate of 15.12% between the forecasted period of 2020-2025 [2].

Whilst high demand has driven research and development in this sector, there still exists numerous challenges which need to be overcome to improve the functionality, reliability, cost, and safety needed for these systems to be utilised in numerous industries. The European Robotics League (ERL) Emergency competition was created to help push the state of the art and meet these challenges, specifically in the sector of autonomous systems for emergency response. The competition seeks to advance development of outdoor and indoor navigation using air and ground vehicle cooperation. Team Bath Drones, a student lead autonomous aircraft competition team from the University of Bath, entered the ERL 2020 Emergency local competition in Poland for this reason.

This competition required the design and building of a collaborative autonomous ground and air-vehicle system. The competition tests the systems capability to work as a team to navigate autonomously in a GPS degraded area with obstacles in the way. To then localise missing persons (mannequins) in the environment and deliver first aid kits within a certain distance of

them. To detect and recognise different markers both in an indoor and outdoor environment and to create a map of the indoor environment. With a nominal budget of £250, the goal of this project was to create a low-cost design for the ground vehicle component of the system with the functionality to carry out the tasks detailed above.

1.2. Aims and Objectives

The aim of this project was to design and create an autonomous Unmanned Ground Vehicle (UGV) that would work with an Unmanned Aerial Vehicle (UAV) to navigate a real-world complex indoor and outdoor environment for the purpose of search and rescue. This mainly involves the design and creation of the autonomous guidance, navigation and control system which includes the creation of the robot's method of localising, mapping, local path planning and actuation. It also involves the design of the method through which the two vehicles would cooperate, and the selection and integration of Commercial Off the Shelf (COTS) hardware, which was used throughout the project due to the tight timeframe.

From the competition's rulebook [3], a number of system and functional requirements were defined, these are detailed in Table 1. These requirements were used to inform the design and scope of the system. In addition to these requirements and the budget limitations of £250, there was also an aim to ensure that the computational demand of the programs was minimised so that they would be able to operate in a temporally efficient real-time manner. It is assumed that the environment will be static, and that no prior information of the environment is given. Using these assumptions, the UGV must be capable of navigating through the environment whilst detecting and generating new paths to avoid obstacles in real-time.

Table 1: List of System and Function Requirements

ID	Title	Requirement
SYS-01	Mass	Must weigh under 350kg.
SYS-02	Size	No size limitations but larger robots will struggle in an indoor environment, so recommended to minimise dimensions.
SYS-03	Actuation	Propulsion and steering method must be through traction with the ground.
SYS-04	Autonomous Operation	The robot's low-level motor control including starting, stopping, and steering, together with medium-level control such as navigation, are performed without human intervention.
SYS-05	Cooperation	Safety concerns prohibit the aircraft and ground vehicle operating simultaneously, only one may be in operation at any given time.
SYS-06	Duration	Must be able to operate for over 30mins.
FNC-01	Emergency Stop	Wireless Emergency Stop System required.
FNC-02	Mounted Emergency Stop	Externally actuated emergency stop capability.
FNC-03	Warning Devices	Warning lights when operating, recommended on vehicles under 20kg, mandatory on vehicles over.
FNC-04	Range	Must operate in a 200x30m area with Visual Line of Sight (VLOS).
FNC-05	Return	Vehicle must return to starting area after all tasks completed.
FNC-06	Communications	Vehicle transmits live position and images to the control station during run.
FNC-07	Waypoint Navigation	Vehicle must autonomously navigate to within 3m of three waypoints, avoiding obstacles along the route.
FNC-08	Mapping	Vehicle builds a geometric indoor map of the indoor building environment.
FNC-09	Mission	Vehicle must autonomously detect missing workers, report positions, and deploy first-aid kits within 1m of the workers.

1.3. Report Layout

This report aims to document the entire design and development process of the UGV's autonomous navigation system, culminating in an operational mobile autonomous robot. First building upon the author's previous path planning literature review completed in the preliminary project background review [4], a further literature review on the subject is carried out covering some areas that were missed previously for the sake of brevity and focusing more on previously recommended methods. The literature review will also contain sections on methods for localisation and mapping and methods of collaboration with the aerial vehicle. From these methods a strategy is selected which will conform to the design's requirements and is then implemented.

The design process associated with the selection of the hardware necessary for the system is detailed and the selections are justified. Then the software developed for the implementations

of the autonomous navigation system's submodules (localisation, mapping, path planning, and locomotion control) is presented, and their functions explained. Each software submodule is tested to prove its functionality and then a full system test is carried out on the completed vehicle where its performance is analysed. A performance conclusion of the system is given with any issues with the current design and its implementation analysed. Finally, any further recommendations for improvements to the functionality or usability that could be made to the system are described.

2. Literature review

This section of the report seeks to provide an overview on the different state-of-the-art methods and techniques in path planning, mapping and localisation, and autonomous collaboration. The author previously completed a literature review focusing on path planning [4]. From this review a path planning method of A* algorithm was outlined as most appropriate due to its robustness and capability to produce the global optimum solution. In this report, one of the sections of the literature review further focuses on variations of the A* algorithm and other path planning techniques which were missed in the author's previous review for the sake of brevity. There is also a section reviewing mapping and localisation techniques due to them being a core competency of autonomous navigation, the review is used to help inform the selection of the most suitable method for this system. A final literature review is given on the potential approaches to the collaboration of the UGV and UAV to achieve the systems functional requirements outlined in Table 1, and a concluding judgment on which method to use is made.

2.1.Path Planning

2.1.1. Sampling based Algorithms

2.1.1.1. Probabilistic RoadMaps (PRMs)

Sampling based approaches differ from approaches like potential fields and cell decomposition, as they do not discretise the search space. Sampling based approaches aim to avoid explicit representations of the search space by relying upon collision checking modules to validate the feasibility of candidate trajectories and make a roadmap of feasible trajectories to connect points in the search space [5]. This makes the sampling approach very efficient for path planning in high dimensional spaces and search spaces with many obstacles.

PRMs are a multiple query method which construct a graph of the search space made up of successful trajectory connections between random nodes and then it enters a query step where the shortest path that connects the initial state with a final state through the roadmap is calculated [6]. Standard smoothing techniques can be applied to make paths calculated suitable for non-holonomic vehicles. The multiple query method is not needed for online path-planning and unstructured environments.

2.1.1.2. Rapidly-exploring Random Trees (RRTs)

RRTs are a single query, tree-based planner, and the online complement to PRMs. They differ in that they are incremental in their sampling of trajectories. This avoids the number of samples of the search space having to be set prior, and the solution from initial state to the final state will be calculated as soon as there is a large enough set of connections to reach it. This results in RRTs being highly computationally efficient, and it has proven to be probabilistically complete [7]. There are some limitations to the method, such as in a highly complex search space with lots of obstacles there is a smaller chance of finding a point that is not inside an obstacle and can connect to the tree. This can make navigation down small passages difficult.

2.1.2. Reeds Shepp

Reeds Shepp curves are an extension of Dubins curves, which is the shortest curve that connects an initial state and a final state in Euclidean space given a constraint to the curvature. In Dubins curves it is assumed that the vehicle can only travel forwards, in the Reeds Shepp curves the vehicle can also reverse [8]. The advantage of Reeds-Shepp path planning is that it does not require discretisation of the work space or to build multiple trajectories, instead using the concatenation of several linear and circular segments can be used to find the shortest path for a non-holonomic vehicle [9]. This method is easy to implement in cluttered environments and will produce routes that a non-holonomic vehicle can follow. It is however not complete and so can fail to find paths between points.

2.1.3. Graph Search Approaches

2.1.3.1. A* search algorithms

As mentioned by the author in a previous literature review, A* is one of the most efficient path-finding algorithms. Using the Euclidean distance heuristic, the A* algorithm tries to find the shortest path to its goal node, this is illustrated in Figure 1. This allows it to achieve better temporal efficiencies than the Dijkstra algorithm on which it is based, it is also complete and optimal, so it is guaranteed to find the shortest route to a problem if it exists. However, the A* algorithm's speed of computation is tied to the heuristic used and for large and complex

grid spaces it can be very slow and unsuited for real-time calculation. For a 2000*2000 grid map, the general A* algorithm had an execution time of 3 hours, unacceptable for real-time applications [10].

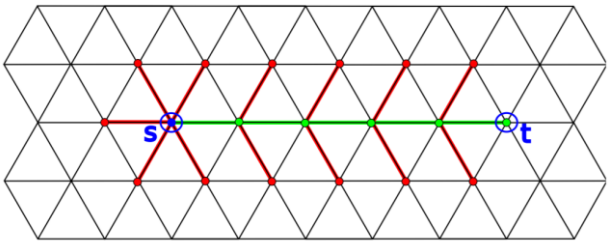

Figure 1: Graph showing A algorithm searching from S to T* [11]

2.1.3.2. Field D*

The Field D* algorithm attempts to solve an issue with the A* algorithm's limitation of planning to the centre of each discretised grid of the search space as seen in Figure 2 (a). It is an extension of the D* algorithm which is like the A* algorithm but an informed incremental search algorithm. This gives the D* greater computational efficiency than the A*. Paths using this method are restricted in their headings to being increments of $\frac{\pi}{4}$, which can result in sub optimal paths, unnecessary turns and just impossible routes if the vehicle isn't holonomic [12]. To solve this the Field D* algorithm uses linear interpolation to produce paths that are not restricted to set increments of headings and as a result can produce smoother more cost optimal paths. It is possible for the Field D* linear interpolation assumption to be wrong and produce suboptimal paths.

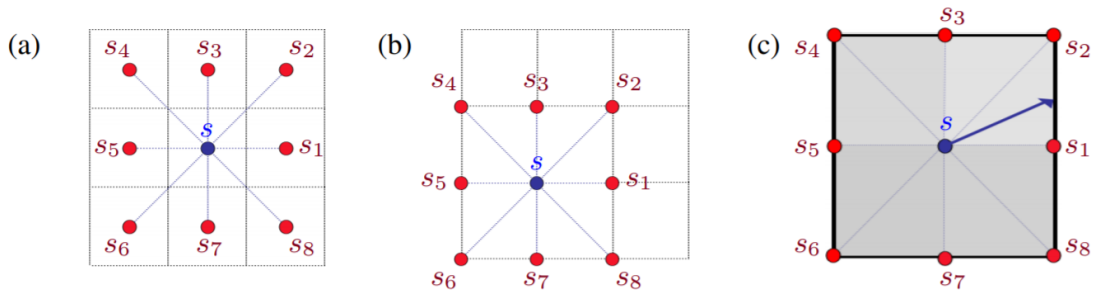

Figure 2: (a) Nodes reside in the centre of the grid cell. (b) Nodes reside in the corners of the grid cell. (c) The shortest path from the node is through one of the edges [12]

2.1.3.3. Hybrid A*

The Hybrid A* path planning algorithm is an extension of the A* algorithm, that overcomes the limitations of kinematics being only considered for holonomic robots and the

consideration of the search space as discrete instead of continuous [13]. A comparison of the two methods can be seen in Figure 3. To consider the continuous nature of the search space, motion primitives are calculated that determine reachable states in each node. This helps produce smooth and efficient paths. The Hybrid A* has been successfully applied as a local path planner by a team from the University of Stanford in the DARPA Urban Challenge, proving its' real-time capabilities in a complex environment [14]. It is not a complete algorithm but improving the fineness of the resolution used for the discretisation of the grid space and the resolution of the motion primitives should improve the path planner's ability to find a path solution between an initial and final state.

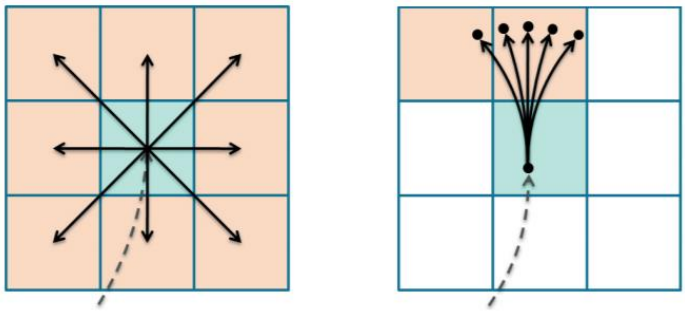

Figure 3: Comparison of regular A (left) and hybrid A* (right)*[13]

2.2. Localisation and Mapping

Localisation and mapping are two tasks which are essential to the development of vehicles capable of autonomous navigation. Localisation involves determining the vehicle's position in an environment. Mapping is the process of creating a representation of the environment surrounding the vehicle so that it can localise itself. These tasks can be combined to be simultaneously executed in a technique called Simultaneous Localisation and Mapping (SLAM), this method often results in more accurate maps [15].

Various sensors exist which are utilised for navigation, such as inertial sensors, infrared (IR) sensors, sonar, and LIDAR. Inertial measurement units can provide orientation and trajectory measurements to give the pose information of a moving vehicle, which is beneficial for localisation. IR sensors are useful for obstacle avoidance due to their inherently fast response time; however they do have downsides in terms of limited range, being affected by the

environment's ambient light sources, as well as being dependent on the reflectivity of the obstacle [16].

Sonar sensors can produce computationally affordable distance and imaging data which has been shown to have sufficient accuracy to classify plant types [17]. However, the performance of the sensors can be compromised by movement of the platform they are mounted to or if the environment is outdoors due to air movement. Loud noises from other machines in the environment can also degrade the sensor's sensitivity and they tend to have poor angular resolution due to their wide beam width [18]. LIDAR overcome some of the major disadvantages of the previously mentioned sensors. They have superior reliability, range accuracy and angular resolution than sonar sensors. They are however much more expensive devices which produce very large datasets, they are also prone to missing transparent objects.

IR, Lidar, and sonar sensors are all considered to be active sensors which alter the environment by emitting light or waves. Passive sensors have an advantage over the active sensor types, they can obtain a much greater amount of spatial and temporal information than active sensors, an example of this type of sensor is the camera. Vision based navigation systems have the advantages of being cheap and producing huge amounts of information when mapping the environment compared to active sensors. Previously, this was limited by the need for complex algorithms but the growth in cheap and fast computing power has opened the potential for real-time navigation systems using these types of sensors [19].

Due to the benefits that these passive sensors can provide, this review explores and contrasts the key performance characteristics of different solutions to localisation and mapping with a focus on vision-based navigation systems. Also, FNC-06 of the functional requirements details that the system is required to transmit live images, making the inclusion of a camera mandatory. Therefore, to reduce power consumption and complexity it makes sense to have a vision-based sensor as the main method of mapping and localisation, rather than having a camera and another sensor. This review is being done to help justify the choice in the method used for both localisation and mapping in a complex environment which has not previously been mapped for local path planning and obstacle avoidance.

2.2.1. Localisation

2.2.1.1. GPS

GPS is one of the most popular methods for localisation today, due to its ability to in real time provide the absolute or relative location data. GPS also suffers less from drift than visual odometry techniques over long distances. However, GPS tends to lack the positional accuracy for localisation and navigation especially in highly cluttered environments. This can be overcome by using Real Time Kinematic (RTK) GPS which has been shown to be capable of having a horizontal rms error of 11mm even in urban environments [20]. However GPS signal and therefore positional accuracy can still be degraded by occlusion of line of sight to satellites when navigating in an indoor environment and the limitation of the bandwidth and update rate could cause a collision [19].

2.2.1.2. IMU

Inertial Measurement Units (IMUs) measure, using accelerometers and gyroscopes, triaxial acceleration and velocity. These sensors can be used in dead reckoning to give the relative position of a mobile robot. This can only be done for short periods of time after which drift occurs, though research has been conducted into ways this can be combined with an absolute positioning method to reduce the drift and improve accuracy [21]. These sensors are best used in conjunction with other sensors.

2.2.1.3. Visual Odometry

Visual Odometry (VO) is the method of estimating the camera's motion in an environment through the analysis of a sequence of camera images. Akin to wheel odometry, which was developed before it, it suffers from drift in the motion estimates over time; Unlike wheel odometry, visual odometry has the advantage of not being affected by wheel slip from uneven terrain. VO's more accurate relative position error ranging from 0.1-2% has led to its increased popularity over the wheel odometry method [22]. Two of the most common VO motion estimation techniques utilise either stereo or monocular vision.

Monocular

Monocular VO relies upon using multiple temporally successive frames during which the camera is moving to triangulate feature points in 3D. A minimum of three different frames

are needed to first observe the features, re-observe them, triangulate the feature's points in the second frame, and calculate the transformation in the third frame. One of the biggest issues in the monocular method is a lack of certainty of scale meaning the transformation between the first and second frames is not fully known and is set at an arbitrary predefined scale. The following transformations and reconstructed points are based off this initial scale. This means the true scale of the global environment is not obtained unless information is given to the system about the first transformation or the actual scale of the surrounding 3D environment. This information can be obtained using IMUs or GPS [23].

Research was carried out by Scaramuzza and Siegwart on VO using a monocular omnidirectional camera [24]. Using the images from the camera as the only input, two different trackers are used, one is a feature-based tracker using SIFT features and RANSAC-based outlier removal for computation of the displacement in the angle of rotation, the other tracker produces high resolution estimates of the rotation angle using an appearance based approach. This system was applied to an automotive platform over 400 m and had an accumulated error of 6.5 m in distance and 5^{o} in orientation due to unavoidable drift. Proving that this method was capable of efficiently and relatively accurately functioning in a large outdoor space.

Stereo

Stereo VO methods, unlike monocular VO, can reconstruct and triangulate its 3D information in a single time step rather than having to go through multiple consecutive frames. This is due to it being able to simultaneously observe the features in both the left and right imagers which are spaced at a known distance away from each other. In the next frame these features are matched with their corresponding features and the transformation that was undergone between the two frames is estimated. Compared to monocular vision sensors, stereo vision produces more accurate and efficient triangulation, however it is more expensive and requires greater calibration [25].

Optical Flow

The optical flow technique attempts to give an estimate of the 3D velocity vector of every point in the image. This is done through analysis of the change of a 2D vector field where each vector is the displacement of the pixels from one frame to the next, caused either by the

movement of the object in the image or due to movement of the camera. This is useful for estimating the relative motion of the camera compared to its environment and has been applied to obstacle avoidance mimicking the natural behaviour of bees [26]. In a paper by Pantilie et al. optical flow has been combined with stereo image information for real-time obstacle detection in an occupancy grid framework[27]. It was able to differentiate between both static and dynamic closely packed obstacles moving in different directions and at different speeds, this improves the ability to distinguish between dynamic obstacles in the foreground and closely positioned background obstacles in the image and prevent collisions.

2.2.2. Mapping

As a robot navigates its surrounding environment it must compose a representation of them from the data it receives from its available sensors, this data model is known as a map. For the purposes of this project only a two-dimensional map is required for the rover to navigate its environment. Two of the most common methods of mapping is using metric maps and topological maps.

2.2.2.1. Metric Maps

For the metric map method, the map generated represents the geometric relationships between the different objects and obstacles all in a fixed reference frame [28]. Metric maps are ideal for the representation of metric information input from mapping sensors and are required when implementing metric path planning algorithms like the shortest path algorithm. This representation, however is usually space inefficient as the resolution does not depend on the complexity of the environment and therefore tends to be best applied for local mapping and planning rather than global planning techniques.

Feature maps

Feature maps, an example of which is shown in Figure 4, are a method of metric mapping which represents the environment as a collection of points, edges, or planes. The robot's sensors extracts geometric features from the environment and utilises this data to create a map [29]. To localise in this map type, features that have already been detected and stored are compared to features that are currently being observed.

One of the advantages of this technique is due to the low number of sparse objects stored, it tends to have a relatively low computational cost, as unlike occupancy maps free space is not represented and doesn't add any cost of storage when localising. However, this technique is sensitive to false data association, which occurs when there is a misassociation between the observed feature and the corresponding stored map feature. This would result in a decrease in the pose estimation certainty and will eventually lead to the robot becoming lost [30]. This misassociation makes this technique very sensitive to observer pose uncertainty and high density of map features. Kagami et al. [31], utilising a laser scanner built a dense 3D map for localisation and obstacle detection and extracted the planar features of the laser scanner input. It was shown that after ten runs of a 250m path the error in XY plane translation was about 15cm and the yaw angle error was 12 degrees. This proved that this technique is feasible for use as a mapping tool for localisation.

Figure 4: Example of a feature map [32]

Occupancy Grids

Occupancy grids are the most common form of mapping technique and one of the simplest to implement. Occupancy grids are arrays of cells each of which represents an area of the environment. The cells are marked either as occupied or unoccupied based upon whether a sensor detected an obstacle in that cell. Unlike feature maps, occupancy grids are not concerned with the geometric shape of any features in the environment, only the probability of whether the cell is occupied or not [32]. An example of an occupancy grid can be seen in Figure 5.

This method has the main advantage of reducing the complexity of path planning tasks, its major drawback is the computational complexity and associated time complexity for large and high-resolution environments. The challenge with occupancy grids is to use a fine enough grid resolution to capture the environmental detail and give accurate pose information whilst also making it feasible in terms of computation time, especially since path planning algorithms also become proportionally more computationally expensive with finer resolution. There are methods to have multi-resolution grids with higher resolutions at areas of greater environmental complexity such as in the paper Burgard et al. [33]. Though this method does introduce integration problems in terms of how to calculate what the optimal resolution should be.

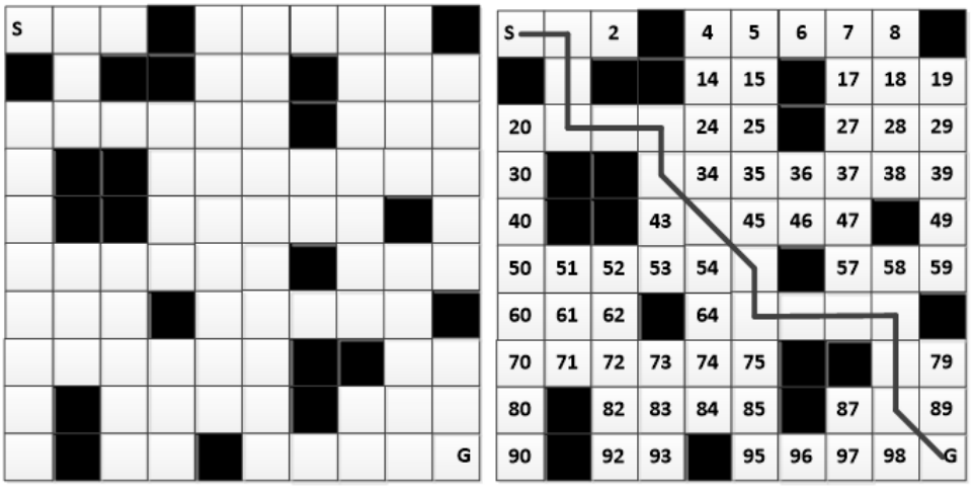

Figure 5: Occupancy grid, black cells represents occupied cells [29]

2.2.2.2. Topological Maps

Topological maps represent the environment in graphs, nodes in the graph correspond to distinct situations and landmarks, if there is a direct path between the nodes then they are connected by an arc. So, these types of graph are concerned with adjacency information between objects unlike metric maps which are concerned with the geometric relations. An example of this method can be seen in Figure 6.

This type of map representation has the benefit of its resolution being complexity dependent, which allows for efficient planning and low space complexity. It does however suffer from difficulty recognising whether two similar looking locations are the same or not and has difficulty constructing large-scale maps of the environment as it can fail to recognise

geometrically close locations especially if the sensor information is ambiguous [34]. This is one of the most critical weaknesses as if a location is not recognised due to a variation of viewpoint, change in the brightness conditions or dynamic obstacles then the topological sequence will break down and the robot will not be able to localise.

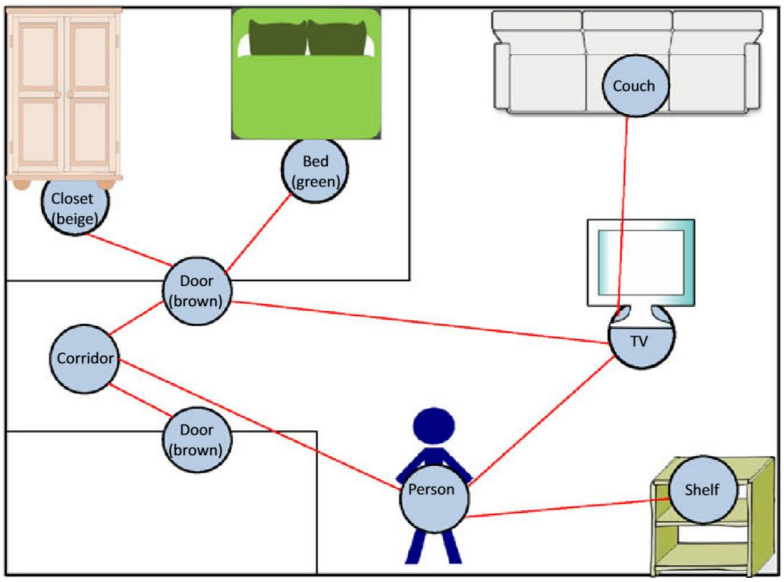

Figure 6: Simple representation of a feature map graph [32]

2.2.2.3. Hybrid Maps

Hybrid maps utilising both techniques do exist, combining the advantage of metric techniques in general superior localisation, and topological maps qualitative abstract representation of the environment into locally connected regions without the need to maintain a global reference frame. These complementary functionalities were showcased in a paper by Ravankar et al. [35], where it was proven that this method could produce maps 96.02% smaller in terms of computational storage size than the conventional GMapping package of ROS which uses an occupancy grid technique. The hybrid method was also shown to be much more temporally efficient and suitable for large scale environment mapping.

2.2.3. Simultaneous Localisation and Mapping

Previously in this literature review the problems of localisation and mapping were discussed separately, SLAM solves both problems simultaneously. SLAM techniques are generally categorised as a filtering approach and a smoothing approach.

2.2.3.1. Filtering Approaches

Extended Kalman Filter Based SLAM

Filtering approaches to SLAM are generally concerned with the online approach, estimating the current state of the robot and the map with measurements from sensors as they are input. The Extended Kalman Filter (EKF) technique is one of the most common approaches for estimating the state of the robot in SLAM. It is based upon the Bayes filter for the prediction of non-linear functions, a first order Taylor series expansion is used to obtain the linear approximations for these non-linear functions. EKF-SLAM is divided into two stages a prediction stage and an observation or update stage. The prediction stage estimates the robot's future pose based upon its current pose and the applied control input; this is usually done using dead reckoning techniques. In the update stage when an existing feature in the environment is re-observed, the feature's bearing and range to the sensor is measured, using the estimate of the current robot's pose the feature's position is estimated and the uncertainty of the state of the robot and the features are updated based upon these observations. Chatterjee et al. implemented EKF based SLAM on a mobile robot platform and proved that the method was effective as there was very small deviations, maximum of 9.5 cm, between the predicted position of the environment's features and their real-world position [36]. EKF SLAM's major advantage over other filtering techniques is its easiness to implement and computational efficiency, however the use of Taylor Series to approximate non-linear functions can result in inaccurate estimates.

Particle Filter Based SLAM

There are many particle-filter based SLAM techniques such as FastSLAM 1.0 which is based upon the Rao-Blackwellized particle filter [37]. FastSLAM has three stages: sampling an arbitrary amount of particles and their features, then once a feature is re-observed performing an EKF update of each particle to calculate its importance factor and finally re-sampling to find the posterior of the location which is the conditional probability based upon the relevant observations. FastSLAM has the advantage over EKF SLAM of overcoming the non-linearity limitations which is important when the motion of the robot is highly non-linear, and it overcomes the curse of dimensionality that other particle filter implementations suffer from [38].

2.2.3.2. Smoothing Approaches

RGB-D SLAM

RGB-D SLAM is a method of Visual SLAM which relies upon sensors giving both colour information, like a normal camera, and depth information. RGB-D sensors were popularised by the Microsoft Kinect which was based on a structure light approach to provide depth information for each pixel. The main advantages of an RGB-D sensor are its low cost and access to both visual and geometric shape information improving robustness when providing accurate visual odometry. Henry et al. proved that large consistent maps of indoor environments could be produced using this technique, though improvements did need to be made by using more than two consecutive frames when estimating the motion of the camera [39].

2.3. Multiple Robot Collaboration

As robots become more common in an increasingly wide variety of situations, it becomes possible for heterogenous robots to be able to collaborate with each other in different ways and take advantage of their respective differences. Robots can provide computational and sensing resources for each other or act as transports or transmission relays. The advantage of a UGV is its high payload capacity and energy efficiency when moving. The advantage of a UAV is its mobility due to being able to move in 3D and its higher vantage point giving its sensors a greater field of view [40].

There are two main control architectures for multi-robot systems, reactive-control and planning based control. The reactive approach uses a series of simple rules to generate complex behaviour, this has a low computational cost but can be time consuming to develop. Planning based control explicitly coordinates the robots through a plan to navigate through the environment and execute whatever tasks the robots needs to [41]. For robots to be able to collaborate with each other and share mapping data or to help in the generation of path plans, the robots must be in a common reference frame. There are numerous methods in the literature for both multiple robot relative and global localisation. Grabowski et al described methods of using omnidirectional sonar sensors to calculate the distance between pairs of robots [42]. Kato et al present a method of identifying the orientation of robots through image

processing of their hue and shape and determining their position through omnidirectional vision sensors creating potential triangles between three robots [43].

2.4. Conclusions

This section of the report has discussed the different methods of path planning, localisation and mapping, and collaboration between multiple autonomous robots. The advantages and disadvantages of the different methods have been analysed. These literature reviews combined with the author's previous literature review [4], helped to inform the key design decisions for the autonomous navigation system.

The varied hardware capable of localisation was presented in detail and it was shown that passive vision-based sensors are cheaper and can produce more information than active hardware sensors such as LIDAR. Then the specific methods capable of mapping were described with a focus on metric and topological maps and their respective advantages and disadvantages. The two approaches to SLAM, filtering and smoothing were also discussed and details were given on the function of several of the most popular implementations such as EKF and RGB-D based SLAM.

3. Development of Autonomous Rover System

This section of the report aims to detail the methods followed in the creation of the UGV system and to justify the choices made in the design process. Based upon the information gained in the literature reviews it was decided that the best method of path planning was a hybrid A* planner due to its capability to produce the optimal path and the need to adapt the path to be suitable for a vehicle with non-holonomic constraints. It was selected over other path planning methods such as APF and RRT, because those methods struggle with path planning through narrow gaps and corridors. The method of mapping selected was an occupancy grid due to its simplicity to implement and integrate with the selected path planner which is important given the tight timeframe. Also, a metric mapping technique was required to meet requirement FNC-08. Finally, it was decided that the most effective way for the UGV and UAV to collaborate, given the restrictions outlined in SYS-05 requirement, was through a planning-based control architecture. The UAV is to act first taking advantage of its superior mobility and greater vision range due to its altitude. The UAV would autonomously explore the search space until the two missing workers, discussed in functional requirement FNC-09, are found. The UAV will then land and using a global path planner will create a series of waypoints to each of the missing workers, these waypoints will be transmitted to the UGV in the form of an array which will use its local path planner to autonomously navigate between them. The localisation between the two vehicles for conversion into the same reference frame could be done with the translational distance being measured with the depth sensor and the orientation of the UAV being measure by simple fiducial markers on its body like in the work by Butzke et al [44].

3.1. Functions

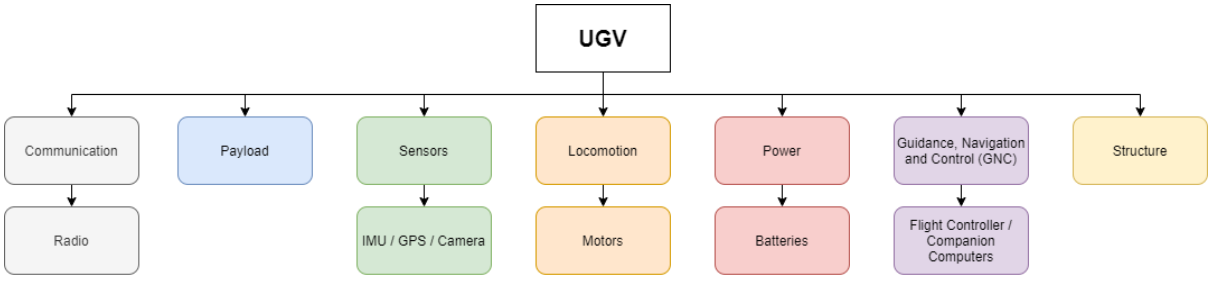

Figure 7: Diagram of the function subsystems of an UGV

Figure 7 gives a breakdown of the major functional subsystems of the UGV and gives examples of their implementations. This was created to help outline what hardware and software components were needed to create a design which would meet the system's requirements. Each of these subsystems will interact with each other either through flow of power and/or data.

The Communication subsystem involves the method of transferring data to and from the UGV. Payload is simply the method of releasing the first aid kit payload to meet the Mission requirement FNC-09. The Sensors subsystem is any hardware which inputs data to the system with the function of localising the UGV or mapping the surrounding environment. Locomotion is the method by which the UGV transports itself from place to place and the power subsystem is the method which the energy for all the other subsystems is generated, stored, and transmitted. The Guidance, Navigation and Control (GNC) Subsystem controls the autonomous movement of the UGV, this includes the Flight Controller (FC) and its companion computer necessary for path planning, localisation, mapping, and the execution of path planning commands through the locomotion subsystem. The Structure subsystem is simply the vehicle's frame.

3.2. Hardware Architecture

3.2.1. Full System

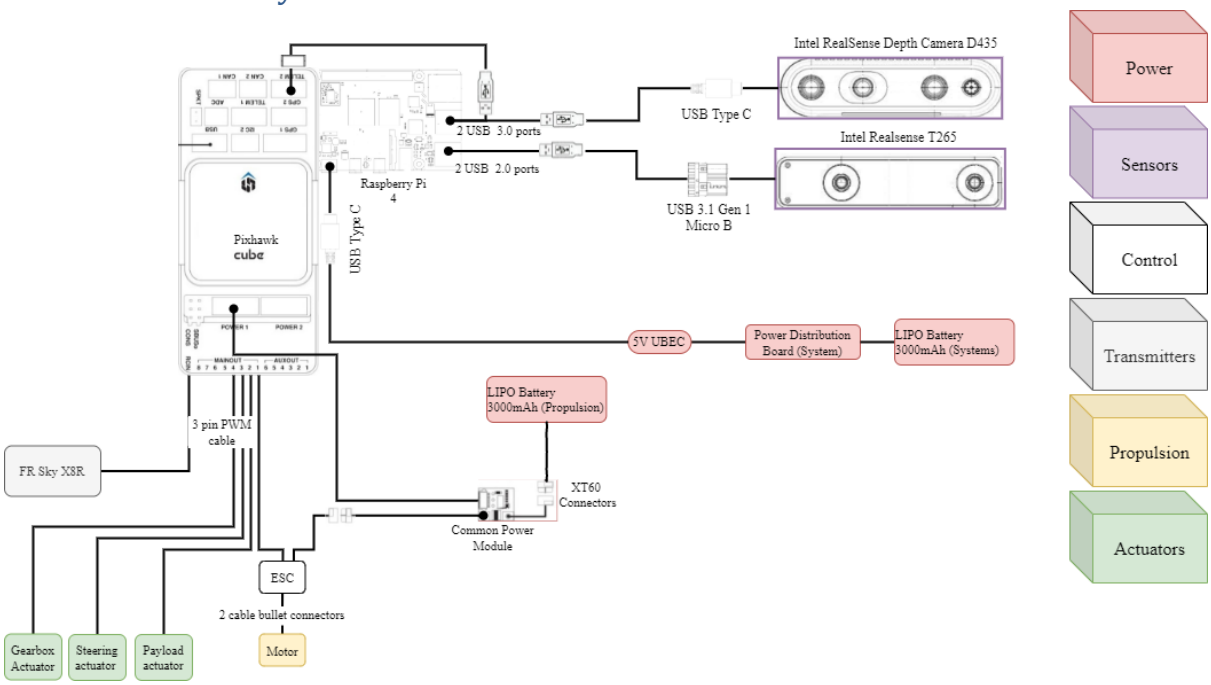

Figure 8: UGV Component Diagram

Figure 8 shows all the electrical and electronic hardware on the UGV and their connecting interfaces.

3.2.2. Structure and Locomotion

The first stage of the design process for the UGV was the selection of the vehicle frame to be used, this would determine whether the system was holonomic, meaning that the controllable degrees of freedom are equal to the total degrees of freedom of the system. It would also determine, the weight and the size of the vehicle and therefore the amount of space available for the mounting of devices to the frame.

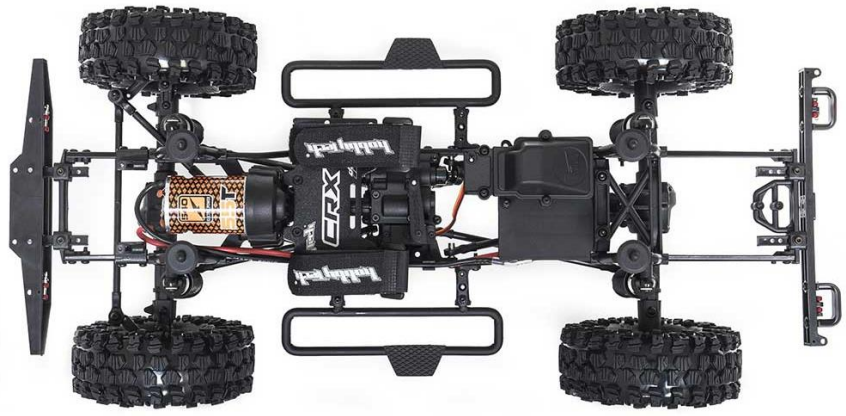

Figure 9: Selected Vehicle Frame [45]

The vehicle frame selected was a four-wheel drive with suspension over each wheel, this was important to ensure damping of any vibrations that could affect mounted sensors. The decision was made for locomotion to be with a wheeled, car-like vehicle instead of tracked or omni-directional wheeled vehicle. This was due to the good energy efficiency of wheeled vehicles, and lower cost and complexity which was important due to budget restrictions. This results in the vehicle being a non-holonomic drive system. The dimensions of the vehicle are displayed in Table 2. The size of the vehicle was minimised to ensure operability in an indoor environment in line with requirement SYS-02. A mounting area over the frame of the vehicle was designed and laser cut, to give an additional area of 300 cm^2 for mounting processors. To give a suitable mount for vision-based sensors, a camera rig was designed in CAD and 3D printed and attached to the front bumper, the design of it can be seen in the Appendix. This was sufficient to meet the space requirements for mounting any sensors and other hardware.

Table 2: Dimensions of Vehicle Frame

Dimension	Value
Length	525mm
Width (front / rear)	253mm
Height	245mm
Ground clearance	40mm
Wheelbase	324mm
Weight (without battery)	2450g
Damper Length (front / rear)	95mm
Rims	48x25mm
Tires	121x45mm
Wheel hexes	12mm
Chassis Spec	Double frame steel frame thickness 1.5mm
Transmission	2 speed gearbox
Gear Type	48dp

For the locomotion, the propulsion of the wheels is done by a built in 540 55T motor controlled by an 80A waterproof Electronic Speed Controller (ESC). The steering actuation is done by a servo integrated into the front wheel steering mechanism, allowing for accurate control of the turning radius and therefore the yaw of the UGV.

3.2.3. Processors for Guidance, Navigation and Control Subsystem

Part of the GNC subsystem is the autonomous capability to navigate, this was developed using ArduPilot Rover firmware which is an open source autopilot. This autopilot firmware was selected due to the author's familiarity with it and due to the extensive documentation and functionality it has. For the Rover firmware to function it has to be running on a supported FC board, which are listed under ArduPilot's documentation [46]. A FC board is

the main processor responsible for controlling the motors, interfacing with sensors, implementing attitude assessment and control law, and is responsible for the navigation and communication with the ground station. Some of the most popular supported FC options were compared as can be seen in Table 3. From these options the Pixhawk "Cube" 2 was selected due to its redundant sensors, high number of interface connection ports and mechanical vibration isolation for the built in IMUs.

Table 3: Comparison of different FC boards

Platform	Processor	Sensors	Interfaces	Power Consumption (W)	Dimensions (mm)	Weight (g)	Features
Pixhawk/PX4	ARM Cortex-M4F	IMU, Barometer, Magnetometer	PWM, UART, SPI, I2C, CAN, ADC, USB	~2	81.5 x 50 x 15.5	38	[47]
Pixhawk Cube	ARM Cortex-M4F	3*IMU, 2*Barometer	PWM, UART, SPI, I2C, CAN, ADC, USB	~2	95 x 45 x 35	39	Mechanical vibration isolation, already available so no purchasing cost [48]
OcPoC	"Xilinx Zynq" FPGA SoC (ARM Cortex-A9)	IMU, Barometer, GPS	PWM, I2C, CAN, Ethernet, SPI, JTAG, OTG, UART, Bluetooth, Wi-Fi	4	42(D) x 20(T)	70	[49]
Beagle Bone Blue	ARM Cortex-A8	IMU, Barometer	Bluetooth, Wi-Fi, SPI, I2C, UART, USB, PWM, ADC	1.75	89 x 54.6	-	[50]
CUAV V5 Plus	ARM Cortex M7	3*IMU, Barometer, Magnetometer	I2C, SPI, CAN, PPM, PWM, UART	-	85.5 x 42 x 33	90	Mechanical vibration isolation [51]
F4BY	ARM Cortex M4F	IMU, Barometer, Compass	I2C, SPI, CAN, USB, PPM, PWM, ADC, UART	-	50 x 50	-	[52]
OpenPilot Revolution	ARM Cortex-M4	IMU, Barometer, Compass	I2C, SPI, USB, PPM, PWM, ADC, UART	-	36 x 36	-	[53]

There is a need for a separate computer known as a companion computer to interface with the selected Cube FC. The companion computer communicates with the FC using the MAVLink

messaging protocol, and so can send and receive data from the autopilot and this can be used for intelligent decisions such as GNC subsystems obstacle avoidance capability. Some of the most popular solutions for single board companion computers were analysed and compared as shown in Table 4. From these the Raspberry Pi 4 was selected due to its high number of USB connection ports, good CPU speed, and due to the large amount of documentation and support available for it online.

Table 4: Comparison of Companion Computer Options

Platform	Processor (speed)	Interfaces	Power (W)	Dimensions (mm)	Weight (g)	Cost	Features
Arduino Nano	ATmega328 (Acceptable)	USB, PWM, GPIO	~0.17	18 x 45	7	Cheapest	[54]
Arduino Mega	ATmega2560 (Good)	USB, PWM, GPIO, SPI, I2C	~1	102 x 53	37	Cheap	[55]
Nvidia Jetson TX2	4x ARM Cortex-A57 + NVIDIA Denver2 (Outstanding)	USB 3.0 + 2.0, Ethernet, HDMI, WIFI, Bluetooth, I2C, I2S, SPI, UART, GPIO	7.5	50 x 87 x 6	85	Expensive	Optimised for AI [56]
Odroid -C4	4 x ARM Cortex-A55 (Excellent)	4x USB 3.0, 1x USB 2.0, Ethernet, HDMI, ADC, PWM, I2C, I2S, SPI, UART, GPIO	~3	85 x 56 x 1	59	Cheap	[57]
Raspberry Pi 4	4x ARM Cortex-A72 (Good)	2x USB 3.0, 2x USB 2.0, Ethernet, HDMI, WIFI, Bluetooth, I2C, I2S, SPI, UART, GPIO	3	88 x 58 x 20	46	Cheap (Already available, no purchasing cost)	[58]

3.2.4. Sensors

From section 2.2 of the literature review, a focus was given to passive vision-based localisation and mapping sensors due to the large amount of information they can provide and the need for a vision sensor per requirement FNC-06. Based on the information collated in the literature review the Intel Tracking T265 camera was selected to provide positional data for the UGV. A V-SLAM method for localisation was selected over GPS due to the large positional errors which can occur making it unsuitable for obstacle avoidance in a highly cluttered environment. Whilst an RTK GPS does have the positional accuracy necessary, the fact it requires a base station increases the complexity and cost of the design, making it

inappropriate. The T265 uses Visual Inertial Odometry SLAM, this means it uses a built in IMU in addition to the stereo fisheye lenses which it uses for feature detection [59]. Since the T265 requires no additional resources to perform the SLAM algorithms, it means that to perform tracking it is platform independent and is very easy to integrate into the rest of the UGV system and therefore there is less computational stress on the GNC companion computer.

Figure 10: Intel Depth Camera D435 [60]

In addition to the T265, a method of capturing depth data to map the surrounding environment for obstacle avoidance is necessary. To meet this need the Intel Depth Camera D435 was selected, it is displayed in Figure 10 [60]. The D435 works similarly to a Microsoft Kinect V1, it is a stereo RGB-D sensor which uses an infrared projector to create a pattern of structured light. An algorithm on the camera analyses the deformation of this pattern when it strikes a surface to calculate depth information [61]. This is used in conjunction with simply calculating depth from stereo vision by estimating disparities in matching key points in the left and right images, this concept is displayed in Figure 11. The Intel D435 was selected over other depth cameras such as the Kinect due to its higher image resolution and framerate, 1280 x 720 and 90 fps respectively [62].

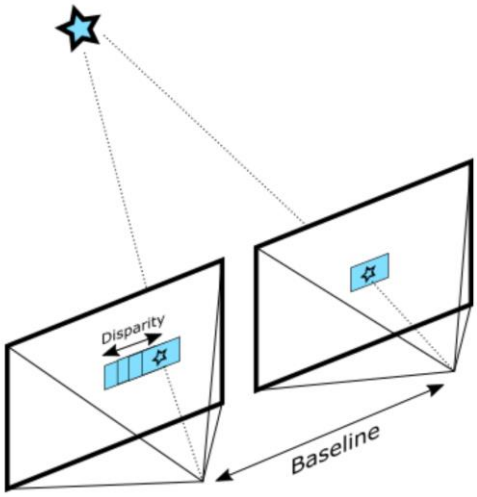

Figure 11: Stereoscopic Depth calculation, algorithm finds disparity by matching blocks in left and right image

3.2.5. Communication

In accordance with requirement FNC-06 there was a need for a wireless emergency stop capability. To accomplish this a FrSky telemetry link was used due to its low latency compared to a Mavlink radio link. The FrSky telemetry link allows the display of ArduPilot information such as flight modes, battery level, and error messages and it allows manual control of the UGV, such as arming and disarming the vehicle and changing the flight mode. One of the flight modes is set to be an emergency stop mode where the UGV immediately ceases all movement. For a FrSky telemetry link a transmitter and receiver are required, for the transmitter the Taranis X9E was selected due to its frequency hopping Advanced Continuous Channel Shifting Technology giving the connection good range and reliability on the 2.4 GHz band, and due to the transmitter already being available so there was no purchasing cost involved. For the receiver, multiple different FrSky receivers were compared as can be seen in Table 5, from this analysis the X8R was selected due to it meeting the range requirement FNC-04, and due to its availability.

Table 5: Comparison of FrSky Receivers

Receiver	Range	Combined output	Digital telemetry input	Dimensions (mm)	Weight (g)
D4R-II	1.5km	CPPM (8)	D.Port	40 x 22.5 x 6	5.8
D8R-XP	1.5km	CPPM (8)	D.Port	55 x 25 x 14	12.4
D8R-II Plus	1.5km	no	D.Port	55 x 25 x 14	12.4
X4R	1.5km	CPPM (8)	Smart Port	40 x 22.5 x 6	5.8
X4R-SB	1.5km	S.Bus (16)	Smart Port	40 x 22.5 x 6	5.8
X6R / S6R	1.5km	S.Bus (16)	Smart Port	47.42 × 23.84 × 14.7	15.4
X8R / S8R	1.5km	S.Bus (16)	Smart Port	46.25 x 26.6 x 14.2	16.6
XSR / XSR-M	1.5km	S.Bus (16) / CPPM (8)	Smart Port	26 x 19.2 x 5	3.8
RX8R	1.5km	S.Bus (16)	Smart Port	46.25 x 26.6 x 14.2	12.1
RX8R PRO	1.5km	S.Bus (16)	Smart Port	46.25 x 26.6 x 14.2	12.1
R-XSR	1.5km	S.Bus (16) / CPPM (8)	Smart Port	16 x 11 x 5.4	1.5
G-RX8	1.5km	S.Bus (16)	Smart Port + integrated vario	55.26178	5.8
R9	10km	S.Bus (16)	Smart Port	43.3 x 26.8 x 13.9	15.8
R9 slim	10km	S.Bus (16)	Smart Port	43.3 x 26.8 x 13.9	15.8

For communicating the UAV's global path plan target locations to the UGV, the UAV will send the array of locations to the GCS which will then send them to the UGV. This will be done over Wi-Fi using the Raspberry Pi's built in network adapter and the Transmission Control Protocol (TCP).

3.2.6. Power

To meet requirement SYS-06 of at least a 30min operational time, the battery size needed was calculated to be 4850mAh, given the power draw of the hardware components shown in Table 6, this is with the assumption that the motor and Raspberry Pi are always drawing maximum current, so in practice the operational time may be longer. To facilitate this operational time, two 3000mAh batteries are used in the design, giving a total operational time of 37 minutes at max power draw. One acts as a propulsion battery, connected to the ESC and motor, and the other battery acts as a system battery connected to the Raspberry Pi. The reasoning for using two batteries instead of one higher capacity battery was the redundancy it offered. The battery chemistry selected was Li-Po due to its high energy

density. A common power module was added to the design to provide steady power to the Pixhawk and to monitor the amount of power left in the battery.

Table 6: Current and Voltage draw of devices

Device	Current (mA)	Voltage (V)
Pixhawk Cube	2250	5
Raspberry Pi 3B+	3000	5
Intel Real Sense T265	300	5
Intel Real Sense D435	700	5
FrSKY X8R	100	5
Motor	3350	7.2
Total:	9700	

3.3. Autonomous Navigation System Software

The Guidance, Navigation and Control Subsystem controls the autonomous movement of the UGV. Part of this is the Rover autopilot flight code which the Ardupilot firmware implements. Part of its responsibility is the PID control algorithms which control the motor response. The other part of the GNC subsystem is the autonomous navigation subsystem which is running on the raspberry pi which interfaces with the Pixhawk and gives the autopilot commands on what the target location and yaw of the UGV should be. The autonomous navigation system is responsible for the determining the position of the UGV from the T265 Tracking Camera, and inputting depth data from D435 Depth Camera into an occupancy grid. The occupancy grid is then utilised by a hybrid A* path planner to determine the optimal route to the set waypoint whilst avoiding detected obstacles in the occupancy grid. A high-level view of the software architecture and interfacing is displayed in Figure 12.

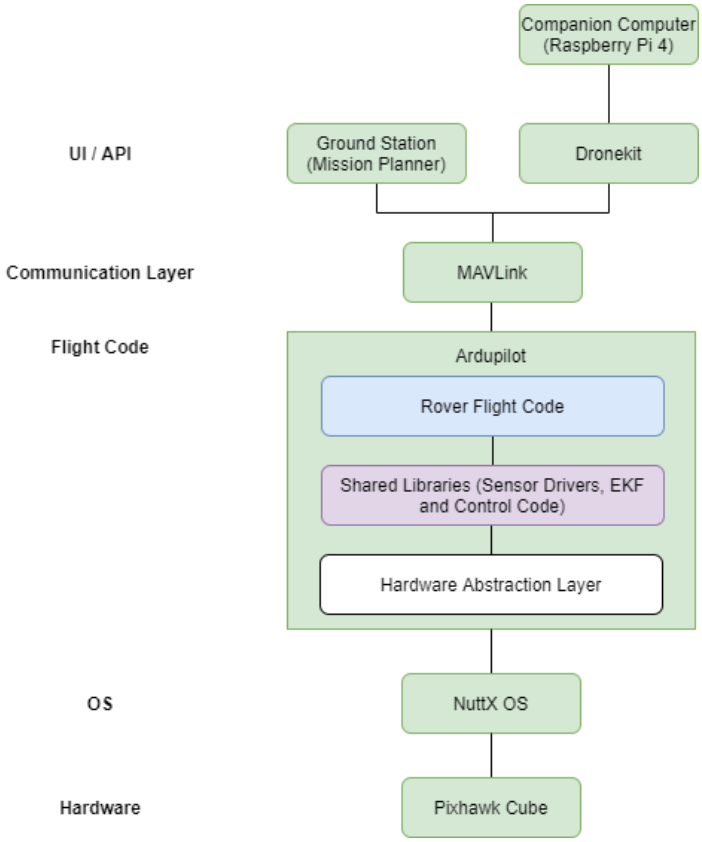

Figure 12: Software Architecture of UGV

3.3.1. Mavlink and Dronekit

Once the path plan is calculated by the companion computer it needs to send a message to the FC to move to that location. This is done using the very lightweight MAVLink messaging protocol. MAVLink is a binary telemetry protocol designed for resource-constrained systems and bandwidth-constrained links, it features methods for checking corruption, packet drops and packet authentication making it both efficient and reliable [63]. To communicate using the MAVLink protocol the Dronekit-Python API was used to simplify the interface for sending commands and accessing information between the companion computer and MAVLink. It allows programmatic access to the UGV's state and parameter information and enables direct control of the UGV's movement and operations from the companion computer. The complete Dronekit-Python API reference can be found here [64], the most important function used was commanding the target position of the autopilot in the local NED coordinate frame using custom MAVLink messages.

3.3.2. Autonomous Navigation Subsystem Structure

The Autonomous Navigation subsystem was created in Python inside the Visual Studio 2017 IDE, all the modules created are shown in the Appendix. A block diagram of the software

modules of the subsystem can be seen in Figure 13. They can be split up into their broad functions of path planning, localisation, mapping, interfacing with the Pixhawk and acting as a utility module to support the functioning of other modules.

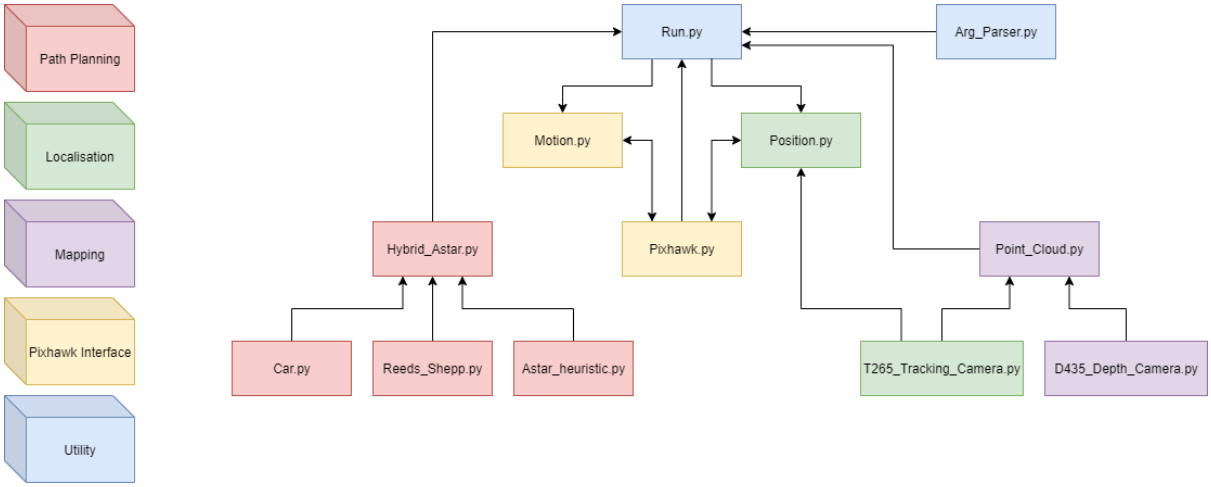

Figure 13: Block diagram of the software modules structure of the Autonomous Navigation subsystem

The Run.py file is a script run designed to be run from the command line and carry out all the functions to perform the autonomous navigation task. The Arg_Parser.py module is a utility module simply used to make a user-friendly command line interface for the Run.py module. Run.py imports the other modules which perform the tasks of localisation, mapping, path planning and interfacing with the Pixhawk. The Pixhawk.py function contained the Dronekit API functions that sends commands in MAVLink to the Pixhawk. The flowchart for how the Run.py file functions can be seen in Figure 14. It is designed to use four threads operating concurrently, this was done so that positional estimates of the UGV from the position thread would update at a greater frequency than if it had to run through the loop of the main thread each time. This was the same reason for using a separate thread for the Motion Thread, which outputs the target MAVLink path position commands to the Pixhawk. It was also the reasoning for the User Input Monitor Thread, which constantly monitors the keyboard for specific inputs which will trigger commands like setting the Pixhawk's home position. The Main Thread is responsible for updating the occupancy grid map and planning a path from the current position to the current goal waypoint which was calculated from the global path planner on the UAV.

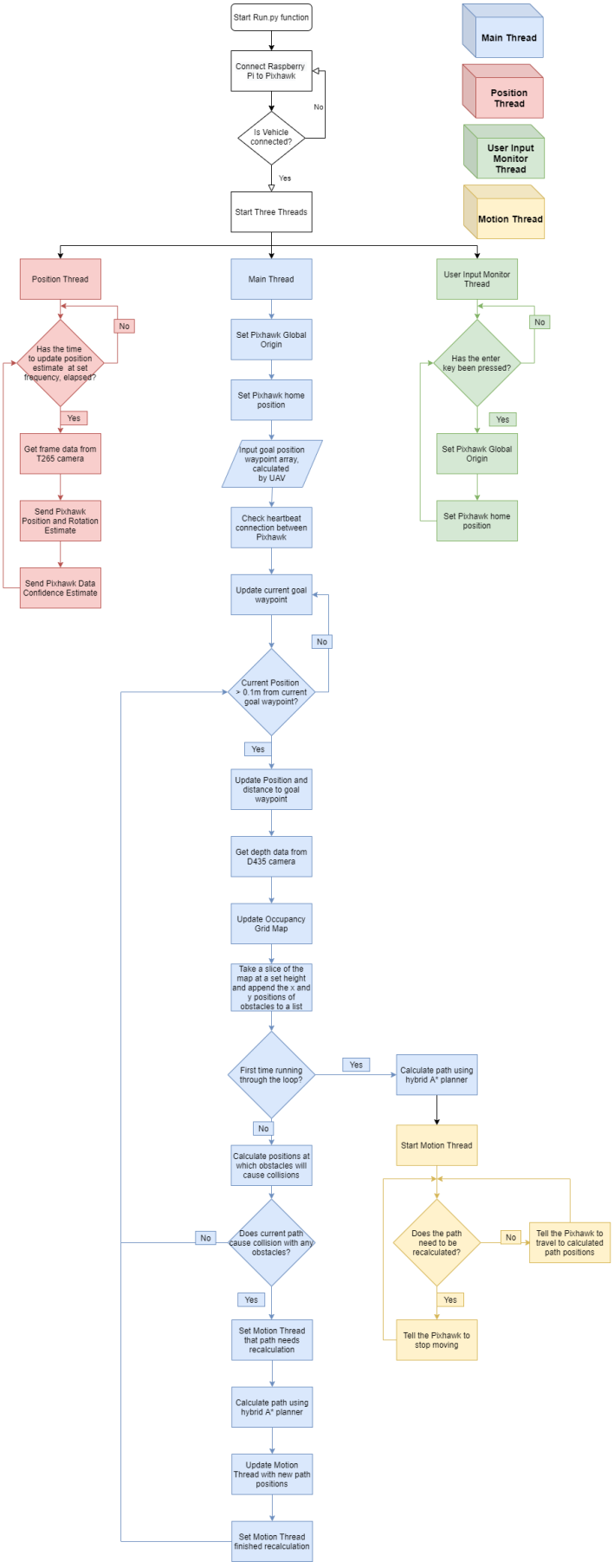

Figure 14: Run.py Flowchart

3.3.3. Localisation

The T265 and D435 cameras are supported by the librealsense API which is a library that provides the functionality for accessing data from the Intel Realsense cameras, this includes the python wrapper pyrealsense2. The data flow of the T265 camera can be seen in Figure 15.

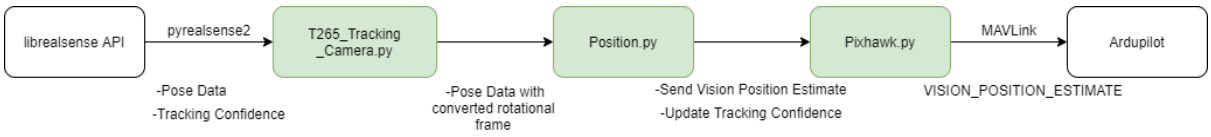

Figure 15: Data Flow of the T265 tracking camera

Once librealsense has been installed on the Raspberry Pi, the next step is to extract the pose data from the T265 cameras using the pyrealsense2 library. At a frequency of 200Hz, this 6-DOF pose data is extracted in the form of an [x, y, z] position vector and an orientation quaternion of the form [x, y, z, w]. Once the data from the camera has been extracted a frame translation needs to be performed to align the T265's internal frame with the NED frame convention used for aircraft. The T265's internal frame uses the defacto AR/VR framework standard coordinate frame, which can be seen in Figure 16. From the figure the positive X direction is towards the right imager, the positive Y direction is towards the top of the device and the positive Z direction faces into the device.

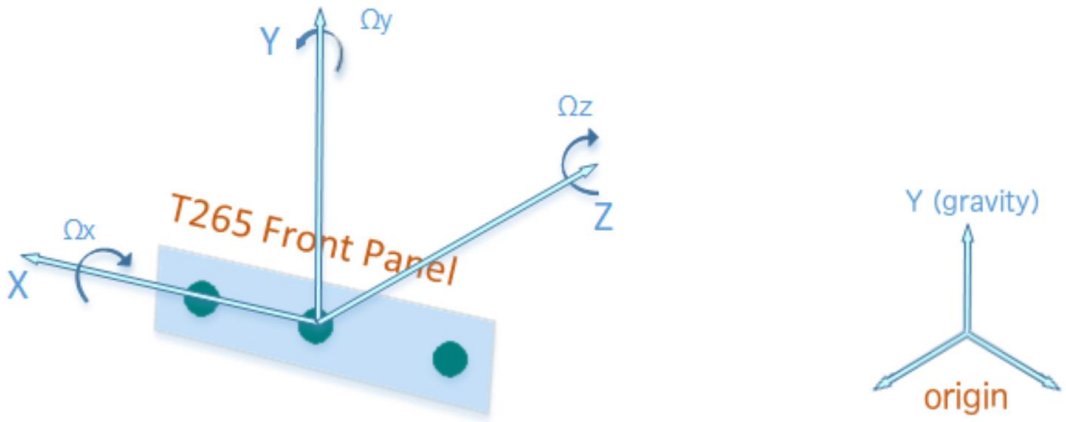

Figure 16: Internal coordinate frame of T265 [65]

To transform this internal frame to the NED frame convention homogenous transformation matrices are used. As can be seen in Figure 17, we currently have the pose data w.r.t the origin frame, coming from the T265 which is the rotation matrix 2. To obtain the rotation matrix in the correct coordinate frame, which is matrix 4, the homogenous transform matrices

1, 2, and 3 need to be multiplied by each other in order. Since the camera is forward facing the 1st and 3rd matrices are simply the inverse of each other, the rotation matrix for 1 is displayed below. The first rotation matrix is the rotation from the aircraft's inertial NED reference frame to the T265 sensor reference frame and the third rotation matrix is the rotation from the T265 body frame to the aircraft NED body frame.

$$Rotation\ Matrix\ 1 = \begin{bmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

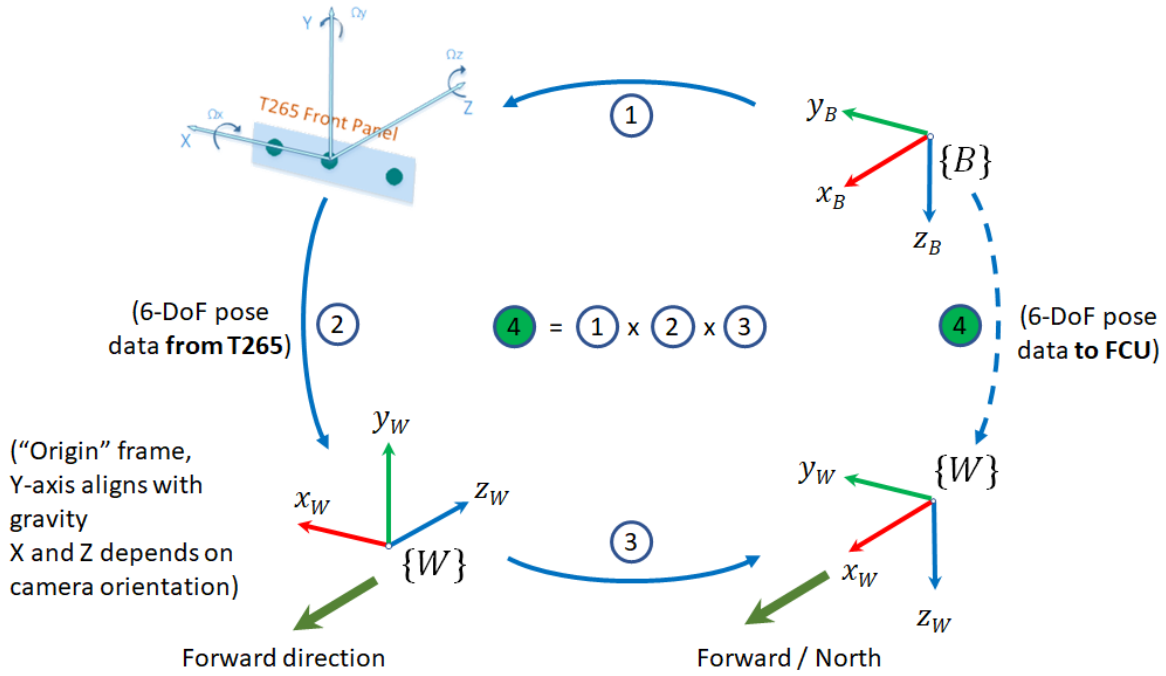

Figure 17: Homogenous transformation matrices for the T265 pose data

With the conversion of the T265 pose data to the NED frame convention Ardupilot's parameters were configured to accept position estimates from external navigation data. The parameters changed are shown below:

```
AHRS_EKF_TYPE = 2
EK2_ENABLE = 1
EK3_ENABLE = 0
EK2_GPS_TYPE = 3
EK2_POSNE_M_NSE = 0.1
EK2_VELD_M_NSE = 0.1
EK2_VELNE_M_NSE = 0.1
BRD_RTC_TYPES = 2
GPS_TYPE = 0
COMPASS_USE = 0
COMPASS_USE2 = 0
COMPASS_USE3 = 0
SERIAL5_BAUD = 921 (the serial port used to connect to Raspberry Pi)
```

```
SERIAL5_PROTOCOL = 1
SYSID_MYGCS = 1
```

The positional data is then input into the Position.py module which calls functions Dronekit API functions in the Pixhawk.py module to send MAVLink messages, updating the position estimate, to Ardupilot at a rate of 15 Hz. A frequency of 15 Hz was used because if the messages were sent at the frequency of 200 Hz which the T265 collects pose data at, it would cause the FC to freeze.

3.3.4. Mapping

The mapping functionality of the Autonomous Navigation subsystem relies on three modules, the D435_Depth_Camera.py, the T265_Tracking_Camera.py and Point_Cloud.py. Whilst all three of these modules have been edited they were originally adapted from an implementation from this source [66]. The D435_Depth_Camera.py module extracts the depth frame and RGB frame data from the D435 camera using the librealsense API and converts it from 2D screen-space coordinates into a local 3D coordinate system. The depth frame and RGB frame output a 640x480 array of depth and RGB pixels at 30 fps which can be utilised to make a pointcloud as shown in Figure 18.

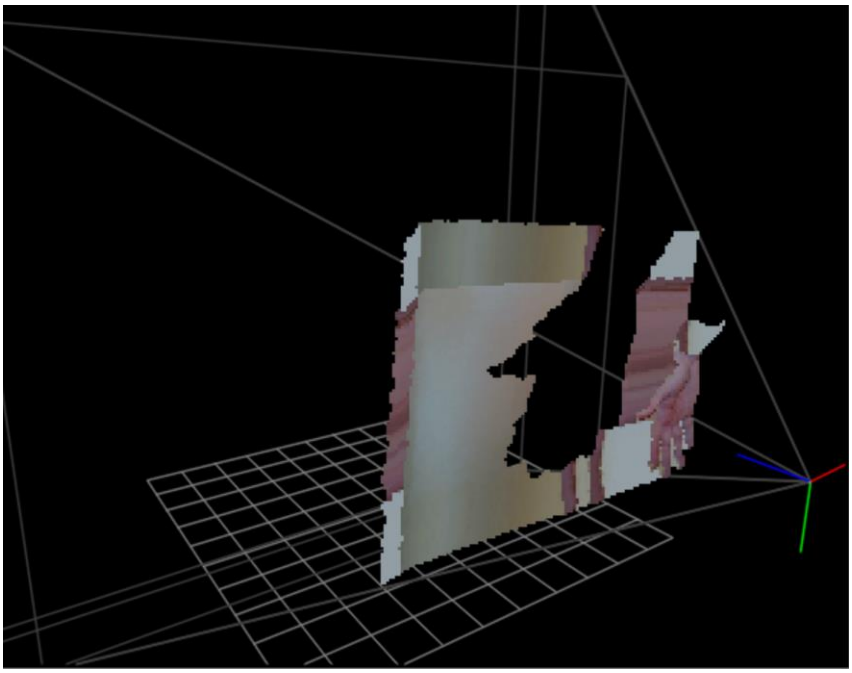

Figure 18: Point Cloud of the RGB and Depth Frames

To transform between the 2D and 3D coordinate frame the intrinsic parameters of the system need to be known, such as the principal point and focal length shown in Figure 19.

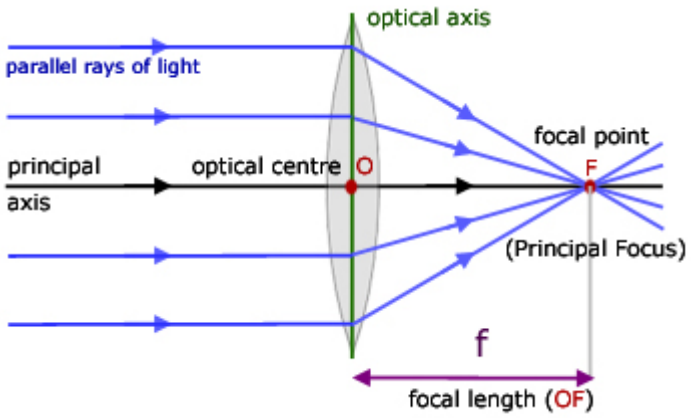

Figure 19: Concept of a lens' principal focus and focal length

The transformation simply involves creating an X and Y deprojection matrix as shown in the code below and multiply the matrices by the depth data to give 3D coordinates for each pixel. In the code width and height are the width and height of the 2D frame in pixels, ppx and ppy are the principal focus coordinates in X and Y and fx and fy are the focal length distance in X and Y.

```
# Create deproject Row/Column Vector
self.xDeprojectRow = (np.arange( self.width ) - self.intrin.ppx) / self.intrin.fx
self.yDeprojectCol = (np.arange( self.height ) - self.intrin.ppy) / self.intrin.fy

# Tile across full matrix height/width
self.xDeprojectMatrix = np.tile( self.xDeprojectRow, (self.height, 1) )
self.yDeprojectMatrix = np.tile( self.yDeprojectCol, (self.width, 1) ).transpose()
```

From the T265_Tracking_Camera.py module, the pose data in the NED frame is sent to the Point_Cloud.py module. The data flow of all the modules responsible for mapping, from the D435 camera to the Point_Cloud module can be seen in Figure 20:

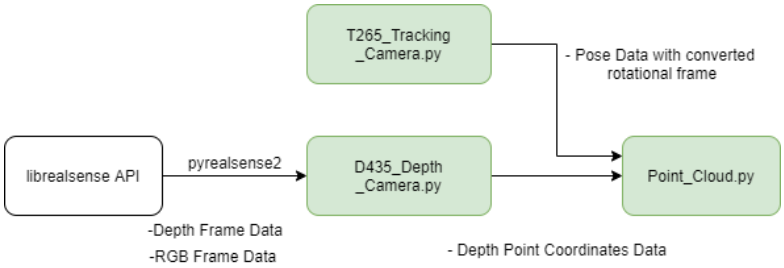

Figure 20: Data Flow of the D435 depth camera

In the Point_Cloud.py module pose data is used in conjunction with the D435 3D coordinates depth data to convert the position of the depth pixels from local coordinates to global coordinates given the origin position of the tracking camera. Once the depth points are converted to the global coordinate frame, they are sorted into voxel bins on their coordinates to create a 3D occupancy grid. This is done using the built in NumPy function, digitise. The value in each grid square is the number of points sorted in that voxel. The size and resolution of the occupancy grid can be changed to optimise it for travelling longer or shorter distances.

The occupancy grid has been designed so that if there is uncertainty about whether a voxel of the occupancy grid is occupied or not, it's value will decay by a set amount each loop until that voxel is no longer marked as occupied. This is accomplished by having a maximum weight on each of the voxels, if the number of depth points detected in that voxel reaches the max weight then it is assumed that the voxel is definitely occupied and no decay is performed. If the voxel value doesn't reach the max weight then either the value of the voxel will eventually decay to zero and the voxel will be assumed to be unoccupied, or new depth points will be detected in the voxel and added to the value until it reaches the max weight.

To convert the 3D occupancy map with voxels to a 2D occupancy map with grid squares for use by the path planner, a slice of the X and Y coordinate positions of occupied grid squares was taken at a range of heights 5 cm above and below the T265's origin height position. These values were selected due to the vehicle being likely to be able to traverse any obstacles lower than the minimum height, and the 5 cm above the origin height is the maximum height of the vehicle. Figure 21 shows an obstacle being detected and added to the 2D occupancy grid map.

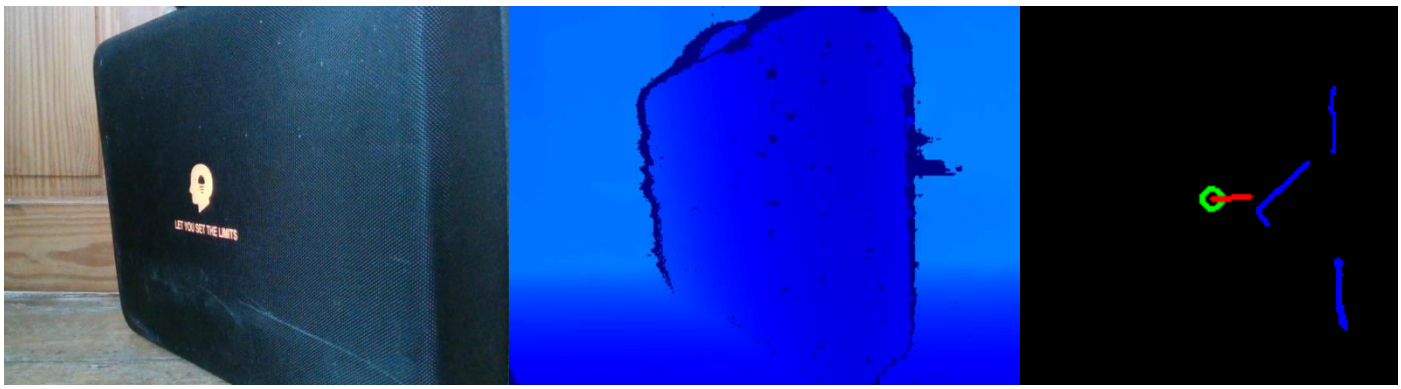

Figure 21: From Left to Right, RGB frame image, Depth frame data, Occupancy Grid with depth points plotted

3.3.5. Path Planning

The UGV needed a local path planner to navigate from its current position to the current target position created by the UAV path planner whilst avoiding obstacles. This is accomplished using the Hybrid A* path planning algorithm. The Hybrid_Astar module takes as its imports several modules including the Car.py, Astar_heuristic.py, Reeds_Shepp.py and the Point_Cloud.py for its occupancy map. The module was adapted to fit the mapping code from an implementation developed by this source [67].

As mentioned previously in the literature review, the hybrid A* algorithm can generate a near cost optimal path to the goal position, if a feasible path exists. It requires as an input, a map to act as the search space for the path, a start position, and a goal position. The start position input was always the current position calculated from the T265 Tracking Camera module and was in the form of an X and Y coordinate and the current yaw angle of the UGV. The goal position waypoint was input into the path planner from the UAV global path planner in the same format as the start position. The output of hybrid A* algorithm was a list of the states of the vehicle from the start position to the goal position, represented in a 4D discrete grid. Two of the dimensions were the X and Y positions of the grid in continuous map coordinates; a third dimension was the yaw angle of the vehicle, and the fourth was the direction the vehicle was travelling, either forwards or in reverse.

The reason a hybrid A* algorithm is used over other algorithms like the A* and Field D* algorithms is that they represent the search space in discrete steps which cannot be followed by a non-holonomic robot, as it assumes that the robot can turn in any direction on the spot. To create a path that can be followed by a non-holonomic robot the algorithm needs to consider the continuous nature of the search space and the minimum turning radius of the UGV, this concept can be seen in Figure 22. To achieve this the hybrid A* algorithm uses a set of precomputed motion primitives, based on the shape of the vehicle using the Car.py module, to calculate a continuous vehicle coordinate in each grid cell that the robot can reach. The motion primitives are calculated using Reeds Shepp curves from the Reeds_Shepp.py module. The curves are the shortest curves between two points in Euclidean space given a minimum radius.

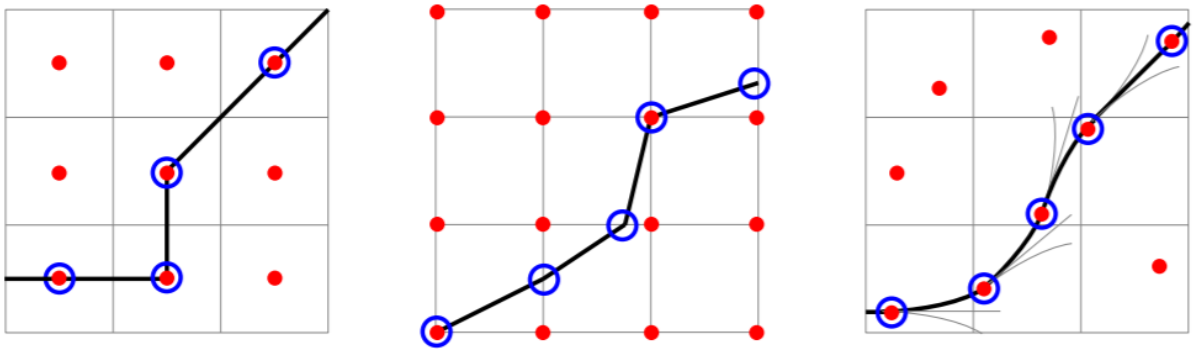

Figure 22: From left to right, comparison of A, Field D* and Hybrid A**

Figure 22 shows how the A* algorithm cannot be executed due to the sharp turns and, whilst the Field D* algorithm does produce a much smoother path it has some kinks in turns which are impossible to make for a non-holonomic vehicle. The Hybrid A*algorithm, through the consideration of continuous nature of the search space, can produce smooth executable paths for a non-holonomic vehicle.

In the A* algorithm a cost function is used to determine what the next best node in the grid to move to is. It is calculated from the cost to move from the current node to an adjacent node and the Euclidean distance from that node to the goal node. The hybrid A* differs from this in that there is a cost also assigned to reversing to encourage the car to drive forwards so that it can detect objects with its sensor. There is also a cost assigned to nearby obstacles to encourage keeping a distance from them.

In addition to these costs, the hybrid A* algorithm uses two heuristics in the calculation of the path. The first heuristic considers the non-holonomic nature of the vehicle but does not consider any obstacles when calculating paths and is useful for reaching the target position at the correct yaw angle. The second heuristic is calculated using the Astar_heuristic.py module and it works similarly to a standard A* algorithm, as it considers whether there are any obstacles in the nodes but also considers the vehicle as holonomic. Its first step is to generate an obstacle map from sensed obstacles input from the Point_Cloud.py module. Based on a virtual radius of the UGV, adjacent nodes to the node where the obstacle is detected will also be set as occupied if they fall within the virtual radius distance. This is done to make certain nodes impassable as a collision would occur if the vehicle were to move into one of these nodes. Then using unoccupied nodes, the heuristic calculates the shortest path of nodes to the goal position.

Once the initial path plan to the goal position is generated using the initially sensed obstacles the Motion.py module outputs the path plan to the Pixhawk.py module at a set frequency, which in turn sends MAVLink commands to the Pixhawk autopilot. As the UGV moves to these positions it is likely it will sense more obstacles than it initially sensed. So, the Run.py function is continuously updating the obstacle map with new updates to the occupancy map. If an obstacle is detected that will cause a collision along the current path, the path planner recalculates the path with the updated obstacle map and the UGV will start following this new path. An example of a path plan that avoids detected obstacles in the occupancy map is shown in Figure 23.

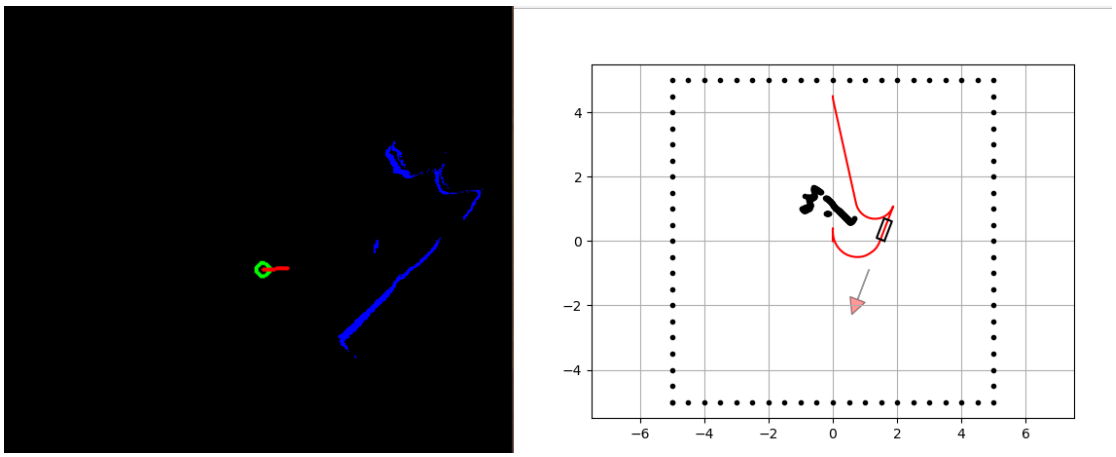

Figure 23: Hybrid A path planner obstacle avoidance capabilities*

4. Autonomous Navigation Subsystem Testing

4.1. Testing the Mapping Functionality

In Figure 24 the accuracy of the mapping functionality is demonstrated, proving that both the positional and depth data were working in conjunction with each other. As can be seen a 10cm thick briefcase is placed against the wall in frame 1, and the 2D occupancy map is clearly able to distinguish that the briefcase from the wall. In Frame 2 however, when the briefcase is removed from the frame, whilst the path planner can map the rest of the wall, it still treats the removed briefcase as an obstacle. This is due to when the briefcase was sensed, the voxels that represent the briefcase reached the max voxel weight value and so will not decay over time. Whilst this method is useful for reducing noise and ensuring that static obstacles that have been sensed will remain on the occupancy grid, it does mean that the mapping function is incapable of dealing with dynamic obstacles as it will treat them as if they are in every position that it sensed the obstacle at.

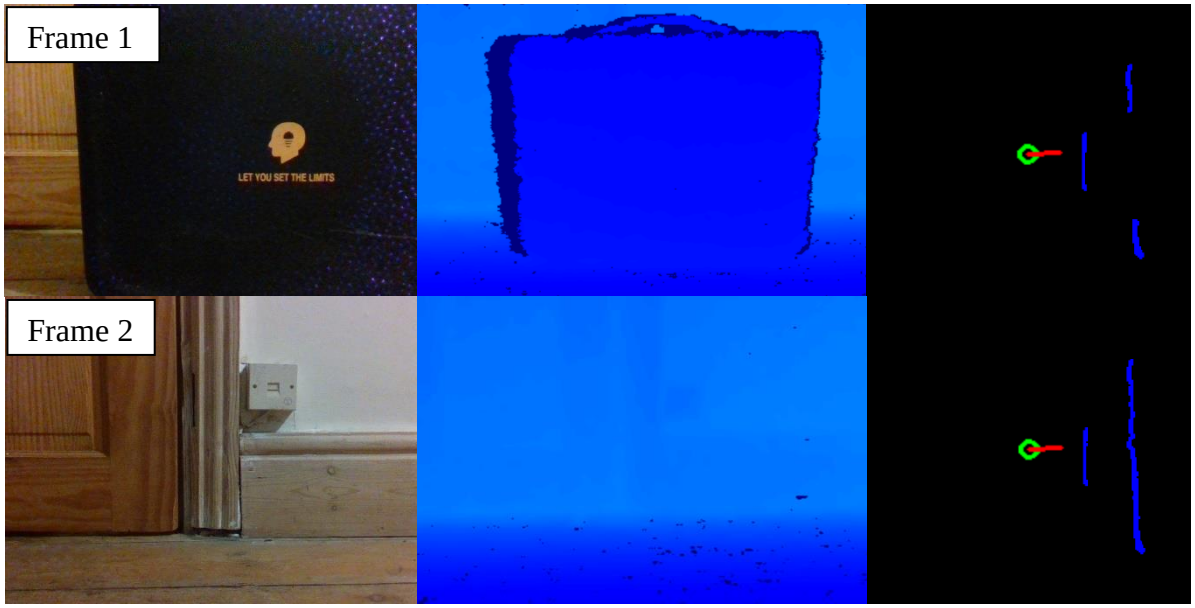

Figure 24: Demonstration of Point_Cloud.py mapping functionality

4.2. Testing Path Planner Functionality

4.2.1. Comparison to other path planers

Figure 25 compares an example of a path being planned by the Hybrid A* algorithm and a path being planned by the standard A* algorithm. Whilst the path planned by the A* algorithm is the shortest route given the resolution of the state space, it is also made for a holonomic vehicle as it requires 90 degree turns instantaneously at points of the path. The A* algorithm also doesn't take into account the shape of the vehicle as the path calculated recommends at points following nodes closest to the wall of obstacles, but given the geometry of the car if it was in any of those nodes it would cause a collision with the obstacles. The path calculated by the Hybrid A* is much smoother and avoids getting close to obstacles, so will be more energy efficient, quicker, and less likely to cause a collision.

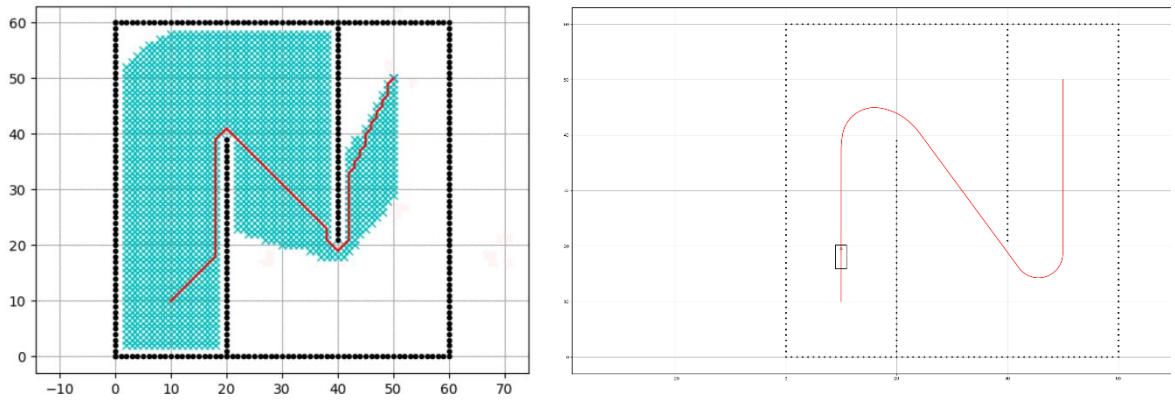

Figure 25: Left to Right, Path plan of A algorithm, Path plan of hybrid A* algorithm*

Figure 26 shows a simulation of the path calculated by the UGV, through a relatively small gap, such as a doorway. This shows the path planner can overcome a common weakness of path planners which is planning a route through narrow passages or gaps in the search space [68]. For example, the artificial potential field path planner would likely experience problems as the potential near the narrow passage would be high. If there is a local minima to the potential field near the entrance of the narrow passage then a potential field path planner is unlikely to be able to escape the local minima, due to the high potential surrounding it in the narrow passage.

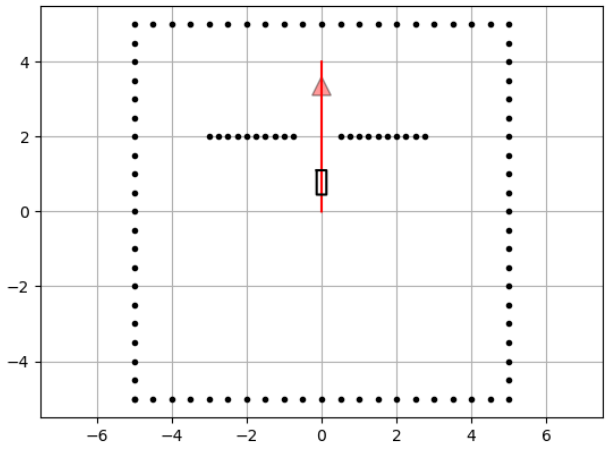

Figure 26: Path planning through narrow gap

4.2.2. Simulation Testing

The resolution of the grid cells of the search space and the resolution of the motion of the vehicle which is used by the motion primitives to generate a continuous path to each discretised grid cell of the search space can be modified. The path in Figure 27, was used to time the path planner function with changes made to the grid resolution and motion resolution. The results are displayed in Table 7. From the results it can be clearly seen that an increase in the motion resolution can have some benefits in the speed of computation.

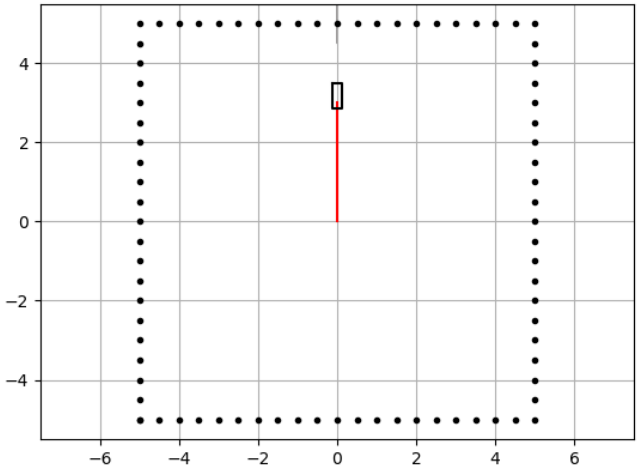

Figure 27: Path Planner resolution timing test

At a motion resolution of 0.1 m and XY grid Resolution of 0.25 m the time for the function to execute was 0.6514 seconds, and with a motion resolution of 0.2 m and the same grid resolution the time to execute was 0.3612 seconds. At motion resolutions greater than 0.2 m however, the time to execute did not decrease from 0.3612 seconds meaning there are

diminishing returns to increasing the motion resolution. Increasing the XY grid resolution yielded greater decreases in the execution time of the program than similar increases to the motion resolution. However, this too saw diminishing returns after an XY grid resolution of 0.5m, as the execution time was 0.2802 seconds at that resolution and 0.3548 seconds at a resolution of 1m. This is probably due to the best possible time for the path planner to go through all the functions it uses is around 0.3 seconds. Based on these tests the XY grid resolution was set to 0.25m and the motion resolution 0.1m, this was done because the path planner was still able to calculate a path in an efficient online manner in under a second and if the discretisation of the search space is made too rough the path planner can fail to find a path as whilst the path planner will yield realisable continuous paths, it is not complete.

Table 7: Execution times of path planning function

Time for Function to Plan Path (s)	XY Grid Resolution (m)	Motion Resolution (m)
0.9651	0.125	0.1
0.6514	0.25	0.1
0.2802	0.5	0.1
0.3548	1.0	0.1
0.3612	0.25	0.2
0.4271	0.25	0.4
0.3631	0.25	1.0

In Figure 28 the ability of the path planner to create a new path plan when new obstacles are detected which would cause a collision with the original path is demonstrated. In this figure the mapped obstacles are all sensed by the D435 depth sensor and a path plotted to avoid the obstacles. As the UGV moves along the original path, Frame 1, it can sense more of the obstacles in front of it and realise that its original route will cause a collision. So, it generates a new path plan, Frame 2, considering the newly sensed obstacles. It should be noted that whilst it did successfully generate an obstacle avoiding path based on obstacles it detected, it could not sense obstacles behind it that would've blocked its planned path due to not having a sensor at the vehicle's rear.

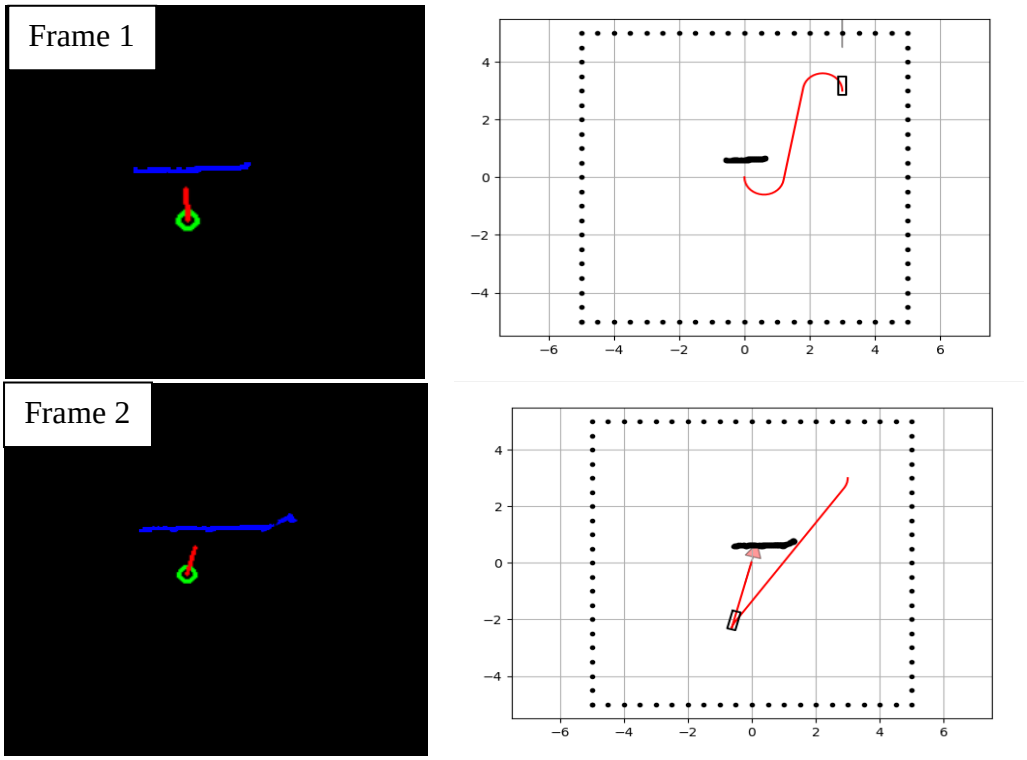

Figure 28: Demonstration of path recalculation

4.2.3. Further Simulation

Figure 29 shows a sub-optimality with the current hybrid A* implementation. When travelling with multiple waypoints the path planner currently only considers the target yaw of the current target goal point and not the location of the waypoint position after that. This results in Figure 29 (right) where to reach the next waypoint the UGV must reverse, whereas if it were already facing towards the waypoint it could drive straight. To overcome this limitation the path planner must plan the yaw angle it should reach its current waypoint at, not only based upon the locations of the current target waypoint, but the next two waypoints.

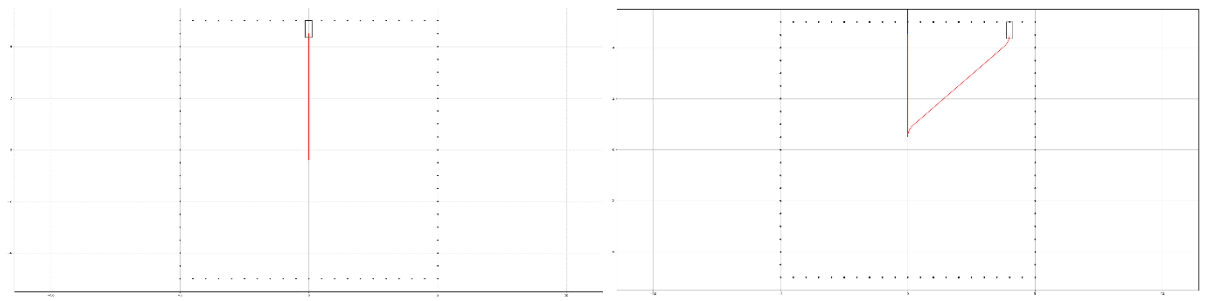

Figure 29: Limitation of planning to just one waypoint. (Left): Path to first goal position. (Right): Path to second goal position.

During simulation testing of the path planner, a simple maze was created to test the path planner's ability to navigate in environments with a high density of obstacles, as can be seen in Figure 30. The start position was set at [1,1] with the yaw angle set facing up and the goal position was set at [9,9] also with the yaw angle facing up. The path planner failed to find a path between the two points but when it was given an intermediate point at [8,3] facing to the right it was able to calculate a path to the intermediate point. Then using the intermediate point as its start position the path planner was able to calculate the path to the initial [9,9] goal position, proving that the path is possible. This could be happening because without using an intermediate point all the paths calculated by the heuristic that considers the vehicle non-holonomic but ignores obstacles are causing collisions. It could also be because the resolution of the XY grid and motion was too coarse.

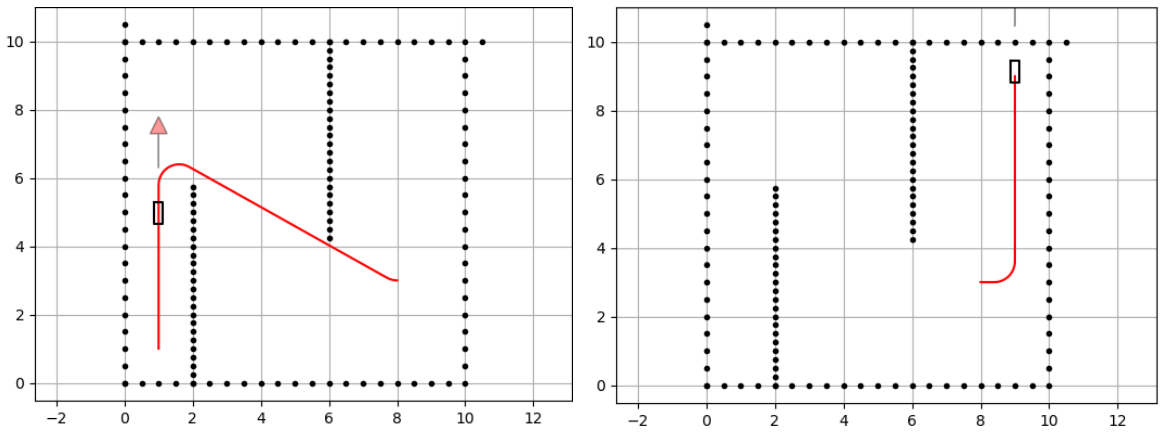

Figure 30: Failure to calculate path

A real-world test of the UGV's ability to autonomously navigate a complex environment with static obstacles is shown in Figure 31. The path planner was tasked with generating a path from the current position of the vehicle to a position 5.385m away in the next room and was successful in both mapping the surrounding environment, localising itself in the environment and path planning from its position to its target position whilst avoiding obstacles.

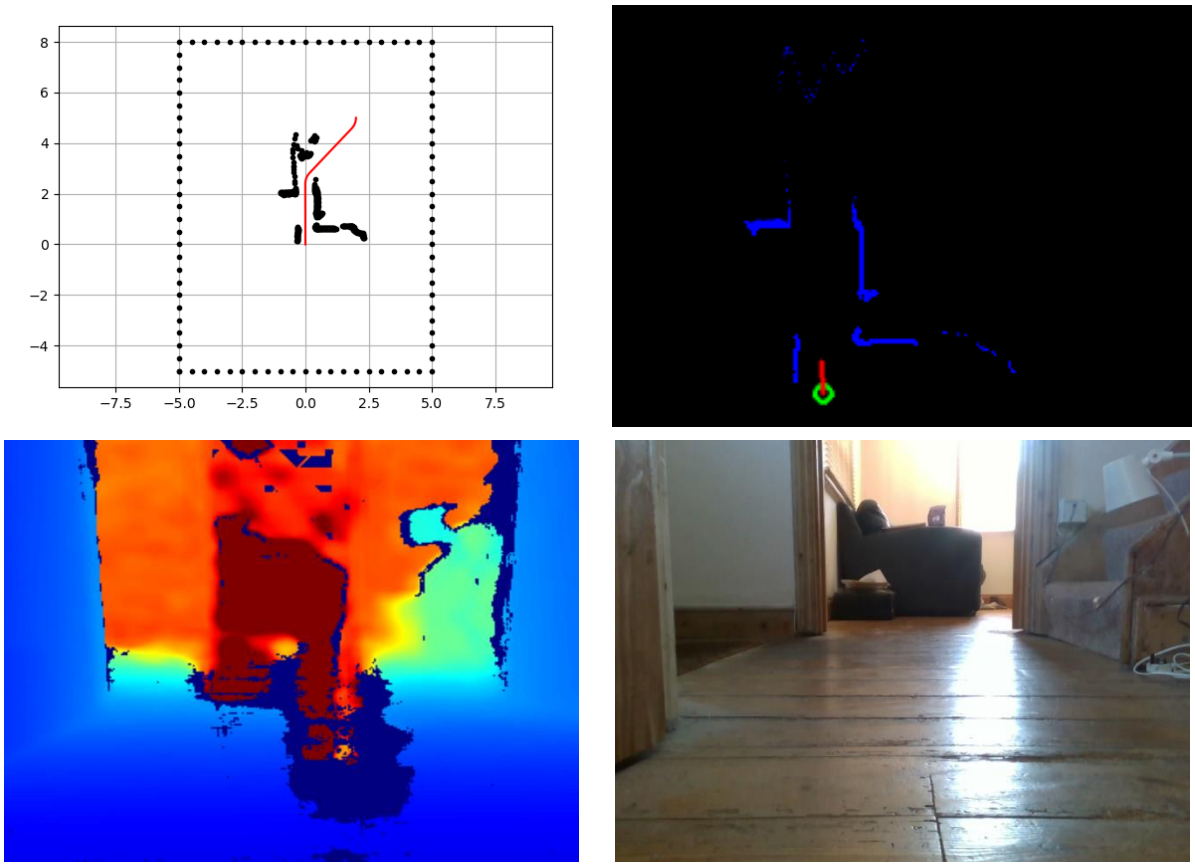

Figure 31: Real world test of autonomous navigation subsystem (Top left): Path Plan with mapped obstacles. (Top right): Occupancy Grid. (Bottom left): Depth frame data. (Bottom right): RGB image

5. Discussion

5.1. Achievement of Project Aims and Objectives

Overall, the project aims of designing and building an autonomous UGV to collaborate with a UAV for the ERL competition was successful. The design of the UGV was broadly split between the design, selection of components and assembly of the hardware architecture, and the design and development of the algorithms for the autonomous navigation subsystem which is a part of the Guidance, Navigation and Control system.

5.1.1. Hardware Architecture

Firstly, the hardware components needed for the system were identified based upon the main functions the UGV needed to carry out, which in turn were identified from the requirements stated in section 1.2. Seconly, the specific hardware components such as processors, radios and sensors were selected. This was done based on information gathered in the literature review for the best methods for mapping and localisation and with multi criteria decision analysis tables. The chassis for the vehicle helped to inform what path planner was used and a camera rig was created to secure the sensors to.

During testing there was very little noise in the sensor data caused by movement of the vehicle. This was due to the mechanically robust design of the camera rig securing stably the sensors to the front bumper, and the selected vehicle frames suspension acting as damping for oscillations. It was noted that during some simulation tests the UGV would plan routes involving reversing manoeuvres despite their being obstacles behind it because it could not sense them. To rectify this, a sensor such as another D435 camera needs to be placed on the rear of the vehicle so that the UGV can more fully map its environment and generate the best path.

5.1.2. Autonomous Navigation Subsystem

The software developed for the autonomous navigation subsystem was very successful as shown in the simulation tests. It was fully capable of localising itself in a real-world complex environment, mapping the environment’s obstacles and planning a path for a non-holonomic vehicle. This meets the FNC-09 functional requirement of being able to autonomously navigate to several waypoints whilst avoiding obstacles.

The resolutions of the XY grid and UGV motion were optimised so navigation could occur in a real-time manner, without sacrificing the fineness of resolution which could result in the path planner failing to calculate a route. The Run.py script was designed to use multiple threads, for updating the position estimate, sending motion commands to the Pixhawk, monitoring user inputs as well as running the main thread responsible for detecting obstacles and planning a path. The use of threading means operations in the modules are done in parallel, allowing for quicker updates of variables like position, this improves the general speed of the code and aids the subsystem in functioning in an online, real-time manner.

5.2. Challenges

There are still some challenges present with the currently implemented design of the system. As was mentioned in section 4.1 when testing the mapping functionality, the produced obstacle map is unable to function with dynamic obstacles. If a voxel of the occupancy grid reaches the maximum weight, then the value of the voxel will not decay over time, this is done to prevent previously sensed obstacles from decaying over time when the depth sensor is not facing them. This can result in a "ghost" of the obstacle in the map which the path planner will treat as impassable. To rectify this, an update rule should be used so that only grid cells in current view of the sensor are cleared and updated. This can be done using Bayesian filter for occupancy grid mapping with the inverse sensor model [69].

Another issue identified with the mapping function is that it simply utilises depth information to determine the grid's occupancy, and each grid square's state is considered binary, occupied, or not occupied. Whilst this is the easiest method to implement, it does not consider grid cells that might be occupied but are still passable, for example if the grid is filled with long grass. This results in the path planner treating certain grid sections of the map as impassable when they are passable. To solve this the RGB data of the sensor could be used to identify the hue and texture of the obstacle using statistical measures [70]. This could then be used to give an occupancy value to the grid square somewhere between 1 and 0 and a cost applied to the path planner using that node dependent on its occupancy value, allowing grid squares that are occupied but still passable to be moved through. A hue segmentation technique could also be used to detect the ground plane, as currently anything above a certain

height is treated as an obstacle, so a relatively steep slope would result in a false positive obstacle.

A limitation of the current implementation of the path planner was discussed in section 4.2.3. There is not currently a method of considering the yaw angle of not just the current target position, but also the waypoint after that. Whilst this means that from waypoint to waypoint the optimal path is usually being generated given the start parameters, over the entire path of all the waypoints it is a suboptimal path. This can be solved by calculating the goal yaw angle of the target waypoint as not only the direct yaw from the current position to the target position but also considering what the yaw between the current target waypoint is and the waypoint after that.

From simulation testing in section 4.2.3 it was discovered that sometimes the path planner will fail to find a possible route in a large complex environment even if a route is possible. To resolve this, it is recommended that when the UAV's global path planner generates target waypoint positions in areas with a high density of obstacles, the distance between the waypoint positions is under 10 metres. However, despite these limitations, the UGV could navigate a complex real-world environment and avoiding obstacles.

6. Conclusions

The primary aim of the project was to design and build a UGV capable of collaborating with a UAV and develop an autonomous guidance navigation and control system for it. This would allow autonomous navigation in a complex unstructured environment to waypoints calculated by a global path planner. The performance of these autonomous navigation capabilities was then to be analysed, through simulation and real-world tests.

Hardware components necessary to achieve the required functions of the UGV were selected based upon recommendations from the literature and analysis of multiple criteria such as power consumption, functionality, interfaces, and cost. Each hardware component was analysed to ensure it was capable of interfacing with the other components and could achieve its functional role in the system.

To meet the functional need of localisation and mapping for the autonomous navigation subsystem the Intel T265 and D435 cameras were selected as sensors. The T265 was selected because it performs all the SLAM algorithms on the device and so is platform independent and does not add any computational stress to the UGV's processors. The D435 was selected due to its high image resolution and framerate.

To create the mapping and localsation functionality of the autonomous navigation subsystem, the data from the two camera sensors was used as inputs. The pose of the UGV was determined from the T265 Tracking Camera, and depth data from D435 Depth Camera was converted from a 2D depth frame to a 3D global coordinate frame and sorted into voxels in a 3D occupancy grid. A value was added to each voxel based upon the number of points sorted into it and if a maximum weight was reached it was assumed that the voxel was occupied and so its value should not decay over time. Whilst this method of mapping was easy to implement, it did create issues such as not being able to distinguish when obstacles had been removed. A potential fix to this issue is the use of Bayesian filter and inverse sensor model to use as an update rule to the occupancy map.

The occupancy grid was then utilised by a hybrid A* path planner to determine the optimal route to the set waypoint whilst avoiding detected obstacles in the occupancy grid. The hybrid A* planner was compared to a traditional A* path planner and shown to be able to

produce more efficient routes, and routes that were possible to follow for a non-holonomic vehicle. The resolutions used by the path planner were optimised for real-time calculation and simulations were carried out. It was shown that the current implementation of the path planner could be improved by considering the next two goal waypoints to produce a more optimal overall path instead of just the next waypoint.

The calculated path was then sent to the FC using MAVLink commands created the Dronekit API. These would then command the Pixhawk to move to the path position and as it was moving it was capable of detecting new obstacles and if they would cause a collision with its current path it would calculate a new path.

Overall, the UGV and its autonomous guidance, navigation and control system are deemed a success due to being able to operate in a real-world scenario. Though, further work could be implemented in the future to further improve the system.

7. Future work

As mentioned previously some work is needed to produce a fully realised system for the ERL competition and meet all the functional requirements specified. This includes integration of object recognition software for recognising the mannequins to meet requirement FNC-09, and the design and integration of a payload delivery system. It also includes the integration of the UAV's global path planner waypoints, which will need to be converted into the UGV's reference frame. As previously mentioned, this requires a technique of cooperative localisation to be implemented, to transform the UGV's and UAV's reference frames to be the same reference frame. An option for this is to use GPS so that the global positions of both the UAV and UGV are known, however Ardupilot does not support both position estimates from the Intel T265 and GPS at the same time so a method of integration would have to be devised. With these features implemented, the result would be a UGV and UAV collaborative team capable of meeting all the functional requirements specified.

To further improve the reliability and functionality of mapping on the UGV, another depth sensor could be mounted to the rear of the vehicle and Bayesian occupancy grid mapping via the inverse sensor model could be used. These changes should prevent the mapping issue of moved obstacles still being detected in their original position and of generating path plan routes to reverse despite there being obstacles behind the vehicle. Another improvement to reliability of the developed system would be for the Hybrid A* path planner to implement the planning via one target waypoint position to the next waypoint position, to provide more optimal paths.

To further improve the capability of the system beyond the requirements of the ERL competition, it could be made capable of navigating autonomously in a dynamic rather than static environment.

8. References

[1] (Research and Markets), "Global Autonomous Mobile Robot Market Expected to Grow in Value to $58.9 Billion During the Forecast Period, 2020-2026," *Business Wire*, 2020. [Online]. Available: https://www.businesswire.com/news/home/20200122005673/en/Global-Autonomous-Mobile-Robot-Market-Expected-Grow. [Accessed: 13-May-2020].

[2] "AUTONOMOUS MOBILE ROBOT MARKET - GROWTH, TRENDS, AND FORECAST (2020 - 2025)," *Mordor Intelligence*, 2020. [Online]. Available: https://www.mordorintelligence.com/industry-reports/autonomous-mobile-robot-market. [Accessed: 13-May-2020].

[3] SciRoc, "ERL Emergency Service Robots 2019 Rulebook," 2019.

[4] F. Hewson, "Project Plan and Background Review," Bath, 2020.

[5] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," *Robot. Sci. Syst.*, vol. 6, pp. 267–274, 2011.

[6] L. E. Kavraki, M. N. Kolountzakis, and J. C. Latombe, "Analysis of probabilistic roadmaps for path planning," *IEEE Trans. Robot. Autom.*, vol. 14, no. 1, pp. 166–171, 1998.

[7] J. J. Kuffner and S. M. La Valle, "RRT-connect: an efficient approach to single-query path planning," *Proc. - IEEE Int. Conf. Robot. Autom.*, vol. 2, no. April, pp. 995–1001, 2000.

[8] J. Reeds and L. Shepp, "Optimal Paths for a Car That Goes Both Forwards and Backwards," *Pacific J. Math.*, vol. 145, no. 2, pp. 367–393, 1990.

[9] A. Bicchi, "Planning Shortest Bounded-Curvature Paths for a Class of Nonholonomic Vehicles among Obstacles," pp. 1349–1354, 1995.

[10] A. Koubaa *et al.*, *Robot Path Planning and Cooperation: Foundations, Algorithms and Experimentations*. 2018.

[11] H. Ortega-Arranz, D. R. Llanos, and A. Gonzalez-Escribano, *The Shortest-Path Problem: Analysis and Comparison of Methods*, vol. 1, no. 1. 2014.

[12] D. Ferguson and A. Stentz, "The Field D* algorithm for improved path planning and replanning in uniform and non-uniform cost environments," *Robot. Inst.*, 2005.

[13] J. Petereit, T. Emter, C. W. Frey, T. Kopfstedt, and A. Beutel, "Application of Hybrid A* to an Autonomous Mobile Robot for Path Planning in Unstructured Outdoor Environments," *Robot. 2012; 7th Ger. Conf. Robot.*, no. 1, pp. 227–232, 2012.

[14] M. Montemerlo *et al.*, “Junior: The stanford entry in the urban challenge,” *Springer Tracts Adv. Robot.*, vol. 56, no. October 2005, pp. 91–123, 2009.

[15] J. Fuentes-Pacheco, J. Ruiz-Ascencio, and J. M. Rendón-Mancha, “Visual simultaneous localization and mapping: a survey,” *Artif. Intell. Rev.*, vol. 43, no. 1, pp. 55–81, 2012.

[16] G. Benet, F. Blanes, J. E. Simo, and P. Perez, “Using infrared sensors for distance measurement in mobile robots,” *Rob. Auton. Syst.*, vol. 40, pp. 255–266, 2002.

[17] N. Harper and P. McKerrow, “Recognising plants with ultrasonic sensing for mobile robot navigation,” *1999 3rd Eur. Work. Adv. Mob. Robot. Eurobot 1999 - Proc.*, vol. 34, pp. 105–112, 1999.

[18] G. Brooker, *Introduction to Sensors for Ranging and Imaging*. SciTech Publishing, Inc., 2009.

[19] A. Chatterjee, A. Rakshit, and N. N. Singh, *Vision Based Autonomous Robot Navigation*. Springer, 2013.

[20] I. M. Elhassan, “Testing RTK GPS Horizontal Positioning Accuracy within an Urban Area,” *Rrjet*, vol. 6, no. 3, pp. 17–24, 2017.

[21] W. Seo and K. R. Baek, “Indoor Dead Reckoning Localization Using Ultrasonic Anemometer with IMU,” *J. Sensors*, vol. 2017, 2017.

[22] D. Scaramuzza and F. Fraundorfer, “Tutorial: Visual odometry,” *IEEE Robot. Autom. Mag.*, vol. 18, no. 4, pp. 80–92, 2011.

[23] G. Nützi, S. Weiss, D. Scaramuzza, and R. Siegwart, “Fusion of IMU and vision for absolute scale estimation in monocular SLAM,” *J. Intell. Robot. Syst. Theory Appl.*, vol. 61, no. 1–4, pp. 287–299, 2011.

[24] D. Scaramuzza and R. Siegwart, “Appearance-guided monocular omnidirectional visual odometry for outdoor ground vehicles,” *IEEE Trans. Robot.*, vol. 24, no. 5, pp. 1015–1026, 2008.

[25] A. Woo, B. Fidan, and W. W. Melek, “Localization for Autonomous Driving,” *Handb. Position Locat.*, pp. 1051–1087, 2019.

[26] J. Santos-Victor, G. Sandini, F. Curotto, and S. Garibaldi, “Divergent stereo for robot navigation: learning from bees,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 1993.

[27] C. D. Pantilie, S. Bota, I. Haller, and S. Nedevschi, “Real-time obstacle detection using dense stereo vision and dense optical flow,” *Proceedings - 2010 IEEE 6th*

International Conference on Intelligent Computer Communication and Processing, ICCP10. pp. 191–196, 2010.

[28] S. Panzieri, F. Pascucci, R. Setola, and G. Ulivi, "A low cost vision based localization system for mobile robots," *Target*, vol. 4, p. 5, 2001.

[29] W. Tian, "The research into methods of map building and path planning on mobile robots," *Proc. 2017 IEEE 2nd Inf. Technol. Networking, Electron. Autom. Control Conf. ITNEC 2017*, vol. 2018-Janua, pp. 1087–1090, 2018.

[30] T. Bailey, "Mobile Robot Localisation and Mapping in Extensive Outdoor Environments," The University of Sydney, 2002.

[31] S. Kagami, R. Hanai, N. Hatao, and M. Inaba, "Outdoor 3D map generation based on planar feature for autonomous vehicle navigation in urban environment," *IEEE/RSJ 2010 Int. Conf. Intell. Robot. Syst. IROS 2010 - Conf. Proc.*, pp. 1526–1531, 2010.

[32] K. Yousif, A. Bab-Hadiashar, and R. Hoseinnezhad, "An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics," *Intell. Ind. Syst.*, vol. 1, no. 4, pp. 289–311, 2015.

[33] W. Burgard, A. Derr, D. Fox, and A. B. Cremers, "Integrating global position estimation and position tracking for mobile robots: The dynamic Markov localization approach," *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2, no. October, pp. 730–735, 1998.

[34] S. Thrun, "Learning metric-topological maps for indoor mobile robot navigation," *Artif. Intell.*, vol. 99, no. 1, pp. 21–71, 1998.

[35] A. A. Ravankar, A. Ravankar, T. Emaru, and Y. Kobayashi, "A hybrid topological mapping and navigation method for large area robot mapping," *2017 56th Annu. Conf. Soc. Instrum. Control Eng. Japan, SICE 2017*, vol. 2017-Novem, pp. 1104–1107, 2017.

[36] A. Chatterjee, O. Ray, A. Chatterjee, and A. Rakshit, "Development of a real-life EKF based SLAM system for mobile robots employing vision sensing," *Expert Syst. Appl.*, vol. 38, no. 7, pp. 8266–8274, 2011.

[37] O. Khatib and F. Groen, *FastSLAM*. Springer, 2007.

[38] P. Qi and L. Wang, "On simulation and analysis of mobile robot SLAM using Rao-Blackwellized particle filters," *2011 IEEE/SICE Int. Symp. Syst. Integr. SII 2011*, pp. 1239–1244, 2011.

[39] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments," *Int. J.*

Rob. Res., vol. 31, no. 5, pp. 647–663, 2012.

[40] J. Peterson, H. Chaudhry, K. Abdelatty, J. Bird, and K. Kochersberger, "Online aerial terrain mapping for ground robot navigation," *Sensors (Switzerland)*, vol. 18, no. 2, 2018.

[41] M. Peasgood, "Cooperative Navigation for Teams of Mobile Robots," University of Waterloo, 2007.

[42] R. Grabowski and P. Khosla, "Localization techniques for a team of small robots," *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2, pp. 1067–1072, 2001.

[43] K. Kato, H. Ishiguro, and M. Barth, "Identifying and localizing robots in a multi-robot system environment," *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2, pp. 966–971, 1999.

[44] J. Butzke, K. Gochev, B. Holden, E. J. Jung, and M. Likhachev, "Planning for a ground-air robotic system with collaborative localization," *Proc. - IEEE Int. Conf. Robot. Autom.*, vol. 2016-June, pp. 284–291, 2016.

[45] Modelsport, "HobbyTech CRX V2 Crawler Kit Version," 2020. [Online]. Available: https://www.modelsport.co.uk/hobbytech-crx-v2-crawler-kit-version/rc-car-products/440250. [Accessed: 14-May-2020].

[46] A. D. Team, "Autopilot Hardware Options," 2020. [Online]. Available: https://ardupilot.org/rover/docs/common-autopilots.html#open-hardware. [Accessed: 15-May-2020].

[47] A. D. Team, "Pixhawk Overview," 2020. [Online]. Available: https://ardupilot.org/rover/docs/common-pixhawk-overview.html. [Accessed: 15-May-2020].

[48] Hex, "Pixhawk 2," 2020. [Online]. Available: http://www.proficnc.com/. [Accessed: 15-May-2020].

[49] Aerotenna, "OcPoC™ with Xilinx Zynq® mini SoC Flight Controller," 2020. [Online]. Available: https://aerotenna.com/shop/ocpoc-zynq-mini/. [Accessed: 15-May-2020].

[50] BeagleBoard.org, "BeagleBone Blue," 2020. [Online]. Available: https://beagleboard.org/blue. [Accessed: 15-May-2020].

[51] CUAV, "CUAV V5+ Autopilot without GPS," 2020. [Online]. Available: https://store.cuav.net/index.php?id_product=94&rewrite=cuav-new-pixhack-v5-autopilot-for-fpv-rc-drone-quadcopter-helicopter-flight-simulator-free-shipping-whole-sale&controller=product. [Accessed: 15-May-2020].

[52] A. D. Team, “F4BY FMU,” 2020. [Online]. Available: https://ardupilot.org/rover/docs/common-f4by.html. [Accessed: 15-May-2020].

[53] L. Documentation, “OpenPilot Revolution,” 2017. [Online]. Available: https://librepilot.atlassian.net/wiki/spaces/LPDOC/pages/26968084/OpenPilot+Revolution. [Accessed: 15-May-2020].

[54] Arduino, “Arduino Nano,” 2020. [Online]. Available: https://store.arduino.cc/arduino-nano. [Accessed: 15-May-2020].

[55] Arduino, “Arduino Mega 2560,” 2020. [Online]. Available: https://store.arduino.cc/arduino-mega-2560-rev3. [Accessed: 15-May-2020].

[56] NVIDIA, “Jetson TX2 Module,” 2020. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2. [Accessed: 15-May-2020].

[57] O. Wiki, “ODROID-C4,” 2020. [Online]. Available: https://wiki.odroid.com/odroid-c4/hardware/hardware. [Accessed: 15-May-2020].

[58] R. Pi, “Raspberry Pi 4 Tech Specs,” 2020. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/. [Accessed: 15-May-2020].

[59] Intel, “Intel RealSense Tracking Camera T265,” 2020. [Online]. Available: https://www.intelrealsense.com/tracking-camera-t265/?_ga=2.253564873.1734198841.1589668832-1401852564.1583416341. [Accessed: 15-May-2020].

[60] Intel, “Intel RealSense Depth Camera D435,” 2020. [Online]. Available: https://www.intelrealsense.com/depth-camera-d435/. [Accessed: 15-May-2020].

[61] D. Scharstein and R. Szeliski, “High-accuracy stereo depth maps using structured light,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 1, 2003.

[62] C. Y. Chiu, M. Thelwell, T. Senior, S. Choppin, J. Hart, and J. Wheat, “Comparison of depth cameras for three-dimensional reconstruction in medicine,” *Proc. Inst. Mech. Eng. Part H J. Eng. Med.*, vol. 233, no. 9, pp. 938–947, 2019.

[63] D. Project, “MAVLink Developer Guide,” 2020. [Online]. Available: https://mavlink.io/en/. [Accessed: 16-May-2020].

[64] 3D Robotics, “Dronekit-Python API Reference,” 2020. [Online]. Available: https://dronekit-python.readthedocs.io/en/stable/automodule.html#api-reference. [Accessed: 16-May-2020].

[65] I. RealSense, “T265 Tracking Camera,” 2020. [Online]. Available:

https://github.com/IntelRealSense/librealsense/blob/master/doc/t265.md. [Accessed: 16-May-2020].

[66] F. Sherratt, "ERL_SmartCities_2019," *Github*, 2020. [Online]. Available: https://github.com/fsherratt/ERL_SmartCities_2019/tree/master/modules. [Accessed: 01-Mar-2020].

[67] Z. Zh, "Hybrid A* path planning," *Github*, 2020. [Online]. Available: https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathPlanning/HybridAStar/hybrid_a_star.py. [Accessed: 16-May-2020].

[68] C. Eldershaw, M. Yim, Y. Zhang, K. Roufas, and D. Duff, "Motion planning with narrow C-space passages," *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2, no. October, pp. 1608–1613, 2003.

[69] E. Kaufman, T. Lee, Z. Ai, and I. S. Moskowitz, "Bayesian occupancy grid mapping via an exact inverse sensor model," *Proc. Am. Control Conf.*, vol. 2016-July, pp. 5709–5715, 2016.

[70] A. SCHEPELMANN, "IDENTIFICATION & SEGMENTATION OF LAWN GRASS BASED ON COLOR & VISUAL TEXTURE CLASSIFIERS," CASE WESTERN RESERVE UNIVERSITY, 2010.

9. Appendices

9.1. D435_Depth_Camera.py

```
import pyrealsense2 as rs
import traceback
import sys
import scipy
import numpy as np
from scipy import signal

class unexpectedDisconnect( Exception):
    # Camera unexpectably disconnected
    pass

class rs_d435:
    min_range = 0.1
    max_range = 10
    def __init__(self, width=640, height=480, framerate=30):
        self.width = width
        self.height = height

        self.framerate = framerate

        self.intrin = None
        self.scale = 1

        self._FOV = (0, 0)

        self.xDeprojectMatrix = None
        self.yDeprojectMatrix = None

    def __enter__(self):
        self.openConnection()

    def __exit__(self, exception_type, exception_value, traceback):
        if traceback:
            print(traceback.tb_frame)

        self.closeConnection()

    # -----------------------------------------------------------------------
    # openConnection
    # return void
    # -----------------------------------------------------------------------
    def openConnection(self):
        self.pipe = rs.pipeline()

        self.cfg = rs.config()
        self.cfg.enable_stream( rs.stream.depth, self.width, self.height, \
                                rs.format.z16, self.framerate )
        self.cfg.enable_stream(rs.stream.color, 640, 480, rs.format.bgr8,
self.framerate)

        self.profile = self.pipe.start( self.cfg  )

        self.initialise_deprojection_matrix()
```

```
        print('rs_d435:D435 Connection Open')

    # ------------------------------------------------------------------------
    # closeConnection
    # return void
    # ------------------------------------------------------------------------
    def closeConnection(self):
        self.pipe.stop()

        print('rs_d435:D435 Connection Closed')

    # ------------------------------------------------------------------------
    # getIntrinsics
    # get camera intrinsics
    # return void
    # ------------------------------------------------------------------------
    def getIntrinsics( self ):
        profile = self.pipe.get_active_profile()

        self.intrin = profile.get_stream( rs.stream.depth
).as_video_stream_profile().get_intrinsics()
        self.scale = profile.get_device().first_depth_sensor().get_depth_scale()

        self.FOV = rs.rs2_fov( self.intrin )
        self.FOV = np.deg2rad( self.FOV )

    # ------------------------------------------------------------------------
    # initialise_deprojection_matrix
    # Conversion matrix from depth to
    # return void
    # ------------------------------------------------------------------------
    def initialise_deprojection_matrix( self ):
        self.getIntrinsics()

        # Create deproject row/column vector
        self.xDeprojectRow = (np.arange( self.width ) - self.intrin.ppx) /
self.intrin.fx
        self.yDeprojectCol = (np.arange( self.height ) - self.intrin.ppy) /
self.intrin.fy

        # Tile across full matrix height/width
        self.xDeprojectMatrix = np.tile( self.xDeprojectRow, (self.height, 1) )
        self.yDeprojectMatrix = np.tile( self.yDeprojectCol, (self.width, 1)
).transpose()

        # self.xDeprojectMatrix = self.shrink(self.xDeprojectMatrix)
        # self.yDeprojectMatrix = self.shrink(self.yDeprojectMatrix)

    # ------------------------------------------------------------------------
    # getFrame
    # Retrieve a depth frame with scale metres from camera
    # return np.float32[width, height]
    # ------------------------------------------------------------------------
    def getFrame(self):
        frames = self.pipe.wait_for_frames()
        #colorizer = rs.colorizer()

        # Get depth data
        depth_frame = frames.get_depth_frame()
        color_frame = frames.get_color_frame()
        if not depth_frame:
            return None
```

```
        #spatial = rs.spatial_filter()
        #spatial.set_option(rs.option.filter_magnitude, 5)
        #spatial.set_option(rs.option.filter_smooth_alpha, 1)
        #spatial.set_option(rs.option.filter_smooth_delta, 50)
        #spatial.set_option(rs.option.holes_fill, 3)
        #filtered_depth = spatial.process(depth_frame)

        #colorizer.colorize() if you want to add colour to the depth image, put before
depth_frame.get_data
        #depth_points = np.asarray( filtered_depth.get_data(), dtype=np.float32 )
        depth_points = np.asarray( depth_frame.get_data(), dtype=np.float32 )
        color_image = np.asanyarray(color_frame.get_data(), dtype=np.uint8)
 #       depth_points = self.shrink(depth_points)

        return depth_points, color_image

    # ---------------------------------------------------------------------------
    # shrink
    # Shrink X, Y and Z by a factor so processing is faster
    # ---------------------------------------------------------------------------
    def shrink(self, frame, factor=4):
        frame = signal.decimate(frame, factor, n=None, ftype='iir', axis=1,
zero_phase=True)
        frame = signal.decimate(frame, factor, n=None, ftype='iir', axis=0,
zero_phase=True)

        return frame


    # ---------------------------------------------------------------------------
    # deproject_frame
    # Conversion depth frame to 3D local 61oordinate system in meters
    # return [[x,y,z]] coordinates of depth pixels
    # ---------------------------------------------------------------------------
    def deproject_frame( self, frame ):
        frame = frame * self.scale
        Z = frame
        X = np.multiply( frame, self.xDeprojectMatrix )
        Y = np.multiply( frame, self.yDeprojectMatrix )

        Z = np.reshape(Z, (-1))
        X = np.reshape(X, (-1))
        Y = np.reshape(Y, (-1))

        # Conversion into aero-reference frame
        points = np.column_stack( (Z,X,Y) )

        #gets rid of points that are placed at the camera where there isn't any actual
data
        inRange = np.where( (points[:,0] > self.min_range) & (points[:,0] <
self.max_range) )
        points = points[inRange]

        return points

if __name__ == "__main__":
    import cv2

    d435Obj = rs_d435( framerate = 30 )

    with d435Obj:
       while True:
```

```
            frame = d435Obj.getFrame()
            threeDFrame = d435Obj.deproject_frame(frame[0])
            cv2.imshow('depth_frame', frame[0])
            cv2.waitKey(1)
```

9.2. T265_Tracking_Camera.py

```
import traceback
import sys
import time
import pyrealsense2 as rs
from scipy.spatial.transform import Rotation as R

class rs_t265:
    """
    Initialise variables for T265 camera
    """
    def __init__(self):
        # Public
        self.tilt_deg = 0
        self.North_offset = 0

        # Private
        self._pipe = None
        self.H_aeroRef_T265Ref = None
        self.H_T265body_aeroBody = None

        self._initialise_rotational_transforms()

    """
    With __enter__ method opens a connected to the T265 camera
    """
    def __enter__(self):
        self.open_connection()

    """
    With __exit__ method closes the connection to the T265 camera
    """
    def __exit__(self, exception_type, exception_value, traceback):
        if traceback:
            print(traceback.tb_frame)
            self._exception_handle("rs_t265: __exit__: `{}`".format(exception_value))

        self.close_connection()

    """
    Open a connected to the T265 camera
    """
    def open_connection(self):
        cfg = rs.config()
        cfg.enable_stream(rs.stream.pose)

        self._pipe = rs.pipeline()

        try:
            self._pipe.start(cfg)
        except RuntimeError as e:
```

```
            self._exception_handle("rs_t265: getFrame: failed to connect to camera")
            raise e

        print('rs_t265:T265 Connection Open')

    """
    Close connected to the T265 camera
    """
    def close_connection(self):
        if self._pipe is None:
            return

        self._pipe.stop()
        self._pipe = None

        print('rs_t265:T265 Connection Closed')

    """
    Retrieve a data from the T265 camera
    """
    def get_frame(self) -> tuple:
        try:
            frames = self._pipe.wait_for_frames()
        except RuntimeError as e:
            self._exception_handle("rs_t265: getFrame: timeout waiting for data
frame")
            raise e

        pose = frames.get_pose_frame()

        try:
            data = pose.get_pose_data()
        except AttributeError:
            self._exception_handle("rs_t265: getFrame: pose frame contains no data")
            return None

        pos = [data.translation.x,
               data.translation.y,
               data.translation.z]

        quat = [data.rotation.x,
                data.rotation.y,
                data.rotation.z,
                data.rotation.w]

        conf = data.tracker_confidence

        quat = self._convert_rotational_frame(quat)
        pos = self._convert_positional_frame(pos)

        return (pos, quat, conf, time.time())

    """
    Initialise rotational transforms between tilted T265 and NED aero body and ref
frames
    """
    def _initialise_rotational_transforms(self):
        H_aeroNEDRef_aeroRef = R.from_euler('z', self.North_offset, degrees=True)
        H_aeroRef_T265Ref = R.from_matrix([[0,0,-1],[1,0,0],[0,-1,0]])
        H_T265Tilt_T265Body = R.from_euler('x', self.tilt_deg, degrees=True)

        self.H_aeroRef_T265Ref = H_aeroNEDRef_aeroRef * H_aeroRef_T265Ref
```

```
        self.H_T265body_aeroBody = H_T265Tilt_T265Body * H_aeroRef_T265Ref.inv()

    """
    Convert T265 rotational frame to aero NED frame
    """
    def _convert_rotational_frame(self, quat) -> list:
        rot = self.H_aeroRef_T265Ref * R.from_quat(quat)  * self.H_T265body_aeroBody

        return rot #.as_quat()

    """
    Convert T264 translation frame to aero NED translation
    """
    def _convert_positional_frame(self, pos) -> list:
        return self.H_aeroRef_T265Ref.apply(pos)

    """
    Function to collate interal class exceptions
    TODO: Log error - requires rabbit MQ stuff
    """
    def _exception_handle(self, err):
        print(err)

if __name__ == "__main__": #pragma: no cover
    t265Obj = rs_t265()

    with t265Obj:
        while True:
            data_frame = t265Obj.get_frame()

            print( ' Pos: {}\t Quat: {}\t Conf:{}'.format(
                    data_frame[0],
                    data_frame[1],
                    data_frame[2]) )

            time.sleep(0.5)
```

9.3. Point_Cloud.py

```
import numpy as np
import py
from scipy import interpolate
from scipy import io
import sys
import traceback
import copy
import time

class mapper:
    num_coordinate = 3

    xRange = [-8, 8]
    yRange = [-8, 8]
    zRange = [-0.5, 0.5]

    voxelSize = 0.02
    voxelMaxWeight = 2000
    voxelWeightDecay = 1
```

```
    xDivisions = int((xRange[1] - xRange[0]) / voxelSize)
    yDivisions = int((yRange[1] - yRange[0]) / voxelSize)
    zDivisions = int((zRange[1] - zRange[0]) / voxelSize)

    def __init__(self):
        self.xBins = np.linspace(self.xRange[0], self.xRange[1], self.xDivisions)
        self.yBins = np.linspace(self.yRange[0], self.yRange[1], self.yDivisions)
        self.zBins = np.linspace(self.zRange[0], self.zRange[1], self.zDivisions)

        self.grid = np.zeros((self.xDivisions, self.yDivisions, self.zDivisions),
dtype=np.float32)

        self.interpFunc = interpolate.RegularGridInterpolator( (self.xBins,
self.yBins, self.zBins),
                                                               self.grid, method =
'linear',
                                                               bounds_error = False,
                                                               fill_value = np.nan )

    # ------------------------------------------------------------------------
    # frame_to_global_points
    # param frame - (3,X,Y) matrix of coordinates from d435 camera
    # param pos - [x,y,z] offset cooridnates
    # param r - scipy local->global rotation object
    # return Null
    # ------------------------------------------------------------------------
    def local_to_global_points(self, local_points, pos, r):
        # Transform into global coordinate frame

        points_global = r.apply(local_points)
        points_global = np.add(points_global, pos)

        return points_global

    # ------------------------------------------------------------------------
    # updateMap
    # param pos - (N,3) list of points to add to the map
    # param rot -
    # return Null
    # ------------------------------------------------------------------------
    def update(self, points, pos, rot):
        # Add to map

        points = self.local_to_global_points(points, pos, rot)
        self.updateMap(points, pos)
        self.interpFunc.values = self.grid


    def digitizePoints(self, points):

        xSort = np.digitize(points[:, 0], self.xBins) -1  #Facing Directly Forward
from the camera
        ySort = np.digitize(points[:, 1], self.yBins) -1  #Direction to the right of
the camera, facing away from it
        zSort = np.digitize(points[:, 2], self.zBins) -1  #Direction straight up from
the camera

        return [xSort, ySort, zSort]

    # ------------------------------------------------------------------------
    # updateMap
```

```
    # param points - (N,3) list of points to qadd to the map
    # return Null
    # ----------------------------------------------------------------------------
    def updateMap(self, points, pos):

        # Update map
        gridPoints = self.digitizePoints(points)

        np.add.at(self.grid, gridPoints, 1)

        # Decay map where map has not reached maxWeight
        self.grid = np.where(self.grid < self.voxelMaxWeight,
                             self.grid - self.voxelWeightDecay, #If True
                             self.grid) #If False

        # Keep all map values below voxelMaxWeight
        self.grid = np.clip(self.grid, a_min=0, a_max=self.voxelMaxWeight)

    # ----------------------------------------------------------------------------
    # queryMap
    # param queryPoints - (N,3) list of points to query against map
    # return (N) list of risk for each point
    # ----------------------------------------------------------------------------
    def queryMap(self, queryPoints):
        return self.interpFunc(queryPoints)

    def saveToMatlab(self, filename):
        io.savemat(filename, mdict=dict(map=self.grid), do_compression=False)



if __name__ == "__main__":
#def mainpc():
   # from modules.realsense
    import T265_Tracking_Camera as t265
    import D435_Depth_Camera as d435
    import Telemetry as telemetry

    import cv2
    import base64
    import time
    import threading

    t265Obj = t265.rs_t265()
    d435Obj = d435.rs_d435(framerate=30, width=480, height=270)

    mapObj = mapper()

    with t265Obj, d435Obj:
        try:
            while True:
                t13 = time.perf_counter()
                # Get frames of data - points and global 6dof
                pos, r, conf, _ = t265Obj.get_frame()

                frame, rgbImg = d435Obj.getFrame()
                points = d435Obj.deproject_frame(frame)
                mapObj.update(points, pos, r)

                depth = cv2.applyColorMap(cv2.convertScaleAbs(frame, alpha=0.1),
cv2.COLORMAP_JET)
                cv2.imshow('frame', depth)
```

```
                cv2.imshow('RGB', rgbImg)
                cv2.waitKey(1)
                #print(conf)
                try:

                    x = np.digitize(pos[0], mapObj.xBins) - 1
                    y = np.digitize(pos[1], mapObj.yBins) - 1
                    z = np.digitize(pos[2], mapObj.zBins) - 1
                    z2= np.digitize(pos[2], mapObj.zBins) - 2
                    z3= np.digitize(pos[2], mapObj.zBins) - 0

                    gridSlice1=copy.copy(mapObj.grid[:,:,z])
                    gridSlice2=copy.copy(mapObj.grid[:,:,z2])
                    gridSlice3=copy.copy(mapObj.grid[:,:,z3])

                    gridSlice = np.sum([gridSlice1, gridSlice2, gridSlice3], axis=0)
                    grid = gridSlice


                    empty = np.zeros((mapObj.xDivisions,
mapObj.yDivisions),dtype=np.float32)

                    img = cv2.merge((grid, empty, empty))
                    img = cv2.transpose(img)

                    img = cv2.circle(img, (x, y), 5, (0, 1, 0), 2)

                    vec = np.asarray([20, 0, 0])
                    vec = r.apply(vec)  # Aero-ref -> Aero-body


                    vec[0] += x
                    vec[1] += y

                    img = cv2.line(img, (x, y), (int(vec[0]), int(vec[1])), (0, 0, 1),
2)
                    img = cv2.resize(img, (540, 540))
                    cv2.imshow('map', img)
                    cv2.waitKey(1)
                    t16 = time.perf_counter()
                    print(f"get frames, deproject and update map and visualise: {t16 -
t13:0.4f} seconds")
                    # time.sleep(0.5)

                except KeyboardInterrupt:
                    raise KeyboardInterrupt
                except:
                    traceback.print_exc(file=sys.stdout)

        except KeyboardInterrupt:
            pass
```

9.4. Position.py

```
import T265_Tracking_Camera as t265

from scipy.spatial.transform import Rotation as R
```

```
import numpy as np
import pymavlink
import time
from threading import Thread

class position:
    def __init__(self, pixhawkObj):
        self.t265 = t265.rs_t265()
        self.t265.open_connection()

        self.pixhawkObj = pixhawkObj

        self._pos = np.asarray([0,0,0], dtype=np.float)
        self._r = R.from_euler('xyz', [0,0,0])
        self._conf = 0

        self.running = True
        self.north_offset = None
        self.current_time_us = 0

    def setNorthOffset(self, north_offset):
        if north_offset is not None and self.north_offset is None:
            t265_yaw = self._r.as_euler('xyz')[0][2]
            north_offset -= t265_yaw
            self.north_offset = R.from_euler('xyz', [0,0,north_offset])

    def __del__(self):
        self.t265.closeConnection()

    def update(self):
        return self._pos, self._r, self._conf

    def loop(self):
        while self.running:
            self._pos, self._r, self._conf, _ = self.t265.get_frame()
            #self.setNorthOffset( self.pixhawkObj.compass_heading )
            self.current_time_us = int(round(time.time() * 1000000))
            # Convert from FRD to NED coordinate system
            if self.north_offset is not None:
                self._pos = self.north_offset.apply(self._pos)
                self._r = self.north_offset * self._r

            if  self.pixhawkObj.enable_msg_vision_position_estimate:
                self.pixhawkObj.send_vision_position_estimate_message(self._pos,
self._r, self.current_time_us)

            if self.pixhawkObj.enable_update_tracking_confidence_to_gcs:
                self.pixhawkObj.send_tracking_confidence_to_gcs(self._conf)


            time.sleep(0.01)

    def close(self):
        self.running = False
```

9.5. Car.py

```
import matplotlib.pyplot as plt
from math import sqrt, cos, sin, tan, pi

WB = 0.33  # rear to front wheel
```

```
W = 0.26  # width of car
LF = 0.5  # distance from rear to vehicle front end
LB = 0.15  # distance from rear to vehicle back end
MAX_STEER = 0.5  # [rad] maximum steering angle

WBUBBLE_DIST = (LF - LB) / 2.0
WBUBBLE_R = sqrt(((LF + LB) / 2.0)**2 + 1)

# vehicle rectangle verticles
VRX = [LF, LF, -LB, -LB, LF]
VRY = [W / 2, -W / 2, -W / 2, W / 2, W / 2]


def check_car_collision(xlist, ylist, yawlist, ox, oy, kdtree):
    for x, y, yaw in zip(xlist, ylist, yawlist):
        cx = x + WBUBBLE_DIST * cos(yaw)
        cy = y + WBUBBLE_DIST * sin(yaw)

        ids = kdtree.search_in_distance([cx, cy], WBUBBLE_R)

        if not ids:
            continue

        if not rectangle_check(x, y, yaw,
                               [ox[i] for i in ids], [oy[i] for i in ids]):
            return False  # collision

    return True  # no collision


def rectangle_check(x, y, yaw, ox, oy):
    # transform obstacles to base link frame
    c, s = cos(-yaw), sin(-yaw)
    for iox, ioy in zip(ox, oy):
        tx = iox - x
        ty = ioy - y
        rx = c * tx - s * ty
        ry = s * tx + c * ty

        if not (rx > LF or rx < -LB or ry > W / 2.0 or ry < -W / 2.0):
            return False  # no collision

    return True  # collision


def plot_arrow(x, y, yaw, length=1.0, width=0.5, fc="r", ec="k"):
    """Plot arrow."""
    if not isinstance(x, float):
        for (ix, iy, iyaw) in zip(x, y, yaw):
            plot_arrow(ix, iy, iyaw)
    else:
        plt.arrow(x, y, length * cos(yaw), length * sin(yaw),
                  fc=fc, ec=ec, head_width=width, head_length=width, alpha=0.4)
        # plt.plot(x, y)


def plot_car(x, y, yaw):
    car_color = '-k'
    c, s = cos(yaw), sin(yaw)

    car_outline_x, car_outline_y = [], []
    for rx, ry in zip(VRX, VRY):
```

```
        tx = c * rx - s * ry + x
        ty = s * rx + c * ry + y
        car_outline_x.append(tx)
        car_outline_y.append(ty)

    arrow_x, arrow_y, arrow_yaw = c * 1.5 + x, s * 1.5 + y, yaw
    plot_arrow(arrow_x, arrow_y, arrow_yaw)

    plt.plot(car_outline_x, car_outline_y, car_color)


def pi_2_pi(angle):
    return (angle + pi) % (2 * pi) - pi


def move(x, y, yaw, distance, steer, L=WB):
    x += distance * cos(yaw)
    y += distance * sin(yaw)
    yaw += pi_2_pi(distance * tan(steer) / L)  # distance/2

    return x, y, yaw


if __name__ == '__main__':
    x, y, yaw = 0., 0., 1.
    plt.axis('equal')
    plot_car(x, y, yaw)
    plt.show()
```

9.6. Reeds_Shepp.py

```
import math
import matplotlib.pyplot as plt
import numpy as np

show_animation = True


class Path:

    def __init__(self):
        self.lengths = []
        self.ctypes = []
        self.L = 0.0
        self.x = []
        self.y = []
        self.yaw = []
        self.directions = []


def plot_arrow(x, y, yaw, length=1.0, width=0.5, fc="r", ec="k"):
    """
    Plot arrow
    """

    if not isinstance(x, float):
        for (ix, iy, iyaw) in zip(x, y, yaw):
            plot_arrow(ix, iy, iyaw)
```

```
    else:
        plt.arrow(x, y, length * math.cos(yaw), length * math.sin(yaw),
                  fc=fc, ec=ec, head_width=width, head_length=width)
        plt.plot(x, y)


def mod2pi(x):
    v = np.mod(x, 2.0 * math.pi)
    if v < -math.pi:
        v += 2.0 * math.pi
    else:
        if v > math.pi:
            v -= 2.0 * math.pi
    return v


def straight_left_straight(x, y, phi):
    phi = mod2pi(phi)
    if y > 0.0 and 0.0 < phi < math.pi * 0.99:
        xd = - y / math.tan(phi) + x
        t = xd - math.tan(phi / 2.0)
        u = phi
        v = math.sqrt((x - xd) ** 2 + y ** 2) - math.tan(phi / 2.0)
        return True, t, u, v
    elif y < 0.0 < phi < math.pi * 0.99:
        xd = - y / math.tan(phi) + x
        t = xd - math.tan(phi / 2.0)
        u = phi
        v = -math.sqrt((x - xd) ** 2 + y ** 2) - math.tan(phi / 2.0)
        return True, t, u, v

    return False, 0.0, 0.0, 0.0


def set_path(paths, lengths, ctypes):
    path = Path()
    path.ctypes = ctypes
    path.lengths = lengths

    # check same path exist
    for tpath in paths:
        typeissame = (tpath.ctypes == path.ctypes)
        if typeissame:
            if sum(tpath.lengths) - sum(path.lengths) <= 0.01:
                return paths  # not insert path

    path.L = sum([abs(i) for i in lengths])

    # Base.Test.@test path.L >= 0.01
    if path.L >= 0.01:
        paths.append(path)

    return paths


def straight_curve_straight(x, y, phi, paths):
    flag, t, u, v = straight_left_straight(x, y, phi)
    if flag:
        paths = set_path(paths, [t, u, v], ["S", "L", "S"])

    flag, t, u, v = straight_left_straight(x, -y, -phi)
    if flag:
```

```
        paths = set_path(paths, [t, u, v], ["S", "R", "S"])

    return paths


def polar(x, y):
    r = math.sqrt(x ** 2 + y ** 2)
    theta = math.atan2(y, x)
    return r, theta


def left_straight_left(x, y, phi):
    u, t = polar(x - math.sin(phi), y - 1.0 + math.cos(phi))
    if t >= 0.0:
        v = mod2pi(phi - t)
        if v >= 0.0:
            return True, t, u, v

    return False, 0.0, 0.0, 0.0


def left_right_left(x, y, phi):
    u1, t1 = polar(x - math.sin(phi), y - 1.0 + math.cos(phi))

    if u1 <= 4.0:
        u = -2.0 * math.asin(0.25 * u1)
        t = mod2pi(t1 + 0.5 * u + math.pi)
        v = mod2pi(phi - t + u)

        if t >= 0.0 >= u:
            return True, t, u, v

    return False, 0.0, 0.0, 0.0


def curve_curve_curve(x, y, phi, paths):
    flag, t, u, v = left_right_left(x, y, phi)
    if flag:
        paths = set_path(paths, [t, u, v], ["L", "R", "L"])

    flag, t, u, v = left_right_left(-x, y, -phi)
    if flag:
        paths = set_path(paths, [-t, -u, -v], ["L", "R", "L"])

    flag, t, u, v = left_right_left(x, -y, -phi)
    if flag:
        paths = set_path(paths, [t, u, v], ["R", "L", "R"])

    flag, t, u, v = left_right_left(-x, -y, phi)
    if flag:
        paths = set_path(paths, [-t, -u, -v], ["R", "L", "R"])

    # backwards
    xb = x * math.cos(phi) + y * math.sin(phi)
    yb = x * math.sin(phi) - y * math.cos(phi)

    flag, t, u, v = left_right_left(xb, yb, phi)
    if flag:
        paths = set_path(paths, [v, u, t], ["L", "R", "L"])

    flag, t, u, v = left_right_left(-xb, yb, -phi)
    if flag:
```

```
        paths = set_path(paths, [-v, -u, -t], ["L", "R", "L"])

    flag, t, u, v = left_right_left(xb, -yb, -phi)
    if flag:
        paths = set_path(paths, [v, u, t], ["R", "L", "R"])

    flag, t, u, v = left_right_left(-xb, -yb, phi)
    if flag:
        paths = set_path(paths, [-v, -u, -t], ["R", "L", "R"])

    return paths


def curve_straight_curve(x, y, phi, paths):
    flag, t, u, v = left_straight_left(x, y, phi)
    if flag:
        paths = set_path(paths, [t, u, v], ["L", "S", "L"])

    flag, t, u, v = left_straight_left(-x, y, -phi)
    if flag:
        paths = set_path(paths, [-t, -u, -v], ["L", "S", "L"])

    flag, t, u, v = left_straight_left(x, -y, -phi)
    if flag:
        paths = set_path(paths, [t, u, v], ["R", "S", "R"])

    flag, t, u, v = left_straight_left(-x, -y, phi)
    if flag:
        paths = set_path(paths, [-t, -u, -v], ["R", "S", "R"])

    flag, t, u, v = left_straight_right(x, y, phi)
    if flag:
        paths = set_path(paths, [t, u, v], ["L", "S", "R"])

    flag, t, u, v = left_straight_right(-x, y, -phi)
    if flag:
        paths = set_path(paths, [-t, -u, -v], ["L", "S", "R"])

    flag, t, u, v = left_straight_right(x, -y, -phi)
    if flag:
        paths = set_path(paths, [t, u, v], ["R", "S", "L"])

    flag, t, u, v = left_straight_right(-x, -y, phi)
    if flag:
        paths = set_path(paths, [-t, -u, -v], ["R", "S", "L"])

    return paths


def left_straight_right(x, y, phi):
    u1, t1 = polar(x + math.sin(phi), y - 1.0 - math.cos(phi))
    u1 = u1 ** 2
    if u1 >= 4.0:
        u = math.sqrt(u1 - 4.0)
        theta = math.atan2(2.0, u)
        t = mod2pi(t1 + theta)
        v = mod2pi(t - phi)

        if t >= 0.0 and v >= 0.0:
            return True, t, u, v

    return False, 0.0, 0.0, 0.0
```

```
def generate_path(q0, q1, max_curvature):
    dx = q1[0] - q0[0]
    dy = q1[1] - q0[1]
    dth = q1[2] - q0[2]
    c = math.cos(q0[2])
    s = math.sin(q0[2])
    x = (c * dx + s * dy) * max_curvature
    y = (-s * dx + c * dy) * max_curvature

    paths = []
    paths = straight_curve_straight(x, y, dth, paths)
    paths = curve_straight_curve(x, y, dth, paths)
    paths = curve_curve_curve(x, y, dth, paths)

    return paths


def interpolate(ind, length, mode, max_curvature, origin_x, origin_y, origin_yaw,
path_x, path_y, path_yaw, directions):
    if mode == "S":
        path_x[ind] = origin_x + length / max_curvature * math.cos(origin_yaw)
        path_y[ind] = origin_y + length / max_curvature * math.sin(origin_yaw)
        path_yaw[ind] = origin_yaw
    else:  # curve
        ldx = math.sin(length) / max_curvature
        ldy = 0.0
        if mode == "L":  # left turn
            ldy = (1.0 - math.cos(length)) / max_curvature
        elif mode == "R":  # right turn
            ldy = (1.0 - math.cos(length)) / -max_curvature
        gdx = math.cos(-origin_yaw) * ldx + math.sin(-origin_yaw) * ldy
        gdy = -math.sin(-origin_yaw) * ldx + math.cos(-origin_yaw) * ldy
        path_x[ind] = origin_x + gdx
        path_y[ind] = origin_y + gdy

    if mode == "L":  # left turn
        path_yaw[ind] = origin_yaw + length
    elif mode == "R":  # right turn
        path_yaw[ind] = origin_yaw - length

    if length > 0.0:
        directions[ind] = 1
    else:
        directions[ind] = -1

    return path_x, path_y, path_yaw, directions


def generate_local_course(total_length, lengths, mode, max_curvature, step_size):
    n_point = math.trunc(total_length / step_size) + len(lengths) + 4

    px = [0.0 for _ in range(n_point)]
    py = [0.0 for _ in range(n_point)]
    pyaw = [0.0 for _ in range(n_point)]
    directions = [0.0 for _ in range(n_point)]
    ind = 1

    if lengths[0] > 0.0:
        directions[0] = 1
    else:
```

```
            directions[0] = -1

        ll = 0.0

        for (m, l, i) in zip(mode, lengths, range(len(mode))):
            if l > 0.0:
                d = step_size
            else:
                d = -step_size

            # set origin state
            ox, oy, oyaw = px[ind], py[ind], pyaw[ind]

            ind -= 1
            if i >= 1 and (lengths[i - 1] * lengths[i]) > 0:
                pd = - d - ll
            else:
                pd = d - ll

            while abs(pd) <= abs(l):
                ind += 1
                px, py, pyaw, directions = interpolate(
                    ind, pd, m, max_curvature, ox, oy, oyaw, px, py, pyaw, directions)
                pd += d

            ll = l - pd - d  # calc remain length

            ind += 1
            px, py, pyaw, directions = interpolate(
                ind, l, m, max_curvature, ox, oy, oyaw, px, py, pyaw, directions)

        # remove unused data
        while px[-1] == 0.0:
            px.pop()
            py.pop()
            pyaw.pop()
            directions.pop()

        return px, py, pyaw, directions


    def pi_2_pi(angle):
        return (angle + math.pi) % (2 * math.pi) - math.pi


    def calc_paths(sx, sy, syaw, gx, gy, gyaw, maxc, step_size):
        q0 = [sx, sy, syaw]
        q1 = [gx, gy, gyaw]

        paths = generate_path(q0, q1, maxc)
        for path in paths:
            x, y, yaw, directions = generate_local_course(
                path.L, path.lengths, path.ctypes, maxc, step_size * maxc)

            # convert global coordinate
            path.x = [math.cos(-q0[2]) * ix + math.sin(-q0[2])
                      * iy + q0[0] for (ix, iy) in zip(x, y)]
            path.y = [-math.sin(-q0[2]) * ix + math.cos(-q0[2])
                      * iy + q0[1] for (ix, iy) in zip(x, y)]
            path.yaw = [pi_2_pi(iyaw + q0[2]) for iyaw in yaw]
            path.directions = directions
            path.lengths = [length / maxc for length in path.lengths]
```

```
        path.L = path.L / maxc

    return paths


def reeds_shepp_path_planning(sx, sy, syaw,
                              gx, gy, gyaw, maxc, step_size=0.2):
    paths = calc_paths(sx, sy, syaw, gx, gy, gyaw, maxc, step_size)

    if not paths:
        return None, None, None, None, None

    minL = float("Inf")
    best_path_index = -1
    for i, _ in enumerate(paths):
        if paths[i].L <= minL:
            minL = paths[i].L
            best_path_index = i

    bpath = paths[best_path_index]

    return bpath.x, bpath.y, bpath.yaw, bpath.ctypes, bpath.lengths


def main():
    print("Reeds Shepp path planner sample start!!")

    start_x = -1.0  # [m]
    start_y = -4.0  # [m]
    start_yaw = np.deg2rad(-20.0)  # [rad]

    end_x = 5.0  # [m]
    end_y = 5.0  # [m]
    end_yaw = np.deg2rad(25.0)  # [rad]

    curvature = 1.0
    step_size = 0.1

    px, py, pyaw, mode, clen = reeds_shepp_path_planning(
        start_x, start_y, start_yaw, end_x, end_y, end_yaw, curvature, step_size)

    if show_animation:  # pragma: no cover
        plt.cla()
        plt.plot(px, py, label="final course " + str(mode))

        # plotting
        plot_arrow(start_x, start_y, start_yaw)
        plot_arrow(end_x, end_y, end_yaw)

        plt.legend()
        plt.grid(True)
        plt.axis("equal")
        plt.show()

    if not px:
        assert False, "No path"


if __name__ == '__main__':
    main()
```

9.7. Astar_heuristic

```
import math
import heapq
import matplotlib.pyplot as plt
import time

show_animation = False


class Node:

    def __init__(self, x, y, cost, pind):
        self.x = x
        self.y = y
        self.cost = cost
        self.pind = pind

    def __str__(self):
        return str(self.x) + "," + str(self.y) + "," + str(self.cost) + "," +
str(self.pind)


def calc_final_path(ngoal, closedset, reso):
    # generate final course
    rx, ry = [ngoal.x * reso], [ngoal.y * reso]
    pind = ngoal.pind
    while pind != -1:
        n = closedset[pind]
        rx.append(n.x * reso)
        ry.append(n.y * reso)
        pind = n.pind

    return rx, ry


def dp_planning(sx, sy, gx, gy, ox, oy, reso, rr):
    """
    sx: start x position [m]
    sy: start y position [m]
    gx: goal x position [m]
    gx: goal x position [m]
    ox: x position list of Obstacles [m]
    oy: y position list of Obstacles [m]
    reso: grid resolution [m]
    rr: robot radius[m]
    """

    nstart = Node(round(sx / reso), round(sy / reso), 0.0, -1)
    ngoal = Node(round(gx / reso), round(gy / reso), 0.0, -1)
    ox = [iox / reso for iox in ox] #divides all of ox by the resolution
    oy = [ioy / reso for ioy in oy] #divides all of oy by the resolution

    obmap, minx, miny, maxx, maxy, xw, yw = calc_obstacle_map(ox, oy, reso, rr)
    #t1=time.perf_counter()
    #defines movement in terms of relative positions and gives the cost of each
movement
    motion = get_motion_model()
```

```
    #initialising both the yet to visit and visited list
    openset, closedset = dict(), dict()
    openset[calc_index(ngoal, xw, minx, miny)] = ngoal
    pq = []
    pq.append((0, calc_index(ngoal, xw, minx, miny)))

    while 1:
        if not pq:
            break
        cost, c_id = heapq.heappop(pq) #c_id is the current index, heapop returns the
smallest data element from the heap pq
        #popping current node out of openset into closed set
        if c_id in openset:
            current = openset[c_id]
            closedset[c_id] = current
            openset.pop(c_id)
        else:
            continue

        # show graph
        if show_animation:  # pragma: no cover
            plt.plot(current.x * reso, current.y * reso, "xc")
            # for stopping simulation with the esc key.
            plt.gcf().canvas.mpl_connect('key_release_event',
                    lambda event: [exit(0) if event.key == 'escape' else None])
            if len(closedset.keys()) % 10 == 0:
                plt.pause(0.001)

        # Remove the item from the open set

        # expand search grid based on motion model, generates child nodes from
adjacent squares
        for i, _ in enumerate(motion):
            node = Node(current.x + motion[i][0],
                        current.y + motion[i][1],
                        current.cost + motion[i][2], c_id)
            n_id = calc_index(node, xw, minx, miny)
            #making sure not to use nodes already visited
            if n_id in closedset:
                continue
            #checking node is within boundary and not in obstacle
            if not verify_node(node, obmap, minx, miny, maxx, maxy):
                continue

            if n_id not in openset:
                openset[n_id] = node  # Discover a new node
                heapq.heappush(
                    pq, (node.cost, calc_index(node, xw, minx, miny)))
            else:
                if openset[n_id].cost >= node.cost:
                    # This path is the best until now. record it!
                    openset[n_id] = node
                    heapq.heappush(
                        pq, (node.cost, calc_index(node, xw, minx, miny)))

    rx, ry = calc_final_path(closedset[calc_index(
        nstart, xw, minx, miny)], closedset, reso)
    #t2=time.perf_counter()
    #print(f"Path Planner in {t2 - t1:0.4f} seconds")
    return rx, ry, closedset
```

```
def calc_heuristic(n1, n2):
    w = 1.0  # weight of heuristic
    d = w * math.sqrt((n1.x - n2.x)**2 + (n1.y - n2.y)**2)
    return d

#Making sure the node is within range and not in an obstacle
def verify_node(node, obmap, minx, miny, maxx, maxy):

    if node.x < minx:
        return False
    elif node.y < miny:
        return False
    elif node.x >= maxx:
        return False
    elif node.y >= maxy:
        return False
    #Need to subtract minx and miny to scale to the obmap indicies correctly
    if obmap[int(round(node.x-minx))][int(round(node.y-miny))]:
        return False

    return True

#function determines whether position would cause a collision with an obstacle
def calc_obstacle_map(ox, oy, reso, vr):

    minx = int(round(min(ox)))
    miny = int(round(min(oy)))
    maxx = int(round(max(ox)))
    maxy = int(round(max(oy)))

    xwidth = round(maxx - minx)
    ywidth = round(maxy - miny)
    # obstacle map generation, determines which positions would cause a collision
    # with an obstacle given the device's radius
    obmap = [[False for i in range(ywidth)] for i in range(xwidth)]
    #for ix in range(xwidth):
    #    x = ix + minx #the current x position
    #    for iy in range(ywidth):
    #        y = iy + miny #the current y position
    #        #  print(x, y)
    #        for iox, ioy in zip(ox, oy):
    #            d = math.sqrt((iox - x)**2 + (ioy - y)**2) #distance from current x
and y position to obstacle
    #            if d <= vr / reso:
    #                obmap[ix][iy] = True
    #                break
    obmap_motion = obmap_motion_model()

    for iox, ioy in zip(ox, oy):
        rox=iox
        roy=ioy
        iox=int(round(iox))
        ioy=int(round(ioy))

        for i, _ in enumerate(obmap_motion):
            adjind = [iox + obmap_motion[i][0],
                      ioy + obmap_motion[i][1]]

            if not verify_obmap(adjind, minx, miny, maxx, maxy):
                continue

            d = math.sqrt((rox - adjind[0])**2 + (roy - adjind[1])**2)
```

```
            if d <= vr / reso:
                ix = adjind[0] - minx
                iy = adjind[1] - miny
                obmap[ix][iy] = True


    return obmap, minx, miny, maxx, maxy, xwidth, ywidth


def calc_index(node, xwidth, xmin, ymin):
    return (node.y - ymin) * xwidth + (node.x - xmin)

#function just giving the option to move in 8 directions(up, down, diagonally, etc..)
#and the costs of moving in those directions
def get_motion_model():
    # dx, dy, cost
    motion = [[1, 0, 1],
              [0, 1, 1],
              [-1, 0, 1],
              [0, -1, 1],
              [-1, -1, math.sqrt(2)],
              [-1, 1, math.sqrt(2)],
              [1, -1, math.sqrt(2)],
              [1, 1, math.sqrt(2)]]

    return motion

# function needs to be changed depending on how the resolution and car radius change
def obmap_motion_model():
    # dx, dy
    obmap_motion = [[0, 0],
                    [1, 0],
                    [2, 0],
                    [0, 1],
                    [0, 2],
                    [-1, 0],
                    [-2, 0],
                    [0, -1],
                    [0, -2],
                    [-1, -1],
                    [-1, 1],
                    [1, -1],
                    [1, 1],
                    [1, 2],
                    [2, 1],
                    [2, 2],
                    [-1, 2],
                    [-2, 1],
                    [-2, 2],
                    [1, -2],
                    [2, -1],
                    [2, -2],
                    [-1, -2],
                    [-2, -1],
                    [-2, -2]]

    return obmap_motion


def verify_obmap(adjind, minx, miny, maxx, maxy):

    if adjind[0] < minx:
```

```
            return False
        elif adjind[1] < miny:
            return False
        elif adjind[0] >= maxx:
            return False
        elif adjind[1] >= maxy:
            return False

        return True


def main():
    print(__file__ + " start!!")

    # start and goal position
    sx = 10.0  # [m]
    sy = 10.0  # [m]
    gx = 50.0  # [m]
    gy = 50.0  # [m]
    grid_size = 2.0  # [m]
    robot_size = 1.0  # [m]

    ox, oy = [], []

    for i in range(60):
        ox.append(i)
        oy.append(0.0)
    for i in range(60):
        ox.append(60.0)
        oy.append(i)
    for i in range(61):
        ox.append(i)
        oy.append(60.0)
    for i in range(61):
        ox.append(0.0)
        oy.append(i)
    for i in range(40):
        ox.append(20.0)
        oy.append(i)
    for i in range(40):
        ox.append(40.0)
        oy.append(60.0 - i)

    if show_animation:  # pragma: no cover
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "xr")
        plt.plot(gx, gy, "xb")
        plt.grid(True)
        plt.axis("equal")

    rx, ry, _ = dp_planning(sx, sy, gx, gy, ox, oy, grid_size, robot_size)

    if show_animation:  # pragma: no cover
        plt.plot(rx, ry, "-r")
        plt.show()


if __name__ == '__main__':
    show_animation = True
    main()
```

9.8. Hybrid_Astar.py

```
import heapq
import scipy.spatial
import numpy as np
import math
import matplotlib.pyplot as plt
import sys
sys.path.append("../ReedsSheppPath/")
try:
    from Astar_heuristic import dp_planning, calc_obstacle_map
    import Reeds_Shepp as rs
    from Car import move, check_car_collision, MAX_STEER, WB, plot_car
except:
    raise


XY_GRID_RESOLUTION = 0.25  # [m]
YAW_GRID_RESOLUTION = np.deg2rad(15.0)  # [rad]
MOTION_RESOLUTION = 0.1  # [m] path interpolate resolution
N_STEER = 20.0  # number of steer command
H_COST = 1.0
VR = 0.15  # robot radius

SB_COST = 100.0  # switch back penalty cost
BACK_COST = 10.0  # backward penalty cost
STEER_CHANGE_COST = 5.0  # steer angle change penalty cost
STEER_COST = 1.0  # steer angle change penalty cost
H_COST = 5.0  # Heuristic cost

show_animation = True


class Node:

    def __init__(self, xind, yind, yawind, direction,
                 xlist, ylist, yawlist, directions,
                 steer=0.0, pind=None, cost=None):
        self.xind = xind #x index
        self.yind = yind #y index
        self.yawind = yawind #yaw index
        self.direction = direction #moving direction, forward=true, backwards=false
        self.xlist = xlist #x position
        self.ylist = ylist #y position
        self.yawlist = yawlist #yaw angle
        self.directions = directions #directions of each point
        self.steer = steer #steer input
        self.pind = pind #parent index
        self.cost = cost #cost


class Path:

    def __init__(self, xlist, ylist, yawlist, directionlist, cost):
        self.xlist = xlist
        self.ylist = ylist
        self.yawlist = yawlist
        self.directionlist = directionlist
        self.cost = cost
```

```
class KDTree:
    """
    Nearest neighbor search class with KDTree
    """

    def __init__(self, data):
        # store kd-tree
        self.tree = scipy.spatial.cKDTree(data)

    def search(self, inp, k=1):
        """
        Search NN
        inp: input data, single frame or multi frame
        """

        if len(inp.shape) >= 2:  # multi input
            index = []
            dist = []

            for i in inp.T:
                idist, iindex = self.tree.query(i, k=k)
                index.append(iindex)
                dist.append(idist)

            return index, dist

        dist, index = self.tree.query(inp, k=k)
        return index, dist

    def search_in_distance(self, inp, r):
        """
        find points with in a distance r
        """

        index = self.tree.query_ball_point(inp, r)
        return index


class Config:

    def __init__(self, ox, oy, xyreso, yawreso):
        min_x_m = min(ox)
        min_y_m = min(oy)
        max_x_m = max(ox)
        max_y_m = max(oy)

        ox.append(min_x_m)
        oy.append(min_y_m)
        ox.append(max_x_m)
        oy.append(max_y_m)

        self.minx = round(min_x_m / xyreso)
        self.miny = round(min_y_m / xyreso)
        self.maxx = round(max_x_m / xyreso)
        self.maxy = round(max_y_m / xyreso)

        self.xw = round(self.maxx - self.minx)
        self.yw = round(self.maxy - self.miny)

        self.minyaw = round(- math.pi / yawreso) - 1
```

```
        self.maxyaw = round(math.pi / yawreso)
        self.yaww = round(self.maxyaw - self.minyaw)


def calc_motion_inputs():

    for   steer   in   np.concatenate((np.linspace(int(-MAX_STEER),   int(MAX_STEER),
int(N_STEER)),[0.0])):
        for d in [1, -1]:
            yield [steer, d]


def get_neighbors(current, config, ox, oy, kdtree):

    for steer, d in calc_motion_inputs():
        node = calc_next_node(current, steer, d, config, ox, oy, kdtree)
        if node and verify_index(node, config):
            yield node


def calc_next_node(current, steer, direction, config, ox, oy, kdtree):

    x, y, yaw = current.xlist[-1], current.ylist[-1], current.yawlist[-1]

    arc_l = XY_GRID_RESOLUTION * 1.5
    xlist, ylist, yawlist = [], [], []
    for _ in np.arange(0, arc_l, MOTION_RESOLUTION):
        x, y, yaw = move(x, y, yaw, MOTION_RESOLUTION * direction, steer)
        xlist.append(x)
        ylist.append(y)
        yawlist.append(yaw)

    if not check_car_collision(xlist, ylist, yawlist, ox, oy, kdtree):
        return None

    d = direction == 1
    xind = round(x / XY_GRID_RESOLUTION)
    yind = round(y / XY_GRID_RESOLUTION)
    yawind = round(yaw / YAW_GRID_RESOLUTION)

    addedcost = 0.0

    if d != current.direction:
        addedcost += SB_COST

    # steer penalty
    addedcost += STEER_COST * abs(steer)

    # steer change penalty
    addedcost += STEER_CHANGE_COST * abs(current.steer - steer)

    cost = current.cost + addedcost + arc_l

    node = Node(xind, yind, yawind, d, xlist,
                ylist, yawlist, [d],
                pind=calc_index(current, config),
                cost=cost, steer=steer)

    return node


def is_same_grid(n1, n2):
```

```
    if n1.xind == n2.xind and n1.yind == n2.yind and n1.yawind == n2.yawind:
        return True
    return False


def analytic_expantion(current, goal, c, ox, oy, kdtree):

    sx = current.xlist[-1]
    sy = current.ylist[-1]
    syaw = current.yawlist[-1]

    gx = goal.xlist[-1]
    gy = goal.ylist[-1]
    gyaw = goal.yawlist[-1]

    max_curvature = math.tan(MAX_STEER) / WB
    paths = rs.calc_paths(sx, sy, syaw, gx, gy, gyaw,
                          max_curvature, step_size=MOTION_RESOLUTION)

    if not paths:
        return None

    best_path, best = None, None

    for path in paths:
        if check_car_collision(path.x, path.y, path.yaw, ox, oy, kdtree):
            cost = calc_rs_path_cost(path)
            if not best or best > cost:
                best = cost
                best_path = path

    return best_path


def update_node_with_analystic_expantion(current, goal,
                                         c, ox, oy, kdtree):
    apath = analytic_expantion(current, goal, c, ox, oy, kdtree)

    if apath:
        plt.plot(apath.x, apath.y)
        fx = apath.x[1:]
        fy = apath.y[1:]
        fyaw = apath.yaw[1:]

        fcost = current.cost + calc_rs_path_cost(apath)
        fpind = calc_index(current, c)

        fd = []
        for d in apath.directions[1:]:
            fd.append(d >= 0)

        fsteer = 0.0
        fpath = Node(current.xind, current.yind, current.yawind,
                     current.direction, fx, fy, fyaw, fd,
                     cost=fcost, pind=fpind, steer=fsteer)
        return True, fpath

    return False, None


def calc_rs_path_cost(rspath):
```

```
    cost = 0.0
    for l in rspath.lengths:
        if l >= 0:  # forward
            cost += l
        else:  # back
            cost += abs(l) * BACK_COST

    # swich back penalty
    for i in range(len(rspath.lengths) - 1):
        if rspath.lengths[i] * rspath.lengths[i + 1] < 0.0:  # switch back
            cost += SB_COST

    # steer penalyty
    for ctype in rspath.ctypes:
        if ctype != "S":  # curve
            cost += STEER_COST * abs(MAX_STEER)

    # ==steer change penalty
    # calc steer profile
    nctypes = len(rspath.ctypes)
    ulist = [0.0] * nctypes
    for i in range(nctypes):
        if rspath.ctypes[i] == "R":
            ulist[i] = - MAX_STEER
        elif rspath.ctypes[i] == "L":
            ulist[i] = MAX_STEER

    for i in range(len(rspath.ctypes) - 1):
        cost += STEER_CHANGE_COST * abs(ulist[i + 1] - ulist[i])

    return cost


def hybrid_a_star_planning(start, goal, ox, oy, xyreso, yawreso):
    """
    start
    goal
    ox: x position list of Obstacles [m]
    oy: y position list of Obstacles [m]
    xyreso: grid resolution [m]
    yawreso: yaw angle resolution [rad]
    """

    start[2], goal[2] = rs.pi_2_pi(start[2]), rs.pi_2_pi(goal[2])
    tox, toy = ox[:], oy[:]

    obkdtree = KDTree(np.vstack((tox, toy)).T)

    config = Config(tox, toy, xyreso, yawreso)

    nstart = Node(round(start[0] / xyreso), round(start[1] / xyreso), round(start[2] /
yawreso),
                  True, [start[0]], [start[1]], [start[2]], [True], cost=0)
    ngoal = Node(round(goal[0] / xyreso), round(goal[1] / xyreso), round(goal[2] /
yawreso),
                 True, [goal[0]], [goal[1]], [goal[2]], [True])

    openList, closedList = {}, {}

    _, _, h_dp = dp_planning(nstart.xlist[-1], nstart.ylist[-1],
                             ngoal.xlist[-1], ngoal.ylist[-1], ox, oy, xyreso, VR)
    #pq is the priority queue
```

```
    pq = []
    openList[calc_index(nstart, config)] = nstart
    #adding elements to the current heap
    heapq.heappush(pq, (calc_cost(nstart, h_dp, ngoal, config),
                        calc_index(nstart, config)))

    while True:
        if not openList:
            print("Error: Cannot find path, No open set")
            return [], [], []

        cost, c_id = heapq.heappop(pq)
        if c_id in openList:
            current = openList.pop(c_id)
            closedList[c_id] = current
        else:
            continue

        if show_animation:  # pragma: no cover
            plt.plot(current.xlist[-1], current.ylist[-1], "xc")
            # for stopping simulation with the esc key.
            plt.gcf().canvas.mpl_connect('key_release_event',
                    lambda event: [exit(0) if event.key == 'escape' else None])
            if len(closedList.keys()) % 10 == 0:
                plt.pause(0.001)

        isupdated, fpath = update_node_with_analystic_expantion(
            current, ngoal, config, ox, oy, obkdtree)

        if isupdated:
            break

        for neighbor in get_neighbors(current, config, ox, oy, obkdtree):
            neighbor_index = calc_index(neighbor, config)
            if neighbor_index in closedList:
                continue
            if neighbor not in openList \
                    or openList[neighbor_index].cost > neighbor.cost:
                heapq.heappush(
                    pq, (calc_cost(neighbor, h_dp, ngoal, config),
                         neighbor_index))
                openList[neighbor_index] = neighbor

    path = get_final_path(closedList, fpath, nstart, config)
    return path

#n=nstart/neighbur
def calc_cost(n, h_dp, goal, c):
    ind = (n.yind - c.miny) * c.xw + (n.xind - c.minx)
    if ind not in h_dp:
        return n.cost + 999999999  # collision cost
    return n.cost + H_COST * h_dp[ind].cost


def get_final_path(closed, ngoal, nstart, config):
    rx, ry, ryaw = list(reversed(ngoal.xlist)), list(
        reversed(ngoal.ylist)), list(reversed(ngoal.yawlist))
    direction = list(reversed(ngoal.directions))
    nid = ngoal.pind
    finalcost = ngoal.cost

    while nid:
```

```
        n = closed[nid]
        rx.extend(list(reversed(n.xlist)))
        ry.extend(list(reversed(n.ylist)))
        ryaw.extend(list(reversed(n.yawlist)))
        direction.extend(list(reversed(n.directions)))

        nid = n.pind

    rx = list(reversed(rx))
    ry = list(reversed(ry))
    ryaw = list(reversed(ryaw))
    direction = list(reversed(direction))

    # adjust first direction
    direction[0] = direction[1]

    path = Path(rx, ry, ryaw, direction, finalcost)

    return path


def verify_index(node, c):
    xind, yind = node.xind, node.yind
    if xind >= c.minx and xind <= c.maxx and yind >= c.miny \
            and yind <= c.maxy:
        return True

    return False


def calc_index(node, c):
    ind = (node.yawind - c.minyaw) * c.xw * c.yw + \
        (node.yind - c.miny) * c.xw + (node.xind - c.minx)
    if ind <= 0:
        print("Error(calc_index):", ind)

    return ind


def main():
    print("Start Hybrid A* planning")
    import time
    import Point_Cloud as map
    import T265_Tracking_Camera as t265
    import D435_Depth_Camera as d435
    import cv2
    import base64
    import threading
    import copy
    import traceback

    t265Obj = t265.rs_t265()
    d435Obj = d435.rs_d435(framerate=30, width=480, height=270)
    mapObj = map.mapper()
    s=0
    with t265Obj, d435Obj:
        try:
            while True:
                tik=time.perf_counter()

                # Get frames of data - points and global 6dof
                pos, r, conf, _ = t265Obj.get_frame()
```

```
                frame, rgbImg = d435Obj.getFrame()
                points = d435Obj.deproject_frame(frame)
                mapObj.update(points, pos, r)


                try:

                    x = np.digitize(pos[0], mapObj.xBins) - 1
                    y = np.digitize(pos[1], mapObj.yBins) - 1
                    z = np.digitize(pos[2], mapObj.zBins) - 1
                    z2= np.digitize(pos[2], mapObj.zBins) - 2
                    z3= np.digitize(pos[2], mapObj.zBins) - 0

                    gridSlice1=copy.copy(mapObj.grid[:,:,z])
                    gridSlice2=copy.copy(mapObj.grid[:,:,z2])
                    gridSlice3=copy.copy(mapObj.grid[:,:,z3])

                    gridSlice = np.sum([gridSlice1, gridSlice2, gridSlice3], axis=0)
                    grid = gridSlice

                    empty                =                np.zeros((mapObj.xDivisions,
mapObj.yDivisions),dtype=np.float32)
                    img = cv2.merge((grid, empty, empty))
                    img = cv2.transpose(img)
                    img = cv2.circle(img, (x, y), 5, (0, 1, 0), 2)

                    vec = np.asarray([20, 0, 0])
                    vec = r.apply(vec)  # Aero-ref -> Aero-body

                    vec[0] += x
                    vec[1] += y

                    img = cv2.line(img, (x, y), (int(vec[0]), int(vec[1])), (0, 0, 1),
2)
                    img = cv2.resize(img, (540, 540))
                    cv2.imshow('map', img)
                    cv2.waitKey(1)

                    #defining x and y coordinates of obstacles
                    ox, oy = [], []



                    for i in np.arange(-5,5,0.5):
                        ox.append(i)
                        oy.append(-5)
                    for i in np.arange(-5,8,0.5):
                        ox.append(5)
                        oy.append(i)
                    for i in np.arange(-5,5.5,0.5):
                        ox.append(i)
                        oy.append(8)
                    for i in np.arange(-5,8,0.5):
                        ox.append(-5)
                        oy.append(i)


                    grid = cv2.transpose(grid)
                    for i in range(grid.shape[0]):
                        if np.max(grid[i]) > 0.0:
                            for j in range(grid.shape[1]):
```

```
                            if grid[i][j] > 0:
                                ox.append(mapObj.xBins[i])
                                oy.append(mapObj.yBins[j])

                    # Should have North as 90 degrees
                    # Set Initial parameters, float
                    yaw_angle = r.as_euler('zyx', degrees=True)

                    start = [pos[1], pos[0], np.deg2rad(90.0 - yaw_angle[0])]#90 faces
to the top, 0 to the right, -90 towards the bottom
                    goal = [2.0, 5.0, np.deg2rad(90.0)]


                    plt.plot(ox, oy, ".k")
                    rs.plot_arrow(start[0], start[1], start[2], fc='g')
                    rs.plot_arrow(goal[0], goal[1], goal[2])
                    plt.grid(True)
                    plt.axis("equal")


                    if s == 0:


                        path    =    hybrid_a_star_planning(start,    goal,    ox,    oy,
XY_GRID_RESOLUTION, YAW_GRID_RESOLUTION)
                        tok=time.perf_counter()
                        print(f"Path Planner in {tok - tik:0.4f} seconds")

                        xpath = path.xlist
                        ypath = path.ylist
                        yawpath = path.yawlist
                        directionpath = path.directionlist
                        s=s+1
                        for ix, iy, iyaw in zip(xpath, ypath, yawpath):
                            plt.cla()
                            plt.plot(ox, oy, ".k")
                            plt.plot(xpath, ypath, "-r", label="Hybrid A* path")
                            plt.grid(True)
                            plt.axis("equal")
                            plot_car(ix, iy, iyaw)
                            plt.pause(0.0001)

                        print(__file__ + " done!!")

                    elif s != 0:
                     #use the obstacle map to check if any of the new obstacles will
cause a collision with the path
                      #if they do then calculate a new path, if they don't then
continue along path
                        ox1 = [iox / XY_GRID_RESOLUTION for iox in ox]
                        oy1 = [ioy / XY_GRID_RESOLUTION for ioy in oy]
                        obmap, minx, miny, maxx, maxy, xw, yw = calc_obstacle_map(ox1,
oy1, XY_GRID_RESOLUTION, VR)

                        plt.cla()
                        plt.plot(ox, oy, ".k")
                        plt.plot(xpath, ypath, "-r", label="Hybrid A* path")
                        plt.grid(True)
                        plt.axis("equal")
                        # need this to run through the x and y values and stop if
they're within range of an obstacle
                        #divide path.xlist by the resolution
```

```
                        # here if path.xlist is empty need a try statement or
something
                        for ind in range(len(path.xlist)):
                            if obmap[int(round((path.xlist[ind]/XY_GRID_RESOLUTION) -
minx))][int(round((path.ylist[ind]/XY_GRID_RESOLUTION) - miny))]:
                                tic=time.perf_counter()
                                path = hybrid_a_star_planning(start, goal, ox, oy,
XY_GRID_RESOLUTION, YAW_GRID_RESOLUTION)
                                toc=time.perf_counter()
                                print(f"Path Planner in {toc - tic:0.4f} seconds")

                                xpath = path.xlist
                                ypath = path.ylist
                                yawpath = path.yawlist
                                directionpath = path.directionlist
                                for ix, iy, iyaw in zip(xpath, ypath, yawpath):
                                    plt.cla()
                                    plt.plot(ox, oy, ".k")
                                    plt.plot(xpath, ypath, "-r", label="Hybrid A*
path")
                                    plt.grid(True)
                                    plt.axis("equal")
                                    plot_car(ix, iy, iyaw)
                                    plt.pause(0.0001)
                                print(__file__ + " done!!")
                                break


                            #else:
                            #    continue

                except KeyboardInterrupt:
                    raise KeyboardInterrupt
                except:
                    traceback.print_exc(file=sys.stdout)

        except KeyboardInterrupt:
            pass

if __name__ == '__main__':
    main()
```

9.9. Motion.py

```
import pymavlink
import time
import numpy as np
from threading import Thread

class motion:
    def __init__(self, pixhawkObj):
        self.running = True
        self.pixhawkObj = pixhawkObj
        self.recalc_path = False
        return
    def update(self, xpath, ypath, yawpath):
        self.xpath = xpath
```

```
        self.ypath = ypath
        self.yawpath = yawpath

    def loop(self):
        while self.running:
            if self.recalc_path == False:
                for self.ixpath, self.iypath, self.iyawpath in zip(self.xpath,
self.ypath, self.yawpath):
                    if self.recalc_path == True:
                        break
                    else:
                        self.pixhawkObj.send_ned_position(self.ixpath, self.iypath, 0)
                        self.pixhawkObj.condition_yaw(self.iyawpath, relative = False)
                        time.sleep(0.1)

            else:
                self.pixhawkObj.send_ned_position(0, 0, 0)
                print("recalculating path")
                time.sleep(0.1)

    def recalc(self, recalc_path):
        self.recalc_path = recalc_path

    def close(self):
        self.running = False
```

9.10. Pixhawk.py

```
# Set the path for IDLE
import sys
import dronekit
sys.path.append("/usr/local/lib/")

# Set MAVLink protocol to 2.
import os
os.environ["MAVLINK20"] = "1"

# Import the libraries
import pyrealsense2 as rs
import numpy as np
#import transformations as tf
import math as m
import time
import scipy.spatial
import argparse
import threading
from time import sleep
from apscheduler.schedulers.background import BackgroundScheduler
from dronekit import connect, VehicleMode
from pymavlink import mavutil

########################################
# Parameters
########################################
class Pix:
    def __init__(self):
        ########Anything with self, you're going to have to change
        # Default configurations for connection to the FCU
        self.connection_string_default = '/dev/ttyUSB0'
```

```
        self.connection_baudrate_default = 921600
        self.connection_timeout_sec_default = 5

        self.camera_orientation_default = 0

        # Enable/disable each message/function individually
        self.enable_msg_vision_position_estimate = True
        self.vision_position_estimate_msg_hz_default = 15

        #self.enable_msg_vision_position_delta = False
        #self.vision_position_delta_msg_hz_default = 15

        self.enable_update_tracking_confidence_to_gcs = True
        self.update_tracking_confidence_to_gcs_hz_default = 1

        # Default global position for EKF home/ origin
        self.enable_auto_set_ekf_home = False
        self.home_lat = 151269321    # Somewhere random
        self.home_lon = 16624301     # Somewhere random
        self.home_alt = 163000       # Somewhere random

        # TODO: Taken care of by ArduPilot, so can be removed (once the handling on AP
side is confirmed stable)
        # In NED frame, offset from the IMU or the center of gravity to the camera's
origin point
        self.body_offset_enabled = 0
        self.body_offset_x = 0  # In meters (m)
        self.body_offset_y = 0  # In meters (m)
        self.body_offset_z = 0  # In meters (m)

        # Global scale factor, position x y z will be scaled up/down by this factor
        self.scale_factor = 1.0

        # Enable using yaw from compass to align north (zero degree is facing north)
        self.compass_enabled = 0

        # pose data confidence: 0x0 - Failed / 0x1 - Low / 0x2 - Medium / 0x3 - High
        self.pose_data_confidence_level = ('Failed', 'Low', 'Medium', 'High')

        # lock for thread synchronization
        self.lock = threading.Lock()

        ########################################
        # Global variables
        ########################################

        # FCU connection variables
        self.vehicle = None
        self.is_vehicle_connected = False

        # Camera-related variables
        self.pipe = None

        # Data variables
        self.data = None
        self.H_aeroRef_aeroBody = None
        self.heading_north_yaw = None
        self.current_confidence_level = None
```

```
    ######################################
    # Functions
    ######################################

    def send_ned_position(self, xpath, ypath, zpath):
        """
        Move vehicle in direction based on specified velocity vectors.
        """
        msg = self.vehicle.message_factory.set_position_target_local_ned_encode(
            0,       # time_boot_ms (not used)
            0, 0,    # target system, target component
            mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
            0b0000111111111000, # type_mask (only speeds enabled)
            xpath, ypath, zpath, # x, y, z positions
            0, 0, 0, # x, y, z velocity in m/s (not used)
            0,  0,  0,  #  x,  y,  z  acceleration  (not  supported  yet,  ignored  in
GCS_Mavlink)
            0, 0)    # yaw, yaw_rate (not supported yet, ignored in GCS_Mavlink)

        # send command
        self.vehicle.send_mavlink(msg)


    def condition_yaw(self, heading, relative=False):
        """
        Send MAV_CMD_CONDITION_YAW message to point vehicle at a specified heading (in
degrees).

        This  method  sets  an  absolute  heading  by  default,  but  you  can  set  the
`relative` parameter
        to `True` to set yaw relative to the current yaw heading.

        By default the yaw of the vehicle will follow the direction of travel. After
setting
        the  yaw  using  this  function  there  is  no  way  to  return  to  the  default  yaw
"follow direction
        of travel" behaviour (https://github.com/diydrones/ardupilot/issues/2427)

        For more information see:
        http://copter.ardupilot.com/wiki/common-mavlink-mission-command-messages-
mav_cmd/#mav_cmd_condition_yaw
        """
        if relative:
            is_relative = 1 #yaw relative to direction of travel
        else:
            is_relative = 0 #yaw is an absolute angle
        # create the CONDITION_YAW command using command_long_encode()
        msg = self.vehicle.message_factory.command_long_encode(
            0, 0,    # target system, target component
            mavutil.mavlink.MAV_CMD_CONDITION_YAW, #command
            0, #confirmation
            heading,    # param 1, yaw in degrees
            0,          # param 2, yaw speed deg/s
            1,          # param 3, direction -1 ccw, 1 cw
            is_relative, # param 4, relative offset 1, absolute angle 0
            0, 0, 0)    # param 5 ~ 7 not used
        # send command to vehicle
        self.vehicle.send_mavlink(msg)


    # https://mavlink.io/en/messages/common.html#VISION_POSITION_ESTIMATE
    def send_vision_position_estimate_message(self, _pos, _r, current_time_us):
```

```
        global is_vehicle_connected
        with self.lock:
            if self.is_vehicle_connected == True:
                rot_eul_angles = _r.as_euler('zyx', degrees=False)
                msg = self.vehicle.message_factory.vision_position_estimate_encode(
                    current_time_us,                    # us Timestamp (UNIX time or
time since system boot)
                    _pos[0],              # Global X position
                    _pos[1],           # Global Y position
                    _pos[2],              # Global Z position
                    rot_eul_angles[2],                          # Roll angle
                    rot_eul_angles[1],                          # Pitch angle
                    rot_eul_angles[0]                           # Yaw angle
                )
                self.vehicle.send_mavlink(msg)
                self.vehicle.flush()

    def send_tracking_confidence_to_gcs(self, _conf):
        global current_confidence_level
        confidence_status_string = 'Tracking confidence: ' + _conf
        self.send_msg_to_gcs(confidence_status_string)

    def send_msg_to_gcs(self, text_to_be_sent):
        # MAV_SEVERITY: 0=EMERGENCY 1=ALERT 2=CRITICAL 3=ERROR, 4=WARNING, 5=NOTICE,
6=INFO, 7=DEBUG, 8=ENUM_END
        # Defined here: https://mavlink.io/en/messages/common.html#MAV_SEVERITY
        # MAV_SEVERITY = 3 will let the message be displayed on Mission Planner HUD,
but 6 is ok for QGroundControl
        if self.is_vehicle_connected == True:
            text_msg = 'T265: ' + text_to_be_sent
            status_msg = self.vehicle.message_factory.statustext_encode(
                6,                      # MAV_SEVERITY
                text_msg.encode()     # max size is char[50]
            )
            self.vehicle.send_mavlink(status_msg)
            self.vehicle.flush()
            print("INFO: " + text_to_be_sent)
        else:
            print("INFO: Vehicle not connected. Cannot send text message to Ground
Control Station (GCS)")

    #        Send        a        mavlink        SET_GPS_GLOBAL_ORIGIN        message
(http://mavlink.org/messages/common#SET_GPS_GLOBAL_ORIGIN), which allows us to use
local position information without a GPS.
    def set_default_global_origin(self):
        if self.is_vehicle_connected == True:
            msg = self.vehicle.message_factory.set_gps_global_origin_encode(
                int(self.vehicle._master.source_system),
                self.home_lat,
                self.home_lon,
                self.home_alt
            )

            self.vehicle.send_mavlink(msg)
            self.vehicle.flush()

    #         Send         a         mavlink         SET_HOME_POSITION         message
(http://mavlink.org/messages/common#SET_HOME_POSITION), which allows us to use local
position information without a GPS.
    def set_default_home_position(self):
        if self.is_vehicle_connected == True:
            x = 0
```

```
            y = 0
            z = 0
            q = [1, 0, 0, 0]   # w x y z

            approach_x = 0
            approach_y = 0
            approach_z = 1

            msg = self.vehicle.message_factory.set_home_position_encode(
                int(self.vehicle._master.source_system),
                self.home_lat,
                self.home_lon,
                self.home_alt,
                x,
                y,
                z,
                q,
                approach_x,
                approach_y,
                approach_z
            )

            self.vehicle.send_mavlink(msg)
            self.vehicle.flush()

    # Request a timesync update from the flight controller, for future work.
    # TODO: Inspect the usage of timesync_update
    def update_timesync(self, ts=0, tc=0):
        if ts == 0:
            ts = int(round(time.time() * 1000))
        msg = vehicle.message_factory.timesync_encode(
            tc,     # tc1
            ts      # ts1
        )
        vehicle.send_mavlink(msg)
        vehicle.flush()

    # Listen to attitude data to acquire heading when compass data is enabled
    def att_msg_callback(self, attr_name, value):
        global heading_north_yaw
        if heading_north_yaw is None:
            heading_north_yaw = value.yaw
            print("INFO:  Received  first  ATTITUDE  message  with  heading  yaw",
heading_north_yaw * 180 / m.pi, "degrees")
        else:
            heading_north_yaw = value.yaw
            print("INFO:    Received    ATTITUDE    message    with    heading    yaw",
heading_north_yaw * 180 / m.pi, "degrees")


    def vehicle_connect(self, connection_string, connection_baudrate):
        global vehicle, is_vehicle_connected

        try:
            self.vehicle  =  connect(connection_string,  wait_ready  =  True,  baud  =
connection_baudrate, source_system = 1)
        except:
            print('Connection error! Retrying...')
            sleep(1)

        if self.vehicle == None:
            self.is_vehicle_connected = False
```

```
            return False
        else:
            self.is_vehicle_connected = True
            return True


    # Monitor user input from the terminal and perform action accordingly
    def user_input_monitor(self, scale_calib_enable):
        global scale_factor
        while True:
            # Specical case: updating scale
            #if scale_calib_enable == True:
            #       scale_factor = float(input("INFO: Type in new scale as float
number\n"))
            #    print("INFO: New scale is ", scale_factor)

            if self.enable_auto_set_ekf_home:
                self.send_msg_to_gcs('Set EKF home with default GPS location')
                self.set_default_global_origin()
                self.set_default_home_position()
                time.sleep(1) # Wait a short while for FCU to start working

            # Add new action here according to the key pressed.
            # Enter: Set EKF home when user press enter
            try:
                c = input()
                if c == "":
                    self.send_msg_to_gcs('Set EKF home with default GPS location')
                    self.set_default_global_origin()
                    self.set_default_home_position()
                else:
                    print("Got keyboard input", c)
            except IOError: pass
```

9.11. Arg_Parser.py

```
import argparse
def GetParser():
        parser = argparse.ArgumentParser(description='Reboots vehicle')
        parser.add_argument('--connect',
                            help="Vehicle connection target string. If not specified,
a default string will be used.")
        parser.add_argument('--baudrate', type=float,
                            help="Vehicle connection baudrate. If not specified, a
default value will be used.")
        parser.add_argument('--vision_position_estimate_msg_hz', type=float,
                            help="Update frequency for VISION_POSITION_ESTIMATE
message. If not specified, a default value will be used.")
        parser.add_argument('--scale_calib_enable', default=False,
action='store_true',
                            help="Scale calibration. Only run while NOT in flight")
        parser.add_argument('--camera_orientation', type=int,
                            help="Configuration for camera orientation. Currently
supported: forward, usb port to the right - 0; downward, usb port to the right - 1, 2:
forward tilted down 45deg")
        parser.add_argument('--debug_enable',type=int,
                            help="Enable debug messages on terminal")
```

```
        return parser
```

9.12. Run.py

```
#!/usr/bin/env python3

import Point_Cloud as map
import T265_Tracking_Camera as t265
import D435_Depth_Camera as d435
import Hybrid_Astar as planner
import Motion
import Pixhawk
import Position
import Arg_Parser
import time
import cv2
import base64
import threading
import copy
import traceback
import numpy as np
import matplotlib.pyplot as plt
import heapq
import scipy.spatial
import sys
import math
sys.path.append("/usr/local/lib/")

# Set MAVLink protocol to 2.
import os
os.environ["MAVLINK20"] = "1"

# Import the libraries
import pyrealsense2 as rs
import threading
from time import sleep
from apscheduler.schedulers.background import BackgroundScheduler
from dronekit import connect, VehicleMode, LocationGlobal, LocationGlobalRelative
from pymavlink import mavutil
if __name__ == "__main__":
    print("*** STARTING ***")

    pixhawkObj = Pixhawk.Pix()

    parser = Arg_Parser.GetParser()
    args = parser.parse_args()

    connection_string = args.connect
    connection_baudrate = args.baudrate
    vision_position_estimate_msg_hz = args.vision_position_estimate_msg_hz
    scale_calib_enable = args.scale_calib_enable
    camera_orientation = args.camera_orientation
    debug_enable = args.debug_enable

    # Using default values if no specified inputs
    if not connection_string:
        connection_string = pixhawkObj.connection_string_default
```

```
        print("INFO: Using default connection_string", connection_string)
    else:
        print("INFO: Using connection_string", connection_string)

    if not connection_baudrate:
        connection_baudrate = pixhawkObj.connection_baudrate_default
        print("INFO: Using default connection_baudrate", connection_baudrate)
    else:
        print("INFO: Using connection_baudrate", connection_baudrate)

    if not vision_position_estimate_msg_hz:
        vision_position_estimate_msg_hz                                                    =
pixhawkObj.vision_position_estimate_msg_hz_default
        print("INFO:       Using       default       vision_position_estimate_msg_hz",
vision_position_estimate_msg_hz)
    else:
        print("INFO:              Using              vision_position_estimate_msg_hz",
vision_position_estimate_msg_hz)

    if pixhawkObj.body_offset_enabled == 1:
        print("INFO:   Using   camera   position   offset:   Enabled,   x   y   z   is",
pixhawkObj.body_offset_x, pixhawkObj.body_offset_y, pixhawkObj.body_offset_z)
    else:
        print("INFO: Using camera position offset: Disabled")

    if pixhawkObj.compass_enabled == 1:
        print("INFO: Using compass: Enabled. Heading will be aligned to north.")
    else:
        print("INFO: Using compass: Disabled")

    if scale_calib_enable == True:
        print("\nINFO: SCALE CALIBRATION PROCESS. DO NOT RUN DURING FLIGHT.\nINFO:
TYPE IN NEW SCALE IN FLOATING POINT FORMAT\n")
    else:
        if pixhawkObj.scale_factor == 1.0:
            print("INFO: Using default scale factor", pixhawkObj.scale_factor)
        else:
            print("INFO: Using scale factor", pixhawkObj.scale_factor)

    if not camera_orientation:
        camera_orientation = pixhawkObj.camera_orientation_default
        print("INFO: Using default camera orientation", camera_orientation)
    else:
        print("INFO: Using camera orientation", camera_orientation)
    if camera_orientation == 0:     # Forward, USB port to the right
        H_aeroRef_T265Ref   = np.array([[0,0,-1,0],[1,0,0,0],[0,-1,0,0],[0,0,0,1]])
        H_T265body_aeroBody = np.linalg.inv(H_aeroRef_T265Ref)
    else:                           # Default is facing forward, USB port to the right
        H_aeroRef_T265Ref   = np.array([[0,0,-1,0],[1,0,0,0],[0,-1,0,0],[0,0,0,1]])
        H_T265body_aeroBody = np.linalg.inv(H_aeroRef_T265Ref)

    if not debug_enable:
        debug_enable = 0
    else:
        debug_enable = 1
        np.set_printoptions(precision=4, suppress=True) # Format output on terminal
        print("INFO: Debug messages enabled.")


    print("INFO: Connecting to vehicle.")
    while (not pixhawkObj.vehicle_connect(connection_string, connection_baudrate)):
        pass
```

```
    print("INFO: Vehicle connected.")


    d435Obj = d435.rs_d435(framerate=30, width=480, height=270)
    posObj = Position.position(pixhawkObj)
    mapObj = map.mapper()
    motionObj = Motion.motion(pixhawkObj)

    #Schedules Mavlink Messages in the Background at predetermined frequencies
    sched = BackgroundScheduler()

    if                pixhawkObj.enable_msg_vision_position_estimate                or
pixhawkObj.enable_update_tracking_confidence_to_gcs:
        sched.add_job(posObj.loop,           'interval',           seconds           =
1/vision_position_estimate_msg_hz)


    # A separate thread to monitor user input
    user_keyboard_input_thread                                                        =
threading.Thread(target=pixhawkObj.user_input_monitor)
    user_keyboard_input_thread.daemon = True
    user_keyboard_input_thread.start()

    sched.start()
    print("INFO: Press Enter to set EKF home at default location")

    pixhawkObj.set_default_global_origin()
    pixhawkObj.set_default_home_position()

    # x, y, yaw, of the waypoints taken from global path planner
    waypoints = np.asarray([[0.0, 4.0, 90.0],
            [0.0, 0.0, 180.0]])

    s = 0 # used to create initial path planning loop
    with d435Obj:
        try:
          while True:
            # Monitor last_heartbeat to reconnect in case of lost connection
            if                    pixhawkObj.vehicle.last_heartbeat                    >
pixhawkObj.connection_timeout_sec_default:
                pixhawkObj.is_vehicle_connected = False
                print("WARNING:  CONNECTION  LOST.  Last  hearbeat  was  %f  sec  ago."%
pixhawkObj.vehicle.last_heartbeat)
                print("WARNING: Attempting to reconnect ...")
                pixhawkObj.vehicle_connect()
                continue

            for iway in range(len(waypoints)):

                curr_goal = waypoints[iway]
                curr_pos, _, _ = posObj.update()

                remaining_distance  =  math.sqrt((curr_goal[0]  -  curr_pos[0])**2  +
(curr_goal[1] - curr_pos[1])**2)
                close_enough = 0.1

                while  remaining_distance  >  close_enough:  #  some  amount  of  distance
away from the curr_position

                    # Get frames of data - points and global 6dof
                    pos, r, conf = posObj.update()
```

```
                    remaining_distance = math.sqrt((curr_goal[0] - pos[0])**2 +
(curr_goal[1]- pos[1])**2)

                    frame, rgbImg = d435Obj.getFrame()
                    points = d435Obj.deproject_frame(frame)
                    mapObj.update(points, pos, r)

                    try:

                        x = np.digitize(pos[0], mapObj.xBins) - 1
                        y = np.digitize(pos[1], mapObj.yBins) - 1
                        z = np.digitize(pos[2], mapObj.zBins) - 1
                        z2= np.digitize(pos[2], mapObj.zBins) - 2
                        z3= np.digitize(pos[2], mapObj.zBins) - 0

                        #Taking a slice of the obstacles x and y coordinates at three
different z heights
                        gridSlice1=copy.copy(mapObj.grid[:,:,z])
                        gridSlice2=copy.copy(mapObj.grid[:,:,z2])
                        gridSlice3=copy.copy(mapObj.grid[:,:,z3])

                        #Adding the slices together
                        gridSlice = np.sum([gridSlice1, gridSlice2, gridSlice3],
axis=0)
                        grid = gridSlice

                        #Defining x and y coordinates of obstacles
                        ox, oy = [], []

                        #Using this to define the maximum search area of the path
planner
                        for i in np.arange(-5,5,0.5):
                            ox.append(i)
                            oy.append(-5)
                        for i in np.arange(-5,5,0.5):
                            ox.append(5)
                            oy.append(i)
                        for i in np.arange(-5,5.5,0.5):
                            ox.append(i)
                            oy.append(5)
                        for i in np.arange(-5,5,0.5):
                            ox.append(-5)
                            oy.append(i)

                        #Appending obstacles x and y coordinates from grid
                        grid = cv2.transpose(grid)
                        for i in range(grid.shape[0]):
                            if np.max(grid[i]) > 0.0:
                                for j in range(grid.shape[1]):
                                    if grid[i][j] > 0:
                                        ox.append(mapObj.xBins[i])
                                        oy.append(mapObj.yBins[j])

                        # Should have North as 90 degrees
                        # Set Initial parameters
                        # Start position is always the current position of the car
                        # Need to have a way of making the function still generate a
path if the start is within range of an obstacle
                        yaw_angle = r.as_euler('zyx', degrees=True)
                        start = [pos[0], pos[1], np.deg2rad(90.0 - yaw_angle[0])]
                        goal = [curr_goal[0], curr_goal[1], np.deg2rad(90.0 -
curr_goal[2])] #90 faces to the top, 0 to the right, -90 towards the bottom
```

```
                        #if path_index == 0:
                        #    path_index = path_index + 1
                        #else:
                        #    if path_index - 1 >= len(xpath):
                        #        break
                        #    else:
                        #        xvel = xpath[path_index - 1]
                        #        yvel = ypath[path_index - 1]
                        #        yaw_set = yawpath[path_index -1]

                        #        #needs to be faster 0.1m every sec is slow as shit
                        #        pixhawkObj.send_ned_position(xvel, yvel, 0 , 1)
                        #        pixhawkObj.condition_yaw(yaw_set, relative = False)

                        #        path_index = path_index + 1

                        #initial path calculation loop
                        if s == 0:

                            #path planner function
                            path = planner.hybrid_a_star_planning(start, goal, ox, oy,
planner.XY_GRID_RESOLUTION, planner.YAW_GRID_RESOLUTION)


                            #list of x,y,yaw and direction for the path
                            xpath = path.xlist
                            ypath = path.ylist
                            yawpath = path.yawlist
                            directionpath = path.directionlist

                            #creating thread for motion
                            motionObj.update(xpath, ypath, yawpath)
                            motion_thread = threading.Thread(target=motionObj.loop)
                            motion_thread.daemon = True
                            motion_thread.start()

                            s=s+1


                        #second path planner loop, new path only calculated if
obstacle detected in route of old path
                        elif s != 0:

                            #use the obstacle map to check if any of the new obstacles
will cause a collision with the path
                            ox1 = [iox / planner.XY_GRID_RESOLUTION for iox in ox]
                            oy1 = [ioy / planner.XY_GRID_RESOLUTION for ioy in oy]
                            obmap,    minx,    miny,    maxx,    maxy,    xw,    yw    =
planner.calc_obstacle_map(ox1, oy1, planner.XY_GRID_RESOLUTION, planner.VR)

                            #runs through the initial path x and y values and creates
a new path if they're within range of an obstacle
                            for ind in range(len(path.xlist)):
                                if
obmap[int(round((path.xlist[ind]/planner.XY_GRID_RESOLUTION)                          -
minx))][int(round((path.ylist[ind]/planner.XY_GRID_RESOLUTION) - miny))]:

                                    # send command to the pixhawk to stop moving and
wait for new path to be calculated
                                    motionObj.recalc(recalc_path = True)
```

```
                                    # plan a new path
                                    path = planner.hybrid_a_star_planning(start, goal,
ox, oy, planner.XY_GRID_RESOLUTION, planner.YAW_GRID_RESOLUTION)

                                    #list of x,y,yaw and direction of the new path
                                    xpath = path.xlist
                                    ypath = path.ylist
                                    yawpath = path.yawlist
                                    directionpath = path.directionlist

                                    # update pixhawk of the directions of the new path
                                    motionObj.update(xpath, ypath, yawpath)
                                    MotionObj.recalc(recalc_path = False)

                                    break
                        except KeyboardInterrupt:
                            raise KeyboardInterrupt
                        except:
                            traceback.print_exc(file=sys.stdout)
            except KeyboardInterrupt:
                pass

            finally:
                motionObj.close()
                d435Obj.closeConnection()
                pixhawkObj.vehicle.close()
                print("INFO: Realsense pipeline and vehicle object closed.")
                sys.exit()
```

9.13. Camera Rig Part Drawing