Imperial College London

Aerial Robotics Lab

MEng Group Design Project

Imperial College London

Department of Aeronautics

Deep Earth Drones

Flight Control and Navigation Report

Author:
Pablo Hermoso Moreno

CID:
01071174

Supervisor:
Dr. Aditya Paranjape

Second Marker:
Dr. Thulasi Mylvaganam

June 18, 2018

Abstract

As part of a project aimed at building a tethered quadcopter capable of autonomous navigation and perching in unexplored environments, this paper focuses on the conserations and solutions to some of the most challenging aspects of robot navigation, namely: sensing, acting, planning, communication architectures, hardware, computational efficiencies and problem solving. A comparative study of path planning algorithms was conducted and Artificial Potential Fields was selected due to its mathematical elegance, implementation simplicity and computational efficiency, which allow real-time obstacle avoidance without prior knowledge of the environment using exclusively on-board computations and Time-of-Flight sensors. Simulated Annealing was employed to solve one of the inherent shortcomings of such approach; Local Minima.

Effectiveness of the proposed algorithm was validated on custom built Python and Gazebo simulation platforms before demonstrating its potential in a real aerial platform using Robot Operating System (ROS) based communication. Full integration of hardware, the robustness of the Simulated Annelaing algorithm and a necessity to further optimise the Potential Field parameters were identified as key areas for improvement. Lastly, this paper outlines the need for a laser range data segmentation and feature extraction algorithm for a more rigorous implementation of Potential Fields in structured environments and discusses the challenges faced in the future implementation of a Simultaneous Localisation and Mapping (SLAM) algorithm.

Acknowledgements

Foremost, I would like to thank my fellow team members: Dylan Almeida, Wee Zhao Chua Khoo, and Roman Kastusik. Their perseverance, innovative attitude and team spirit have been remarkable over the last six weeks.

My sincere thanks also goes to Dr. Aditya Paranjape and Dr. Thulasi Mylvaganam for their encouragement, guidance and technical advice. Their experience, questions and feedback have been invaluable in delivering a feasible and practical solution in the time allocated for this project.

Besides my supervisors, I am grateful to Dr. Mirko Kovac for providing the vision and inspiration behind such an interesting, challenging and hands-on project and for facilitating access to the *Aerial Robotics Lab*.

Last but not least, I would like to express my gratitude to Dr. Chang Liu and Mr. Pisak Chermprayong for their knowledge, advice and patience. Their continued support throughout our project despite the demands of their own research is proof of their helping attitude.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In recent years, the Mining Industry has faced increased public pressure due to the human cost of completing such manual labour intensive tasks in a high risk environment [1]. Traditionally risk averse mining companies have been urged by external consultants [2] [3] to overcome innovation barriers and adopt technologies including autonomous vehicles, drones, and 3D printing to capture real time data and carry out some of the mine surveying, inspection and mapping tasks that are currently being completed by humans. Increased automation will not only minimise human risk and improve health and well-being of miners, but will also come with important social and economic implications. The image of mining will improve, operational productivity will increase, insurance costs will decrease and all this together will help transform stakeholder relationships.

As part of this vision, utilising UAVs equipped with sensors to carry out some of the mapping and inspection tasks offers a cost and time effective solution. Valuable data that would otherwise be impossible to collect due to the difficult access to some of these places can therefore be collected in a safe, repeatable and efficient manner. These factors coupled with more accessible commercial UAV platforms and the numerous open source hardware and software initiatives [4] [5] [6] have caused this field of research to become increasing popular [7] [8].

1.2 Project Overview

The aim of this project was to design, build, and test a tethered quadcopter capable of autonomously navigating though an unknown environment to a perching location before returning to base.

As part of the Flight Control and Navigation group, we identified 3 main tasks:

- Autonomous Navigation & Obstacle Avoidance
- Perching Target Acquisition
- Hardware and Software Integration

This report will focus on the Autonomous Navigation & Obstacle Avoidance Tasks since this is the task the author dedicated most of his time and effort to. Other contributions that lie outside the remit of the trajectory generation tasks, namely the creation of simulation environments and the integration of sensors within the ROS (Robot Operating System) environment will be outlined.

Within the time frame allocated to this project, the main limitation we encountered was the inability to incorporate precise Localisation and Mapping in a GPS-denied environment. Though Simultaneous Localisation and Mapping (SLAM) algorithms [10] have been successfully implemented in these conditions, they frequently assume availability of dense information about the environment. With the available sensor being a sparse sensing TeraRanger Tower (8x45° separated Time-of-Flight sensors) we explored other solutions such as that presented in [11]. Nevertheless, we decided to focus our efforts on the Localisation and Mapping of the UAV with respect to the target perching location after cruise with the aid of Computer Vision.

2 Literature Review

2.1 Problem Statement

Motion Path Planning can be defined as the act of computing a continuous sequence of collision-free robot configurations connecting the initial and goal configurations [12]. A Configuration Space is a data structure which allows the robot to specify the position (location and orientation) of any objects and the robot [13]. As such, a robot configuration is a specification of the positions of all robot points relative to a fixed coordinate system. In 3D, and for the purpose of this aerial vehicle, we can define the configuration of the robot by a vector containing the position and attitude of the UAV $q = (x, y, z, \psi, \theta, \phi)$.

Robot path planning traditionally focuses solely on the translations and rotations required to move the robot. Nevertheless, in the process of selecting the most appropriate path planning algorithm for this exercise, the following criteria were identified and used for the comparison of path planning algorithms:

- Completeness: This criterion is fulfilled by an algorithm if it will always find a solution to the motion planning problem when one exists or indicate failure in finite time. Though highly desirable, complete solutions become computationally intractable as the number of degrees of freedom increase. Weaker forms of such condition include *resolution completeness* and *probabilistic completeness*, where the first indicates the planner will find a solution at a given environment discretisation and the latter indicates that the planner will find an existing solution as time goes to infinity.
- Optimality: This can be defined according to many objective functions such as path length or execution time. Generally, optimal motion requires increased computational complexity [17].
- Computational Complexity: In particular, the space complexity or spacial memory was considered since this dictates how scalable the algorithms become as we increase the inputs or the complexity of the environment. For the purpose of navigation in unknown environments, the selected *online* or *sensor-based* algorithm must produce low computational time for the real-time re-computation of the path as new obstacles are detected.
- Impact of Sensor Uncertainty: For example, path

planning algorithms were carefully reviewed to ensure that sufficient measures were taken to avoid obstacles in the presence of drift.

2.2 Motion Planning Algorithms

2.2.1 Roadmaps

Roadmaps are a class of topological[1] maps embedded in free space where the nodes and edges also carry some physical meaning. Path planners can construct a collision-free path on to the roadmap, traverse the roadmap to the vicinity of the goal and then find a collision free path to the goal. An optimal path is finally computed using for example Dijkstra's single source shortest path algorithm [13]. Two of the most influential types of roadmaps are:

1. Visibility Graphs: In the construction of visibility graphs, the initial and goal configurations are connected via straight-line segments that connect two line-of-sight nodes [13]. Though it is a complete algorithm, Visibility Graphs were discarded since they try to stay as close as possible to obstacles as to compute the shortest path. In our scenario, any execution error would lead to a collision.
2. Voronoi Diagrams [18]: Motion across Voronoi Diagrams is based on moving along line segments or vertices which are equidistant to two or more obstacles respectively. Such class of representations are most suitable for polygonal worlds and become very difficult to compute in higher dimensions [17].

Overall, Roadmaps are most suited to 2D polygonal configuration spaces and their computation in 3D is complex. Furthermore, they work best when the robot navigates in a known environment, else, the robot will have to rely on its sensors to construct a roadmap that can be used for future excursions. In this exercise, this would lead to problems with the spacial memory required to hold such maps in 3D, the time complexity of this process, and the impact of localisation errors.

2.2.2 Cell Decomposition & Graph Search

Cell Decomposition offers an alternative representation of the configuration space. These structures represent free space by the union of simple non-overlapping discrete grid regions called cells. For each discretized cell, a point inside that cell is selected to be the representative point of the cell (often the center of the cell). Following, an *adjacency graph* is constructed by connecting the representative point from each cell to the representative points from each neighboring cell. In 2D this can lead can be 4-connected or 8-connected grids if we allow connections to be drawn diagonally. For each cell, the algorithm must then check if the representative point of each cell intersects with an obstacle, in which case, the cell is marked occupied and removed from the *adjacency graph* [19]. Finally, graph search is run on the *adjacency graph*, and if a path is found it is presented as a solution.

Within the family of exact cell decompositions, *trapezoidal decomposition* was discarded due to its reliance on a polygonal space. Thus, we investigated approximate cell decompositions, namely Regular Grids and Octrees (3D) [13]. The latter aims to alleviate the *digitization bias* and wasted space problems of Regular Grids by the use of a recursive grid.

Given a suitable *adjacency graph*, a path between the initial node and the goal node can be computed using graph search algorithms. Many of those algorithms require the program to visit each node on the graph to determine the shortest path between the initial and goal nodes. For reduced computational expense, path planners which do a "branch and bound" style of search; that is, ones which prune off paths which aren't optimal were investigated, in particular, A* and D* algorithms [20][21][22].

A* algorithm generates an optimal path incrementally; at each update, it considers the nodes that could be added to the path and picks the best one[13]. The heart of the method is the formula (or evaluation function) for measuring the plausibility of a node:

$$f^*(n) = g^*(n) + h^*(n) \tag{1}$$

$f^*(n)$ measures how good the move to node n is. $g^*(n)$ measures the cost of getting to node n from the initial node. $h^*(n)$ is the cheapest cost of getting from n to goal – Euclidean (straight line) distance.

D* is the dynamic version of A*. Cost functions are fixed in A* algorithm while they are recomputed whenever obstacles are identified in D* algorithm. As cost functions computed remain fixed, A* algorithm is efficient if the location of the obstacles are known prior to planning. Cost computation in D* does not involve heuristics. It is computed backwards from the target cell. As cost functions are re-computed in D* algorithm, it provides shortest path in all scenarios, including unknown environments.

Despite D* algorithm providing the shortest path, the cost of re-computation is large in terms of memory requirements and execution time for vast grids. As such, modified versions of the original D* Algorithm presented in [20] and [23] were explored.

In summary, the advantages of such class of algorithms is that they are *resolution complete* and *resolution optimal* [19]. However, they work best with known maps, they may not find a path if the resolution is not sufficient and discretisation is expensive,

[1]Topological maps aim at representing environments with a graphlike structure, with nodes representing "landmarks" and edges representing adjacency relationship between nodes [17].

Table 1: Summary of Comparison of Relevant Motion Planning Algorithms According to the Criteria Established in 2.1.

	Roadmaps	Cell Decomposition + Graph Search	Sample Based	Potential Fields
Completeness	Yes	Resolution Complete	Probabilistically Complete	Probabilistically and Resolution Complete
Optimality	Visuality - Yes Voronoi - No	Resolution Dependent	Generally No	No
Suitable for Unknown Environments	No	D* - Yes / A* - No	No	Yes
Good 3D Freedom Scalability	No	No	Yes	Yes

which means that the number of grids increases exponentially as we increase of the number of DOFs.

2.2.3 Sampling Based Algorithms

By relying explicitly on a collision checking module, Sampling Based Algorithms eliminate the requirement of an exact representation of the obstacles in the configuration space. They connect a set of points sampled from the obstacle-free space in order to build a graph (roadmap) of feasible trajectories. The roadmap is then used to construct the solution to the original motion-planning problem. The most influential sampling-based motion planning algorithms include: Probabilistic RoadMaps (PRMs) (Kavraki et al., 1996, 1998) and Rapidly-Exploring Random Trees (RRTs) (Kuffner and LaValle, 2000; LaValle and Kuffner, 2001; LaValle, 2006).

These class of algorithms are widely used today, since they provide large amounts of computational savings by avoiding explicit construction of obstacles in the state space and are *probabilistically complete*. However, in dynamic or unknown environments, the collision checking module must be continuously updated and path re-computations become increasingly computationally expensive.

2.2.4 Artificial Potential Fields

Artificial Potential Fields were first proposed by Khatib (1986) [25]. These search algorithms incrementally explore the obstacle free configuration space by directing the robot as if it were a particle moving in a gradient vector field. In practice, we work on a model of a positively charged robot attracted to a negatively charged goal which is repelled from positively charged obstacles. Since the path is the result of the interaction of vector fields, the path finding problem becomes does no longer involve a direct construction of an optimum path thus no explicit representation of the configuration space is required.

This comes with numerous advantages. Firstly, its high simplicity, safety and mathematical elegance make it one of the most popular methods within research. It is also suitable for real-time applications due to its fast computation time [26]. The robot takes into account the realities of the current environment as the sensors detect new obstacles. Furthermore, planning and control are merged into a single function in the *gradient descent* method, thus making its implementation significantly easier.

However, the problem that plagues all gradient descent methods is the possible covergence to of local minima in the potential field which is different from the goal, which represents the global minima. In other words, many potential functions do not lead to complete path planners. Additionally, other non-desirable phenomena such as oscillations in narrow passages or impossibility of passing between closely spaced obstacles have been observed.

2.3 Motion Planning Algorithm Selection

A summary of the global comparison of the motion planning algorithms is presented in Table 1. For the purpose of this exercise, implementation simplicity ruled over optimal trajectory generation and efficiency under an unknown environment was deemed critical. Furthermore, optimal trajectories generally come very close to obstacles, which was not desired considering we would be navigating in an unknown environment with sparse sensing. Sensor uncertainty or undetected obstacles could cause collision when trying to pursue the optimal path in an on-line built obstacle map. As shown in Table 1, Potential Fields was the only algorithm that satisfied all criteria.

3 The Potential Fields Method

Mathematically, a *potential function* is defined as a differentiable real-valued function $U : \mathbb{R}^m \rightarrow \mathbb{R}$. Physically, its value can be viewed as energy, hence its gradient $\nabla U(q) = [\frac{\partial U}{\partial q_1}(q), ..., \frac{\partial U}{\partial q_m}(q)]^T$, which also indicates the direction that locally maximises $U(q)$ can be interpreted as a force. We can use this property to construct a vector field that drives the robot following the negated gradient of the potential function towards the goal. This technique is known as *gradient descent* (2) and for lower order systems where the higher order dynamics can be neglected, the gradients can be viewed as velocity vectors [17]:

$$\dot{q(t)} = -\nabla U(q(t)) \tag{2}$$

Robot motion will terminate at a critical point q^* (maximum, saddle or minimum) such that $\nabla U(q^*) = 0$. Mimima constitute the only class of stable critical points. The problem of terminating motion in a local minima which is not the goal is a problem that will be carefully addressed in 3.3.

In the field of robot motion, many potential functions exist, with some of the most influentials being those presented by Khatib [25] and Canny [27]. Due to its mathematical simplicity and intuitive nature, we used the additive attractive / repulsive potential [17], which defines the potential at a configuration as:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{3}$$

3.1 The Attractive Potential

To obtain a continuously differentiable attractive potential that monotonically increases with distance from the goal (q_{goal}), we employed a combined quadratic and conic potential (4) (5) such that the conic potential attracts the robot when it is distant from the q_{goal} and the quadratic potential does so when it is close to q_{goal}. Quadratic potentials offer a positive and continuous differentiable function which attains its zero minimum at the $q = q_{goal}$. However, they lead to large velocity outputs far away from the goal. Conic potentials offer a solution to such problem, but they contain a discontinuity at the origin.

$$U_{att}(q) = \begin{cases} \frac{1}{2}\zeta d^2(q, q_{goal}), & d(q, q_{goal}) \leq d^*_{goal} \\ d^*_{goal}\zeta d(q, q_{goal}) - \frac{1}{2}\zeta d^*_{goal}, & d(q, q_{goal}) > d^*_{goal} \end{cases} \tag{4}$$

$$\nabla U_{att}(q) = \begin{cases} \zeta(q - q_{goal}), & d(q, q_{goal}) \leq d^*_{goal} \\ \frac{d^*_{goal}\zeta(q - q_{goal})}{d(q, q_{goal})}, & d(q, q_{goal}) > d^*_{goal} \end{cases} \tag{5}$$

Note that d^* is the threshold distance where the planner switches from conic to quadratic, ensuring the gradient is well defined at the boundary.

3.2 The Repulsive Potential

For the purpose of this exercise, we decided to use a purely repulsive potential based on the original FIRAS function (7)(8) presented in [25], whose value tends to infinity as the robot approaches the obstacle surface ($d_i(q) \to 0$). Following the implementation of such function, it was observed that the robot would oscillate between equidistant obstacles so we decided to sum the effects of individual obstacles as opposed to only using the closest obstacle [17] (6).

$$U_{rep}(q) = \sum_{i=1}^{n} U_{rep,i}(q) \tag{6}$$

$$U_{rep,i}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{d_i(q)} - \frac{1}{Q^*}), & d_i \leq Q^* \\ 0, & d_i > Q^* \end{cases} \tag{7}$$

$$\nabla U_{rep,i}(q) = \begin{cases} \eta(\frac{1}{Q^*} - \frac{1}{d_i(q)})\frac{\nabla d_i(q)}{d_i^2(q)}, & d_i \leq Q^* \\ 0, & d_i > Q^* \end{cases} \tag{8}$$

To avoid undesirable perturbing forces beyond the obstacle's vicinity, the influence of this repulsive field must be limited to a region of radius $Q^* \in \mathbb{R}$ surrounding the obstacle.

3.3 Solving the Local Minima Problem

Numerous approaches with varying levels of difficulty exist to solve the local minima issue. Some of the simplest approaches include: Backtracking from the local minimum and then using another strategy to avoid it or randomised motion at the local minima in the hope this will help escape such minimum. Evidently, these are simple to implement yet become time consuming and impractical in higher order complex configuration spaces. Wave-front planners [17] are also practical for grid based configuration spaces but this was not the case in this excercise. Next, some authors proposed the use of a special class of potential functions such as Navigation Functions [29] and Harmonic Functions for building artificial potential fields which satisfy the mathematical condition $\nabla^T \nabla U = 0$ and thus avoid the local minima problem by only having one local minimum at the goal. However, these solutions are usually limited to simple or restricted class of conservatively bounded obstacles [17], or they need the prior construction of configuration space. Similarly, a set of improved potential functions, which include the concept of Virtual Force Fields have been presented [31] and they have proved to work well in different scenarios although they do not guarantee local minimum avoidance.

The most complete solution would be to augment the potential field approach with a higher-level (global) search based planner (e.g. D*) so that the robot can use the information derived from its sensor, but still plan globally [32]. As mentioned in 2.2.2, these are not efficient in unknown environments.

Having identified implementation simplicity, effectiveness and robustness as key criteria for the selection of a local minima solver, we moved our attention to a class of stochastic global optimisation algorithm named Simmulated Annealing, which was first introduced in [33][34]. In the context of motion planning coupled with artificial potential fields, it has been successfully implemented as demonstrated in [35][36][37][38] to cause the robot to escape from any local minima and converge towards the global minima. The main advantages include the fact that it is *probabilistically complete*, does not involve pre-computation of configuration space or adjacency graphs, it is suitable for robots with many DOFs and requires low spacial memory. As with any stochastic

method however, it can lead to increased flight time as the robot tries to escape the local minima.

3.4 Integration of Simulated Annealing with Potential Fields

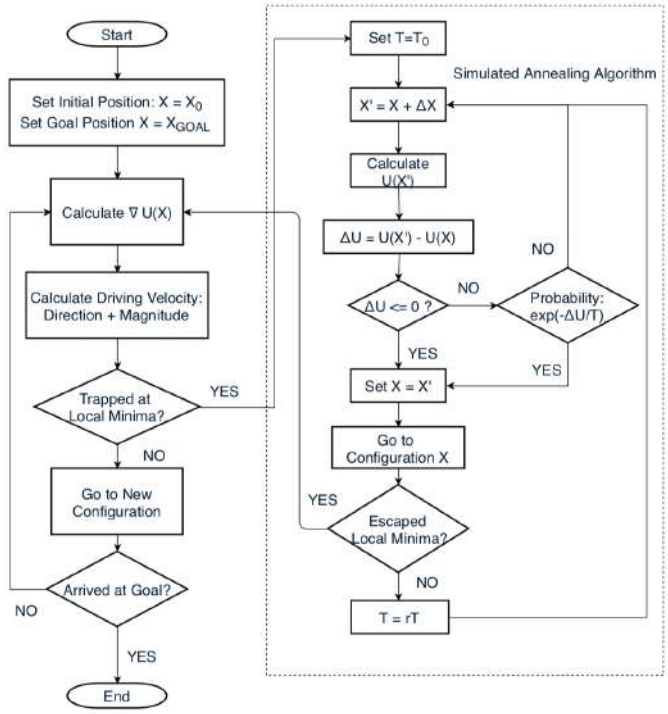


Figure 2: Graphical Representation of Our Integration of Simulated Annealing within Potential Fields Algorithm

Path planning is treated as a global optimisation problem, where the non-convex cost function is a Potential Field constructed in the configuration space and the obstacles represent the constraints.

With the Simulated Annealing approach [37], at each step a new solution X' is chosen randomly from a set of neighbours of the current solution X. The new solution is accepted unconditionally if $U(X') \leq U(X)$ or else with (uphill move) probability of $e^{\frac{-\Delta U}{T}}$. Here U is the cost function (i.e. Potential Function), and T denotes the temperature. If X' is not accepted, the algorithm proceeds to the next step and another new solution is chosen randomly. After each step the temperature is decreased by cooling rate r. This is repeated until a small value near zero is reached or escape from the local minimum has occurred. Therefore the probability that an uphill move will be accepted diminishes as the temperature reduces.

To avoid unnecessary oscillations and reduce flight time, this algorithm is only triggered when the robot detects it is trapped in a local minima and terminates when it considered that it has escaped it (as shown in Figure 2).

3.5 Local Minima Detection & Selection of New Configuration States

Despite the driving force being zero at a local minimum, the dynamics of the drone generally cause the UAV to oscillate around this point. For this reason, a local minimum was detected when the trajectory traced by the robot following a specified number of movements fell within a small specified volume.

To avoid any loss in generality of the environment and incorporate local minima escape in 3D, we adopted spherical coordinates such that the new possible configuration states tested by the Simulated Annealing Algorithm lied within a specified radius and angles: $\theta = \tan^{-1}(\frac{y}{x}), \phi = \cos^{-1}(\frac{z}{r})$. These were constrained by: $\theta \in [0, 2\pi), \phi \in [0, \pi)$ and were subsequently converted to Cartesian coordinates for interpretation in global frame of reference.

4 System Design

4.1 Hardware Architecture

For the experiments we used a custom built UAV with a Pixhawk Flight Controller running the PX4 Flight Stack [40]. This has a built in accelerometer, gyroscope, barometer and magnetometer which become essential for state estimation. An ODroid-XU4 running Ubuntu 16.04 LTS was selected as the companion computer and was in charge of the communication with the autopilot using MAVLink Communation Protocol [41], in particular, its communication node for the ROS environment Mavros [42].

The sensors available included TeraRanger Ones (x2), a TeraRanger Tower [9] (8 x 45° Separated TeraRanger Ones), a PX4FLOW Optical Flow Camera with inbuilt sonar and a small camera for target acquisition using computer vision.

With a range of 0.2-14m, a field of view of 3° and an accuracy of up to $\pm 4cm$, TeraRanger One infrared Time-Of-Flight distance sensors were used for obstacle detection and altitude readings. Postion (x, y, z) and angular position (ϕ, θ, ψ) were obtained via Pixhawk's integration of PX4FLOW optical velocities in (x, y) and its own IMU (Inertial Measuring Unit) for roll, pitch and yaw data. To improve the accuracy of the Optical Flow outputs, we substituted the use of the inbuilt sonar for a TeraRanger One and equipped the drone with a small lighting unit below. Nevertheless, the use of this technology for localisation as opposed for example to GPS leads to a significant accumulation of error. A further discussion of the localisation errors is presented in Section 8.

4.2 Software Architecture

Robot Operating System (ROS) [5] was used as the communication interface between the two on-board controllers. Compatibility with a wide array of peripherals through community developed packages, the fact that it supports Python and C++ and the extensive documentation available make it one of the best platforms for such task. A full plan of the ROS node topology and its integration with the other teams in this project is presented in Section 5. At

this point, it is important to highlight that the TeraRanger Tower was connected directly to ODroid as opposed to Pixhawk due to the required voltage input. As a result, communication to such sensors had to be performed via a separately built node as opposed to Mavros. QGround Control was the software used for calibrating and detailing the sensor fusing strategy for state estimation in Pixhawk.

Lastly, we had to set up the SSH connection between ODroid and the Ground Station computer as to load and run the navigation scripts. The *Imperial-WPA* college network proved unstable in the laboratory environment but more importantly, it assigns dynamic IP addresses, which greatly complicates the connection task to ODroid. To resolve this issue, we introduced a router and built a Local Area Network (LAN). Once the network was set, a static IP reservation protocol was implemented in the router settings for the ODroids MAC Address and the Port Folding/Port Triggering settings were modified to enable SSH connection through a specified port (Port 22).

5 Implementation

5.1 Mission Profile

1. Takeoff & Hover: Both attitude (ϕ, θ, ψ) and position (x, y, z) deviations were corrected by a simple PID algorithm presented in Appendix A.
2. Cruise to Perching Location: This stage was constructed by the author of this report and the Autonoumous Navigation and Obstacle Avoidance tasks were completed by the Potential Fields with Simulated Annealing algorithm presented in Appendix B.
3. Target Acquisition & Perching: This stage began with a grid-based search, which in parallel ran a computer vision algorithm powered by *OpenCV* [43] and *scikit-image* [44]. For practical purposes, the target was set to be bounded by a red rectangle. The pre-processing stage was done on MatLab and involved importing pictures of the target as to find an accurate HSV value for the red target. Once *on-line*, the computer vision algorithm would then filter the real-time images of the target to enhance redness, select the biggest "island" as to remove noise and then calculate the centroid coordinates of the red bounded section of the image. These coordinates relative to the body fixed reference of the UAV were then processed by a PID for velocity outputs that commanded the UAV towards this centroid.
4. Return to Base: This stage was powered by the same Potential Fields with Simulated Annealing algorithm presented in the cruise stage.

5.2 ROS Environment

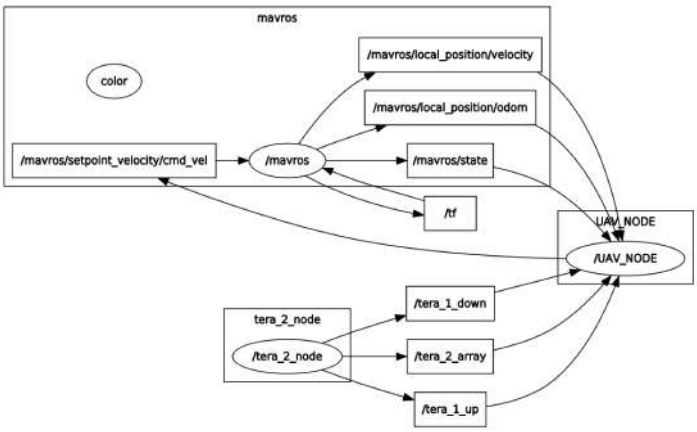


Figure 3: ROS Computation Graph Showing Node Topology and Interconnection via ROS Topics.

Mavros is the main communication ROS node with the Pixhawk autopilot via MAVLink Communication Protocol and it publishes state estimation data. The TeraRanger Node is labeled in Figure 3 by *tera_2_node* and publishes three separate topics with the range data from the differently oriented sensors.

The protocol followed to establish a connection between the TeraRanger sensors and the relevant ROS node was the following. Firstly, the ROS Package for TeraRanger Array Solutions provided by Terabee [48] was cloned and built in the *catkin_ws* ROS workspace in the ODroid. The *setup.bash* file was then sourced so as to add the environment variables of the new ROS package. Lastly, the connection port name was identified and the *tera_2_node* was run during tests. Note that in the real setup, the ROS node names are different.

At execution of the master navigation script, *UAV_NODE* was launched to subscribe to the state estimation and sensor data and command velocity inputs to the autopilot. The Master Navigation Script (Appendix B) was built by the author of this report and served as a backbone for the relevant mission profile tasks. It imports required libraries and message structures, defines key execution parameters, sets a class-based structure, initialises the communication protocols and defines the mission profile in a stage-based flight approach. Autonomous Navigation and Obstacle Avoidance are included in such script, but for flexibility, Perching and Target Acquisition was built separately and called by the master script.

In collaboration with the Power Pack team, we also built the communication protocol for a separate *TETHER_NODE* (Not presented in Figure 3). It's purpose was to subscribe to the *odom* and *velocity* state estimations from *mavros* so as to design the spooling algorithm. Additionally, we collaborated with such team to design the spooling algorithm, which worked on two main principles: spooling back if no load is sensed and trajectory integration when

moving towards the goal.

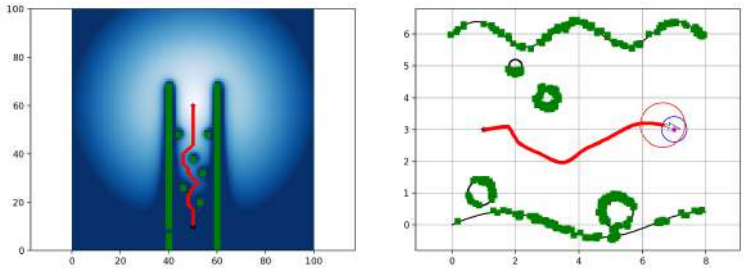
5.3 Potential Fields Algorithm

Firstly, a drone object is initialised from the UAV class, with its starting and goal positions, and physical parameters such as it's maximum radius. The philosophy behind navigation in unknown environments dictates that the drone will incrementally build it's own map by saving the (x, y, z) coordinates of the surrounding obstacles together with its own trajectory. Given the increased interest in research related to multi-robot path planning and coordination [45][46][47], structuring the script in such a classed based approach was thought to be beneficial since, with few modifications, saving the obstacle map as an internal variable within each drone object enables multi-robot autonomous navigation and obstacle avoidance (see Figure 5).

Next, the Potential Field must be calculated following the equations presented in Section 3. The first iteration of the navigation algorithm followed a grid based approach, calculating $U(q) = U_{att}(q) + U_{rep}(q)$ at each of the neighbouring pixels and moving to that which minimises $U(q)$. This results in the potentials map in Figure 4a. However, this proved computationally expensive, required high spatial memory and did not output precise velocity commands. Incorporating vectorised motion (see Figure 4b) where the resultant vector $\nabla U(q) = \nabla U_{att}(q) + \nabla U_{rep}(q)$ is used to determine the new configuration of the robot proved more accurate and computationally less expensive.

While the attractive potential was simple to implement, there are several key features to highlight in the implementation of the repulsive potential:

- Obstacle Detection: As the UAV navigates through the unknown environment, it must fuse the range sensor data received with the its configuration $(x, y, z, \phi, \theta, \psi)$ to populate its obstacle map. The geometric transformations required to map the obstacle locations from the body fixed F.O.R to a world F.O.R are presented in Appendix C.
- Collision Avoidance: As the UAV is not a point particle in space, we must ensure links collision avoidance [25], were links refer for example to the propellers, the perching mechanism or the drone legs. [17] and [25] propose methods based on Jacobian transformations between the local workspace and the global configuration space to ensure collision free motion in a geometrically defined robot with links. To simplify the modeling of the UAV and given that dimensions of some of the links (e.g. perching mechanism) were not available, we took a simplistic approach and defined the UAV as a sphere with radius equal to the maximum of that of the drone (r_{uav}). As a result, $\nabla U_{rep} \to \infty$ as $d_{obs} \to r_{uav}$ and we ensure collision free motion.
- Repulsive Memory: Localisation errors will give rise to the UAV recording a distorted obstacle map. In expectation, the error associated with the relative position of an obstacle will grow in time. To avoid misleading repulsive vectors, it was decided that the UAV would only use the latest *memory_repulsive* detected obstacles, where *memory_repulsive* is a pre-defined scalar value. Its tuning was a balance between the effectiveness and efficiency of obstacle avoidance and the expected localisation error. The final selected value gave the UAV a repulsive memory of $\approx 3s$.



(a) Grid-Based Motion **(b)** Vectorised Motion

Figure 4: Python Simulations Using Grid and Vector Motion

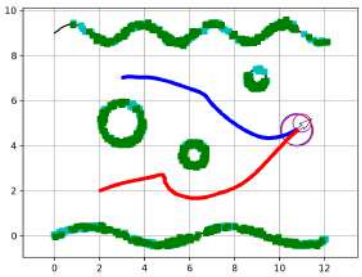


Figure 5: Multi-Robot Navigation. Blue UAV maps cyan obstacles and Red UAV maps green obstacles.

5.4 Simulation Environments

5.4.1 Python Worlds

Initially, a platform to test ideas and navigation algorithms in a time effective and computationally efficient manner was required. Python Worlds (Appendix D) offered a simple and easily modifiable platform for initial debugging and tuning of the Potential Fields algorithm presented in 5.3.

As opposed to Gazebo and Real Setup tests, where the script has no prior knowledge of the environment, in the Python World script, the user defines an environment object with parameters including the map size but most importantly the obstacles and their resolution. In the latest version presented in Appendix D, the obstacles available include: point obstacles, straight horizontal/vertical walls, sinusoidal horizontal/vertical walls with specified amplitude and frequency and circular obstacles. This greatly increased the freedom available to the user and allowed us to test the robustness of the navigation script in many scenarios.

In the same fashion as the ROS-based solution in Appendix B, the script will initialise a drone object from the UAV class. However, there are some notable differences worth mentioning:

1. TeraRanger Scan: Instead of collecting range data from sensors, a search is performed across the user-defined environment obstacle map. Firstly, the obstacle is checked to be within range (0.2-14m) and within line of sight such that the angle between the obstacle and UAV corresponds to that of one of the TeraRanger One sensors. Next, if more than one obstacle is within line of sight, we only record the closest one as to avoid vision across obstacles. If all criteria are satisfied, the obstacle in the user defined environment is appended to the UAV's private obstacle map.
2. UAV Dynamics: Updating position following the resultant velocity command from the *gradient descent* algorithm was simplified by assuming the UAV's velocity was constant and equal to that commanded. A more rigorous model to include inertia is obtained via Gazebo simulations.

In summary, the key features include: Vectorised motion, class based algorithm, imperfect sensors (precision and Gaussian noise incorporated), Local Minima solver via Simulated Annealing and visualisation and data logging capabilities for analysis. The main limitations arise from the simplified dynamics, the assumption of perfect localisation, and the fact that it is a static 2D environment.

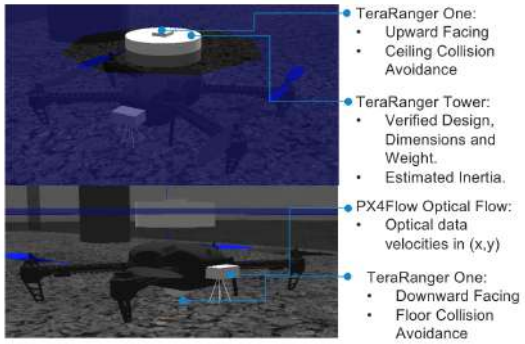


Figure 6: Gazebo Model of Quadcopter Incorporating Sensors Used for Flight Control and Navigation Purposes.

5.4.2 ROS-Gazebo Environment

Gazebo is a powerful 3D simulation environment for autonomous robots due to its advanced and robust physics engine, high quality graphics, high fidelity sensors with added noise and more importantly its open source nature.

Firstly, we began by modeling the quadcopter and the installed sensors. By default, Gazebo's model library incorporates an *iris_opt_flow* model, which includes and accurate representation of a quadcopter with an Optical Flow sensor for state estimation. The remaining 2 x TeraRange One and the TeraRanger tower had to be installed (See Figure 6).

In the interest of facilitating this task for future applications, following is a breakdown of the process, taking as an example the TeraRanger Tower [9]. Relevant scripts are presented in Appendix E.

1. Create Model SDF File: Physical properties such as mass, volume [49], inertia, pose and orientation, connections including links and joints and sensor range and noise models have to be included. For the purposes of initial testing, the LiDAR Noise models were set as Gaussian with a standard deviation of 0.01.
2. Create Plugin Source File: Dynamics and sensor outputs are included. Message passing API to connect with Gazebo also has to be defined.
3. Connection with ROS: As opposed to connecting the sensor to a physical port and ROS launching the corresponding package on that port, the Gazebo interface does not support this. We contemplated different options including modifying the Mavros ROS package and the relevant plugins to include the TeraRanger tower, but we decided against this due to the high risk of damaging such an important package. Next, we investigated using the *Pygazebo* library [50], which supports publishing and subscribing to any Gazebo topics using a straightforward Python API. However, we faced problems with asynchronous programming and message passing strategy being different to that used in the ROS environment.
   Lastly, we opted for a middle-ground solution between the both previously mentioned options: Writing a simultaneous C++ Gazebo subscriber and ROS publisher. Firstly, we set up a new workspace and initialised a ROS package for the new sensors. Next, we built the *listener.cc* C++ script (Appendix E). This file will include all the relevant Gazebo Message Passing, Gazebo-ROS, and ROS library headers together with the relevant message structures for the sensors. It will then initialise a ROS node for the sensors, as seen in the ROS Computation Graph (Figure 3), set up ROS publishers for the sensors and initialise a Gazebo communication node. Finally, it will subscribe to the relevant sensor Gazebo topic and in the CallBack function will process the data and publish it to the relevant ROS topic.
   Once the *listener.cc* was built and sourced, the ROS package Makefile (Appendix E), had to be modified to include Gazebo libraries and to add as an executable the *listener.cc*, such that it was compiled using the *catkin_make* command. Running the newly created ROS node within the sensor ROS package proved very robust.

For the purposes of testing and evaluation of performance, an accurate model of the replica mine constructed by the Manufacturing team was built on

Gazebo. In addition, 10 Gazebo Mine Worlds with increasing levels of difficulty were built. An image of these, together with the defined functionalities to be tested in each is presented in Appendix F.

5.5 Experimental Setup

Pixhawk Autopilot attitude estimation (ϕ, θ, ψ) required fusion of IMU and compass data and this lead to a major problem. In the laboratory environment, Pixhawks's compass suffered strong magnetic interference, thus leading to an unstable world F.O.R and thus causing the UAV to yaw with the changing world frame as to maintain heading. Expecting Pixhawk's world F.O.R to be distorted but stable, we tried to correct the misalignment through a series of transformation matrices presented in Appendix G. Unfortunately, the compass heading drifted significantly during flight.

We tried to reduce the magnetic interference on the compass by increasing the distance between Pixhawk and the battery, shortening cables between the battery and ESCs and aluminum shielding wires and the plate between the Pixhawk and the battery.

Thanks to the *Aerial Robotics Lab*, we were given permission to use the VICON motion capture to set the global heading angle in the same fashion it was used during *L3 Applications*.

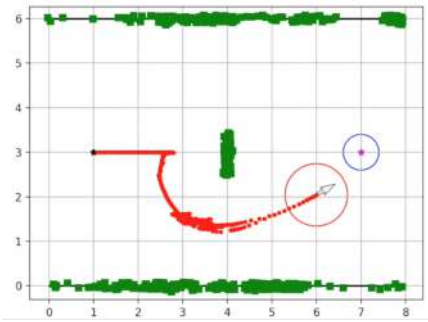


Figure 8: Simulation Results for a Custom Built Python World Including Presence of Local Minimum

6 Experiments and Results

6.1 Simulation Results

6.1.1 Python Worlds

The Potential Fields with Simulated Annealing Algorithm was implemented in Python as described in 5.4.1. Figure 4b represents a general map constructed with circular obstacles and sinusoidal walls. For further analysis, Figure 8 presents the results of a specifically built simulation of a symmetric environment to test the effectiveness of the Local Minima Solver via Simulated Annealing.

6.1.2 ROS-Gazebo Environment

Figure 7 presents the results of the cruise to perching location stage using Potential Fields with Simulated Annealing to avoid 2 curved walls. For a better representation of the effect of Simulated Annealing in ROS-Gazebo environment, Figure 9 presents the trajectory and obstacle mapping in a Gazebo world which triggers the apparition of Local Minima.

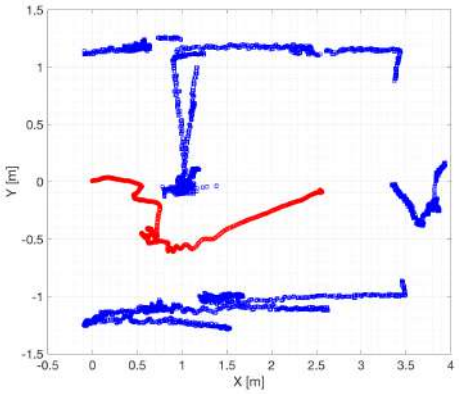
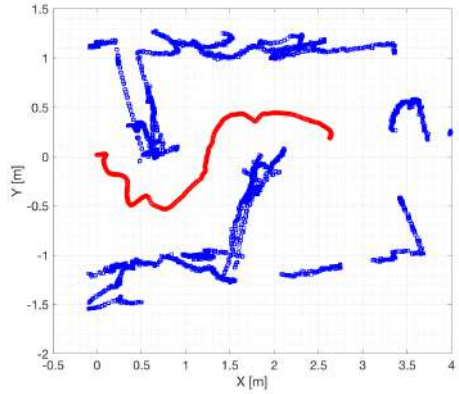


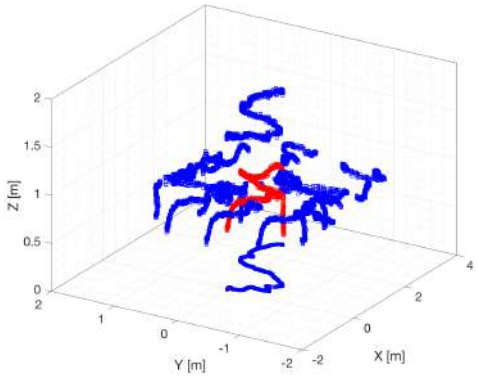
Figure 9: Simulation Results for a Custom Built Gazebo World Including Presence of Local Minimum

6.2 Real Flight Test Results

For the final set of flight tests, whose results are presented in Figure 10 and 11, we integrated the hardware setup described in Section 4 with the perching

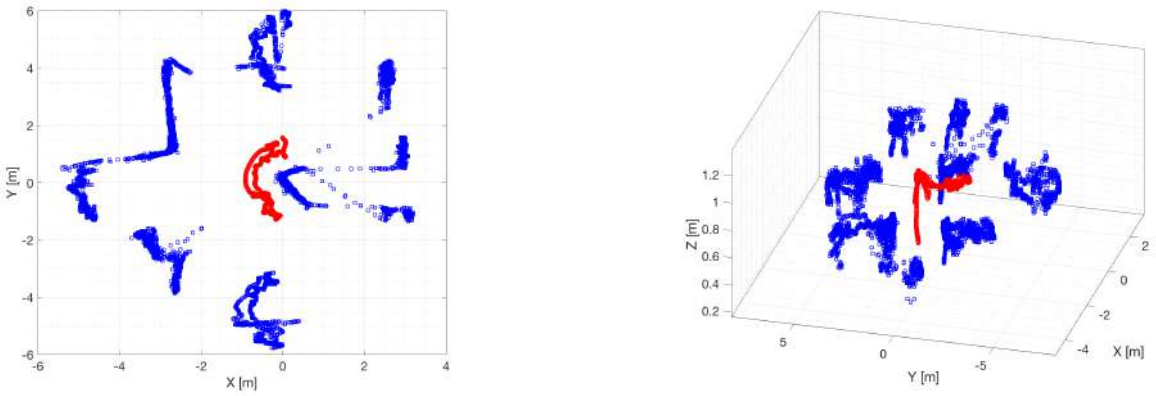


(a) X-Y UAV Trajectory and Detected Obstacles



(b) 3D UAV Trajectory and Detected Obstacles

Figure 7: Simulation Results in Custom Built Gazebo World for using Potential Fields Algorithm. Blue squares represent the mapped obstacles and red dots the trajectory of the UAV.



(a) X-Y UAV Trajectory and Detected Obstacles Using VICON. **(b)** 3D UAV Trajectory and Detected Obstacles Using VICON
Figure 10: Real Flight Test Results For a Single Obstacle Map Using Potential Fields Algorithm. Blue squares represent the mapped obstacles and red dots the trajectory of the UAV.

mechanism designed by the mecatronics team and the power tether system.

On the flight control side, we were unable to install an upward facing TeraRanger One, due to an inability from Pixhawk to differentiate from identically connected sensors. Nevertheless, to mitigate the possible overshoot in height, a fictitious virtual ceiling was implemented in the Potential Fields Algorithm as a new set of detected obstacles. An alternative to this would have been to incorporate a Distance Keeping repulsive potential as opposed to the purely repulsive potential in $z - axis$. Such approach was designed as a backup strategy against the possible delay of the TeraRanger Tower and is presented in Appendix H.

Furthermore, we experienced technical issues with the setup of PX4 Optical Flow, and as to maximise the utility of the time in the Flight Arena, it was decided to test the control algorithm using VICON for position estimation and heading angle. Note that grid-based search and computer vision were not tested in the Flight Arena due to issues with the setup of the test environment.

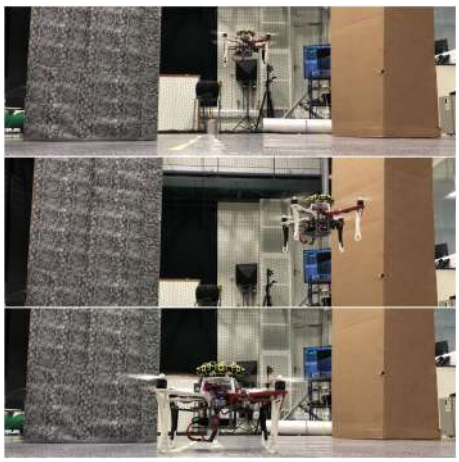


Figure 11: Images of Flight Test in a Environment With 2 Obstacles. Top two images show first and second obstacle avoidance maneuvers.

7 Path Planning Optimisation

Classic Artificial Potential Fields [25] has well-known drawbacks including: local minima problems, no passage between closely spaced obstacles and oscillations between obstacles. To overcome such shortcomings and as to improve the efficiency of the path planning algorithm, we attempted to tune the Potential Fields, Gradient Descent and Simulated Annealing parameters. This would aid to avoid any uncertaincy caused collisions in such a complex non-linear system while giving the drone the flexibility and agility required to navigate through closely spaced obstacles. But first, we defined the quantitative objective functions to minimise and the qualitative evaluation criteria.

On the quantitative side, we aimed to minimise the resultant path length and a function defined as *Danger Index*, which is the sum of the inverse of the distances to the closest obstacle at each time step (9).

$$f(x_i) = \sum_{t=0}^{T} \frac{1}{d_{min,i,t}} \tag{9}$$

Quatilatively, we recorded the aggressiveness of the response, the presecence of any instabilities / oscillations and whether the UAV could pass through closely spaced obstacles.

At first, we began with manual tuning, offering a heuristic first approach to "optimal" parameters. A full list of the parameters tuned with their respective tuning decisions is presented in Appendix I.

At this point, it is worth mentioning the high sensitivity of the Simulated Annealing parameters, mainly the initial temperature T_0 and the cooling rate r. Simulated Annealing should only serve as an additional help to escape the local minima and for this reason, both T_0 and r had to be kept low as to minimise the likelihood of uphill moves, which generally involving getting closer to an obstacle. Furthermore, it was observed that using an increased repulsive gain

(η) exclusively for Simulated Annealing was beneficial to reduce the likelihood of collisions.

Next, we moved our attention to more sophisticated tuning strategies, which could offer better results in this multi-objective and multi-parameter optimisation problem. In particular, we investigated Particle Swarm Optimisation (PSO) [53][54] and Differential Evolution [55], which belong to a class of stochastic population based methods known as evolutionary algorithms.

With the help of the Python SciPy library [56], we implemented a simple single-objective multi-parameter optimisation for potential fields attractive and repulsive gains using Differential Evolution. This involved building an *optimiser.py* script and the corresponding *bash_training.sh* bash script which would initialise the ROS-Gazebo environment, run a simulation with the parameters dictated by Differential Evolution and record the results. Both scripts are presented in Appendix J. Despite the optimised results being relatively close to those predicted manually, it was quickly identified that such methods require large computational power and remain impractical unless simulations could be run in parallel. To alleviate this issue, in [54], Artificial Neural Networks are suggested to infer step-by-step variations of the multiple inputs with the swarm actual trajectory and thus lower the number of simulations.

8 Discussion

8.1 Interpretation Of Results

Despite the critical limitations of the Python World presented in 5.4.1, we can observe from Figure 4b that even with the addition of noise, the trajectory generated was smooth and satisfactory. With regards to the implementation of the Simulated Annealing Local Minima Solver presented in 3.4, we can observed from Figure 8 that the heuristic nature of this algorithm leads to random walks which increase running time. Furthermore, some undesired oscillatory motion was noticed. Nevertheless, it was successful at backtracking from the local minimum, finding an alternative route and helping solve one of the drawbacks of potential fields: passage across closely space obstacles.

The results obtained from the ROS-Gazebo environment presented in Figures 7 and 9 are promising, with Figure 7 proving successful navigation through more complex environments when the goal is not readily visible. While the general overview of the map is successfully recorded, the apparent distortion is due to the localisation errors introduced by the state estimating sensors, namely PX4Flow and the IMU. The position of the detected obstacle with respect to the UAV is accurate but the exact position of the drone in the environment has not been corrected by a localisation algorithm. Regarding the Simulated Annealing implementation in ROS, it can be noted from Figure 9 that, despite its effectiveness, it lead to slightly aggressive maneuvers which reduced the safety margin distance with the obstacles, thus suggesting further tuning of the annealing parameters is required. To alleviate any dangerous movements, fast cooling rates (r) and low initial temperatures (T_0) should be introduced, but as discussed in [35], this may lead to "premature freezing" and the local minima not being escaped.

Real flight tests began with a takeoff stage, which was very successfully controlled in $(x, y, z, \phi, \theta, \psi)$ by a well tuned PID, leading to a smooth takeoff and hover. As demonstrated by Figures 10 and 11, the potential fields autonomous navigation and collision avoidance algorithm was very effective, maintaining good safety margins and being able to computationally process the real time map generation and obstacle avoidance. Stability during flight was satisfactory, suggesting the potential field and gradient descent parameters were well tuned. Nevertheless, this stability and the precision of movements was greatly influenced by the use of VICON. In further tests, this should be compared to the results using only onboard sensors. Figure 10 shows impressive and dense mapping, whose precision is impulsed again by the use of VICON, as opposed to the distorted maps presented from Gazebo in Figure 7. Regardless, this suggests that given more time, SLAM algorithms could be implemented without the acquisition of additional sensors. Due to the "perfect" (VICON) against imperfect (Gazebo) localisation difference, a more detailed analysis of the noise models is not readily obvious. However, it should be noted that the noise levels introduced in Gazebo were validated via static tests on a single TeraRanger One Time-Of-Flight Sensor.

8.2 From Simulation to Experiments

In terms of hardware, several issues can be noted. The quality of the IMU angular position estimates was affected by the integration of angular rate from gyroscope, leading to an accumulation of error. As mentioned in 6.2, technical issues with the PX4Flow sensors meant we were unable to test them, and together with the heading issues associated with the compass in Pixhawk suggest that this experiment should be validated, possibly outdoors where both sensors should work optimally. Lastly, the mounting of the hardware on the UAV platform should be revisited as to minimise any de-stabilising large moments of inertia and lower the centre of gravity.

In terms of software, it should be noted that the physics engine in Gazebo has not been optimised for aerial robotics. It assumes perfectly rigid body dynamics, the torque produced by motors and pro-

pellers as described mathematically in [58] are not modeled, and most of the aerodynamic effects presented in [59] are not modeled. These include: possible lag in thrust generation, loss in the aerodynamic efficiency of the propellers, drag apparition and the effect of ground, ceiling and wall plates interacting with the propeller generated wing. In particular, what is known as *ground effect* is due to the rotor wake rapidly expanding as it approaches the surface, transitioning from the almost vertical downwash to radial outwash parallel to the ground and thus affecting rotor thrust and power [59]. Additionally, flat plate approximations are taken for lifting surfaces.

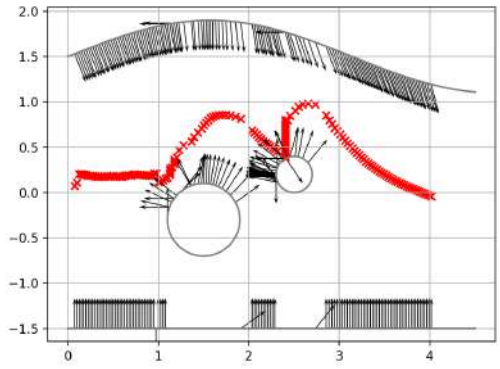


Figure 12: Simulation Results for Feature Extracting Potential Fields Algorithm in Python World

8.3 Improvements & Future Work

We begin by reviewing the autonomous navigation and obstacle avoidance algorithms employed. In our implementation of the Artificial Potential Fields Algorithm, we used laser range data to construct only purely repulsive and attractive point fields. While this approach offers increased simplicity, a number of more complex potential fields including perpendicular or uniform fields [60][61] can be implemented to address different needs. In particular, a more rigorous and robust solution to the problem of constructing a repulsive vector from an obstacle could be obtained by modeling the surface and constructing the repulsive vector using the gradient of such surface.

Within the Python World, we attempted to construct a 2D segmentation and feature extraction algorithm for laser range data which, using the data at consecutive time steps could calculate the gradient and hence construct the normal repulsive vector to a surface. As shown in Figure 12, the gradient finding strategy was generally successful, but the implementation within the Potential Fields algorithm was not robust, leading to occasional collision with obstacles. This was due to many reasons, including the sparse sensing and the complexity of the script, which had to handle many scenarios. Most importantly however, the addition of noise destabilised the system and the curvature of any features was difficult to extract.

For this reason, it is necessary to perform a laser range data segmentation, which can cluster the range data points into a set of groups representing environmental features and then construct a segments map. Some of the methods reviewed in literature include: Point-Distance Based (PDBS) and Kalman Filter Based (KFBS) [62] [63]. This task would additionally aid the localisation task, thus bringing us closer to the implementation of a SLAM algorithm. This is because SLAM algorithms use laser scans to correct the estimated positions given by odometry using for example Extended Kalman Filters (EKF). Overall, the general scheme of the SLAM process [10] can be summarised as: Landmark extraction, data association, state estimation, state update and landmark update.

An alternative to handle feature extraction and landmark detection for localisation purposes would be the use of computer vision [63]. Within the timeframe allocated to this project, this was not considered feasible.

In many real-life implementations where the goal is very close to an obstacle, classical Potential Fields [25] frequently suffers from the well documented *Goal Non-Reachability due to Obstacle Nearby* (GNRON) problem. Essentially, if the repulsive action is much larger than the attractive one, the goal is not a minimum of the total potential function. By taking into consideration between the robot and goal configurations, [57] presents a robust modified repulsive vector to solve this issue:

$$U_{rep,i}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{d_{obs,i}(q)} - \frac{1}{Q^*})d_{goal,i}^n, & d_{obs,i} \leq Q^* \\ 0, & d_{obs,i} > Q^* \end{cases} \tag{10}$$

Appendix K analyses the great effectiveness of its implementation in the Gazebo environment. However, to avoid collisions when navigating close to the obstacle, it was not tested in the real setup.

Obstacle detection in the $z - axis$ was also limited, given that we only had a single downward facing TeraRanger One. A feasible solution would be to rearrange the position of the TeraRanger Ones within the tower as to expand the field of view in this axis.

Lastly, we attempted to model the effect of the tether system on the dynamics of the quad-copter. Approaches included modeling a force vector towards the origin when the tether was taught or a weight/unit length when the tether was slack. While in practice the solution would be between those two, the main limitation we suffered was the modeling of such dynamics forces in the Gazebo environment, which requires the modification of plugins for the dynamic handling of forces.

9 Conclusion

From a flight control perspective, the autonomous navigation and obstacle avoidance tasks have been demonstrated to work both in simulation and real flight tests. However, the overall capabilities of the UAV have been partially showcased, with the target acquisition strategy using computer vision working independently from the navigation. Component integration and the use of onboard sensors for state estimation also proved challenging and must be tested in a future to validate the robustness of the proposed navigation algorithms.

In particular, this report has addressed a range of topics, including the categorisation and selection of path planning algorithms, the sensor fusion strategy for obstacle avoidance, the solving of the potential fields local minima problem and the simulation and real flight test case studies.

The key difference between human and robot navigation remains the quantum difference in perceptual capability. Human can detect, classify, and identify environmental features under different environmental conditions. The presented robot has limited perceptual and decisional capabilities in detection and avoidance while moving in obstacle environments. For this reason, we began investigating the process of laser range data segmentation, feature extraction and subsequent map generation, which would bring us closer to obtaining a Simultaneous Localisation and Mapping (SLAM) algorithm.

Lastly, while Simulated Annealing proved a valid and effective Local Minima solver, the sensitivity to parameter tuning and the aggressiveness of the response in different scenarios suggest that a comparison against other methods, mainly the use of Virtual Force Methods [31] has to be performed.

References

[1] Nomsa Maseko, *The Human Cost Of South Africa's Mining Industry.* BBC News, 02 Nov 2015. Available from: https://www.bbc.co.uk/news/av/world-africa-34696669/the-human-cost-of-south-africa-s-mining-industry [Accessed 8th June 2018].

[2] Deloitte, *Tracking the Trends 2018: The Top 10 Issues Facing Mining in the Year Ahead.* Published in 2018. Available from: http://www.mining.com/wp-content/uploads/2018/01/Deloitte-Tracking-the-Trends-Global-Mining-Study-FINAL.pdf [Accessed 8th June 2018].

[3] EY, *Business Risks Facing Mining and Metals 2017-2018.* Published in 2018. Available from: http://www.ey.com/gl/en/industries/mining—metals/business-risks-in-mining-and-metals [Accessed 8th June 2018].

[4] Gazebo Robot Simulation Main Website. Available from: http://gazebosim.org [Accessed 8th June 2018].

[5] Robot Operating System (ROS) Main Website. Available from: http://www.ros.org [Accessed 8th June 2018].

[6] Open Source for Drones - PX4 Open Source Autopilot. Available from: http://px4.io [Accessed 8th June 2018].

[7] Frans Knox - Head of Production - Mining BMA, *How Drones Are Changing Mining.* Available from: https://www.bhp.com/media-and-insights/prospects/2017/04/how-drones-are-changing-mining [Accessed 8th June 2018].

[8] Sandrine Ceurstemont, *Flying and Rolling Drone will Map Underground Mines On its Own.* News Scientist, 28 September 2017. Available from: https://www.newscientist.com/article/2148877-flying-and-rolling-drone-will-map-underground-mines-on-its-own/ [Accessed 8th June 2018].

[9] TeraRanger Tower System Description. Available from: https://www.terabee.com/portfolio-item/teraranger-tower-scanner-for-slam-and-collision-avoidance/ [Accessed 8th June 2018].

[10] T. Bailey and H. F. Durrant-Whyte. Simultaneous Localisation and Mapping (SLAM): Part I. Robotics and Autonomous Systems (RAS).

[11] K. R. Beevers and W. H. Huang, *"SLAM with sparse sensing,"* Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006., Orlando, FL, 2006, pp. 2285-2290. doi: 10.1109/ROBOT.2006.1642043

[12] Jean-Claude Latombe. 1991. *Robot Motion Planning.* Kluwer Academic Publishers, Norwell, MA, USA.

[13] Robin R. Murphy. 2000. Introduction to AI Robotics (1st ed.). MIT Press, Cambridge, MA, USA.

[14] J. Kuffner Jr and J.-C. Latombe. *Interactive manipulation planning for animated characters.* In Computer Graphics and Applications, 2000. Proceedings. The Eighth Pacific Conference on, pages 417–418. IEEE, 2000.

[15] R. H. Taylor and D. Stoianovici. *Medical robotics in computer-integrated surgery.* Robotics and Automation, IEEE Transactions on, 19(5):765–781, 2003.

[16] S. M. LaValle. *Planning algorithms.* Cambridge university press, 2006.

[17] Choset, H., Lynch K., Hutchison S., Kantor G., Burgard W., Kavraki L., Thrun S. (2004). Principles of robot motion : Theory, algorithms, and implementation (Intelligent robots and autonomous agents). Cambridge, Mass ; London: MIT.

[18] Aurenhammer F. *Voronoi Diagrams - A survey of fundamental geometric structure.* ACM Computing Surveys, 23:345-405, 1991.

[19] Dragan A. *CS 294-115 Algorithmic Human-Robot Interaction*, Department of Electrical Engineering and Computer Sciences at UC Berkeley, Fall 2017.

[20] C. Saranya, K. Koteswara Rao, Manju Unnikrishnan, Dr. V. Brinda, Dr. V.R. Lalithambika and M.V. Dhekane, *Real Time Evaluation of Grid Based Path Planning Algorithms: A comparative study.* Third International Conference on Advances in Control and Optimization of Dynamical Systems March 13-15, 2014. Kanpur, India.

[21] Pradipta D., Konar A., Laishram R. *Path Planning of Mobile Robot in Unknown Environment.* Manipur Institute of Technology Imphal, Manipur-795001, India.

[22] O.B.P.M. Rodenberg, E. Verbree, S. Zlatanova, *Indoor A* Pathfinding Algorithm through an Octree Representation of a Point Cloud.* ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume IV-2/W1, 2016 11th 3D Geoinfo Conference, 20–21 October 2016, Athens, Greece.

[23] Carsten J., Ferguson D., Stentz A., *3D Field D*: Improved Path Planning and Replanning in Three Dimensions* Carnegie Mellon University Pittsburgh, PA, 2006.

[24] Karaman S., Frazzoli E. *Sampling-based Algorithms for Optimal Motion Planning*, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA.

[25] Khatib O., *Real-Time Obstacle Avoidance for Manipulators and Mobile Robots.* Artificial Intelligence Laboratory Stanford University Stanford, California 94305. The International Journal of Robotics Research, Vol. 5, No. 1, Spring 1986.

[26] Sabudin E. N, Omar. R and Che Ku Melor C. K. A. N. H, *Potential Field Methods and their Inherent Approached for Path Planning.* Faculty of Electrical and Electronic Engineering, Universiti Tun Hussein Onn Malaysia, Parit Raja, Batu Pahat, Johor, Malaysia.

[27] Canny J.F., Lin M.C., *An opportunistic global path planner*, IEEE Int. Conf. on Robotics and Automation, 1554-1559 (1990).

[28] J. Canny. *The complexity of robot motion planning.* The MIT press, 1988.

[29] Rimon, E. and Koditschek, D. E, *Exact robot Navigation in geometrically complicated but topologically simple spaces*, Proc. IEEE Int. Conf. Robotics and Automation, Cincinnati, OH, 1990, pp. 1937-1942.

[30] Kim J., Khosla P., *Real-time Obstacle Avoidance Using Harmonic Potential Functions.* IEEE Trans. On Robotic and Automation, vol 8, no 3, 338-349 - 1992.

[31] Borenstein J. and Koren Y., *Real-Time Obstacle Avoidance for Fast. IEEE trans. Syst. Man, Cybern. - Part A*, Syst. Humans, vol. 19, no. 5, pp. 1179-1187 - 1989.

[32] Dudek G. and Jenkin M. ,*Computational principles of mobile robotics*, Cambridge University Press, Cambridge, 2000, ISBN: 0-521-56021-7.

[33] Kirkpatrick, S., Gellat, C. D., and Vecchi, M. P., *"Optimization by Simulated Annealing,"* Science, Vol. 220, No. 4598, May 1983, pp. 671-680.

[34] Cemy, V., *"Thermodynamical approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm,"* J. Optimization Theory and Applications, Vol. 45, NO. 1, 1985, pp. 41-51.

[35] F. Janabi-Sharifi and D. Vinke, *Integration of the artificial potential field approach with simulated annealing for robot path planning,* Proceedings of the 1993 IEEE International Symposium on Intelligent Control, pp. 536-541, 1993.

[36] M.G. Park, M.C. Lee. *Experimental Evaluation of Robot Path Planning by Artificial Potential Field Approach with Simulated Annealing*, SICE 2002. Aug.5-7, 2002, Osaka.

[37] Janabi-Sharifi F., Vinke D., *Integration of the Artificial Potential Field Approach with Simulated Annealing for Robot Path Planning.* Department

of Electrical and Computer Engineering University of Waterloo Waterloo, Ontario, 1993.

[38] Qidan Z. , Yan Y., Xing Z., *Robot Path Planning Based on Artificial Potential Field Approach with Simulated Annealing.* College of Automation, Haerbin Engineering University, 2006.

[39] Juliá M., Gil A., Payá L., Reinoso O., *Local Minima Detection in Potential Field Based Cooperative Multi-robot Exploration* Systems Engineering Department, Miguel Hernández University, 03202 Elche (Alicante), Spain.

[40] PixHawk Flight Controller Home Website. Available from: https://pixhawk.org [Accessed 12th June 2018].

[41] MAVLink Micro Air Vehicle Communication Protocol. Available from: http://qgroundcontrol.org/mavlink/start [Accessed 12th June 2018].

[42] Mavros ROS Wiki Main Website. Available from: http://wiki.ros.org/mavros [Accessed 12th June 2018].

[43] OpenCV (Open Source Computer Vision Library) Main Website. Available from: https://opencv.org [Accessed 13th June 2018].

[44] scikit-image (Open Source Image Processing Python Library) Main Website. Available from: http://scikit-image.org [Accessed 13th June 2018].

[45] Sung-hwan Kim, Gyungtae Lee, Inpyo Hong, Young-Joo Kim and Daeyoung Kim, *New potential functions for multi robot path planning : SWARM or SPREAD,* 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE), Singapore, 2010, pp. 557-561. doi: 10.1109/ICCAE.2010.5451658

[46] Raffaele Grandi, Riccardo Falconi, Claudio Melchiorri, A Navigation Strategy for Multi-Robot Systems Based on Particle Swarm Optimization Techniques, IFAC Proceedings Volumes, Volume 45, Issue 22, 2012, Pages 331-336, ISSN 1474-6670, ISBN 9783902823113, https://doi.org/10.3182/20120905-3-HR-2030.00060.

[47] D. Sun, A. Kleiner and B. Nebel, "Behavior-based multi-robot collision avoidance," 2014 IEEE International Conference on Robotics and Automation (ICRA), Hong Kong, 2014, pp. 1668-1673. doi: 10.1109/ICRA.2014.6907075.

[48] ROS package for TeraRanger array solutions by Terabee - GitHub Library. Available from: https://github.com/Terabee/teraranger_array [Accessed 13th June 2018].

[49] TeraRanger Tower Specifications Sheet. Available from: https://www.terabee.com/wp-content/uploads/2017/09/Towerspecificationsheet-1.pdf [Accessed 14th June 2018].

[50] Pygazebo Python Library GitHub Main Page. Available from: https://github.com/jpieper/pygazebo [Accessed 14th June 2018].

[51] Palacios R., *AE3-401 Advanced Mechanics of Flight. Part II - Atmospheric Flight Dynamics.* Imperial College London, Department of Aeronautics, South Kensington Campus, SW7 2AZ, UK, 2017/2018.

[52] VICON Motion Capture Systems - Main Website. Available from: https://www.vicon.com [Accessed 14th June 2018].

[53] Zhiyu Z., Junjie W., Zefei Z., Donghe Y., Jiang W., *Tangent navigated robot path planning strategy using particle swarm optimized artificial potential field*,Optik,Volume 158,2018,Pages 639-651,ISSN 0030-4026, https://doi.org/10.1016/j.ijleo.2017.12.169.

[54] Furferi R., Conti R., Meli E., Ridolfi A., *Optimization of potential field method parameters through networks for swarm cooperative manipulation tasks*, International Journal of Advanced Robotic Systems. https://doi.org/10.1177/1729881416657931. First Published November 28, 2016

[55] Storn R., Price K., *Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces.* Journal of Global Optimization 11: 341–359, 1997. Kluwer Academic Publishers.

[56] Python SciPy Library Main Website. Available from: https://www.scipy.org [Accessed 15th June 2018].

[57] S. S. Ge and Y. J. Cui, *New potential functions for mobile robot path planning*, in IEEE Transactions on Robotics and Automation, vol. 16, no. 5, pp. 615-620, Oct 2000. doi: 10.1109/70.880813.

[58] Wei D., Guo-Ying G., Xiangyang Z., Han D., *Modeling and Control of a Quadrotor UAV with Aerodynamic Concepts.* World Academy of Science, Engineering and Technology International

Journal of Aerospace and Mechanical Engineering Vol:7, No:5, 2013.

[59] Sanchez-Cuevas P., Heredia G-, Ollero A., *Characterization of the Aerodynamic Ground Effect and Its Influence in Multirotor Control.* International Journal of Aerospace Engineering Volume 2017, Article ID 1823056, 17 pages https://doi.org/10.1155/2017/1823056.

[60] Abdullah M., *Mobile Robot Navigation using potential fields and market based optimization.* International Master's Thesis, Department of Technology at Örebro University, 2013.

[61] W.M.M. Tharindu Weerakoon, *Artificial Potential Field and Feature Extraction Method for Mobile Robot Path Planning in Structured Environments.* Department of Brain Science and Engineering, Graduate School of Life Science and Systems Engineering, Kyushu Institute of Technology. 2016.

[62] T. Weerakoon, K. Ishii and A. A. F. Nassiraei, *Geometric feature extraction from 2D laser range data for mobile robot navigation*, 2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS), Peradeniya, 2015, pp. 326-331. doi: 10.1109/ICIINFS.2015.7399032

[63] S. Wang, *A Review of Gradient-Based and Edge-Based Feature Extraction Methods for Object Detection,* 2011 IEEE 11th International Conference on Computer and Information Technology, Pafos, 2011, pp. 277-282. doi: 10.1109/CIT.2011.51.

A Appendix - Takeoff Script

```python
#!/usr/bin/env python
# Takeoff Script - PID Controlled Flight
# Author: Pablo Hermoso Moreno & Roman Kastusik

# import libraries:
import rospy
import math
import sys
import time
import numpy as np

# ROS Message Structures

from mavros_msgs.msg import State  # import state message structure
from sensor_msgs.msg import Range  # import range message structure
from sensor_msgs.msg import LaserScan  # import LaserScan message structure
from geometry_msgs.msg import TwistStamped  # used to set velocity messages
from nav_msgs.msg import Odometry  # import range message structure
from mavros_msgs.srv import *  # import for arm and flight mode setting

from tf.transformations import euler_from_quaternion  # angle transformation


updrate = float(20)

# Initialise Parameters
range_ground = 0  # Range From Ground [m]
range_ceiling = 0  # Range From Ceiling [m]
velocity = np.array([0, 0, 0])
angpos = np.array([0, 0, 0])
position = np.array([0, 0, 0])
hover_time = 5


tookoff = 0
loopcount = 0

i=0
currentpos=[0,0,0]
previouspos=[0,0,0]
is_hovering=0
has_started_flying=0
goto1st=0
goto2nd=0
check=[0]
firstphase1run=1
counter=0
j=0


# Setpoint - This enables the drone to correct for initial deviations from the [0,0,0] in the
environment setup.

setpoint = np.array([0,0,0.6])

################################################################################

# ROS CLASSES:


class velControl:
    def __init__(self, attPub):  # attPub = attitude publisher
        self._attPub = attPub
        self._setVelMsg = TwistStamped()
        self._targetVelX = 0
        self._targetVelY = 0
        self._targetVelZ = 0

        self._targetAngVelX = 0
        self._targetAngVelY = 0
        self._targetAngVelZ = 0

    def setVel(self, coordinates):
        self._targetVelX = float(coordinates[0])
        self._targetVelY = float(coordinates[1])
        self._targetVelZ = float(coordinates[2])
        print(coordinates)

    def setAngVel(self, coordinates):
        self._targetAngVelX = float(coordinates[0])
        self._targetAngVelY = float(coordinates[1])
        self._targetAngVelZ = float(coordinates[2])
```

```
    def publishTargetPose(self, stateManagerInstance):
        self._setVelMsg.header.stamp = rospy.Time.now()  # construct message to publish with time,
loop count and id
        self._setVelMsg.header.seq = stateManagerInstance.getLoopCount()
        self._setVelMsg.header.frame_id = 'fcu'

        self._setVelMsg.twist.linear.x = self._targetVelX
        self._setVelMsg.twist.linear.y = self._targetVelY
        self._setVelMsg.twist.linear.z = self._targetVelZ

        self._setVelMsg.twist.angular.x = self._targetAngVelX
        self._setVelMsg.twist.angular.y = self._targetAngVelY
        self._setVelMsg.twist.angular.z = self._targetAngVelZ

        self._attPub.publish(self._setVelMsg)


class stateManager:  # class for monitoring and changing state of the controller
    def __init__(self, rate):
        self._rate = rate
        self._loopCount = 0
        self._isConnected = 0
        self._isArmed = 0
        self._mode = None

    def incrementLoop(self):
        self._loopCount = self._loopCount + 1

    def getLoopCount(self):
        return self._loopCount

    def stateUpdate(self, msg):
        self._isConnected = msg.connected
        self._isArmed = msg.armed
        self._mode = msg.mode
        #rospy.logwarn("Connected is {}, armed is {}, mode is {} ".format(self._isConnected, self.
_isArmed,
                                                                        # self._mode))  # some status
info

    def armRequest(self):
        rospy.wait_for_service('/mavros/set_mode')
        try:
            modeService = rospy.ServiceProxy('/mavros/set_mode',
                                              mavros_msgs.srv.SetMode)  # get mode service and set to
offboard control
            modeService(custom_mode='OFFBOARD')
        except rospy.ServiceException as e:
            print("Service mode set faild with exception: %s" % e)

    def offboardRequest(self):
        rospy.wait_for_service('/mavros/cmd/arming')
        try:
            arm = rospy.ServiceProxy('/mavros/cmd/arming',
                                     mavros_msgs.srv.CommandBool)  # get arm command service and arm
            arm(True)
        except rospy.ServiceException as e:  # except if failed
            print("Service arm failed with exception :%s" % e)

    def waitForPilotConnection(self):  # wait for connection to flight controller
        #rospy.logwarn("Waiting for pilot connection")
        while not rospy.is_shutdown():  # while not shutting down
            if self._isConnected:  # if state isConnected is true
                #rospy.logwarn("Pilot is connected")
                return True
            self._rate.sleep
        #rospy.logwarn("ROS shutdown")
        return False


################################################################################

# CONTROL CLASSES:

class PID:
    def __init__(self,P,I,D,_integrator=0,_derivator=0,integrator_max=500,integrator_min=-500):
        self.Kp=P
        self.Ki=I
        self.Kd=D
        self.Derivator=_derivator #stores derivative of the error --> multiply this by Kd
        self.Integrator=_integrator #stores integral of the error --> multiply this by Ki
```

```
        self.set_point=0.0
        self.error=0.0 # --> multiply this by Kp
        self.velToSet=0

    def setPoint(self,set_point):
    self.set_point = set_point
    self.Integrator=0
    self.Derivator=0

    def update(self,position):
        self.error=self.set_point - position
        #set the P, I and D
        self.propTerm=self.Kp*self.error
        self.derTerm=self.Kd*(self.error-self.Derivator)
        self.Derivator=self.error #for the next loop

        self.Integrator=self.Integrator + self.error
        self.intTerm=self.Ki*self.Integrator
        self.velToSet=self.propTerm+self.intTerm+self.derTerm
        return self.velToSet

    def getSetpoint(self) :
        return self.set_point



##############################################################################


# CALLBACK FUNCTIONS:

def distanceCheck(msg):
    global range_ground  # import global range
    # print(msg.range)  # for debugging
    range_ground = msg.range  # set range = recieved range


def teraArrayCheck(msg):
    global range_tera_array

    # GAZEBO SIMULATION ENVIRONMENT
    # Understanding Message Structure of "msg.ranges"
    # Degrees : [180,135,90,45,0,315,270,225,-] - Saved as GLOBAL Array
    # IMP : Real Setup No 180deg
    range_tera_array = msg.ranges[1:7]

def teraUpCheck(msg):
    global range_ceiling
    range_ceiling = msg.ranges[0]

def teraDownCheck(msg):
    global range_ground
    range_ground = msg.ranges[0]

def odomCheck(msg):
    global position
    global angpos
    global range_ground
    angpos = euler_from_quaternion([msg.pose.pose.orientation.x, msg.pose.pose.orientation.y,
                                    msg.pose.pose.orientation.z, msg.pose.pose.orientation.w])
    xPosition = msg.pose.pose.position.x
    yPosition = msg.pose.pose.position.y
    zPosition = msg.pose.pose.position.z

    #zPosition = range_ground * math.cos(math.radians(angpos[1])) * math.cos(math.radians(angpos[0]))

    position = np.array([xPosition, yPosition, zPosition])

def velCheck(msg):
    global velocity
    xVel = msg.twist.linear.x
    yVel = msg.twist.linear.y
    zVel = msg.twist.linear.z
    velocity = np.array([xVel, yVel, zVel])

############################################################## FUNCTIONS RUNNING IN MAIN
 ##########################################################################

def startup() : # Getting off the ground phase
    global goto1st
    global has_started_flying
    global xpositionControl014
```

```
    global zpositionControl014
    global yawControl
    global xVelocity
    global yVelocity
    global zVelocity
    global rollRate
    global pitchRate
    global yawRate
    global counter
    global position

    global setpoint

    if position[2]<=0.2 :
        counter=counter+1
        xpositionControl014.setPoint(setpoint[0]) #setting them within the while loop as it doesnt
really matter where we do it
        zpositionControl014.setPoint(setpoint[2]) #this only changes in phase4()
        ypositionControl.setPoint(setpoint[1])
        yawControl.setPoint(0.0) #never changes
        rollControl.setPoint(0.0)
        pitchControl.setPoint(0.0)

    else :
        goto1st=1
        has_started_flying=1
    xVelocity = xpositionControl014.update(position[0]) #need to update xVelocity in each stage
seperately as some use PID and others don't.
    yVelocity = ypositionControl.update(position[1])
    zVelocity = zpositionControl014.update(position[2])
    if zVelocity>0.3:
        zVelocity=0.3
    rollRate = yawControl.update(angpos[0])
    pitchRate = yawControl.update(angpos[1])
    yawRate = yawControl.update(angpos[2])

def phase1() : # HOVER at 0.6m before terminating stage.
    #global posxx
    global goto1st
    global goto2nd
    global currentime
    global prevtime
    global currentpos
    global previouspos
    global check
    global i
    global is_hovering
    global firstphase1run
    global xpositionControl014
    global xVelocity
    global zVelocity
    global yVelocity
    global rollRate
    global pitchRate
    global yawRate
    global j
    global zpositionControl014
    global stateManagerInstance

    xpositionControl014.setPoint(setpoint[0])
    zpositionControl014.setPoint(setpoint[2])
    ypositionControl.setPoint(setpoint[1])
    yawControl.setPoint(0.0) #never changes
    rollControl.setPoint(0.0)
    pitchControl.setPoint(0.0)
    #rospy.loginfo("\zsetpoint: {} \n ---".format(zpositionControl014.getSetpoint()))

    if(firstphase1run): # Only runs the first time phase1() runs so that hover checking variables are
initialised
        is_hovering=0
        check=[0]
        i=0
        prevtime=time.time()
        previouspos=[position[0],position[1],position[2]]
        firstphase1run=0
        j=0


    currenttime=time.time()
    if (currenttime-prevtime>0.5) :
        i=i+1
        previouspos=currentpos
```

```
        currentpos=[position[0],position[1],position[2]]
        prevtime=currenttime #prev time records 0.4 second intervals
        if abs(currentpos[0]-previouspos[0])>0.05 or abs(currentpos[1]-previouspos[1])>0.05 or abs(
currentpos[2]-previouspos[2])>0.05 :
            check.append(0)
        else :
            check.append(1)
            j=j+1

        if (j>=20): #checking how the drone has behaved in the last 5 intervals of 0.4 seconds (i.e
over the last 2 seconds)
            checkfiltered=[x for x in check if (i-4)<=x<=i] #taking the last 5 values of check
            if all(x==1 for x in checkfiltered) : # if drone has moved minimally in the given time
period then checkfiltered=[1,1,1,1,1]
                is_hovering=1
                print('Hover Complete')
                goto2nd=1 #stop returning to phase1() and proceed
                goto1st=0

   #set points for PID in this stage are exactly the same as for startup so no need to modify
    xVelocity = xpositionControl014.update(position[0]) #need to update xVelocity in each stage
seperately as some use PID and others don't.
    yVelocity = ypositionControl.update(position[1])
    zVelocity = zpositionControl014.update(position[2])
    if zVelocity>0.3:
        zVelocity=0.3
    rollRate = yawControl.update(angpos[0])
    pitchRate = yawControl.update(angpos[1])
    yawRate = yawControl.update(angpos[2])

#########################################################################################################
#####################################

##### CONTROLLER = creating objects

zpositionControl014 = PID(1.2, 0.00005, 0.55) #create PID controller objects with specified gains
xpositionControl014 = PID(1.2, 0.000015, 0.6)
ypositionControl = PID(1.2, 0.000002, 0.45)
yawControl = PID(1, 0, 0)
rollControl = PID(1, 0, 0)
pitchControl = PID(1, 0, 0)

#########################################################################################

def takeoff():


    global tookoff
    global stateManagerInstance
    global firstphase1run

    rate = rospy.Rate(updrate)  # rate will update publisher
    stateManagerInstance = stateManager(rate)  # create new statemanager

    # State:
    rospy.Subscriber("/mavros/state", State,
                     stateManagerInstance.stateUpdate)  # Autopilot State including Arm State,
Connection Status and Mode
    #rospy.Subscriber("/tera_2_array", LaserScan, teraArrayCheck)
    # Upward Facing LiDAR:
    #rospy.Subscriber("/tera_1_up", LaserScan, teraUpCheck)
    # Downward Facing LiDAR:
    #rospy.Subscriber("/tera_1_down", LaserScan, teraDownCheck)
    # REAL SETUP ONLY -> ZHAO CHANGED DOWN FACING LIDAR TO BE:
    rospy.Subscriber("/mavros/distance_sensor/hrlv_ez4_pub", Range,distanceCheck)  # Current Distance
from Ground
    # Local Position Topics
    rospy.Subscriber("/mavros/local_position/odom", Odometry,odomCheck)  # Fuses Sensor Data
    rospy.Subscriber("/mavros/local_position/velocity", TwistStamped, velCheck)

    # 7. Publishers:
    # Velpub allows to set Linear and Angular Velocity with a message structure of type TwistStamped.
Queue Size balance between latest and completeness of data.
    velPub = rospy.Publisher("/mavros/setpoint_velocity/cmd_vel", TwistStamped, queue_size=2)

    # 8. Velocity Control Object - Initialise with Velocity Publisher.
    controller = velControl(velPub)  # Create New Controller Class and Pass in Publisher and State
Manager
    stateManagerInstance.waitForPilotConnection()  # Wait for Connection to Flight Controller

    # Avoid Proceeding Before Subscritptions Have Been Completed
    k = 0
```

```
    while (k == 0):
        if any(v == 0 for v in (position[1], range_ground)):
            print('Waiting for all of the subscriptions')
            rate.sleep()
        else:
            print('All subscriptions are active ')
            k = 1

    # DEFINE setPoint
    global setpoint
    setpoint = np.array([position[0] + setpoint[0],position[1] + setpoint[1], setpoint[2]])


    while not tookoff:

        if(has_started_flying==0):
            startup() #set goto1st as true after grange>0.2 or so. Use PID for height control. Set
has_started_flying=1
        if(goto1st):
            phase1() #Still using PID for height control but checking for 5sec hover. Set goto2nd=1
when is_hovering=true and goto1st=0
        if(goto2nd):
            tookoff = 1
            #print('Took off')


#####################################################################


        controller.setVel([xVelocity,yVelocity, zVelocity])
        controller.setAngVel([rollRate, pitchRate, yawRate])


        controller.publishTargetPose(stateManagerInstance)
        stateManagerInstance.incrementLoop()

        rate.sleep()  # sleep at the set rate

        # Don't know what that is, but better not to touch it
        if stateManagerInstance.getLoopCount() > 100 :  # to prevent offboard rejection
            stateManagerInstance.offboardRequest()  # request control from external computer
            stateManagerInstance.armRequest()  # arming must take place after offboard is requested

    #rospy.spin()


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

B Appendix - ROS Master Navigation Scripts Including Cruise With Potential Fields And Simulated Annealing

```python
#!/usr/bin/env python

# Potential Field Based Path Planner - ROS Based Solution
# Author: Pablo Hermoso Moreno

# Version: 15_06_2018 - V14:

# - Potential Field Based Navigation.
# - 3D (x-y-z) Plane
# - Class Based Python Algorithm
# - Radius of Drone -> d_eff = d - r_drone : Avoid Collision.
# - Radius of Goal -> Once inside: Target Acquisition Mechanism Triggered.
# - Vectorised Motion: Output Velocity Commands to UAV
# - Position Estimate from IMU Data
# - Continuous Walls and Obstacles
# - Imperfect Localisation - This does not affect Obstacle Avoidance
# - Imperfect Sensors
# -> TeraRanger Tower (8 * 45 deg apart)
#    - Stated Range Precision = +- 4cm
#    - Stated Field of View = 3 deg
# -> 2 X TeraRanger One (Downward & Upward Facing)
#    - Stated Range Precision = +- 4cm
#    - Stated Field of View = 3 deg
# - Static Environment
# - Local Minima Solver
#    - Via Simulated Annealing
# - Return to Base Incorporated

# Work In Progress for - V11:
# - Localisation and Mapping for Perching STAGE
# - Grid Based Stage
# - Perching Strategy

# Description:
# Potential Field based planners work using gradient descent to go in direction which locally
minimises potential function.
# Obstacles are positively charged and Goal is negatively charged.


########################################################################################################################

# Import Libraries
import numpy as np

# Plotting Library
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
plt.rcParams['legend.fontsize'] = 10

# ROS Specific Libraries:

import rospy
import math
import sys

# Takeoff Stage - Controlled by specifically built PID for increased accuracy and robustness
import TAKEOFF

# Import Message Structures:
from mavros_msgs.msg import State  # import state message structure
from sensor_msgs.msg import Range  # import range message structure
from sensor_msgs.msg import LaserScan  # import LaserScan message structure
from geometry_msgs.msg import TwistStamped  # used to set velocity messages
from nav_msgs.msg import Odometry  # import range message structure
from mavros_msgs.srv import *  # import for arm and flight mode setting

# Update: Import Message Structure For Real TeraRanger Tower
from teraranger_array.msg import RangeArray

########################################################################################################################

# ROS Specific Parameters:

updrate = float(20)  # Rate at Which Publisher is Updated
deltat = float(1 / updrate)  # Timestep

# Sensor Outputs:
range_ground = 0  # Range From Ground [m]
range_ceiling = 0  # Range From Ceiling [m]
range_tera_array = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]) # TeraRanger Array Ranges [m] -
```

```
Total of 8
angles_tera_array = np.array([180, 135, 90, 45, 0, -45, -90, -135])  # TeraRanger Array Angles [deg
] - Total of 8

# State Parameters
position = np.array([0, 0, 0])
first_pos = np.array([0, 0, 0])
velocity = np.array([0, 0, 0])
angpos = np.array([0, 0, 0])

# Stage Parameters:

# Stage 1 - Takeoff
stage1Height = 0.6  # [m]
stage1Climb_Vel = 0.10 #[m/s]

# Stage 2 & Stage 4:
hover_time = 5 # [s]

# Stage 3 - Cruise to Perching Location:
go = np.array([0, -2.75, stage1Height])  # Goal x and y and z position [m]

# Stage 4 - Hover 2
stage4_hover_pos = np.array([0, 0, 0]) # This is populated once the drone is within goal radius.

# Stage 6 - First Descent:
stage6_land_pos = np.array([0, 0, 0]) # This is populated once the drone is within goal radius.
stage7Des_Vel = -0.20 #[m/s]

# Stage 7 - Second Descent:
stage8Des_Vel = -0.15 #[m/s]

# Stages: 1-Takeoff; 2-Hover 1; 3-Cruise; 4-Hover 2; 5-Return to Base; 6-Hover; 7-Landing 1; 8-Landing
 2
stage1Completed = 0
stage2Completed = 0
stage3Completed = 0
stage4Completed = 0
stage5Completed = 0
stage6Completed = 0
stage7Completed = 0
# No need to initialise last landing stage

# PID Parameters:

# Initialise PID integration term
errorint = np.array([0, 0, 0])

##############################################################################################################
##################

# Potential Fields Parameters
KP = 3.5 # Attractive Potential Gain (zeta in PRM Book)
ETA = 0.00175  # Repulsive Potential Gain

# Repusive Action - Number of Obstacles - After meeting with Dr Paranjape 29_05_2018:
# e.g -> For 1 LIDAR obstacle detection / update -> For 1 second memory -> deltat = 0.05 -> n = 20
# Calculation: 10 Sensors - deltat = 0.05 - Desired Memory ~ 3s
memory_repulsive = 600 # Number of Repulsive Obstacles Memory.

# Drone Radius and Goal Radius
r_drone = 0.45 # Update: This Safety Margin Was Increased From 0.35m
r_goal = 0.30

# Gradient Descent Parameters:
# - alpha_pot : Gradient Descent Factor Gain Factor (Critical to limit velocity outputs)
alpha_pot = 0.025

# Maximum and Minimum Velocities:
max_vel = 0.325
min_vel = 0.05

# Potential Function Parameters:

# Attractive Potential:
# Threshold Distance d^{*}_{goal}
d_star = 1.0  # [m]

# Repulsive Potential:
# Critical Range Q^{*}
q_star = 1.0  # [m]
```

```
# Local Minima Solving: Simulated Annealing ->

# 1. Identify Local Minimum State:
r_local_min = 0.15  # [m]
num_traj_local_min = 400 # 14s Local Minima

# 2. Trigger Simulated Annealing Algorithm:
r_sim_ann = 0.35  # [m] - Same as Drone Radius
disc_theta_sim_ann = 5  # [deg]
disc_phi_sim_ann = 5  # [deg]

T_0_init = 300  # [K]
T_0_curr = 300  # [K]

cooling_r = 0.9 # Simulated Annealing Cooling Rate

# show_animation -> if false - Trajectory and Vectors not plotted.
show_animation = True


#########################################################################################################################
###################

# Initial Calibration of UAV in Indoor Environments
is_first_reading = 0
T_GM = np.array([[],[],[]])

#########################################################################################################################
###################

# USEFUL FUNCTIONS:

# 1. Modified Vertical Stack:
# - Includes check for empty arrays.
def vstack_mod(or_array, new_array):
    if len(or_array) > 1:
        return np.vstack([or_array, new_array])
    else:
        return np.array(new_array)

#########################################################################################################################
###################

# POTENTIAL FIELD CLASS DEFINITIONS:

class uav:

    # ROS Implementation:

    # Initialisation:
    # Key Parameter:
    # - obs_quad : n*2 array containing (x,y) of obstacles DETECTED BY UAV.
    # - r_quad : Trajectory (x,y) History [m]
    def __init__(self, cp):
        self.cp = cp

        # Drone's Obstacle Map:
        self.obs_quad = np.array([[]])

        # Drone's Trajectory
        self.r_quad = np.array([cp])

    # Distance Return: Hypotenuse to any location.
    def dist(self, pos):

        # 3D
        d = np.linalg.norm(position - pos)

        return d

    # Calculate Unit Vector to Given Location in 3D Space
    def unit_vec(self, pos):

        if ( (self.dist(pos)) != 0 ) and ( (self.dist(pos)) != float('Inf') ):
            # Pre_Divide distance to save computational power.
            d_inv = 1 / (self.dist(pos))
            return np.array([d_inv * (pos[0] - position[0]), d_inv * (pos[1] - position[1]), d_inv * (
pos[2] - position[2])])
        else:
            return np.array([0,0,0])

    def terraranger_scan(self):
```

```
        # Append Obstacles to obs_quad

        # 1. Down Facing TeraRanger One:
        x_obs_down = position[0] - range_ground * np.sin(np.deg2rad(angpos[1]))
        y_obs_down = position[1] + range_ground * np.sin(np.deg2rad(angpos[0]))
        z_obs_down = position[2] - range_ground * np.cos(np.deg2rad(angpos[0])) * np.cos(np.deg2rad(
angpos[1]))
        #z_obs_down = 0

        self.obs_quad = vstack_mod(self.obs_quad, [x_obs_down, y_obs_down, z_obs_down])

        # # 2. Up Facing TeraRanger One:
        # x_obs_up = position[0] + range_ceiling * np.sin(np.deg2rad(angpos[1]))
        # y_obs_up = position[1] - range_ceiling * np.sin(np.deg2rad(angpos[0]))
        # z_obs_up = position[2] + range_ceiling * np.cos(np.deg2rad(angpos[0])) * np.cos(np.deg2rad(
angpos[1]))
        #
        # self.obs_quad = vstack_mod(self.obs_quad, [x_obs_up, y_obs_up, z_obs_up])

        # 2. TeraRanger Tower:
        for i in range(len(range_tera_array)):
            if range_tera_array[i] != float('Inf'):
                x_obs_array = position[0] + range_tera_array[i] * np.cos( np.deg2rad(
angles_tera_array[i] - angpos[2] ) ) * np.cos(np.deg2rad(angpos[1]))
                y_obs_array = position[1] - range_tera_array[i] * np.sin( np.deg2rad(
angles_tera_array[i] - angpos[2] ) ) * np.cos(np.deg2rad(angpos[0]))
                z_obs_array = position[2] - range_tera_array[i] * np.sin( np.deg2rad(
angles_tera_array[i] ) ) * np.sin(np.deg2rad(angpos[0])) - range_tera_array[i] * np.cos( np.deg2rad(
angles_tera_array[i] ) ) * np.sin(np.deg2rad(angpos[1]))

                self.obs_quad = vstack_mod(self.obs_quad, [x_obs_array, y_obs_array, z_obs_array])

        # For final testing: No Ceiling Available
        # Create Virtual Ceiling at z = 1.8m
        self.obs_quad = vstack_mod(self.obs_quad, [position[0], position[1], 1.8])

    return self.obs_quad

    def calc_potential_field_quad(self, go):

        # Scan the environment and add LIDAR detected obstacles.
        obs_quad = self.terraranger_scan()

        # For the current position, calculate the Resultant Gradient Descent Vector:

        # 3D
        [u_g, v_g, w_g] = self.calc_attractive_potential_vector(go)

        # Calculate Repulsive Potential With Respect to Obstacles

        # 3D
        [u_o, v_o, w_o] = self.calc_repulsive_potential_vector()

        return [u_g + u_o, v_g + v_o, w_g + w_o], obs_quad

    def calc_attractive_potential_vector(self, go):
        # Need to implement Combined (Quadratic + Conic) Variation.

        # Reason: In some cases, it may be desirable to have distance functions
        # that grow more slowly to avoid huge velocities far from the goal.

        # 3D

        # Distance to Goal
        d = self.dist(go)

        # Unit Vector to Goal:
        d_hat = self.unit_vec(go)

        # Gradient Calculation
        if d <= d_star:
            return [KP * d * d_hat[0], KP * d * d_hat[1], KP * d * d_hat[2]]
        else:
            return [d_star * KP *  d_hat[0], d_star * KP *  d_hat[1], d_star * KP *  d_hat[2]]

    def calc_repulsive_potential_vector(self):

        # Better implementation of the repulsive potential would be to sum the actions of all the
individual repulsive
        # potentials of the obtacles within Vision Range.
```

```
        # Previous model would search only for closest obsacle and take the action of only that one.
        # Problem: When numerically implementing this solution, a path may form that oscillates around
 points that are
        #  two-way equidistant from obstacles, i.e., points where D is nonsmooth. To avoid these
oscillations,
        # instead of defining the repulsive potential function in terms of distance to the closest
obstacle,
        # the repulsive potential function (4.6) is redefined in terms of distances to individual
obstacles
        # where d i ( q )i s the distance to obstacle.

        # Modified Repulsive Potential Vector - After meeting with Dr Paranjape 29_05_2018:
            # Take only the latest "memory_repulsive" detected obstacles
            # "range(len(self.obs_quad))" -> "range(min(len(self.obs_quad),20))"
            # Start reading from the bottom obs obsquad -> [-i]

        u_o = 0
        v_o = 0
        w_o = 0

        if len(self.obs_quad) > 1:
            for i in range(min(len(self.obs_quad),memory_repulsive)):
                if len(self.obs_quad.shape) != 1:

                    # 3D
                    d = self.dist(self.obs_quad[-i])

                    # Unit Vector to Obstacle
                    d_hat = self.unit_vec(self.obs_quad[-i])
                else:

                    # 3D
                    d = self.dist(self.obs_quad)

                    d_hat = self.unit_vec(self.obs_quad)


                # Collision Avoidance:
                d -= r_drone

                if (d <= q_star):
                    u_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[0]
                    v_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[1]
                    w_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[2]

        return [u_o, v_o, w_o]

    # Simulated Annealing Algorithm - NOT IMPLEMENTED WITHOUT TERARANGER TOWER:
    def calc_attractive_potential(self, new_pos, go):
        # Combined (Quadratic + Conic) Variation.

        # Reason: In some cases, it may be desirable to have distance functions
        # that grow more slowly to avoid huge velocities far from the goal.

        # Distance to Goal
        # 3D
        d = np.linalg.norm(go - new_pos)

        if d <= d_star:
            return 0.5 * KP * (d ** 2)
        else:
            return (d_star * KP * d) - (0.5 * KP * (d_star ** 2))

    def calc_repulsive_potential(self, new_pos):

        # Better implementation of the repulsive potential would be to sum the actions of all the
individual repulsive
        # potentials of the obtacles within Vision Range.

        # Previous model would search only for closest obsacle and take the action of only that one.
        # Problem: When numerically implementing this solution, a path may form that oscillates around
 points that are
        #  two-way equidistant from obstacles, i.e., points where D is nonsmooth. To avoid these
oscillations,
        # instead of defining the repulsive potential function in terms of distance to the closest
obstacle,
        # the repulsive potential function (4.6) is redefined in terms of distances to individual
obstacles
        # where d i ( q )i s the distance to obstacle.

        # MODIFIED ETA FOR SIMULATED ANNEALING:
        ETA = 0.005
```

```
        u_rep = 0

        if len(self.obs_quad) > 1:
            for i in range(min(len(self.obs_quad),memory_repulsive)):

                if len(self.obs_quad.shape) != 1:
                    # 3D
                    d = np.linalg.norm(self.obs_quad[-i] - new_pos)
                else:

                    #3D
                    d = np.linalg.norm(self.obs_quad - new_pos)

                # Collision Avoidance:
                # d_eff = d - r_drone: Avoid Collision
                d -= r_drone

                # Check if the obstacle if within Vision Range Q^{*}.
                if (d <= q_star):
                    u_rep += 0.5 * ETA * ((1.0 / d) - (1.0 / q_star)) ** 2

        return u_rep

    def check_local_min(self):

        # Check if the latest "num_traj_local_min" trajectory lies with radius "r_local_min" from
current trajectory:
        if len(self.r_quad) > num_traj_local_min:

            d = self.dist(self.r_quad[-num_traj_local_min])

            if d < r_local_min:  # Radius: "r_local_min"
                return True
            else:
                return False

    def sim_annealing_alg(self, go):

        # Set Initial Temperature:
        global T_0_curr

        # Current U(x)
        ug_cp = self.calc_attractive_potential(position, go)
        uo_cp = self.calc_repulsive_potential(position)


        # Spherical Coordinates:
        # Strategy:
        # -> First test in "theta" (X-Y)
        # -> If no success, test "phi" (Z)

        theta = 0
        phi = 90
        sign_theta = -1
        counter_theta = 0
        sign_phi = -1
        counter_phi = 0

        while abs(phi - 90) < 30:
            while abs(theta) < 180:

                # 1. Calculate x' = x + delta_x
                # Spherical Coordinates:
                x_new = self.cp[0] + r_sim_ann * np.cos(np.deg2rad(theta)) * np.sin(np.deg2rad(phi))
                y_new = self.cp[1] + r_sim_ann * np.sin(np.deg2rad(theta)) * np.sin(np.deg2rad(phi))
                z_new = self.cp[2] + r_sim_ann * np.cos(np.deg2rad(phi))

                # 2. Calculate U(x')
                ug_new = self.calc_attractive_potential([x_new, y_new, z_new], go)
                uo_new = self.calc_repulsive_potential([x_new, y_new, z_new])

                # 3. Calculate delta_U
                delta_U = (-ug_new + uo_new) - (-ug_cp + uo_cp)

                # 4. Probabilistic Algorithm:
                if delta_U < 0:
                    # Found Direction With Negative Gradient:
                    break
                else:
                    # Probability:
                    p_new = np.exp((-delta_U) / (T_0_curr))
```

```
                    if np.random.random_sample() < p_new:
                        break

                theta = (sign_theta * counter_theta * disc_theta_sim_ann)
                sign_theta = sign_theta * -1

                if sign_theta == 1:
                    counter_theta += 1

            phi = 90 + (sign_phi * counter_phi * disc_phi_sim_ann)

            sign_phi = sign_phi * -1

            if sign_phi == 1:
                counter_phi += 1

        # Reduce T_0 by cooling rate:
        T_0_curr = cooling_r * T_0_curr

        # Once the angle has been set, the drone will try to escape local minima at minimum velocity:
        u_res = min_vel * np.cos(np.deg2rad(theta)) * np.sin(np.deg2rad(phi))
        v_res = min_vel * np.sin(np.deg2rad(theta)) * np.sin(np.deg2rad(phi))
        w_res = min_vel * np.cos(np.deg2rad(phi))

        return u_res, v_res, w_res

########################################################################################################################

# ROS SPECIFIC CLASS DEFINITIONS

class stateManager:  # class for monitoring and changing state of the controller
    def __init__(self, rate):
        self._rate = rate
        self._loopCount = 0
        self._isConnected = 0
        self._isArmed = 0
        self._mode = None

    def incrementLoop(self):
        self._loopCount = self._loopCount + 1

    def getLoopCount(self):
        return self._loopCount

    def stateUpdate(self, msg):
        self._isConnected = msg.connected
        self._isArmed = msg.armed
        self._mode = msg.mode
        # rospy.logwarn("Connected is {}, armed is {}, mode is {} ".format(self._isConnected, self._isArmed,self._mode))  # some status info

    def armRequest(self):
        rospy.wait_for_service('/mavros/set_mode')
        try:
            modeService = rospy.ServiceProxy('/mavros/set_mode',
                                              mavros_msgs.srv.SetMode)  # get mode service and set to offboard control
            modeService(custom_mode='OFFBOARD')
        except rospy.ServiceException as e:
            print("Service mode set faild with exception: %s" % e)

    def offboardRequest(self):
        rospy.wait_for_service('/mavros/cmd/arming')
        try:
            arm = rospy.ServiceProxy('/mavros/cmd/arming',
                                     mavros_msgs.srv.CommandBool)  # get arm command service and arm
            arm(True)
        except rospy.ServiceException as e:  # except if failed
            print("Service arm failed with exception :%s" % e)

    def waitForPilotConnection(self):  # wait for connection to flight controller
        # rospy.logwarn("Waiting for pilot connection")
        while not rospy.is_shutdown():  # while not shutting down
            if self._isConnected:  # if state isConnected is true
                # rospy.logwarn("Pilot is connected")
                return True
            self._rate.sleep
        # rospy.logwarn("ROS shutdown")
        return False

class velControl:
```

```
    def __init__(self, attPub):  # attPub = attitude publisher
        self._attPub = attPub
        self._setVelMsg = TwistStamped()
        self._targetVelX = 0
        self._targetVelY = 0
        self._targetVelZ = 0

    def setVel(self, coordinates):
        self._targetVelX = float(coordinates[0])
        self._targetVelY = float(coordinates[1])
        self._targetVelZ = float(coordinates[2])
        # rospy.logwarn("Target velocity is \nx: {} \ny: {} \nz: {}".format(self._targetVelX, self.
_targetVelY,self._targetVelZ))

    def publishTargetPose(self, stateManagerInstance):
        self._setVelMsg.header.stamp = rospy.Time.now()  # construct message to publish with time,
loop count and id
        self._setVelMsg.header.seq = stateManagerInstance.getLoopCount()
        self._setVelMsg.header.frame_id = 'fcu'

        self._setVelMsg.twist.linear.x = self._targetVelX
        self._setVelMsg.twist.linear.y = self._targetVelY
        self._setVelMsg.twist.linear.z = self._targetVelZ

        self._attPub.publish(self._setVelMsg)

class ROS:
    # Functions Called When Subscribers Are Updated

    def distanceCheck(self, msg):
        global range_ground  # import global range
        range_ground = msg.range  # set range = received range

    def teraArrayCheck(self, msg):
        global range_tera_array

        # GAZEBO SIMULATION ENVIRONMENT
        # Understanding Message Structure of "msg.ranges"
        # Degrees : [180,135,90,45,0,315,270,225,-] - Saved as GLOBAL Array
        # IMP : Real Setup No 180deg
        # NOTE: Array Slicing in Python Requires to end index one item before.

        # Gazebo:
        #range_tera_array = np.array(msg.ranges[0:8])

        # REAL Setup
        for i in range(8):
            range_buffer = msg.ranges[i].range
            bool_buffer = 0

            # Inf Handling for cases where tera ranger detects > 14m (return +Inf)
            # or < 0.2m (return -Inf) range - prevents potential numerical errors when calculating
potential field or doing angular corrections
            if np.isinf(range_buffer):
                if range_buffer > 0:
                    range_buffer = 100 # set to 100 m, if range is further than max range
                    bool_buffer = 1
                else:
                    range_buffer = 0.05 # set to 0.05 m, if range is closer than min range
                    bool_buffer = -1

            range_tera_array[i] = range_buffer


    def teraUpCheck(self, msg):
        global range_ceiling
        range_ceiling = msg.ranges[0]

    def teraDownCheck(self, msg):
        global range_ground
        range_ground = msg.ranges[0]

    def odomCheck(self, msg):
        global position
        global angpos
        global range_ground
        angpos = self.quaternion_to_euler_angle(msg.pose.pose.orientation.w, msg.pose.pose.orientation
.x,
                                                msg.pose.pose.orientation.y, msg.pose.pose.orientation
.z)
```

```
        global is_first_reading
        global T_GM

        if is_first_reading == 0:
            # ASSUMING T_BG|t=0 = T_MG

            #T_GM = np.array([[ np.cos(np.deg2rad(angpos[1]))*np.cos(np.deg2rad(angpos[2])) , np.sin(
np.deg2rad(angpos[0]))*np.sin(np.deg2rad(angpos[1]))*np.cos(np.deg2rad(angpos[2])) - np.cos(np.deg2rad
(angpos[0]))*np.sin(np.deg2rad(angpos[2])) , np.cos(np.deg2rad(angpos[0]))*np.sin(np.deg2rad(angpos[1
]))*np.cos(np.deg2rad(angpos[2])) + np.sin(np.deg2rad(angpos[0]))*np.sin(np.deg2rad(angpos[2])) ],
                            #[ np.cos(np.deg2rad(angpos[1]))*np.sin(np.deg2rad(angpos[2])) ,np.sin(np.
deg2rad(angpos[0]))*np.sin(np.deg2rad(angpos[1]))*np.sin(np.deg2rad(angpos[2])) + np.cos(np.deg2rad(
angpos[0]))*np.cos(np.deg2rad(angpos[2]))  , np.cos(np.deg2rad(angpos[0]))*np.sin(np.deg2rad(angpos[1
]))*np.sin(np.deg2rad(angpos[2])) - np.sin(np.deg2rad(angpos[0]))*np.cos(np.deg2rad(angpos[2])) ],
                            #[-np.sin(np.deg2rad(angpos[1])), np.sin(np.deg2rad(angpos[0]))*np.cos(np.
deg2rad(angpos[1])) , np.cos(np.deg2rad(angpos[0]))*np.cos(np.deg2rad(angpos[1])) ]]).transpose() #
Transformation from GLOBAL to MINE

            # When testing with VICON, transformation not required:
            T_GM = np.array([[1,0,0],
                             [0,1,0],
                             [0,0,1]]) # Transformation from GLOBAL to MINE

            is_first_reading = 1


        xPosition = msg.pose.pose.position.x
        yPosition = msg.pose.pose.position.y

        # EXPERIMENT - ONLY USE WITH VICON
        zPosition = msg.pose.pose.position.z

        # NOTE:
        # I found that using "msg.pose.pose.position.z" gives smoother results for current height.
        # For some reason "range_ground" was able to detect I was climbing?
        # For the moment being, use PID in height once the terraranger tower has been installed

        # Z-Position Correction with Angles - theta_x & theta_y :

        # x_Angle = angpos[0]
        # y_Angle = angpos[1]
        # z_Angle = angpos[2]

        #zPosition = range_ground * math.cos(math.radians(angpos[1])) * math.cos(math.radians(angpos[0
]))

        position = np.array([xPosition, yPosition, zPosition])

    def velCheck(self, msg):
        global velocity
        xVel = msg.twist.linear.x
        yVel = msg.twist.linear.y
        zVel = msg.twist.linear.z
        velocity = np.array([xVel, yVel, zVel])

    # Quartenion to Euler Angle Conversion

    def quaternion_to_euler_angle(self, w, x, y, z):
        ysqr = y * y

        t0 = +2.0 * (w * x + y * z)
        t1 = +1.0 - 2.0 * (x * x + ysqr)
        X = math.degrees(math.atan2(t0, t1))

        t2 = +2.0 * (w * y - z * x)
        t2 = +1.0 if t2 > +1.0 else t2
        t2 = -1.0 if t2 < -1.0 else t2
        Y = math.degrees(math.asin(t2))

        t3 = +2.0 * (w * z + x * y)
        t4 = +1.0 - 2.0 * (ysqr + z * z)
        Z = math.degrees(math.atan2(t3, t4))
        angpos = np.array([X, Y, Z])
        return angpos

    # PID Controller:
    def PID(self, position, vel, setpoint,const):  # Inputs: current position, current velocity,
desired state, PID constants

        global errorint
        global deltat
```

```
        error = np.array(setpoint) - position
        prop = const[0] * error
        integ = const[1] * errorint
        der = -1 * const[2] * vel  # NB: d(error)/dt = d(setpoint)/dt - d(position)/dt = -velocity

        output = prop + integ + der
        errorint = errorint + error * deltat

        for i in [0, 1, 2]:
            output[i] = np.sign(output[i]) * min(abs(output[i]), max_vel)

        return output

########################################################################################################################
##################
def main():
    print("Potential_Field_Planning_Start")

    # ARTIFICIAL POTENTIAL FIELDS:

    # 1. Set the Environment

    # MAKE SURE THE STARTING POSITION IS NOT ON A BORDER
    st = np.array([0, 0, 0])  # Start x and y position [m]
    global go  # Goal x and y position [m]

    # 2. Set the Drone:

    # Drone Class:
    drone = uav(st)

    # 3. Initialise ROS Class:
    ros_class = ROS()

    # 4. Initialise ROS Node:
    rospy.init_node('UAV_NODE')

    # 5. Set Up State Manager:
    rate = rospy.Rate(updrate)  # Rate will update publisher at 20hz, higher than the 2hz minimum
before timeouts occur
    stateManagerInstance = stateManager(rate)  # Create StateManager Object and initialise with rate.

    # 6. Subscriptions:
    # We require:
    # - State
    # - Distance From Ground
    # - Distance From Ceiling
    # - TeraRanger Tower Distances
    # - Velocity


    # State:
    rospy.Subscriber("/mavros/state", State,
                     stateManagerInstance.stateUpdate)  # Autopilot State including Arm State,
Connection Status and Mode

    # Range Topics: REMEMBER TO SUBSTITUTE FOR REAL TOPICS IN REAL SETUP

    # TeraRanger Tower
    #rospy.Subscriber("/tera_2_array", LaserScan, ros_class.teraArrayCheck)

    # SUBSTITUTE FOR REAL SETUP:
    rospy.Subscriber("/ranges", RangeArray, ros_class.teraArrayCheck)

    # Upward Facing LiDAR:
    #rospy.Subscriber("/tera_1_up", LaserScan, ros_class.teraUpCheck)

    # Downward Facing LiDAR:

    #rospy.Subscriber("/tera_1_down", LaserScan, ros_class.teraDownCheck)
    # REAL SETUP ONLY -> ZHAO CHANGED DOWN FACING LIDAR TO BE:
    rospy.Subscriber("/mavros/distance_sensor/hrlv_ez4_pub", Range,ros_class.distanceCheck)  # Current
 Distance from Ground

    # Local Position Topics
    rospy.Subscriber("/mavros/local_position/odom", Odometry,ros_class.odomCheck)  # Fuses Sensor Data
    rospy.Subscriber("/mavros/local_position/velocity", TwistStamped, ros_class.velCheck)

    # 7. Publishers:
    # Velpub allows to set Linear and Angular Velocity with a message structure of type TwistStamped.
Queue Size balance between latest and completeness of data.
```

```
    velPub = rospy.Publisher("/mavros/setpoint_velocity/cmd_vel", TwistStamped, queue_size=2)

    # 8. Velocity Control Object - Initialise with Velocity Publisher.
    controller = velControl(velPub)  # Create New Controller Class and Pass in Publisher and State
Manager
    stateManagerInstance.waitForPilotConnection()  # Wait for Connection to Flight Controller

    # 9. Start Giving Instructions:
    global position

    # Structure to avoid proceeding before subscriptions are complete:
    k = 0
    while (k == 0):
        if any(v == 0 for v in (position[1], range_ground)):
            print('Waiting for all of the subscriptions')
            rate.sleep()
        else:
            print('All subscriptions are active ')
            k = 1


    # MODIFY GOAL TO ACCOUNT FOR THE DEVIATION OF VICON!
    global first_pos
    go = np.array([position[0] + go[0],position[1] + go[1], go[2]])
    first_pos = position


    while not rospy.is_shutdown():

        if 'range_ground' in globals():

            global stage1Completed
            global stage2Completed
            global stage3Completed
            global stage4Completed
            global stage5Completed
            global stage6Completed
            global stage7Completed

            global angpos
            global deltat
            global velocity
            global errorint
            global T_0_curr
            global T_0_init


            # CALIBRATION PARAMETERS:
            global T_GM

            # Stage parameters:
            global stage1Height

            # Stages Description:

            # 1. Takeoff to Height = stage1Height
            # Use PID Control Design for L3
            # Climb at stage1ClimbVel m/s

            # 2. Hover for 5s @ Height = stage1Height

            # 3. Cruise to Perching Location
            # Potential Fields Algorithm

            # 4. Hover for 5s @ Height = stage1Height

            # 5. Return to Base

            # 6. First Descent

            # 7 Second & Last Descent

            if stage1Completed == 0:

                # Initiliaise Takeoff & Hover From Separate Script
                TAKEOFF.takeoff()

                stage1Completed = 1
                stage2Completed = 1
                print('Stage 1 - Takeoff - COMPLETE')
                print('Stage 2 - Hover - COMPLETE')
```

```
            # Hover for 5 seconds
            loopCount = stateManagerInstance.getLoopCount()

        # IMPORTANT UPDATE: Hover has been incorporated in Takeoff Script
        # This Stage Is Left for Backup Purposes
        if (stage1Completed == 1) & (stage2Completed == 0):

            # 2. Hover for 5s

            # Add Trajectory To History
            drone.r_quad = np.vstack([drone.r_quad, position])

            if stateManagerInstance.getLoopCount() - loopCount < (hover_time * updrate):
                PIDout = ros_class.PID(position, velocity, [0, 0, stage1Height], [3.3, 0.3, 0.1])

                PIDout_GLOBAL = T_GM.dot(PIDout)
                controller.setVel([PIDout_GLOBAL[0], PIDout_GLOBAL[1], PIDout_GLOBAL[2]])

            else:
                print('Stage 2 - Hover - COMPLETE')

                # Move to stage 3
                stage2Completed = 1
                errorint = np.array([0, 0, 0])

        if (stage1Completed == 1) & (stage2Completed == 1) & (stage3Completed == 0):

            # ARTIFICIAL POTENTIAL FIELDS:

            # Global Distance from Start to Goal:
            d = drone.dist(go)  # Global Distance To Target

            # Add Trajectory To History
            drone.r_quad = np.vstack([drone.r_quad, position])

            if d >= r_goal:  # UAV within Goal Radius

                # Calculate Gradient and Velocity Command in Inertial Frame:
                [u_res, v_res, w_res], obs_quad = drone.calc_potential_field_quad(go)

                # Check for Existance of Local Minima State:
                if drone.check_local_min():
                    print('LOCAL MINIMA STATE')
                    # Trigger Simulated Annealing Algorithm:
                    u_res, v_res, w_res = drone.sim_annealing_alg(go)
                else:
                    # If Drone Escapes Local Minima - Reinitialise Current Temperature
                    T_0_curr = T_0_init

                alpha = alpha_pot

                # 3D
                if np.linalg.norm([alpha * u_res, alpha * v_res, alpha * w_res]) == 0:
                    alpha = 0
                elif np.linalg.norm(
                        [alpha * u_res, alpha * v_res, alpha * w_res]) > max_vel:  # Maximum
Velocity Limit:
                    alpha = max_vel / (np.linalg.norm([u_res, v_res, w_res]))
                elif np.linalg.norm(
                        [alpha * u_res, alpha * v_res, alpha * w_res]) < min_vel:  # Minimum
Velocity Limit:
                    alpha = min_vel / (np.linalg.norm([u_res, v_res, w_res]))

                # Update Position by Sending Velocity Command:
                # Update by transformation to Pixhawk World F.O.R
                VEL_G = T_GM.dot( np.array([alpha * u_res, alpha * v_res, alpha * w_res]) )

                controller.setVel([VEL_G[0], VEL_G[1], VEL_G[2]])


            else: # Withing Goal Radius

                print('Stage 3 - Cruise to Perching Location - COMPLETE')

                #Move to stage 4.
                stage3Completed = 1
                # Hover for 2 second
                loopCount = stateManagerInstance.getLoopCount()
                # Restart errorint after each stage.
                errorint = np.array([0, 0, 0])

                # For Inital Phase Testing - Set Current Pos as Landing Pos
```

```
                    stage4_hover_pos = position

            if (stage1Completed == 1) & (stage2Completed == 1) & (stage3Completed == 1) & (
stage4Completed == 0):

                # Add Trajectory To History
                drone.r_quad = np.vstack([drone.r_quad, position])

                # Begin by hovering:
                # Hover for 2 second
                if stateManagerInstance.getLoopCount() - loopCount < (hover_time * updrate):
                    PIDout = ros_class.PID(position, velocity, stage4_hover_pos, [3, 1.10, 0.23])

                    PIDout_GLOBAL = T_GM.dot(PIDout)
                    controller.setVel([PIDout_GLOBAL[0], PIDout_GLOBAL[1], PIDout_GLOBAL[2]])

                else:
                    print('Stage 4 - Hover - COMPLETE')

                    # Move to stage 5.
                    stage4Completed = 1
                    errorint = np.array([0, 0, 0])

            if (stage1Completed == 1) & (stage2Completed == 1) & (stage3Completed == 1) & (
stage4Completed == 1) & (stage5Completed == 0):

                # Return to Base - New Goal = [0,0,0] = Origin
                return_goal = np.array([first_pos[0],first_pos[1],stage1Height])

                # ARTIFICIAL POTENTIAL FIELDS:

                # Global Distance from Start to Goal:
                d = drone.dist(return_goal)  # Global Distance To Target

                # Add Trajectory To History
                drone.r_quad = np.vstack([drone.r_quad, position])

                if d >= r_goal:  # UAV within Goal Radius

                    # Calculate Gradient and Velocity Command in Inertial Frame:
                    [u_res, v_res, w_res], obs_quad = drone.calc_potential_field_quad(return_goal)

                    # Check for Existance of Local Minima State:
                    if drone.check_local_min():
                        print('LOCAL MINIMA STATE')
                        # Trigger Simulated Annealing Algorithm:
                        u_res, v_res, w_res = drone.sim_annealing_alg(return_goal)
                    else:
                        # If Drone Escapes Local Minima - Reinitialise Current Temperature
                        T_0_curr = T_0_init

                    alpha = alpha_pot

                    # 3D
                    if np.linalg.norm([alpha * u_res, alpha * v_res, alpha * w_res]) == 0:
                        alpha = 0
                    elif np.linalg.norm(
                            [alpha * u_res, alpha * v_res, alpha * w_res]) > max_vel:  # Maximum
Velocity Limit:
                        alpha = max_vel / (np.linalg.norm([u_res, v_res, w_res]))
                    elif np.linalg.norm(
                            [alpha * u_res, alpha * v_res, alpha * w_res]) < min_vel:  # Minimum
Velocity Limit:
                        alpha = min_vel / (np.linalg.norm([u_res, v_res, w_res]))

                    # Update Position by Sending Velocity Command:
                    VEL_G = T_GM.dot(np.array([alpha * u_res, alpha * v_res, alpha * w_res]))

                    controller.setVel([VEL_G[0], VEL_G[1], VEL_G[2]])


                else: # Withing Goal Radius

                    print('Stage 5 - Return to Base - COMPLETE')

                    #Move to stage 6.
                    stage5Completed = 1
                    # Hover for 2 second
                    loopCount = stateManagerInstance.getLoopCount()
                    # Restart errorint after each stage.
                    errorint = np.array([0, 0, 0])
```

```
                    # For Inital Phase Testing - Set Current Pos as Landing Pos
                    stage6_land_pos = position

            if (stage1Completed == 1) & (stage2Completed == 1) & (stage3Completed == 1) & (
stage4Completed == 1) & (stage5Completed == 1) & (stage6Completed == 0):

                # Add Trajectory To History
                drone.r_quad = np.vstack([drone.r_quad, position])

                # Begin by hovering:
                # Hover for 2 second
                if stateManagerInstance.getLoopCount() - loopCount < (hover_time * updrate):
                    PIDout = ros_class.PID(position, velocity, stage6_land_pos, [3, 1.10, 0.23])

                    PIDout_GLOBAL = T_GM.dot(PIDout)
                    controller.setVel([PIDout_GLOBAL[0], PIDout_GLOBAL[1], PIDout_GLOBAL[2]])

                else:
                    print('Stage 6 - Hover - COMPLETE')

                    # Move to stage 7.
                    stage6Completed = 1
                    errorint = np.array([0, 0, 0])

            if (stage1Completed == 1) & (stage2Completed == 1) & (stage3Completed == 1) & (
stage4Completed == 1) & (stage5Completed == 1) & (stage6Completed == 1) & (stage7Completed == 0):

                # Add Trajectory To History
                drone.r_quad = np.vstack([drone.r_quad, position])

                if position[2] > 0.4:
                    # Control PID in x and y

                    PIDout = ros_class.PID(position, velocity, stage6_land_pos, [3, 1.01, 0.25])
                    PIDout_GLOBAL = T_GM.dot(np.array([PIDout[0],PIDout[1],stage7Des_Vel]))
                    controller.setVel([PIDout_GLOBAL[0], PIDout_GLOBAL[1], PIDout_GLOBAL[2]])

                else:
                    print('Stage 7 - First Descent - COMPLETE')

                    stage7Completed = 1
                    errorint = np.array([0, 0, 0])

            if (stage1Completed == 1) & (stage2Completed == 1) & (stage3Completed == 1) & (
stage4Completed == 1) & (stage5Completed == 1) & (stage6Completed == 1) & (stage7Completed == 1):

                # Add Trajectory To History
                drone.r_quad = np.vstack([drone.r_quad, position])

                if position[2] > 0.24:
                    # Control PID in x and y

                    PIDout = ros_class.PID(position, velocity, stage6_land_pos, [3, 1.01, 0.25])
                    PIDout_GLOBAL = T_GM.dot(np.array([PIDout[0],PIDout[1],stage8Des_Vel]))
                    controller.setVel([PIDout_GLOBAL[0], PIDout_GLOBAL[1], PIDout_GLOBAL[2]])

                else:
                    print('Stage 8 - Second Descent - COMPLETE')

                    ############################################################

                    # FINISHED TRAJECTORY:

                    ############################################################

                    # Visualisation of Obstacles & Trajectory:

                    if show_animation:

                        # 2D Plots

                        # X-Y Plane:

                        # Create Figure + Subplot
                        fig1, ax1 = plt.subplots()

                        ax1.plot(drone.r_quad[:,0], drone.r_quad[:,1], '.r' , label='UAV Trajectory')

                        if drone.obs_quad.size != 0:
                            if drone.obs_quad.size > 3:
                                ax1.plot(drone.obs_quad[:,0] ,drone.obs_quad[:,1] , 'sg', label='
Detected Obstacles')
```

```
                            else:
                                ax1.plot(drone.obs_quad[0], drone.obs_quad[1], 'sg', label='Detected
Obstacles')

                        ax1.legend()
                        ax1.set_xlabel('X / [m]')
                        ax1.set_ylabel('Y / [m]')
                        ax1.set_title('X-Y UAV Trajectory and Detected Obstacles')

                        # Y-Z Plane:

                        # Create Figure + Subplot
                        fig3, ax3 = plt.subplots()

                        ax3.plot(drone.r_quad[:, 1], drone.r_quad[:, 2], '.r', label='UAV Trajectory')

                        if drone.obs_quad.size != 0:
                            if drone.obs_quad.size > 3:
                                ax3.plot(drone.obs_quad[:, 1], drone.obs_quad[:, 2], 'sg', label='
Detected Obstacles')
                            else:
                                ax3.plot(drone.obs_quad[1], drone.obs_quad[2], 'sg', label='Detected
Obstacles')

                        ax3.legend()
                        ax3.set_xlabel('Y / [m]')
                        ax3.set_ylabel('Z / [m]')
                        ax3.set_title('Y-Z UAV Trajectory and Detected Obstacles')

                        # X-Z Plane:

                        # Create Figure + Subplot
                        fig4, ax4 = plt.subplots()

                        ax4.plot(drone.r_quad[:, 0], drone.r_quad[:, 2], '.r', label='UAV Trajectory')

                        if drone.obs_quad.size != 0:
                            if drone.obs_quad.size > 3:
                                ax4.plot(drone.obs_quad[:, 0], drone.obs_quad[:, 2], 'sg', label='
Detected Obstacles')
                            else:
                                ax4.plot(drone.obs_quad[0], drone.obs_quad[2], 'sg', label='Detected
Obstacles')

                        ax4.legend()
                        ax4.set_xlabel('X / [m]')
                        ax4.set_ylabel('Z / [m]')
                        ax4.set_title('X-Z UAV Trajectory and Detected Obstacles')

                        # 3D Trajectory Path & Obstacles Detected:

                        #Create 3D Figure + Subplots:

                        fig2 = plt.figure()
                        ax2 = fig2.add_subplot(111, projection='3d')

                        if drone.r_quad.size != 0:
                            if drone.r_quad.size > 3:
                                #ax2.plot(drone.r_quad[:,0],drone.r_quad[:,1],drone.r_quad[:,2])
                                ax2.scatter(drone.r_quad[:,0],drone.r_quad[:,1],drone.r_quad[:,2], c='
r', marker='o', label='UAV Trajectory')
                                ax2.scatter(drone.obs_quad[:,0],drone.obs_quad[:,1],drone.obs_quad[:,2
], c='g', marker='s', label='Detected Obstacles')
                            else:
                                #ax2.plot(drone.r_quad[0],drone.r_quad[1],drone.r_quad[2])
                                ax2.scatter(drone.r_quad[0],drone.r_quad[1],drone.r_quad[2], c='r',
marker='o',label='UAV Trajectory')
                                ax2.scatter(drone.obs_quad[0],drone.obs_quad[1],drone.obs_quad[2], c='
g', marker='s', label='Detected Obstacles')

                        ax2.legend()
                        ax2.set_xlabel('X / [m]')
                        ax2.set_ylabel('Y / [m]')
                        ax2.set_zlabel('Z / [m]')
                        ax2.set_title('3D UAV Trajectory and Detected Obstacles')

                        plt.draw()
                        plt.show()


                    # Save to text:
```

```
                    # 1. Trajectory
                    np.savetxt('drone.r_quad.out', drone.r_quad, delimiter=',')
                    # 2. obs_quad
                    np.savetxt('drone.obs_quad.out', drone.obs_quad, delimiter=',')

                    # Goodbye!
                    print('Goodbye!')

                    sys.exit()

        else:
            print('Waiting for Range')

        controller.publishTargetPose(stateManagerInstance)

        stateManagerInstance.incrementLoop()
        rate.sleep()  # sleep at the set rate
        if stateManagerInstance.getLoopCount() > 100:  # need to send some position data before we can
 switch to offboard mode otherwise offboard is rejected
            stateManagerInstance.offboardRequest()  # request control from external computer
            stateManagerInstance.armRequest()  # arming must take place after offboard is requested
    rospy.spin()  # keeps python from exiting until this node is stopped


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

C Appendix - Geometric Transformations From Body-Fixed Sensors to World Frame Of Reference

Subscript $_{obs}$ denotes obstacle position in World F.O.R and subscript $_{uav}$ denotes the UAV's position in World F.O.R. R denotes the range data from the Time-Of-Flight sensor.

C.1 Downward Facing TeraRanger One

$$\begin{aligned} x_{obs} &= x_{uav} - R\sin(\theta), \\ y_{obs} &= y_{uav} + R\sin(\phi), \\ z_{obs} &= z_{uav} - R\cos(\phi)\cos(\theta). \end{aligned} \tag{11}$$

C.2 Upward Facing TeraRanger One

$$\begin{aligned} x_{obs} &= x_{uav} + R\sin(\theta), \\ y_{obs} &= y_{uav} - R\sin(\phi), \\ z_{obs} &= z_{uav} + R\cos(\phi)\cos(\theta). \end{aligned} \tag{12}$$

C.3 TeraRanger Tower

Note that γ is the Time-of-Flight sensor angle with respect to x in clockwise direction.

$$\begin{aligned} x_{obs} &= x_{uav} + R\cos(\gamma - \psi)\cos(\theta), \\ y_{obs} &= y_{uav} - R\sin(\gamma - \psi)\cos(\phi), \\ z_{obs} &= z_{uav} - R\sin(\gamma)\sin(\phi) - R\cos(\gamma)\sin(\theta). \end{aligned} \tag{13}$$

D Appendix - Python World Main Script Including Cruise With Potential Fields And Simulated Annealing

```
"""
Potential Field Based Path Planner
Author: Pablo Hermoso Moreno

Version: 24_05_2018 - V4:

- Potential Field Based Navigation.
- 2D (x-y) Plane
- Class Based Python Algorithm
- Radius of Drone -> d_eff = d - r_drone : Avoid Collision.
- Radius of Goal -> Once inside: Target Acquisition Mechanism Triggered.
- Vectorised Motion: Assumed Constant Velocity Motion s.t delta_x = u * delta_t
- Continuous Walls and Obstacles
- Perfect Localisation
- Imperfect Sensors -> Teraranger Tower (8 * 45 deg apart)
    - Range Precision of +- 4cm has been incorporated
    - Field of View has NOT been modelled. i.e Field of View = 0 deg
- Static Environment
- Local Minima Solver
    - Via Simulated Annealing

Work In Progress for - V5:
- Coupled Localsiation and Mapping

Description:
Potential Field based planners work using gradient descent to go in direction which locally minimises
potential function.
Obstacles are positively charged and Goal is negatively charged.

# IMPORTANT NOTES:
- Coordinates (CP) are rounded to 4 d.p -> For Teraranger detection purposes.
    - This GLOBAL variable has been left to the used for modification.

"""

# Import Libraries
import numpy as np
import matplotlib.pyplot as plt

# Parameters
KP = 5 # Attractive Potential Gain (zeta in PRM Book)
ETA = 0.1  # Repulsive Potential Gain

# Map Dimensions & Resolution:
mapSizeX = 4
mapSizeY = 3
res_map = 0.5

# Drone Radius and Goal Radius
r_drone = 0.35
r_goal = 0.20

# Precision Parameters Model:
dp_coord = 4
dp_ang = 3

# Dynamics of Drone:

# Key Parameter Definitions:
        # - delta_t : Set timestep between position updates.
        # - alpha_pot : Gradient Descent Factor Gain Factor (Critical to limit velocity outputs)

delta_t = 0.1
alpha_pot = 0.0125

# Maximum and Minimum Velocities:
max_vel = 0.3
min_vel = 0.1

# Sensor Parameters:
lidar_range = 14 # [m]
lidar_precision = 0.04 # [m] - Remember +-

# White Noise Addition:
add_sensor_noise = True

# Potential Function Parameters:

# Attractive Potential:
# Threshold Distance d^{*}_{goal}
d_star = 3  # [m]
```

```
# Repulsive Potential:
# Critical Range Q^{*}
q_star = 1.0 # [m]

# Local Minima Solving: Simulated Annealing ->

# 1. Identify Local Minimum State:
r_local_min = 0.1 # [m]
num_traj_local_min = 20

# 2. Trigger Simulated Annealing Algorithm:
r_sim_ann = 0.1 #[m]
disc_theta_sim_ann = 5 #[deg]
T_0 = 500 #[K]

# show_animation -> if false - Trajectory and Vectors not plotted.
show_animation = True

#################################################################################

# USEFUL FUNCTIONS:

# 1. Modified Vertical Stack:
    # - Includes check for empty arrays.
def vstack_mod(or_array,new_array):
    if len(or_array) > 1:
        return np.vstack([or_array, new_array])
    else:
        return new_array

#################################################################################

# CLASS DEFINITIONS:

# ENVIRONMENT:

class environment:

    # Initialisation:
        # Key Parameter:
        # - obs_env : n*2 array containing (x,y) of obstacles in Environment.
        # - st : Start (x,y) Location [m]
        # - go : Goal (x,y) Location [m]
    def __init__(self, size_x , size_y , go = np.array([]) , st = np.array([])):
        self.size_x = size_x
        self.size_y = size_y
        self.obs_env = np.array([ np.array([]) ])
        self.go = go
        self.st = st

    # ADDING OBSTACLES
    # Terminology:
        # - disc : Discretisation of Obstacle (x,y)
        # (For Sinusoidal)
        # - amp : Amplitude of Oscillation
        # - freq : Frequency of Oscillation

    # 1. Simple Point:
    def add_obs_env_simple(self,co_obs_simple):
        self.obs_env = vstack_mod( self.obs_env , co_obs_simple )

    # 2. Straight Horizontal Wall:
    def add_straight_hor_wall(self,x_in,y_in,length,disc):

        # Coordinates Of Wall:
        x_wall = np.arange(x_in, x_in + length, disc)
        y_wall = np.ones(x_wall.size) * y_in

        for i in range(x_wall.size):
            self.obs_env = vstack_mod(self.obs_env, [round(x_wall[i], dp_coord), round(y_wall[i],
dp_coord)] )

    # 2. Straight Vertical Wall:
    def add_straight_ver_wall(self, x_in, y_in, length, disc):

        # Coordinates Of Wall:
        y_wall = np.arange(y_in, y_in + length, disc)
        x_wall = np.ones(y_wall.size) * x_in

        for i in range(y_wall.size):
            self.obs_env = vstack_mod(self.obs_env, [round(x_wall[i], dp_coord), round(y_wall[i],
dp_coord)])
```

```
    # 3. Sinusoidally Curved Horizontal Wall:
    def add_sinusoid_hor_wall(self,x_in,y_in,length,disc,amp,freq):

        # Coordinates Of Wall:
        x_wall = np.arange(x_in, x_in + length, disc)
        y_wall = y_in + (amp * np.sin(x_wall * freq))

        for i in range(x_wall.size):
            self.obs_env = vstack_mod(self.obs_env, [round(x_wall[i], dp_coord), round(y_wall[i],
dp_coord)] )

    # 3. Sinusoidally Curved Vertical Wall:
    def add_sinusoid_ver_wall(self, x_in, y_in, length, disc, amp, freq):

        # Coordinates Of Wall:
        y_wall = np.arange(y_in, y_in + length, disc)
        x_wall = x_in + (amp * np.sin(y_wall * freq))

        for i in range(x_wall.size):
            self.obs_env = vstack_mod(self.obs_env, [round(x_wall[i], dp_coord), round(y_wall[i],
dp_coord)])

    # 4. Circular Obstacle;
    def add_circ_obs(self,x_c,y_c,r_obs,disc_theta):
        theta = 0
        while theta < 360:
            self.obs_env = vstack_mod(self.obs_env, [round(x_c + r_obs * np.cos(np.deg2rad(theta)),
dp_coord), round(y_c + r_obs * np.sin(np.deg2rad(theta)), dp_coord)])
            theta = theta + disc_theta

    # VISUALISATION
    def plot_map(self):

        # Equal Grid
        plt.grid(True)
        plt.axis("equal")

        # Plot Walls & Obstacle Map
        if len(self.obs_env) > 1:
            plt.plot( self.obs_env[:,0] / res_map, self.obs_env[:,1] / res_map, '.k', markersize=1)

        # Start And Goal Positions:
        plt.plot(self.st[0] / res_map, self.st[1] / res_map, "*k")  # START
        plt.plot(self.go[0] / res_map, self.go[1] / res_map, "*m")  # GOAL

class uav:

    # Initialisation:
    # Key Parameter:
    # - obs_quad : n*2 array containing (x,y) of obstacles DETECTED BY UAV.
    # - r_quad : Trajectory (x,y) History [m]
    def __init__(self, cp):
        self.cp = cp

        # Drone's Obstacle Map:
        self.obs_quad = np.array([[]])

        # Drone's Trajectory
        self.r_quad = np.array([cp])

    # Distance Return: Hypotenuse to any location.
    def dist(self, pos):
        d = np.hypot(self.cp[0] - pos[0], self.cp[1] - pos[1])
        return d

    def update_pos(self,u_res,v_res,go):


        # Check for Existance of Local Minima State:
        if self.check_local_min():
            # Trigger Simulated Annealing Algorithm:
            [u_res,v_res] = self.sim_annealing_alg(go)

        # Gradient Descent Method:

        # Avoid Backward Motion of Drone -> Eliminate Tendency of Escaping: # NOT NECESSARY!
        #if u_res < 0:
            #u_res = 0

        # Velocity limits are set by modification of the gradient descent scaling factor alpha.
```

```
        alpha = alpha_pot

        if np.hypot(alpha * u_res, alpha * v_res) == 0:
            alpha = 0
        elif np.hypot(alpha * u_res, alpha * v_res) > max_vel: # Maximum Velocity Limit:
            alpha = max_vel / (np.hypot(u_res, v_res))
        elif np.hypot(alpha * u_res, alpha * v_res) < min_vel: # Minimum Velocity Limit:
            alpha = min_vel / (np.hypot(u_res, v_res))

        # Update Position:
            # - We assume constant velocity in delta_t
        self.cp[0] += alpha * u_res * delta_t
        self.cp[1] += alpha * v_res * delta_t

        # Round to 4 d.p
            # - Rounding is critical for numerical errors.
        self.cp[0] = round(self.cp[0], 4)
        self.cp[1] = round(self.cp[1], 4)

        # Update the trajectory:
        self.r_quad = np.vstack([self.r_quad, self.cp])

    def terraranger_scan(self,obs_env):


        # Buffer: These arrays will hold the obstacles detected by each sensor. Only the closest one
will be appended to obs_quad.
        buffer_0 = np.array([[]])
        buffer_180 = np.array([[]])
        buffer_90 = np.array([[]])
        buffer_270 = np.array([[]])
        buffer_45 = np.array([[]])
        buffer_135 = np.array([[]])
        buffer_225 = np.array([[]])
        buffer_315 = np.array([[]])

        if obs_env.size != 0:
            for i in range(len(obs_env)):

                if len(obs_env.shape) != 1:
                    d = np.hypot(self.cp[0] - obs_env[i][0], self.cp[1] - obs_env[i][1])
                else:
                    d = np.hypot(self.cp[0] - obs_env[0], self.cp[1] - obs_env[1])


                # Obstancle Detection:
                # 1. Must be within visible range
                # 2. Should not have been recorded previously
                # 3. Only record closest obstacle in sight of LIDAR (no vision across obstacles).

                if (d <= lidar_range):

                    # Calculate Angle to Obstacle:
                    if len(obs_env) != 0:
                        theta = np.arctan2(obs_env[i][1] - self.cp[1], obs_env[i][0] - self.cp[0])
# -pi / +pi

                    else:
                        theta = np.arctan2(obs_env[1] - self.cp[1], obs_env[0] - self.cp[0])  # -pi
 / +pi

                    # Round to Specifiend Precision -> For Numerical Innacuracies
                    theta = round(theta,dp_ang)

                    # 0 Degrees:
                    if theta == round(0, dp_ang):
                        buffer_0 = vstack_mod(buffer_0,obs_env[i])

                    # 45 Degrees:
                    if theta == round(np.pi / 4, dp_ang):
                        buffer_45 = vstack_mod(buffer_45, obs_env[i])

                    # 90 Degrees:
                    if theta == round(np.pi / 2, dp_ang):
                        buffer_90 = vstack_mod(buffer_90, obs_env[i])

                    # 135 Degrees:
                    if theta == round(3 * np.pi / 4, dp_ang):
                        buffer_135 = vstack_mod(buffer_135, obs_env[i])

                    # 180 Degrees:
                    if theta == round(np.pi, dp_ang):
```

```
                        buffer_180 = vstack_mod(buffer_180, obs_env[i])

                    # 225 Degrees:
                    if theta == round(- 3 * np.pi / 4, dp_ang):
                        buffer_225 = vstack_mod(buffer_225, obs_env[i])

                    # 270 Degrees:
                    if theta == round( - np.pi / 2, dp_ang):
                        buffer_270 = vstack_mod(buffer_270, obs_env[i])

                    # 315 Degrees:
                    if theta == round( - np.pi / 4, dp_ang):
                        buffer_315 = vstack_mod(buffer_315, obs_env[i])

        # Find the closest distance obstacle in the buffer for the sensors and append to obs_quad:

        # NOTE:
        # minid = -1 - EMPTY
        # minid = -2 - 1 ENTRY
        # minid >= 0 - MORE THAN 1 ENTRY

        # Noise addition:
        if add_sensor_noise:

            # Range Precision:

            # 0 deg:
            buffer_0 += (2 * lidar_precision * (np.random.random(buffer_0.shape))) - lidar_precision
            # 45 deg:
            buffer_45 += (2 * lidar_precision * (np.random.random(buffer_45.shape))) - lidar_precision
            # 90 deg:
            buffer_90 += (2 * lidar_precision * (np.random.random(buffer_90.shape))) - lidar_precision
            # 135 deg:
            buffer_135 += (2 * lidar_precision * (np.random.random(buffer_135.shape))) -
lidar_precision
            # 180 deg:
            buffer_180 += (2 * lidar_precision * (np.random.random(buffer_180.shape))) -
lidar_precision
            # 225 deg:
            buffer_225 += (2 * lidar_precision * (np.random.random(buffer_225.shape))) -
lidar_precision
            # 270 deg:
            buffer_270 += (2 * lidar_precision * (np.random.random(buffer_270.shape))) -
lidar_precision
            # 315 deg:
            buffer_315 += (2 * lidar_precision * (np.random.random(buffer_315.shape))) -
lidar_precision

        minid_front = -1
        dmin_front = float("inf")
        minid_back = -1
        dmin_back = float("inf")
        minid_right = -1
        dmin_right = float("inf")
        minid_left = -1
        dmin_left = float("inf")
        minid_45 = -1
        dmin_45 = float("inf")
        minid_135 = -1
        dmin_135 = float("inf")
        minid_225 = -1
        dmin_225 = float("inf")
        minid_315 = -1
        dmin_315 = float("inf")

        if buffer_0.size != 0:  # BUFFER NOT EMPTY
            for i in range(len(buffer_0)):
                if len(buffer_0.shape) != 1:
                    d = np.hypot(self.cp[0] - buffer_0[i][0], self.cp[1] - buffer_0[i][1])
                else:
                    minid_front = -2
                    break

                # Search for closest distance obstacle.
                if dmin_front >= d:
                    dmin_front = d
                    minid_front = i

            # Collection of if statements used to find the appropiate way to append the new obstacle
to obs_quad:
            if (self.obs_quad.size == 0) & (minid_front == -2):
                self.obs_quad = buffer_0
```

```
            elif (self.obs_quad.size == 0) & (minid_front >= 0):
                self.obs_quad = buffer_0[minid_front]
            elif (self.obs_quad.size == 2) & (minid_front == -2):
                if (not (self.obs_quad == buffer_0).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_0])
            elif (self.obs_quad.size == 2) & (minid_front >= 0):
                if (not (self.obs_quad == buffer_0[minid_front]).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_0[minid_front]])
            elif (self.obs_quad.size > 2) & (minid_front == -2):
                if (not (self.obs_quad == buffer_0).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_0])
            elif (self.obs_quad.size > 2) & (minid_front >= 0):
                if (not (self.obs_quad == buffer_0[minid_front]).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_0[minid_front]])

        if buffer_180.size != 0:  # BUFFER NOT EMPTY
            for i in range(len(buffer_180)):
                if len(buffer_180.shape) != 1:
                    d = np.hypot(self.cp[0] - buffer_180[i][0], self.cp[1] - buffer_180[i][1])
                else:
                    minid_back = -2
                    break

                # Search for closest distance obstacle.
                if dmin_back >= d:
                    dmin_back = d
                    minid_back = i

            # Collection of if statements used to find the appropiate way to append the new obstacle
to obs_quad:
            if (self.obs_quad.size == 0) & (minid_back == -2):
                self.obs_quad = buffer_180
            elif (self.obs_quad.size == 0) & (minid_back >= 0):
                self.obs_quad = buffer_180[minid_back]
            elif (self.obs_quad.size == 2) & (minid_back == -2):
                if (not (self.obs_quad == buffer_180).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_180])
            elif (self.obs_quad.size == 2) & (minid_back >= 0):
                if (not (self.obs_quad == buffer_180[minid_back]).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_180[minid_back]])
            elif (self.obs_quad.size > 2) & (minid_back == -2):
                if (not (self.obs_quad == buffer_180).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_180])
            elif (self.obs_quad.size > 2) & (minid_back >= 0):
                if (not (self.obs_quad == buffer_180[minid_back]).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_180[minid_back]])

        if buffer_90.size != 0:  # BUFFER NOT EMPTY
            for i in range(len(buffer_90)):
                if len(buffer_90.shape) != 1:
                    d = np.hypot(self.cp[0] - buffer_90[i][0], self.cp[1] - buffer_90[i][1])
                else:
                    minid_right = -2  # Only one obstacle found
                    break

                # Search for closest distance obstacle.
                if dmin_right >= d:
                    dmin_right = d
                    minid_right = i

            # Collection of if statements used to find the appropiate way to append the new obstacle
to obs_quad:
            if (self.obs_quad.size == 0) & (minid_right == -2):
                self.obs_quad = buffer_90
            elif (self.obs_quad.size == 0) & (minid_right >= 0):
                self.obs_quad = buffer_90[minid_right]
            elif (self.obs_quad.size == 2) & (minid_right == -2):
                if (not (self.obs_quad == buffer_90).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_90])
            elif (self.obs_quad.size == 2) & (minid_right >= 0):
                if (not (self.obs_quad == buffer_90[minid_right]).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_90[minid_right]])
            elif (self.obs_quad.size > 2) & (minid_right == -2):
                if (not (self.obs_quad == buffer_90).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_90])
            elif (self.obs_quad.size > 2) & (minid_right >= 0):
                if (not (self.obs_quad == buffer_90[minid_right]).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_90[minid_right]])

        if buffer_270.size != 0:  # BUFFER NOT EMPTY
            for i in range(len(buffer_270)):
                if len(buffer_270.shape) != 1:
```

```
                d = np.hypot(self.cp[0] - buffer_270[i][0], self.cp[1] - buffer_270[i][1])
            else:
                minid_left = -2
                break

            # Search for closest distance obstacle.
            if dmin_left >= d:
                dmin_left = d
                minid_left = i

        # Collection of if statements used to find the appropiate way to append the new obstacle
to obs_quad:
        if (self.obs_quad.size == 0) & (minid_left == -2):
            self.obs_quad = buffer_270
        elif (self.obs_quad.size == 0) & (minid_left >= 0):
            self.obs_quad = buffer_270[minid_left]
        elif (self.obs_quad.size == 2) & (minid_left == -2):
            if (not (self.obs_quad == buffer_270).all().any()):
                self.obs_quad = np.vstack([self.obs_quad, buffer_270])
        elif (self.obs_quad.size == 2) & (minid_left >= 0):
            if (not (self.obs_quad == buffer_270[minid_left]).all().any()):
                self.obs_quad = np.vstack([self.obs_quad, buffer_270[minid_left]])
        elif (self.obs_quad.size > 2) & (minid_left == -2):
            if (not (self.obs_quad == buffer_270).all(1).any()):
                self.obs_quad = np.vstack([self.obs_quad, buffer_270])
        elif (self.obs_quad.size > 2) & (minid_left >= 0):
            if (not (self.obs_quad == buffer_270[minid_left]).all(1).any()):
                self.obs_quad = np.vstack([self.obs_quad, buffer_270[minid_left]])

    if buffer_45.size != 0:  # BUFFER NOT EMPTY
        for i in range(len(buffer_45)):
            if len(buffer_45.shape) != 1:
                d = np.hypot(self.cp[0] - buffer_45[i][0], self.cp[1] - buffer_45[i][1])
            else:
                minid_45 = -2
                break

            # Search for closest distance obstacle.
            if dmin_45 >= d:
                dmin_45 = d
                minid_45 = i

        # Collection of if statements used to find the appropiate way to append the new obstacle
to obs_quad:
        if (self.obs_quad.size == 0) & (minid_45 == -2):
            self.obs_quad = buffer_45
        elif (self.obs_quad.size == 0) & (minid_45 >= 0):
            self.obs_quad = buffer_45[minid_45]
        elif (self.obs_quad.size == 2) & (minid_45 == -2):
            if (not (self.obs_quad == buffer_45).all().any()):
                self.obs_quad = np.vstack([self.obs_quad, buffer_45])
        elif (self.obs_quad.size == 2) & (minid_45 >= 0):
            if (not (self.obs_quad == buffer_45[minid_45]).all().any()):
                self.obs_quad = np.vstack([self.obs_quad, buffer_45[minid_45]])
        elif (self.obs_quad.size > 2) & (minid_45 == -2):
            if (not (self.obs_quad == buffer_45).all(1).any()):
                self.obs_quad = np.vstack([self.obs_quad, buffer_45])
        elif (self.obs_quad.size > 2) & (minid_45 >= 0):
            if (not (self.obs_quad == buffer_45[minid_45]).all(1).any()):
                self.obs_quad = np.vstack([self.obs_quad, buffer_45[minid_45]])

    if buffer_135.size != 0:  # BUFFER NOT EMPTY
        for i in range(len(buffer_135)):
            if len(buffer_135.shape) != 1:
                d = np.hypot(self.cp[0] - buffer_135[i][0], self.cp[1] - buffer_135[i][1])
            else:
                minid_135 = -2
                break

            # Search for closest distance obstacle.
            if dmin_135 >= d:
                dmin_135 = d
                minid_135 = i

        # Collection of if statements used to find the appropiate way to append the new obstacle
to obs_quad:
        if (self.obs_quad.size == 0) & (minid_135 == -2):
            self.obs_quad = buffer_135
        elif (self.obs_quad.size == 0) & (minid_135 >= 0):
            self.obs_quad = buffer_135[minid_135]
        elif (self.obs_quad.size == 2) & (minid_135 == -2):
            if (not (self.obs_quad == buffer_135).all().any()):
```

```
                    self.obs_quad = np.vstack([self.obs_quad, buffer_135])
            elif (self.obs_quad.size == 2) & (minid_135 >= 0):
                if (not (self.obs_quad == buffer_135[minid_135]).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_135[minid_135]])
            elif (self.obs_quad.size > 2) & (minid_135 == -2):
                if (not (self.obs_quad == buffer_135).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_135])
            elif (self.obs_quad.size > 2) & (minid_135 >= 0):
                if (not (self.obs_quad == buffer_135[minid_135]).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_135[minid_135]])

        if buffer_225.size != 0:  # BUFFER NOT EMPTY
            for i in range(len(buffer_225)):
                if len(buffer_225.shape) != 1:
                    d = np.hypot(self.cp[0] - buffer_225[i][0], self.cp[1] - buffer_225[i][1])
                else:
                    minid_225 = -2
                    break

                # Search for closest distance obstacle.
                if dmin_225 >= d:
                    dmin_225 = d
                    minid_225 = i

            # Collection of if statements used to find the appropiate way to append the new obstacle
to obs_quad:
            if (self.obs_quad.size == 0) & (minid_225 == -2):
                self.obs_quad = buffer_225
            elif (self.obs_quad.size == 0) & (minid_225 >= 0):
                self.obs_quad = buffer_225[minid_225]
            elif (self.obs_quad.size == 2) & (minid_225 == -2):
                if (not (self.obs_quad == buffer_225).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_225])
            elif (self.obs_quad.size == 2) & (minid_225 >= 0):
                if (not (self.obs_quad == buffer_225[minid_225]).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_225[minid_225]])
            elif (self.obs_quad.size > 2) & (minid_225 == -2):
                if (not (self.obs_quad == buffer_225).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_225])
            elif (self.obs_quad.size > 2) & (minid_225 >= 0):
                if (not (self.obs_quad == buffer_225[minid_225]).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_225[minid_225]])

        if buffer_315.size != 0:  # BUFFER NOT EMPTY
            for i in range(len(buffer_315)):
                if len(buffer_315.shape) != 1:
                    d = np.hypot(self.cp[0] - buffer_315[i][0],self.cp[1] - buffer_315[i][1])
                else:
                    minid_315 = -2
                    break

                # Search for closest distance obstacle.
                if dmin_315 >= d:
                    dmin_315 = d
                    minid_315 = i

            # Collection of if statements used to find the appropiate way to append the new obstacle
to obs_quad:
            if (self.obs_quad.size == 0) & (minid_315 == -2):
                self.obs_quad = buffer_315
            elif (self.obs_quad.size == 0) & (minid_315 >= 0):
                self.obs_quad = buffer_315[minid_315]
            elif (self.obs_quad.size == 2) & (minid_315 == -2):
                if (not (self.obs_quad == buffer_315).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_315])
            elif (self.obs_quad.size == 2) & (minid_315 >= 0):
                if (not (self.obs_quad == buffer_315[minid_315]).all().any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_315[minid_315]])
            elif (self.obs_quad.size > 2) & (minid_315 == -2):
                if (not (self.obs_quad == buffer_315).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_315])
            elif (self.obs_quad.size > 2) & (minid_315 >= 0):
                if (not (self.obs_quad == buffer_315[minid_315]).all(1).any()):
                    self.obs_quad = np.vstack([self.obs_quad, buffer_315[minid_315]])

        return self.obs_quad

    def calc_potential_field_quad(self, go, obs_env):

        # Scan the environment and add LIDAR detected obstacles.
        obs_quad = self.terraranger_scan(obs_env)
```

```
        # For the current position, calculate the Resultant Gradient Descent:
        [u_g, v_g] = self.calc_attractive_potential_vector(go)

        # Calculate Repulsive Potential With Respect to Obstacles
        [u_o, v_o] = self.calc_repulsive_potential_vector()

        return [u_g + u_o, v_g + v_o], obs_quad

    def calc_attractive_potential_vector(self,go):
        # Need to implement Combined (Quadratic + Conic) Variation.

        # Reason: In some cases, it may be desirable to have distance functions
        # that grow more slowly to avoid huge velocities far from the goal.

        # Distance to Goal
        d = np.hypot(go[0] - self.cp[0], go[1] - self.cp[1])

        # Angle to Goal:
        theta = np.arctan2(go[1] - self.cp[1], go[0] - self.cp[0])  # -pi / +pi

        # Gradient Calculation
        if d <= d_star:
            return [KP * d * np.cos(theta), KP * d * np.sin(theta)]
        else:
            return [d_star * KP * np.cos(theta), d_star * KP * np.sin(theta)]

            # Previous version would only use d and not d**2:
            # return 0.5 * KP * np.hypot(x - go[0], y - go[1])

    def calc_repulsive_potential_vector(self):

        # Better implementation of the repulsive potential would be to sum the actions of all the
individual repulsive
        # potentials of the obtacles within Vision Range.

        # Previous model would search only for closest obsacle and take the action of only that one.
        # Problem: When numerically implementing this solution, a path may form that oscillates around
 points that are
        #  two-way equidistant from obstacles, i.e., points where D is nonsmooth. To avoid these
oscillations,
        # instead of defining the repulsive potential function in terms of distance to the closest
obstacle,
        # the repulsive potential function (4.6) is redefined in terms of distances to individual
obstacles
        # where d i ( q )i s the distance to obstacle.

        u_rep = 0
        v_rep = 0

        if self.obs_quad.size != 0:
            for i in range(len(self.obs_quad)):
                theta = 0
                if len(self.obs_quad.shape) != 1:
                    d = np.hypot(self.cp[0] - self.obs_quad[i][0], self.cp[1] - self.obs_quad[i][1])

                    # Dylan's Collision Avoidance:
                    #d_eff = d - r_drone: Avoid Collision
                    d -= r_drone

                    theta = np.arctan2(self.obs_quad[i][1] - self.cp[1], self.obs_quad[i][0] - self.cp
[0])  # -pi / +pi
                else:
                    d = np.hypot(self.cp[0] - self.obs_quad[0], self.cp[1] - self.obs_quad[1])

                    # Dylan's Collision Avoidance:
                    d -= r_drone

                    theta = np.arctan2(self.obs_quad[1] - self.cp[1], self.obs_quad[0] - self.cp[0])
# -pi / +pi

                if (d <= q_star):
                    u_rep += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * np.cos(theta)
                    v_rep += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * np.sin(theta)

        return [u_rep, v_rep]

    # Simulated Annealing Algorithm:
    def calc_attractive_potential(self,x, y, go):
        # Combined (Quadratic + Conic) Variation.

        # Reason: In some cases, it may be desirable to have distance functions
        # that grow more slowly to avoid huge velocities far from the goal.
```

```
        # MODIFIED KP FOR SIMULATED ANNEALING:
        #KP = 5

        # Distance to Goal
        d = np.hypot(go[0] - x, go[1] - y)

        if d <= d_star:
            return 0.5 * KP * (d ** 2)
        else:
            return (d_star * KP * d) - (0.5 * KP * (d_star ** 2))

    def calc_repulsive_potential(self,x, y):

        # Better implementation of the repulsive potential would be to sum the actions of all the
individual repulsive
        # potentials of the obtacles within Vision Range.

        # Previous model would search only for closest obsacle and take the action of only that one.
        # Problem: When numerically implementing this solution, a path may form that oscillates around
 points that are
        #  two-way equidistant from obstacles, i.e., points where D is nonsmooth. To avoid these
oscillations,
        # instead of defining the repulsive potential function in terms of distance to the closest
obstacle,
        # the repulsive potential function (4.6) is redefined in terms of distances to individual
obstacles
        # where d i ( q )i s the distance to obstacle.

        # MODIFIED ETA FOR SIMULATED ANNEALING:
        ETA = 5

        u_rep = 0

        for i in range(len(self.obs_quad)):
            if len(self.obs_quad.shape) != 1:
                d = np.hypot(x - self.obs_quad[i][0], y - self.obs_quad[i][1])
            else:
                d = np.hypot(x - self.obs_quad[0], y - self.obs_quad[1])

            # Dylan's Collision Avoidance:
            # d_eff = d - r_drone: Avoid Collision
            d -= r_drone

            # Check if the obstacle if within Vision Range Q^{*}.
            if (d <= q_star):
                u_rep += 0.5 * ETA * ((1.0 / d) - (1.0 / q_star)) ** 2

        return u_rep

    def check_local_min(self):

        # Check if the latest "num_traj_local_min" trajectory lies with radius "r_local_min" from
current trajectory:
        if len(self.r_quad) > num_traj_local_min:

            d = self.dist(self.r_quad[-num_traj_local_min])

            if d < r_local_min: # Radius: "r_local_min"
                return True
            else:
                return False

    def sim_annealing_alg(self,go):

        # Set Initial Temperature:
        #global T_0

        # Current U(x)
        ug_cp = self.calc_attractive_potential(self.cp[0], self.cp[1], go)
        uo_cp = self.calc_repulsive_potential(self.cp[0], self.cp[1])

        theta = 0
        while theta < 360:

            # 1. Calculate x' = x + delta_x
            x_new = self.cp[0] + r_sim_ann * np.cos(np.deg2rad(theta))
            y_new = self.cp[1] + r_sim_ann * np.sin(np.deg2rad(theta))

            # 2. Calculate U(x')
            ug_new = self.calc_attractive_potential(x_new,y_new,go)
            uo_new = self.calc_repulsive_potential(x_new,y_new)
```

```
            # 3. Calculate delta_U
            delta_U = (-ug_new + uo_new) - (-ug_cp + uo_cp)

            # 4. Probabilistic Algorithm:
            if delta_U < 0:
                # Found Direction With Negative Gradient:
                break
            else:
                # Probability:
                p_new = np.exp( (-delta_U) / (T_0) )
                if np.random.random_sample() < p_new:
                    break

            theta += disc_theta_sim_ann

        # Reduce T_0:
        #T_0 = 0.90 * T_0
        #print(T_0)

        # Once the angle has been set, the drone will try to escape local minima at minimum velocity:
        u_res = min_vel * np.cos(np.deg2rad(theta))
        v_res = min_vel * np.sin(np.deg2rad(theta))
        return [u_res,v_res]

def main():
    print("Potential_Field_Planning_Start")

    # 1. Set the Environment

    # MAKE SURE THE STARTING POSITION IS NOT ON A BORDER
    st = np.array([0.5, 1.5])  # Start x and y position [m]
    go = np.array([3.5, 1.5])  # Goal x and y position [m]

    # Environment Class:
    env = environment(mapSizeX, mapSizeY, go, st)

    # Add Walls & Obstacles:

    # env.add_sinusoid_hor_wall(0,0,mapSizeX,0.01,0.2,2)
    # env.add_sinusoid_hor_wall(0,mapSizeY,mapSizeX,0.01,0.2,4)

    # Regular Obstacles: No Local Minima ->

    # This obstacle causes local minima apparition:
    #env.add_circ_obs(2,1.5,0.05,0.5)

    # env.add_circ_obs(1.5, 2, 0.15, 0.5)
    # env.add_circ_obs(1.0, 2.5, 0.1, 0.5)
    # env.add_circ_obs(0.45, 0.5, 0.20, 0.5)
    # env.add_circ_obs(2.6, 0.2, 0.25, 0.5)


    # Local Minima Obstacles ->
    # Straight Vertical Wall

    env.add_straight_hor_wall(0,0,4,0.005)
    env.add_straight_hor_wall(0,3,4,0.005)
    env.add_straight_ver_wall(2,1.25,0.45,0.001)


    # Visualisation:
    env.plot_map()

    # 2. Set the Drone:

    # Drone Class:
    drone = uav(st)

    # 3. Potential Fields Solver:

    # Global Distance from Start to Goal:
    d = drone.dist(env.go)  # Global Distance To Target

    # VISUALISATION: Initialise ->

    # Subplot Base:
    ax = plt.gca()

    # Plot Drone Radius:
    circle_drone = plt.Circle((drone.cp[0] / res_map, drone.cp[1] / res_map), r_drone / res_map, color
='r', fill=False)
```

```
    ax.add_artist(circle_drone)

    # Plot Goal Radius:
    circle_goal = plt.Circle((env.go[0] / res_map, env.go[1] / res_map), r_goal / res_map, color='b',
fill=False)
    ax.add_artist(circle_goal)

    # Plot Resultant Vector:
    vector_drone = plt.quiver(drone.cp[0] / res_map, drone.cp[1] / res_map, 0, 0, edgecolor='k',
facecolor='None',linewidth=.5)
    ax.add_artist(vector_drone)

    while d >= r_goal:  # UAV within Goal Radius

        # Calculate Gradient
        [u_res, v_res], obs_quad = drone.calc_potential_field_quad(env.go,env.obs_env)

        drone.update_pos(u_res,v_res,env.go)

        # Re-Calculate the Distance:
        d = drone.dist(env.go)

        if show_animation:

            plt.plot(drone.cp[0] / res_map, drone.cp[1] / res_map, ".r")

            if drone.obs_quad.size != 0:
                if drone.obs_quad.size > 2:
                    plt.plot(drone.obs_quad[:,0]/res_map ,drone.obs_quad[:,1]/res_map ,"sg")
                else:
                    plt.plot(drone.obs_quad[0] / res_map, drone.obs_quad[1] / res_map, "sg")

            # Subplot Exists -> Update:

            # Plot Drone Radius:
            circle_drone.remove()
            circle_drone = plt.Circle((drone.cp[0] / res_map, drone.cp[1] / res_map), r_drone /
res_map,color='r', fill=False)
            ax.add_artist(circle_drone)

            # Plot Resultant Vector:
            vector_drone.set_visible(False)
            vector_drone = plt.quiver(drone.cp[0] / res_map, drone.cp[1] / res_map, u_res, v_res,
edgecolor='k', facecolor='None',linewidth=.5)
            ax.add_artist(vector_drone)

            plt.pause(0.00000001)

    print("Goal!!")


    if show_animation:
        plt.show()

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

E Appendix - Gazebo Files for Installation Of TeraRanger Sensors - C++ Listener & Makefile

```
/*
 * Listener Script - Simultaneous Gazebo Subscriber and ROS Publisher
 * Author: Pablo Hermoso Moreno
 *
*/

#include <gazebo/transport/transport.hh>
#include <gazebo/msgs/msgs.hh>
#include <gazebo/gazebo_client.hh>
#include <gazebo/sensors/GpuRaySensor.hh>
#include <gazebo/sensors/SensorTypes.hh>

#include "gazebo_plugins/gazebo_ros_gpu_laser.h"
#include <gazebo_plugins/gazebo_ros_utils.h>

#include "ros/ros.h"
#include "std_msgs/Float64MultiArray.h"
#include "std_msgs/String.h"

#include <iostream>
#include <sstream>


ros::Publisher tera_2_pub_rplidar;
ros::Publisher tera_1_pub_up;
ros::Publisher tera_1_pub_down;

typedef const boost::shared_ptr<const gazebo::msgs::LaserScanStamped> ConstLaserScanStampedPtr;

void cb_rplidar(ConstLaserScanStampedPtr &_msg)
{

   // We got a new message from the Gazebo sensor.  Create Corresponding ROS message and publish it.
   sensor_msgs::LaserScan laser_msg;
   laser_msg.header.stamp = ros::Time(_msg->time().sec(), _msg->time().nsec());
   //laser_msg.header.frame_id = this->frame_name_;
   laser_msg.angle_min = _msg->scan().angle_min();
   laser_msg.angle_max = _msg->scan().angle_max();
   laser_msg.angle_increment = _msg->scan().angle_step();
   laser_msg.time_increment = 0;  // instantaneous simulator scan
   laser_msg.scan_time = 0;  // not sure whether this is correct
   laser_msg.range_min = _msg->scan().range_min();
   laser_msg.range_max = _msg->scan().range_max();
   laser_msg.ranges.resize(_msg->scan().ranges_size());
   std::copy(_msg->scan().ranges().begin(),
             _msg->scan().ranges().end(),
             laser_msg.ranges.begin());
   laser_msg.intensities.resize(_msg->scan().intensities_size());
   std::copy(_msg->scan().intensities().begin(),
                _msg->scan().intensities().end(),
             laser_msg.intensities.begin());
   //this->pub_queue_->push(laser_msg, this->pub_);

   //std::cout << laser_msg.range_max->DebugString();

   // Publish in ROS topic
   tera_2_pub_rplidar.publish(laser_msg);

}


void cb_up(ConstLaserScanStampedPtr &_msg)
{
  // Dump the message contents to stdout.
  //std::cout << _msg->DebugString();

   // We got a new message from the Gazebo sensor.  Stuff a
   // corresponding ROS message and publish it.
   sensor_msgs::LaserScan laser_msg;
   laser_msg.header.stamp = ros::Time(_msg->time().sec(), _msg->time().nsec());
   //laser_msg.header.frame_id = this->frame_name_;
   laser_msg.angle_min = _msg->scan().angle_min();
   laser_msg.angle_max = _msg->scan().angle_max();
   laser_msg.angle_increment = _msg->scan().angle_step();
   laser_msg.time_increment = 0;  // instantaneous simulator scan
   laser_msg.scan_time = 0;  // not sure whether this is correct
   laser_msg.range_min = _msg->scan().range_min();
   laser_msg.range_max = _msg->scan().range_max();
   laser_msg.ranges.resize(_msg->scan().ranges_size());
   std::copy(_msg->scan().ranges().begin(),
             _msg->scan().ranges().end(),
             laser_msg.ranges.begin());
```

```
    laser_msg.intensities.resize(_msg->scan().intensities_size());
    std::copy(_msg->scan().intensities().begin(),
                _msg->scan().intensities().end(),
              laser_msg.intensities.begin());
    //this->pub_queue_->push(laser_msg, this->pub_);

    //std::cout << laser_msg.range_max->DebugString();

    // Publish in ROS topic
    tera_1_pub_up.publish(laser_msg);

}


void cb_down(ConstLaserScanStampedPtr &_msg)
{
  // Dump the message contents to stdout.
  //std::cout << _msg->DebugString();

    // We got a new message from the Gazebo sensor.  Stuff a
    // corresponding ROS message and publish it.
    sensor_msgs::LaserScan laser_msg;
    laser_msg.header.stamp = ros::Time(_msg->time().sec(), _msg->time().nsec());
    //laser_msg.header.frame_id = this->frame_name_;
    laser_msg.angle_min = _msg->scan().angle_min();
    laser_msg.angle_max = _msg->scan().angle_max();
    laser_msg.angle_increment = _msg->scan().angle_step();
    laser_msg.time_increment = 0;  // instantaneous simulator scan
    laser_msg.scan_time = 0;  // not sure whether this is correct
    laser_msg.range_min = _msg->scan().range_min();
    laser_msg.range_max = _msg->scan().range_max();
    laser_msg.ranges.resize(_msg->scan().ranges_size());
    std::copy(_msg->scan().ranges().begin(),
              _msg->scan().ranges().end(),
              laser_msg.ranges.begin());
    laser_msg.intensities.resize(_msg->scan().intensities_size());
    std::copy(_msg->scan().intensities().begin(),
                _msg->scan().intensities().end(),
              laser_msg.intensities.begin());
    //this->pub_queue_->push(laser_msg, this->pub_);

    //std::cout << laser_msg.range_max->DebugString();

    // Publish in ROS topic
    tera_1_pub_down.publish(laser_msg);

}

//////////////////////////////////////////////
int main(int _argc, char **_argv)
{

  ros::init(_argc, _argv, "tera_2_node");
  ros::NodeHandle n;

  tera_2_pub_rplidar = n.advertise<sensor_msgs::LaserScan>("tera_2_array", 1000);
  tera_1_pub_up = n.advertise<sensor_msgs::LaserScan>("tera_1_up", 1000);
  tera_1_pub_down = n.advertise<sensor_msgs::LaserScan>("tera_1_down", 1000);
  ros::Rate loop_rate(20);

  // Load gazebo
  gazebo::client::setup(_argc, _argv);

  // Create our node for communication
  gazebo::transport::NodePtr node(new gazebo::transport::Node());
  node->Init();

  // Listen to Gazebo world_stats topic
  // gazebo::transport::SubscriberPtr sub = node->Subscribe("~/world_stats", cb);
  gazebo::transport::SubscriberPtr sub_rplidar = node->Subscribe("~/iris_opt_flow/rplidar/link/laser/
scan",cb_rplidar);
  gazebo::transport::SubscriberPtr sub_up = node->Subscribe("~/iris_opt_flow/lidar2/link/laser/scan",c
b_up);
  gazebo::transport::SubscriberPtr sub_down = node->Subscribe("~/iris_opt_flow/lidar/link/laser/scan",
cb_down);


  // Busy wait loop...replace with your own code as needed.
  while (true)
    gazebo::common::Time::MSleep(10);

  // Make sure to shut everything down.
```

```
  gazebo::client::shutdown();


  ros::spinOnce();
  loop_rate.sleep();

}
```

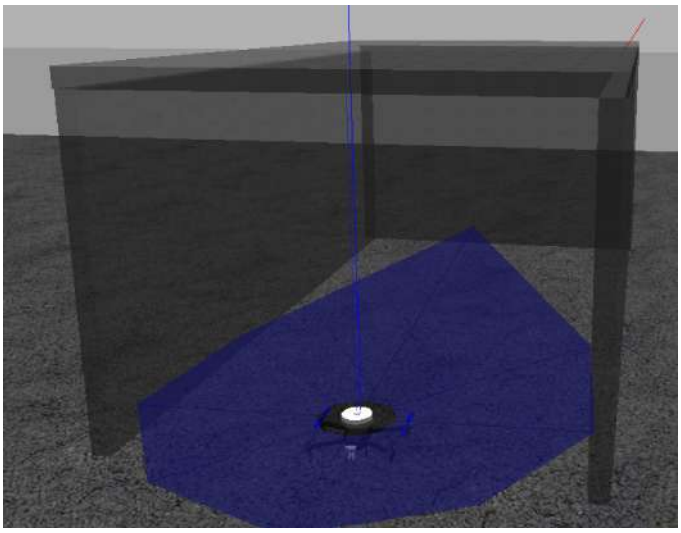
```
cmake_minimum_required(VERSION 2.8.3)
project(tera_2_pkg)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)

catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES tera_2_pkg
#  CATKIN_DEPENDS roscpp rospy std_msgs
#  DEPENDS system_lib
)


## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)



find_package(gazebo REQUIRED)

include_directories(${GAZEBO_INCLUDE_DIRS})

link_directories(${GAZEBO_LIBRARY_DIRS})

list(APPEND CMAKE_CXX_FLAGS "${GAZEBO_CXX_FLAGS}")

add_executable(tera_2_node src/listener.cc)

target_link_libraries(tera_2_node ${catkin_LIBRARIES} ${GAZEBO_LIBRARIES} pthread)


## System dependencies are found with CMake's conventions
# find_package(Boost REQUIRED COMPONENTS system)


## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
# catkin_python_setup()

################################################
## Declare ROS messages, services and actions ##
################################################

## To declare and build messages, services or actions from within this
## package, follow these steps:
## * Let MSG_DEP_SET be the set of packages whose message types you use in
##   your messages/services/actions (e.g. std_msgs, actionlib_msgs, ...).
## * In the file package.xml:
##   * add a build_depend tag for "message_generation"
##   * add a build_depend and a run_depend tag for each package in MSG_DEP_SET
##   * If MSG_DEP_SET isn't empty the following dependency has been pulled in
##     but can be declared for certainty nonetheless:
##     * add a run_depend tag for "message_runtime"
## * In this file (CMakeLists.txt):
##   * add "message_generation" and every package in MSG_DEP_SET to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * add "message_runtime" and every package in MSG_DEP_SET to
##     catkin_package(CATKIN_DEPENDS ...)
##   * uncomment the add_*_files sections below as needed
##     and list every .msg/.srv/.action file to be processed
##   * uncomment the generate_messages entry below
##   * add every package in MSG_DEP_SET to generate_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
```

```
#   Message2.msg
# )

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )

################################################
## Declare ROS dynamic reconfigure parameters ##
################################################

## To declare and build dynamic reconfigure parameters within this
## package, follow these steps:
## * In the file package.xml:
##   * add a build_depend and a run_depend tag for "dynamic_reconfigure"
## * In this file (CMakeLists.txt):
##   * add "dynamic_reconfigure" to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * uncomment the "generate_dynamic_reconfigure_options" section below
##     and list every .cfg file to be processed

## Generate dynamic reconfigure parameters in the 'cfg' folder
# generate_dynamic_reconfigure_options(
#   cfg/DynReconf1.cfg
#   cfg/DynReconf2.cfg
# )

###################################
## catkin specific configuration ##
###################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if you package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need


############
## Build ##
############


## Declare a C++ library
# add_library(${PROJECT_NAME}
#   src/${PROJECT_NAME}/tera_2_pkg.cpp
# )

## Add cmake target dependencies of the library
## as an example, code may need to be generated before libraries
## either from message generation or dynamic reconfigure
# add_dependencies(${PROJECT_NAME} ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
# add_executable(${PROJECT_NAME}_node src/tera_2_pkg_node.cpp)

## Rename C++ executable without prefix
## The above recommended prefix causes long target names, the following renames the
## target back to the shorter version for ease of user use
## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
# set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")

## Add cmake target dependencies of the executable
## same as for the library above
```

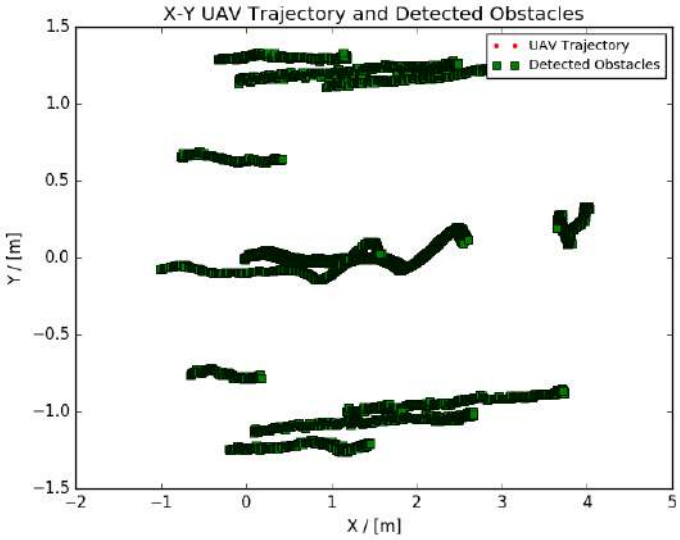
```
# add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS}
)

## Specify libraries to link a library or executable target against
# target_link_libraries(${PROJECT_NAME}_node
#   ${catkin_LIBRARIES}
# )

#############
## Install ##
#############

# all install targets should use catkin DESTINATION variables
# See http://ros.org/doc/api/catkin/html/adv_user_guide/variables.html

## Mark executable scripts (Python etc.) for installation
## in contrast to setup.py, you can choose the destination
# install(PROGRAMS
#   scripts/my_python_script
#   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )

## Mark executables and/or libraries for installation
# install(TARGETS ${PROJECT_NAME} ${PROJECT_NAME}_node
#   ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
#   LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
#   RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )

## Mark cpp header files for installation
# install(DIRECTORY include/${PROJECT_NAME}/
#   DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
#   FILES_MATCHING PATTERN "*.h"
#   PATTERN ".svn" EXCLUDE
# )

## Mark other files for installation (e.g. launch and bag files, etc.)
# install(FILES
#   # myfile1
#   # myfile2
#   DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
# )

#############
## Testing ##
#############

## Add gtest based cpp test target and link libraries
# catkin_add_gtest(${PROJECT_NAME}-test test/test_tera_2_pkg.cpp)
# if(TARGET ${PROJECT_NAME}-test)
#   target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})
# endif()

## Add folders to be run by python nosetests
# catkin_add_nosetests(test)
```

F Appendix - Gazebo Worlds For Testing

F.1 World 1 - No Obstacles



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 13: Gazebo Test World 1 - Simulation Environment and Resultant trajectory and Detected Obstacles

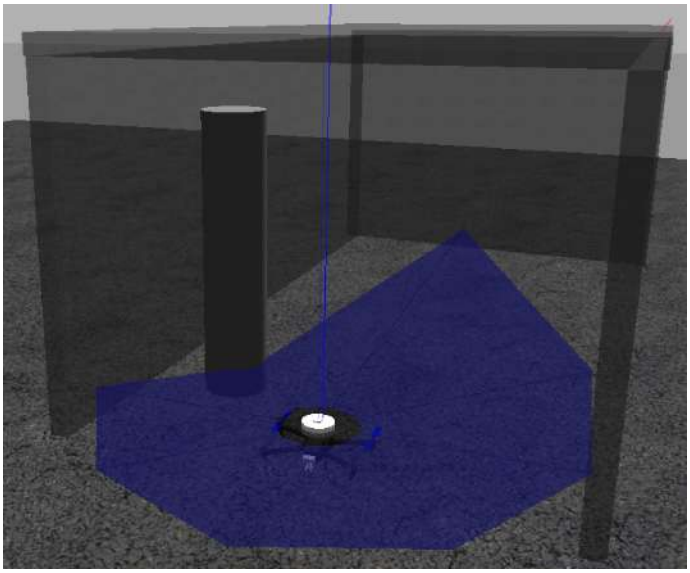
Functionalities to be tested:

- Stage-based flight
- Takeoff PID control
- Obstacle mapping

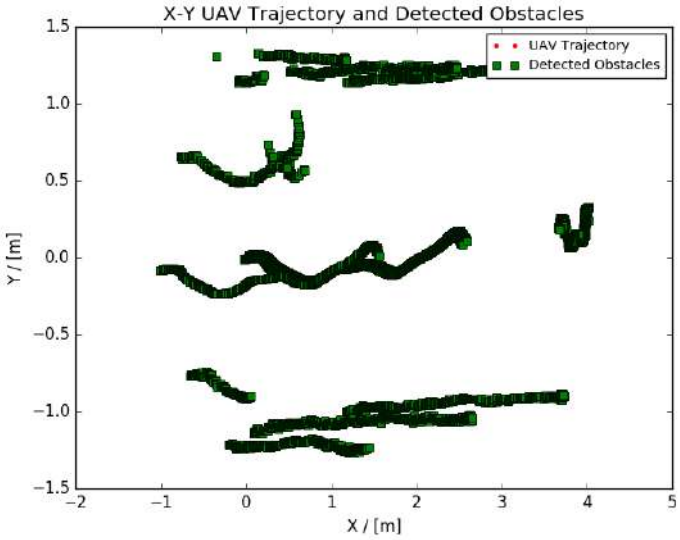
Summary of Results:

- All successful

F.2 World 2 - 1 Obstacle



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 14: Gazebo Test World 2 - Simulation Environment and Resultant trajectory and Detected Obstacles

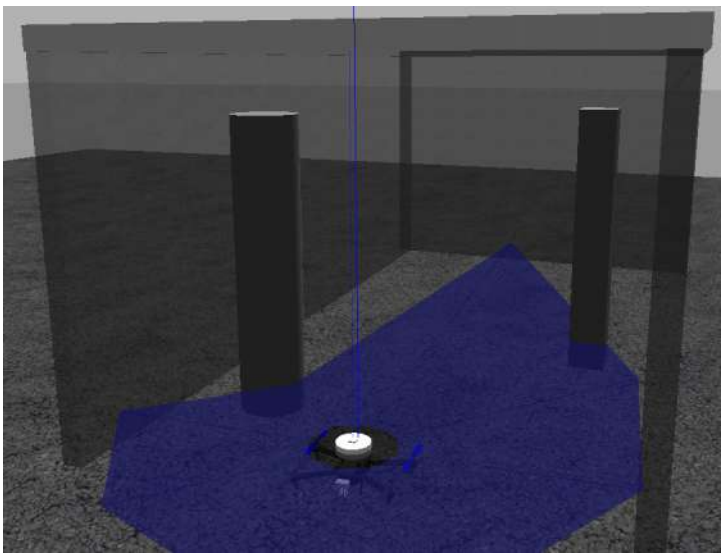
Functionalities to be tested:

- Simple obstacle avoidance
- Mapping of obstacle

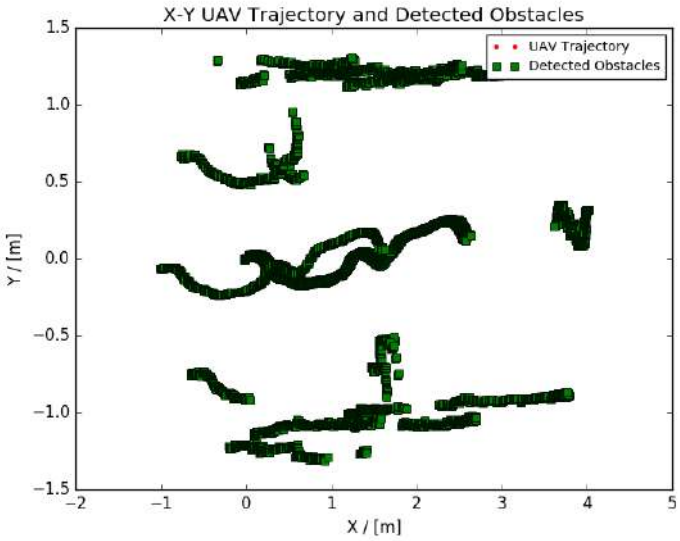
Summary of Results:

- Successful - System was responsive but not aggressive.

F.3 World 3 - 2 Obstacles Far Apart



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 15: Gazebo Test World 3 - Simulation Environment and Resultant trajectory and Detected Obstacles

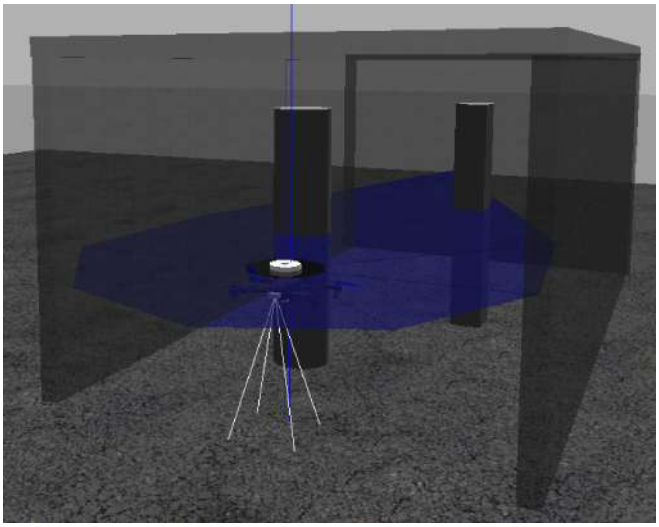
Functionalities to be tested:

- Multiple obstacle avoidance
- Safety distances maintained
- Aggressiveness of multiple maneuvers

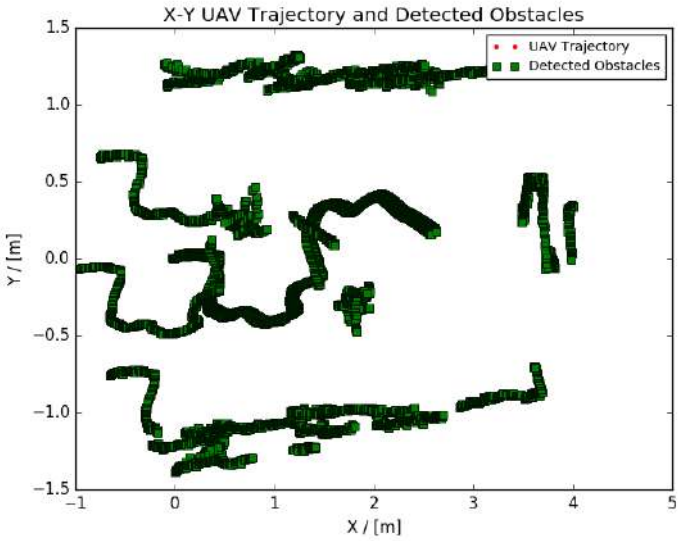
Summary of Results:

- Successful - System was responsive but not aggressive.

F.4 World 4 - 2 Obstacles Close Together



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 16: Gazebo Test World 4 - Simulation Environment and Resultant trajectory and Detected Obstacles

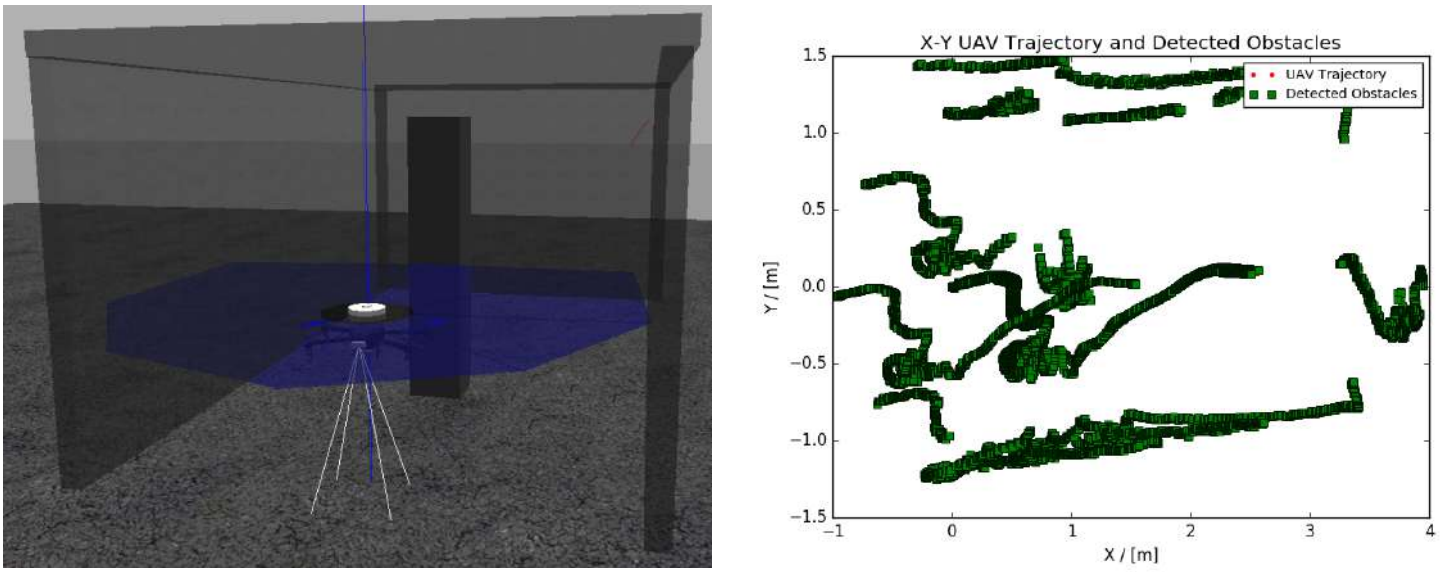
Functionalities to be tested:

- Important test for aggressiveness
- Precise motion
- Mapping of closely spaced obstacles

Summary of Results:

- Successful - System was responsive but not aggressive.

F.5 World 5 - Local Minima Solver With 1 Obstacle



(a) Gazebo Simulation Environment

(b) X-Y UAV Trajectory and Detected Obstacles

Figure 17: Gazebo Test World 5 - Simulation Environment and Resultant trajectory and Detected Obstacles

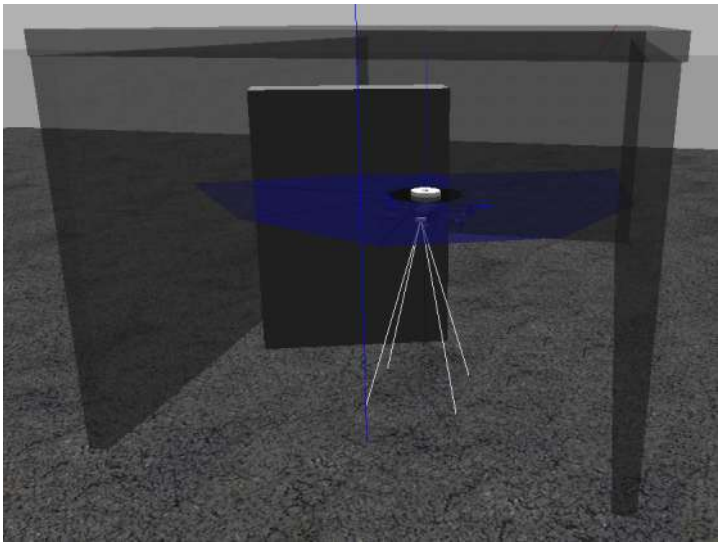
Functionalities to be tested:

- Local Minima solver via Simulated Annealing
- Aggressiveness of response
- Motion across closely spaced obstacles

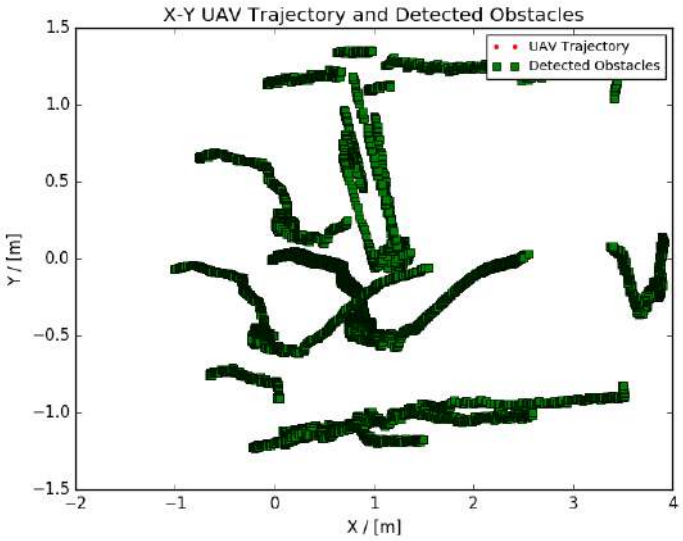
Summary of Results:

- Successful - System was responsive but a little more aggressive than desired.

F.6 World 6 - Single Curved Wall



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 18: Gazebo Test World 6 - Simulation Environment and Resultant trajectory and Detected Obstacles

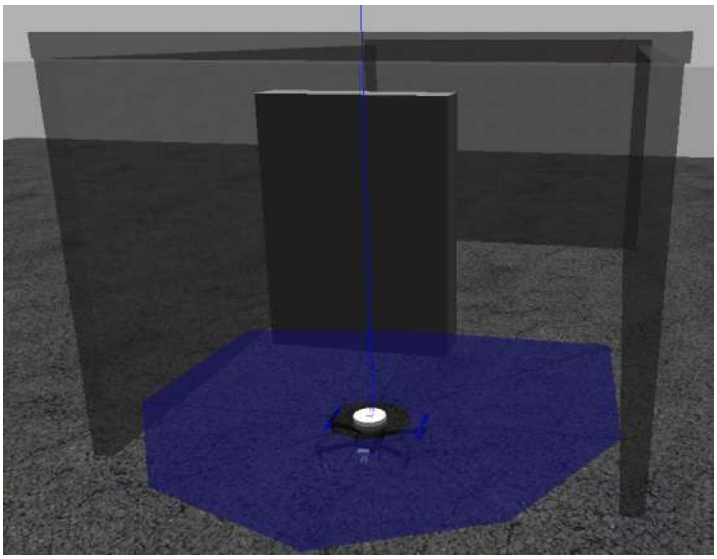
Functionalities to be tested:

- Large obstacles
- Safety distances

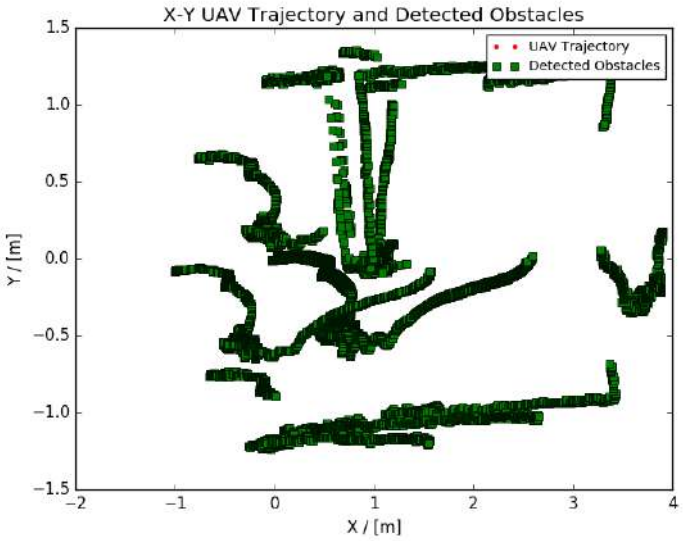
Summary of Results:

- Successful - System was responsive but not aggressive. Small oscillations were noted when trying to go through closely spaced obstacle region.

F.7 World 7 - Local Minima Solver With Large Wall



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 19: Gazebo Test World 7 - Simulation Environment and Resultant trajectory and Detected Obstacles

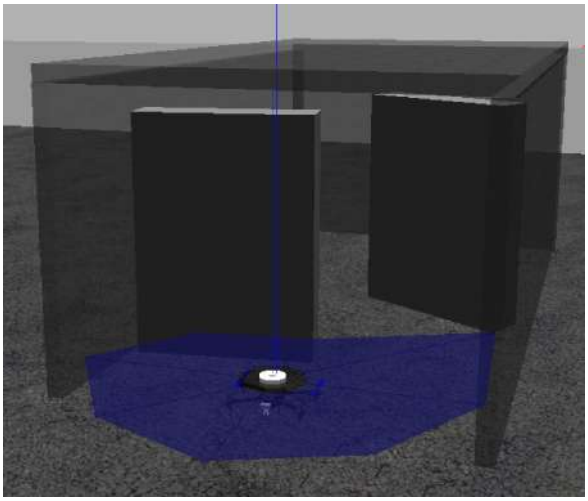
Functionalities to be tested:

- Local Minima solver via Simulated Annealing
- Large obstacles
- Safety distances

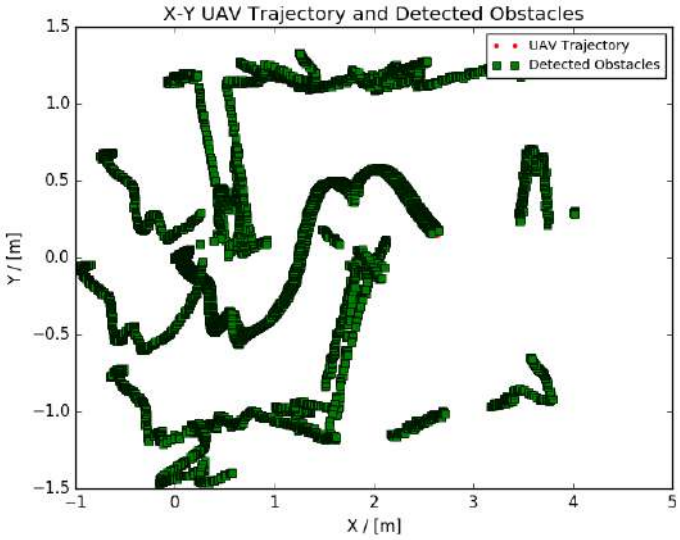
Summary of Results:

- Good - UAV was able to reach goal but was not very robust when passing through such closely-spaced obstacles. Simulated Annealing parameters require further tuning.

F.8 World 8 - 2 Curved Walls



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 20: Gazebo Test World 8 - Simulation Environment and Resultant trajectory and Detected Obstacles

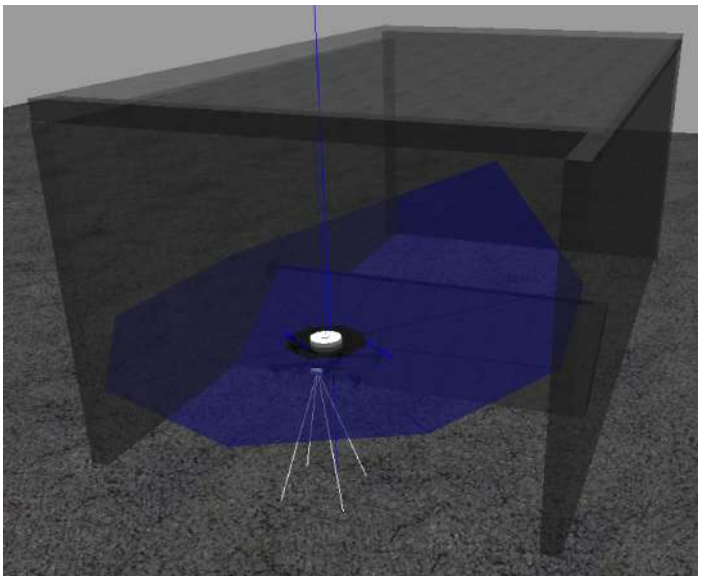
Functionalities to be tested:

- Navigation when goal is not readily visible
- Local minima solver in $y - axis$
- Repulsive memory
- Mapping capabilities

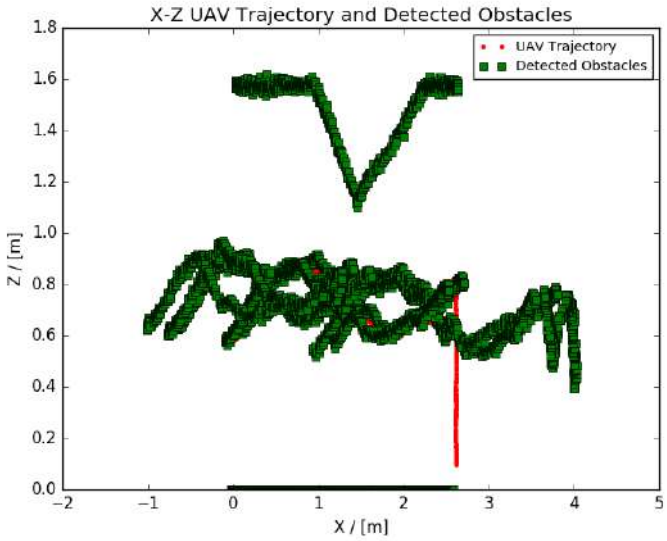
Summary of Results:

- Successful - System was responsive but not aggressive. UAV was able to correct motion after first obstacle.

F.9 World 9 - Double Sided Ramp



(a) Gazebo Simulation Environment



(b) X-Z UAV Trajectory and Detected Obstacles

Figure 21: Gazebo Test World 9 - Simulation Environment and Resultant trajectory and Detected Obstacles

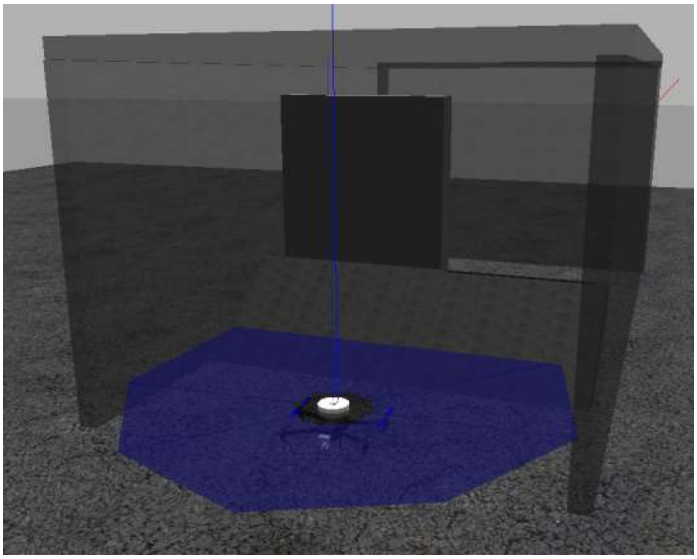
Functionalities to be tested:

- Obstacle detection and avoidance in $z - axis$
- Mapping in $z - axis$

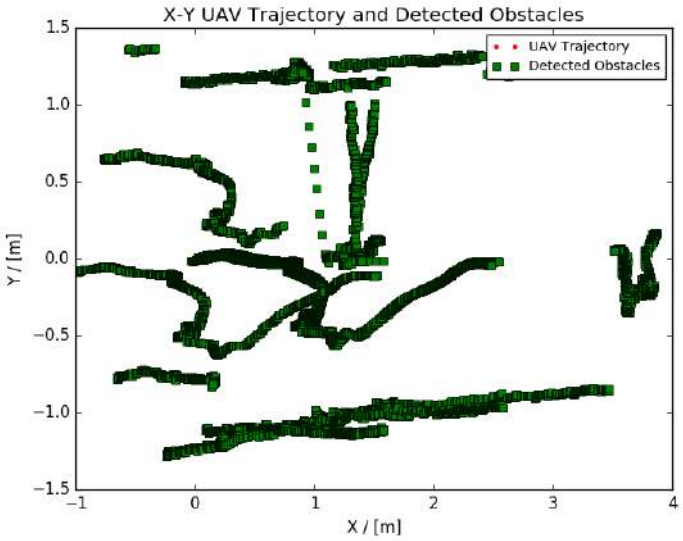
Summary of Results:

- Good - Obstacle avoidance was very good but the mapping in $z - axis$ was upside-down. This is due to the fusion of range data in mavros and should be further investigated.

F.10 World 10 - Double Sided Ramp with Wall



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 22: Gazebo Test World 10 - Simulation Environment and Resultant trajectory and Detected Obstacles

Functionalities to be tested:

- Obstacle avoidance in all 3 directions
- Local Minima solver in all 3 directions
- Aggressiveness of response
- Optical flow drift across ramp

Summary of Results:

- Good - both obstacles were successfully avoided but the motion was severely oscillatory and aggressive, thus the parameters of both Potential Fields and Simulated Annelaing should be revisited.

G Appendix - Transformation Strategy From Pixhawk's World F.O.R to Mine F.O.R

Initially, our expectation was that Pixhawk's world frame of reference would be distorted but would remain fairly stable. As such, our strategy was to align the drone with the Mine ($_M$) frame of reference and record the initial transformation matrix [51] from Body-Fixed ($_B$) to World ($_W$) coordinates (T_{WB}) (14). Assuming both the mine and Pixhawk world F.O.R were fixed, any velocity commands in the mine F.O.R could be converted to Pixhawk's world F.O.R using (15).

$$T_{WB} = \begin{bmatrix} C_\theta C_\psi & S_\phi S_\theta C_\psi - C_\phi S_\psi & C_\phi S_\theta C_\psi + S_\phi S_\psi \\ C_\theta S_\psi & S_\phi S_\theta S_\psi + C_\phi C_\psi & C_\phi S_\theta S_\psi - S_\phi C_\psi \\ -S_\theta & S_\phi C_\theta & C_\phi C_\theta \end{bmatrix} \tag{14}$$

$$V_W = T_{WM} \cdot V_M = T_{WB|t=0} \cdot V_M \tag{15}$$

Unfortunately, the compass heading drifted significantly during flight so this approach was not valid.

H Appendix - Distance Keeping Repulsive Potential

During the course of Week 4 of this project, we suffered some technical issues with the TeraRanger Tower [9] and had to order a replacement part. As a result, we devised backup strategy whose purpose was to prove the feasibility of potential fields as a navigation algorithm. With only one available TeraRanger One placed at 90° to the goal heading angle, we intented to substitute the purely repulsive potentials presented in Section 3 with the distance keeping repulsive potential presented below (16). Though not optimal, this solution would help us navigate towards the goal safely avoiding the left wall (maintaining a distance $d_{y,des}$), as shown in Figure 25. The results of such implementation in the Gazebo environment is presented in Figure 24.

$$\nabla U_{rep,y}(q) = \eta(d_y^2 - d_{y,des}^2)\frac{\mathbf{d_y}}{||\mathbf{d_y}||^2} \tag{16}$$

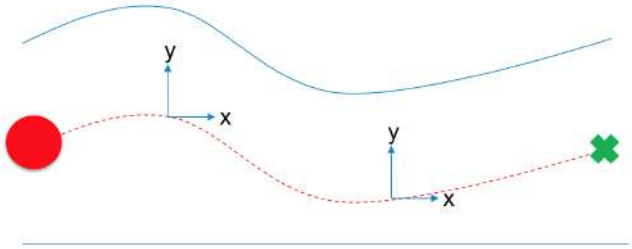
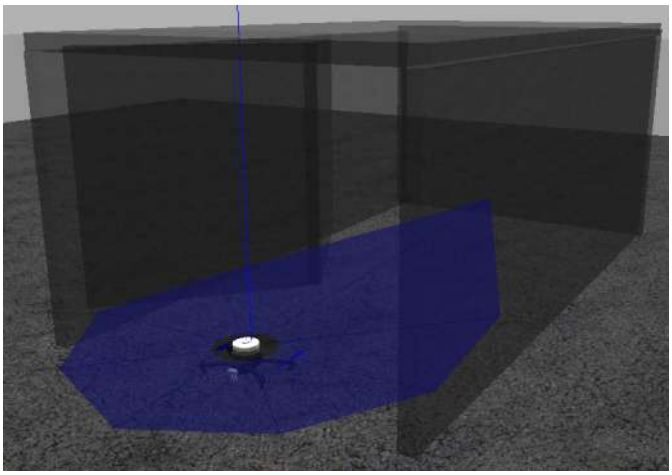
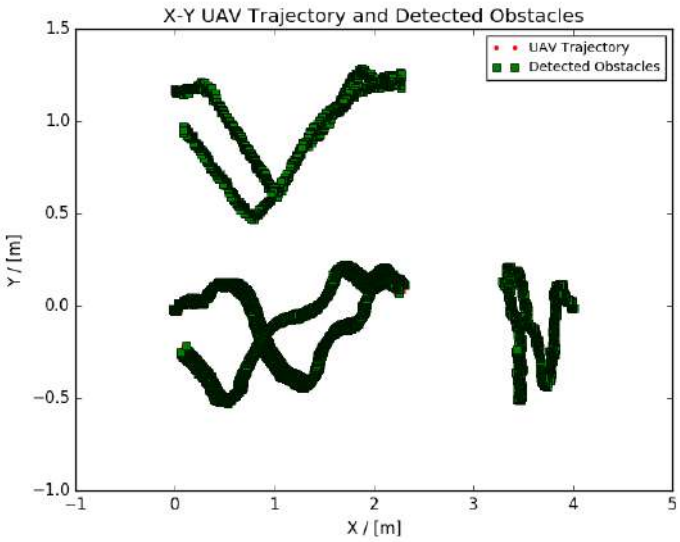


Figure 23: Distance Keeping Repulsive Backup Strategy Representation



(a) Gazebo Simulation Environment



(b) X-Y UAV Trajectory and Detected Obstacles

Figure 24: Results from Implementation of Backup Strategy Using Distance Keeping Potentials in Gazebo.

I Appendix - Tuning of Gradient Descent, Potential Fields and Simulated Annealing Parameters

I.1 Gradient Descent

- α : High values of α would lead to large velocity inputs. Coupled with the unstable dynamics of the drone this could lead to a safety issue. On the other hand, low values of α increase time of flight.

I.2 Potential Fields

- $\frac{\zeta}{\eta}$: If the ratio of the attractive / repulsive gains is high, this could lead to uncertainty-caused collisions as the drone may come close to the obstacle. On the other hand, if the ratio is low, the drone may not reach the goal, it will increase the time of flight, increase the total path length and will get stuck in more local minima.

- $memory\_repulsive$: If the repulsive memory is increased, η should be decreased to compensate for the additional obstacles accounted for. Increasing repulsive memory will lead to a more imprecise repulsive vector.

I.3 Simulated Annealing

- r_local_min : This parameter represents the radius which, if has not been escaped after a specified time will trigger the Local Minima Solver. This should be kept low as to avoid the apparition of fictitious Local Minima.

- Cooling Rate (r) : Low values of r decrease the probability of "uphill" moves but may lead to "premature freezing", which leads to not escaping Local Minima. On the other hand, high values of r could lead to more riskier trajectories being considered and collisions with obstacles may occur due to drone instabilities.

- Initial Temperature (T_0) : Similar to r, high initial temperatures lead to increased likelihood of "uphill" moves and low initial temperatures can lead to the Local Minima not being escaped.

J Appendix - Optimisation Scripts Using Differential Evolution - Optmiser and Bash Scripts

```python
#!/usr/bin/env python
# Optimiser Script: Diferential Evolution is used to tune KP and ETA for Potential Fields
# Author: Pablo Hermoso Moreno

# Import Differtial Evolution Script from SciPy Library
from scipy.optimize import differential_evolution

# Running Bash Scripts ->
from subprocess import *
import subprocess

import csv
import rospy
import math
import sys
import time

def F(x):

    global iteration

    neu_dict = {'POT_FIELDS': []}
    neu_dict['POT_FIELDS'].append(x[0])
    neu_dict['POT_FIELDS'].append(x[1])

    print neu_dict
    print "Iteration Number: ", iteration

    # Save Current Potential Fields Parameters for Current Script:
    with open("POT_FIELDS_VAL_CURRENT.txt", "w") as f:
        # Overwrite Previous Entries
        f.write("%f %f" % (x[0],x[1]))

    # Save History of Selected Potential Field Parameters:
    with open("POT_FIELD_VAL_HISTORY.txt", "a") as f:
        f.write("%f %f \n" % (x[0],x[1]))

    subprocess.call(['./bash_training.sh'])

    # Processing Error - Total Path Length:
    with open("drone.path_length.txt", "r") as f:
        error = float(f.read())

    print "Total Path Length : ", error

    # Save History of Error:
    with open("ERROR_HISTORY.txt", "a") as f:
        f.write("%f \n" % error)

    iteration += 1

    return error


def main():

    global iteration
    iteration = 1

    # Optmisation Scheme for ETA and KP:

    bounds = [(0,5), (0,0.1)]

    # Run Differential Evolution Optimisation:
    res = differential_evolution(F, bounds, maxiter=10, popsize=5,disp=True)

    print("The Outputs from Optimizer Main are: ")
    print("Solution x= ", res.x)
    print("Success bool: ", res.success)
    #print("Status: ", res.status)
    print("Message: ", res.message)
    print("Function minimum found, error = ", res.fun)
    print("Number of evaluations: ", res.nfev)
    print("Number of iterations performed: ", res.nit)



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

```
#!/bin/bash
cd ~/src/Firmware
gnome-terminal -e './gazeboVmWork.sh "make posix_sitl_default gazebo_iris_opt_flow"'
sleep 10s
cd ~/src/Firmware
gnome-terminal -e './rosLaunchDrone.sh'
sleep 6s
gnome-terminal -e 'rosrun tera_2_pkg tera_2_node'
sleep 3s
cd ~/Desktop/GDP/POT_FIELDS/06_06_2018_EA/
gnome-terminal -e './main_06_06_2018.py'
sleep 140s
killall -15 "gazeboVmWork.sh"
killall -15 "gzserver"
killall -15 "make"
killall -15 "ninja"
killall -15 "sitl_run.sh"
killall -15 "px4"
killall -15 "roslaunch"
killall -15 "rosmaster"
killall -15 "rosout"
killall -15 "tera_2_node"
killall -15 "mavros_node"
killall -15 "rosLaunchDrone.sh"
killall -15 "sh"
# Last but not least, kill Navigation Algorithm
pgrep -f main_06_06_2018.py # will give you its pid
pkill -9 -f main_06_06_2018.py # kills the matching pid
exit 0
```

K Appendix - Solving the Goal Non-Reachable With Obstacles Nearby (GNRON) Problem

The GNRON problem arises because the global minimum of the total potential field is not at the goal position when the goal is within the influence distance of the obstacle. This problem is due to the fact that as the robot approaches the goal, the repulsive potential increases as well. This motivates us to incorporate a new repulsive potential function which takes the relative distance between the robot and the goal into consideration [57].

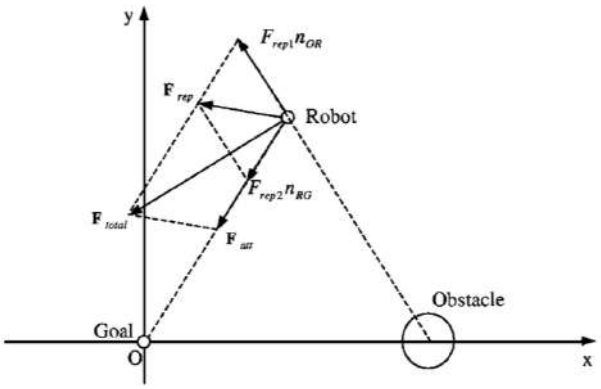


Figure 25: Vector Representation of the New Repulsive Potential Function. [57]

$$U_{rep,i}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{d_{obs,i}(q)} - \frac{1}{Q^*})d_{goal,i}^n, & d_{obs,i} \leq Q^* \\ 0, & d_{obs,i} > Q^* \end{cases} \tag{17}$$

$$\nabla U_{rep}(q) = \nabla U_{rep,1}(q) + \nabla U_{rep,2}(q) \tag{18}$$

$$\nabla U_{rep,1}(q) = \eta(\frac{1}{d_{obs}(q)} - \frac{1}{Q^*})\frac{d_{goal}^n}{d_{obs}^2} \cdot \nabla d_{obs} \tag{19}$$

$$\nabla U_{rep,2}(q) = \frac{n}{2}\eta(\frac{1}{d_{obs}(q)} - \frac{1}{Q^*})^2 d_{goal}^{n-1} \cdot -\nabla d_{goal} \tag{20}$$

The result of the implementation of such repulsive potentials was surprisingly good. Figures 27 and 28 show the resultant trajectory of the UAV in a custom built Gazebo world where the goal is deliberately set very close to a set of obstacles (see Figure 26). While the UAV with the classic FIRAS [25] repulsive function is unable to reach the goal, the UAV with this modified repulsive potential is successful at doing so.

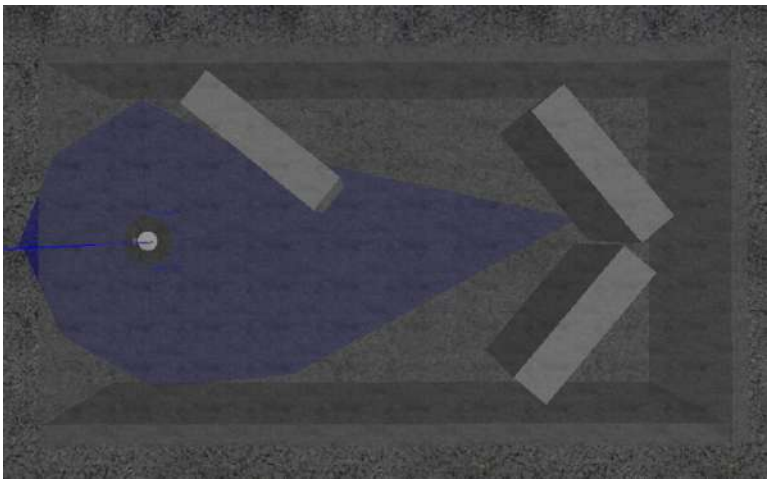
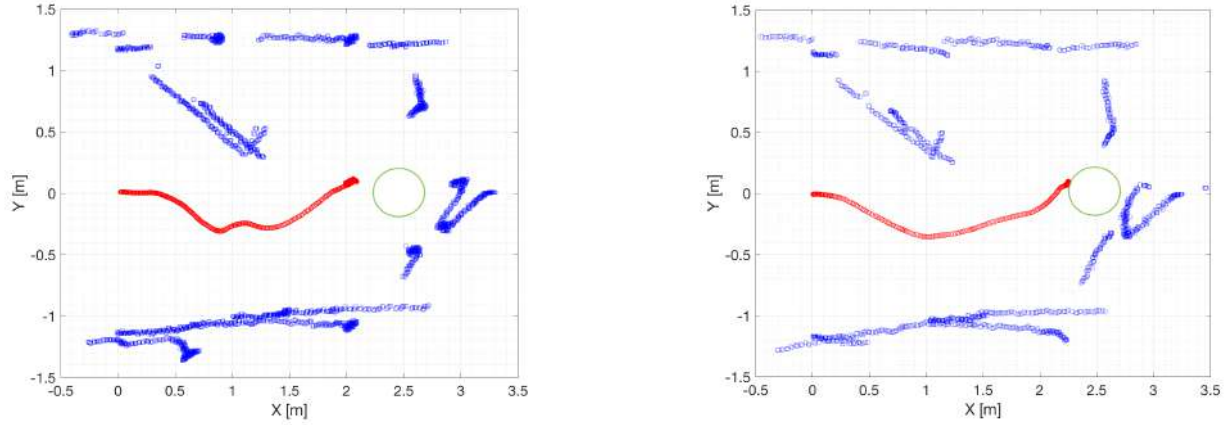


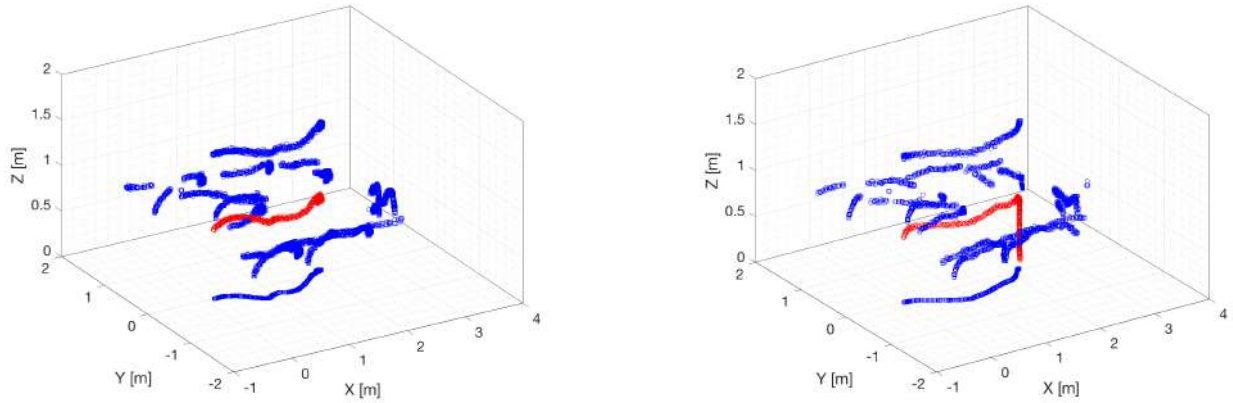
Figure 26: Custom Built Gazebo Environment With Goal Close to Obstacle to Test Effectiveness of Modified Potential in Solving the GNRON Problem



(a) X-Y UAV Trajectory and Detected Obstacles with FIRAS Potential

(b) X-Y UAV Trajectory and Detected Obstacles with Modified Potential

Figure 27: X-Y Resultant Trajectory and Obstacles Mapped using FIRAS against Modified Repulsive Potential. Note how only the UAV with the Modified Potential reaches the goal bounded by the green circle.



(a) 3D UAV Trajectory and Detected Obstacles with FIRAS Potential

(b) 3D UAV Trajectory and Detected Obstacles with Modified Potential

Figure 28: 3D Resultant Trajectory and Obstacles Mapped using FIRAS against Modified Repulsive Potential. Note how only the UAV with the Modified Potential reaches the goal bounded by the green circle and proceeds to land.

```python
if (d <= q_star):
    # CLASSIC
    u_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[0] #* d_goal**2
    v_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[1] #* d_goal**2
    w_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[2] #* d_goal**2
    # GNRON Solver
    #u_o += ETA * (((1 / q_star) - (1 / d))**2) * d_goal * d_hat_goal[0]
    #v_o += ETA * (((1 / q_star) - (1 / d))**2) * d_goal * d_hat_goal[1]
    #w_o += ETA * (((1 / q_star) - (1 / d))**2) * d_goal * d_hat_goal[2]
```

(a) Python Implementation of Classic FIRAS Repulsive Potential

```python
if (d <= q_star):
    # CLASSIC
    u_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[0] * d_goal**2
    v_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[1] * d_goal**2
    w_o += ETA * ((1 / q_star) - (1 / d)) * ((1 / d) ** 2) * d_hat[2] * d_goal**2
    # GNRON Solver
    u_o += ETA * (((1 / q_star) - (1 / d))**2) * d_goal * d_hat_goal[0]
    v_o += ETA * (((1 / q_star) - (1 / d))**2) * d_goal * d_hat_goal[1]
    w_o += ETA * (((1 / q_star) - (1 / d))**2) * d_goal * d_hat_goal[2]
```

(b) Python Implementation of Modified Repulsive Potential

Figure 29: Python Implementation of both FIRAS and Modified Repulsive Potentials