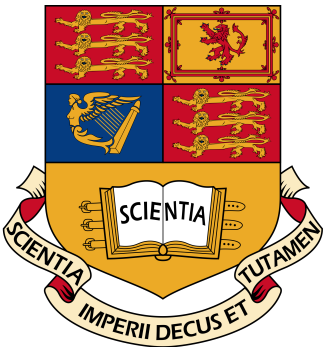

Imperial College London

C362

Third year Software Engineering Project Report

Autonomous Air Traffic Control

Authors:
Paul Balaji
David Cattle
Andrea Janoscikova
Galina Peycheva
Jan Matas
Samuel Wood

Supervisor:
Dr. William Knottenbelt.

January 8, 2017

Contents

1 Executive Summary **3**
1.1 What is AATC? . . . 3
1.2 Why is AATC needed? . . . 3
1.3 Targeted audience . . . 3

2 Introduction **4**
2.1 Background and Motivation . . . 4
2.2 The problem AATC is solving . . . 4
2.3 Achievements . . . 5

3 Project management **6**
3.1 Project Plan . . . 6
3.2 Software Development Strategy . . . 6
3.2.1 Team Organisation . . . 7
3.2.2 Development Practices . . . 7
3.3 Means of Communication . . . 9
3.4 Tools . . . 10

4 Design Architecture Overview & Implementation Details **11**
4.1 The AATC layer model . . . 11
4.2 The AATC test bench . . . 11
4.2.1 Motivations . . . 11
4.2.2 Test Harness Design/Architecture . . . 12
4.2.3 Test Case Description Format . . . 12
4.2.4 Simulated Drone Model . . . 13
4.2.5 Test Harness Output . . . 13
4.3 Global Layer . . . 14
4.3.1 Purpose . . . 14
4.3.2 Cartesian coordinates from GPS coordinates . . . 15
4.3.3 Representing Space: The Global Layer Grid . . . 15
4.3.4 Grid Algorithms . . . 17
4.3.5 Algorithm: Placing No Fly Zones . . . 18
4.3.6 Algorithm: Minimum distance to a NFZ . . . 18
4.3.7 Algorithm: A* . . . 19
4.3.8 Theta* . . . 20
4.3.9 An improvement to Theta*: Lazy Theta* . . . 21
4.3.10 Path-finding Algorithms Benchmark Study . . . 21
4.3.11 Conclusion and possible Extensions . . . 23
4.4 Reactive layer . . . 23
4.4.1 Algorithm Overview . . . 24
4.4.2 Original mathematics and how we changed it . . . 26
4.4.3 Implementation . . . 27
4.4.4 Genetic Algorithm to improve constants . . . 28
4.5 Drone service . . . 29
4.6 Visualiser . . . 30
4.6.1 Usage . . . 31
4.6.2 Implementation . . . 31
4.6.3 Reproducibility . . . 31

4.6.4 Live Mode . . . 32
4.6.5 Live Mode Usage . . . 32
4.6.6 Live Mode Implementation . . . 33

5 Conclusion and future extensions **34**
5.1 Deliverables . . . 34
5.2 Feedback and Results . . . 34
5.3 Future work . . . 34
5.3.1 Global Scale Navigation Design . . . 35
5.3.2 Scaling and Reliability . . . 36

A Visualiser Live Mode WebSocket Communication **37**

B Bibliography **38**

1 Executive Summary

1.1 What is AATC?

AATC is a robust unmanned aviation management system dealing directly with drones and their integration in the skies. This service receives real-time information from drones in the air and computes optimal paths for them to reach their destinations undisturbed. The system now works for a local area of one city and a few dozen drones, but we propose simple methods for scaling the system up to be deployable on a country-wide scale.

1.2 Why is AATC needed?

The drone industry is expanding rapidly, and drone technology proves to be useful in an increasing number of sectors. As a result, the skies are progressively becoming more and more congested. The problem of collision with other aerial bodies is turning into a real issue, which needs to be resolved before proceeding with populating the airspace with more drones. AATC provides a fair solution to the presented problem using efficient algorithms. We believe it will be possible to extend our system or a similar system to become an industry standard solution for drone navigation in real time and prevent what could be some severe accidents.

1.3 Targeted audience

The project was proposed by *Microsoft* and *Altitude Angels*, a company who works towards integrating drones into the airspace. They will use the outcome of our project as a proof of concept code for an Unmanned Traffic Management system which could be adapted and integrated with their existing services. In the long term, the project should be made available to anyone who needs autonomous drones for any purposes.

2 Introduction

2.1 Background and Motivation

Nowadays drone technology is becoming more and more popular in different areas of interest like media, for filming concerts or sports events, exploration, for accessing difficult or dangerous places, law enforcement, deliveries of small items, wildlife observation, etc.

Drone technology is quickly advancing to remove the necessity on human input - it is tending towards becoming fully autonomous. The fact that drones do not carry a human passenger and can operate in areas unreachable or too dangerous for humans make them an attractive prospect to businesses across all sectors. Subsequently, there are many companies showing interest in integrating drones into their work in the next few years.

As a result of these advances, the airspace will become increasingly congested with drones, leading to higher risk of collisions. There have already been several occurrences of drones crashing into passenger airplanes, including one near Heathrow Airport as a matter of fact. To ensure the safe flight of drones, an air traffic management system is needed to synchronize and navigate them in the air to their destination.

2.2 The problem AATC is solving

When a drone sets off in the air to complete some mission, be it reach a goal (deliver an item) or follow commands of some sort to explore an area, it does not hold information about any other flying objects present in the air which could interrupt its flight. The obstacles which the drone should ideally avoid can be summed up into three main categories:

- No Fly Zones (which we will abbreviate as NFZ) - these are static obstacles or zones which the drones should not fly over. Examples are skyscrapers, Buckingham Palace, airports, military premises, etc.
- Manned aviation - any airborne vehicle with humans on board is considered to be manned aviation. These are moving objects which in our representation of the world we will take as flying in a straight line from their start point to their end point.
- Other drones – these are other machines occupying the airspace. We assume we can collect information from them (apart from occasional communication drops) but we might not be able to control them. They do not necessarily move in a straight line.

Our project focuses on designing an efficient, fair and robust approach to navigating all drones from their starting point to their destination. NFZs are predefined in our system as polygons that drones are not allowed to cross at any altitude. Manned aviation is also predefined in our system with information on start and end points of the planes' flights. AATC has no control over their course, so any potential collision involving a manned aviation should be avoided by instructing the drone to avert the danger.

Other drones are moving objects and AATC can send signals to all of them for path correction measures when necessary. The system should also be able to handle 'rogue' drones which do not follow the paths suggested by our system or drones that stop sending their telemetry data for a short time.

To ensure that the system we are building is behaving as we desire, we also needed to create a test harness which we would use to test, improve and train our system using data simulations.

2.3 Achievements

AATC uses a client-server model, where each drone currently active in the airspace is a client. They register with the server at the beginning of their flight and follow path recommendations from the server. The server has information about no-fly zones, manned aviation, and drones. Every second , each drone sends an update to the server, providing its telemetry information (position and heading) and goal waypoints (the places it wants to visit). Using this information, the server computes the best path to avoid all potential collisions and sends a suggestion back to the drone.

We had to do a lot of research and explore different path planning and path finding algorithms to complete the task. Trying to choose the most suitable one for our purpose, we decided to divide the problem into a two-layer control model and deal with static and non-static obstacles in two stages.

Firstly, in the global layer, we plan a route for the drone avoiding no-fly zones. Then, in the reactive layer, we used a variation of artificial potential fields to correct the velocities and directions of all drones. This approach in the reactive layer depends on a lot of constants which we adjusted using a genetic algorithm.

Testing our system on real drones would be quite expensive and impractical, so we built a test harness which uses data simulations to model drones and ask for path recommendations from our server. We created a browser based visualiser that displays the situation we simulate and makes it easy to see the results from our work.

3 Project management

3.1 Project Plan

At the beginning of the term, we met with our supervisor *Dr.* William Knottenbelt and with our project mentors *Mr.* Chris Foster and *Mr.* Lawrence Gripper from Altitude Angels to specify project requirements and set up the rhythm of the project. We agreed on four checkpoints with goals as described in Table 1 below.

Check-point	Checkpoint Goals
1	- Create a very simple simulated environment with a single drone. - Develop a test harness to simulate the presence and behavior of the drone. - Implement a basic visualization of the environment.
2	- Extend the core environment with all the entities the drones need to look out for, namely, ground hazards, no fly zones, manned aviation and other drones. - Build the collision detection system, that will flag each collided drone as dead. - Implement metrics such as battery consumption.
3	- Develop algorithms for the recommended actions such as the creation of interface with the drones (according to customer specifications). - Design algorithms to choose the best action based on battery life, size, mission and type of the drones involved in potential collision.
4	- Add a machine learning element to the system. - Enable gradual improvement of the optimality of actions or working on minor convenient features for the customer.

Table 1: Project plan created in the first meeting.

We managed to complete the first three checkpoints on time successfully. Unfortunately due to the complexity of the online machine learning element of the project and approaching exams we changed the goal of the fourth iteration slightly. In the final iteration, we optimized the collision avoidance algorithms, implemented an offline machine learning algorithm that can improve performance by executing multiple simulations, and implemented live visualisation with a controllable drone that has been discussed as an appealing extension of the project.

3.2 Software Development Strategy

After some research into the different *Agile* methodologies that were proposed, we decided to follow the *eXtreme Programming* (XP) software development methodology. We outline below a few reasons as to why we choose it.

- As Kent Beck outlines in *eXtreme Programming explained*[x], XP is an efficient strategy for small to medium sized teams facing vague or rapidly changing requirements. This was indeed our case: as a team of six, we initially weren't certain about our future product's requirements for our client, and wished to adapt rapidly on any demand change.

Further, the total duration of our project was fairly short and hence features needed to be added quickly.

- Some of us, over the summer, had the opportunity to witness the success of this strategy in the industry and simply considered that to be a good justification for testing the methodology ourselves.

3.2.1 Team Organisation

Following the XP methodology implied having to work iteratively. We decided to work in short one-to-two week cycles. Since daily meetings weren't possible given our different timetables, we met 2 to 3 times per week to discuss the progress of each team member, allocate new tasks and address issues that appeared since the last meeting. Most of the time we worked together in the labs. Thanks to this approach, everyone had an idea about the functionality of each part of the overall structure and could contribute where it was necessary at each moment in time.

In the beginning, each member worked on a particular project component depending on his (or her) preferences and previous experience.

However, after the first few weeks everyone started to contribute to various parts of the project structure, and the tasks were assigned depending on their length and difficulty to make sure that the work is split equally.

Sam Wood was the team leader. He organized meetings and communicated with our supervisor, another group who worked on the same project and with Altitude Angels and Microsoft. He mostly worked on the visualisation of test cases and algorithms implemented in the reactive layer.

Jan Matas was the brain of the group. He did most of the research work related to algorithms for collision avoidance. He also solved multiple compatibility matters relating to the libraries and tools we used and significantly contributed to global and reactive layer.

David Cattle was the shortest path master. He mainly worked on the grid algorithms in the global layer. Moreover, he did most of the work on the test engine component and also contributed to the collision detection algorithms.

Paul Balaji was 'Dev Ops'. He set up and took care of most of the version control and continuous integration tools, and managed our early migration from GitHub to VSTS. He also worked on the controllable drone by its user and most of the unit testing.

Galia Peycheva was the grid expert. She worked on the representation of the environment and the positioning of no-fly zones within it. She also implemented most of the API tests and managed task assignments.

Andrea Janoscikova was the visualiser designer. She implemented multiple features for visualization of the test cases. She worked on the conversion of GPS locations into 3D models and also contributed to the global layer and API tests.

3.2.2 Development Practices

Following *Extreme Programming* entailed being committed to the practices preached by the method. We did so as they lead to higher quality software, and we outline a few of the said practices here:

1. *Pair Programming*: This process enabled us to learn from one another, and allowed to have two pairs of eyes on the same piece of code at one given time, which greatly reduced bug introduction.
2. *Continuous Integration*: As we knew the importance of being able to add features smoothly.
3. *Simple Design*: 'The greatest ideas are the simplest.' So are codes. By endeavoring to continuously *refactor* and simplify our code, we fought *technical debt*.
4. *Unit Testing*: Ensured a behaviour driven approach and acted as an executable specification.
5. *Spike*: Helped us estimate the required work time for features in advance.

Furthermore, at the start of the project, we decided to use a Trello board to adopt agile methods and techniques for project management effectively. We decided to use the green, yellow and black boxes to evaluate easy, medium and hard difficulty levels for each task. We also used a red toolbar for tasks with the high importance that had to be done urgently. The Trello Board was structured into multiple columns, and is explained in the table below. This convention facilitated project management overall.

Column name	Column content
Todo	- Tasks that haven't been assigned yet or haven't been started. - Tasks requiring something to be fixed or tested.
Rejected	- Features or components that weren't necessary anymore (e.g. because of the change of the structure or an algorithm). - Tasks that didn't pass peer code reviews and had to be discussed.
InProgress	- Tasks that were in the development stage with already assigned difficulty level, importance, and the pair working on them.
ForReview	- Tasks that were ready for deployment but were waiting to be reviewed by a person assigned that particular task.
Iteration n Done	- All the tasks completed in the particular iteration. - Important notes, graphs and structures created during the meetings that were relevant for the particular iteration.
Reports	- Reports written in all the stages of the project. - Links to important sources and papers that were found during research.

Table 2: Structure of the Trello board.

The use of the Trello board wasn't just limited to task control. It also sped up the process of decision making in cases where meeting with all team members wasn't possible. Thanks to features such as checklists or comments, we could discuss particular problems in detail and quickly come back to them to go through any outstanding issues.

A screenshot of our final Trello board can be seen in figure 1 below.

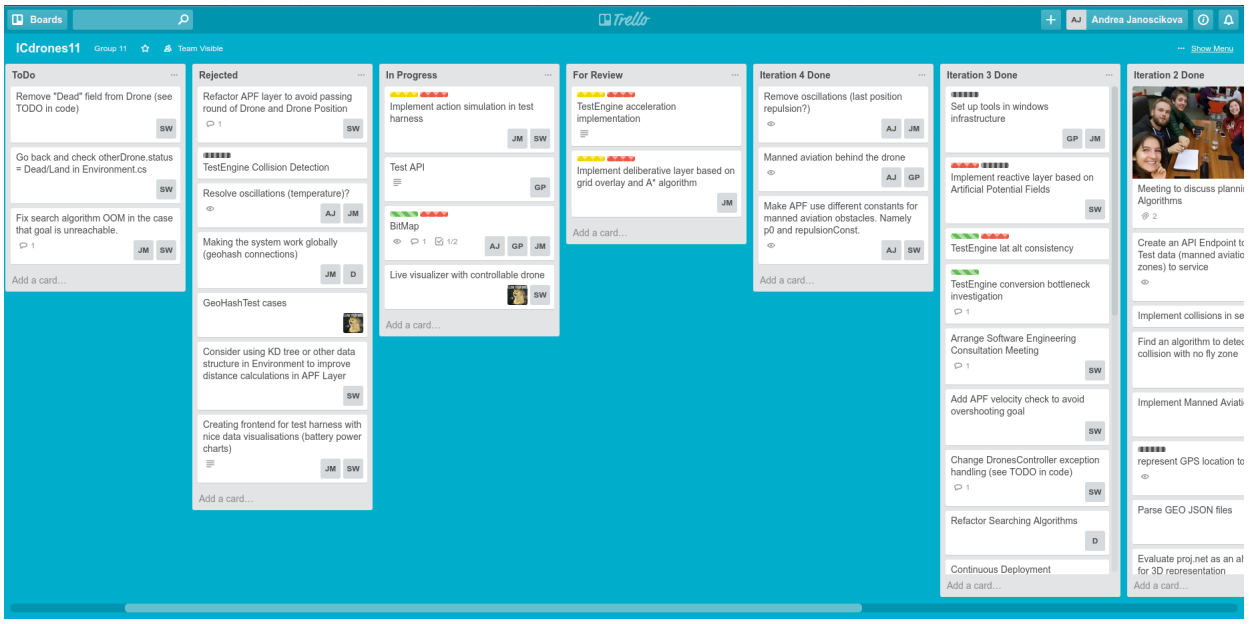

Figure 1: Trello Board.

3.3 Means of Communication

Since the nature of this project is slightly varied from a common group project, keeping strong lines of communication between numerous stakeholders was crucial. As a group project, every member of the team needed to know what their job was, how it links to the overall picture, as well as who they can tap the shoulder of should they need assistance with their work.

We also needed to make sure our supervisor was aware of and verified our work at periodic intervals. In addition to this, the team as a whole needed to make sure our goals aligned with the requirements of Altitude Angels. The extra complication to our communication being that our group had to ensure that the testing harness API were the same across our group and the other group who were undertaking the same project.

Within the team, we made use of Facebook Messenger at first, moving to Slack once we wanted to make use of its 'channels' feature. Google Hangouts were also common when we wanted to hold remote team meetings. Obviously, we would also meet in person several times a week, almost every day in fact, by setting a time to meet and pair program.

To organise meetings with our supervisor, our team leader Sam first sent an email to arrange a time. We then met in *Dr.* Knottenbelt's office as a group and talked through the overall progress we would have made since the previous meeting. As it so happened, meetings coincided with checkpoints.

To communicate with Altitude Angels, we initially met on campus and then exchanged emails and Skype details to arrange follow up calls. Later, we were given a Slack channel where we could ask any questions we wanted and resources could be shared with the other AATC group. This channel was a good way to arrange meeting times with both the other group and with Altitude Angels, to make sure that we could all discuss our progress and expectations, and continuously sync our test harness API.

3.4 Tools

As mentioned prior, Trello and Slack were vital tools used to manage tasks, spread the workload, and ensure clear communication within the team. Due to the diversity of personal machines in our group, a variety of development tools were used for local development. When working on the server code, Windows users were required to use Visual Studio. Mac users used Xamarin, and Ubuntu users made use of Mono Develop. However, numerous problems were encountered with Xamarin and Visual studio, which was decreasing our productivity. It is very likely that vast majority of those issues were caused by our lack of experience with those tools. For test bench development, we used PyCharm as our IDE.

Later during the project, we switched our C# development to Rider by JetBrains. This IDE was almost immediately adopted by the whole team and we enjoyed using it. Despite being in Early access, the IDE provided amazing automatic code completions, great Git integration and amazing stability (we have not encountered a single bug). It was also very convenient to use Rider because it has similar interface to Pycharm so switching between server and test code was seamless.

Regarding version control and continuous integration, we initially made use of GitHub and Travis CI. Due to prior experience with these tools, we found these to be good choices to begin project development.

However, Altitude Angels required us to use VSTS (Visual Studio Team Services) with a repository created for us to commit to. Being new to VSTS, it was difficult to configure tests to be run on each commit. Nonetheless, after enough time reading the documentation, we managed to set up continuous integration properly. As an aside, our unit tests made use of the NUnit open-source framework.

For deployment, we made use of Azure credit. The benefit of Azure over another product, such as Amazon Web Services was that both VSTS and Azure are Microsoft products.
When it came setting up continuous deployment, it worked out that this was just as simple as setting up an Azure service and linking these service details into VSTS.

4 Design Architecture Overview & Implementation Details

4.1 The AATC layer model

Our task was to design a system that can route a small number of drones (say 10) in a small area, but that should be able to scale, with more computing power, to wider areas (up to the scale of countries and continents) and to potentially track an unlimited number of drones. Keeping this restriction in mind, we started an extensive research of available literature. It became obvious that we need to separate concerns to fulfill the scalability criteria, so we decided to split the problem into multiple control layers:

- The Global layer. This layer should hold the static representation of the area and should be able to pre-compute an optimal static route ignoring all moving obstacles (the static route is a list of GPS way-points a drone can follow).
- Reactive layer. This layer should be able to compute the actual goal velocity the drone should travel at It should consider all obstacles that could possibly affect the drone in the near future and adjust the speed and direction of the drone accordingly. The computation on this layer needs to be simple enough to work in real time (*i.e* sub 1-second type latency), so that the drone can immediately react to any new situation.

The two layers in the model are designed to complement each other.

4.2 The AATC test bench

4.2.1 Motivations

The main motivations for building the test harness were to test the *correctness* and *safety* of the AATC system. Moreover, with an API which accepts positions from drones and gives back flight suggestions, we can simulate drone flights in a virtual environment. This is firstly so that we can test the system inexpensively and safely without the use of real drones and secondly to form the backbone of an automated test suite.

Further, to check the *correctness* of AATC, the test harness verifies that the instructions the service provides allow drones to be routed from start point to destination while minimising the amount of battery usage and time spent flying, and without entering a NFZ along the way.
Similarly, to check the *safety* of the system, the test harness verifies that during the simulated flight, none of the drones:

- Collide with another drone.
- Collide with manned aviation.
- Get stuck before reaching their destination.

4.2.2 Test Harness Design/Architecture

The test harness, developed in Python, takes any number of test case 'environment' descriptions and simulates the movement of the virtual drones described in each environment description. The simulated drones communicate their positions to the server at discrete timed intervals and respond to the server's instructions, changing their position and velocity accordingly.

Once all simulations are completed, the test harness logs data files for each 'environment' simulation log, that is then interpreted by a separate visualizer component (*c.f* subsection 4.5) to be replayed. Please see Figure 2 for a graphical representation of the architecture.

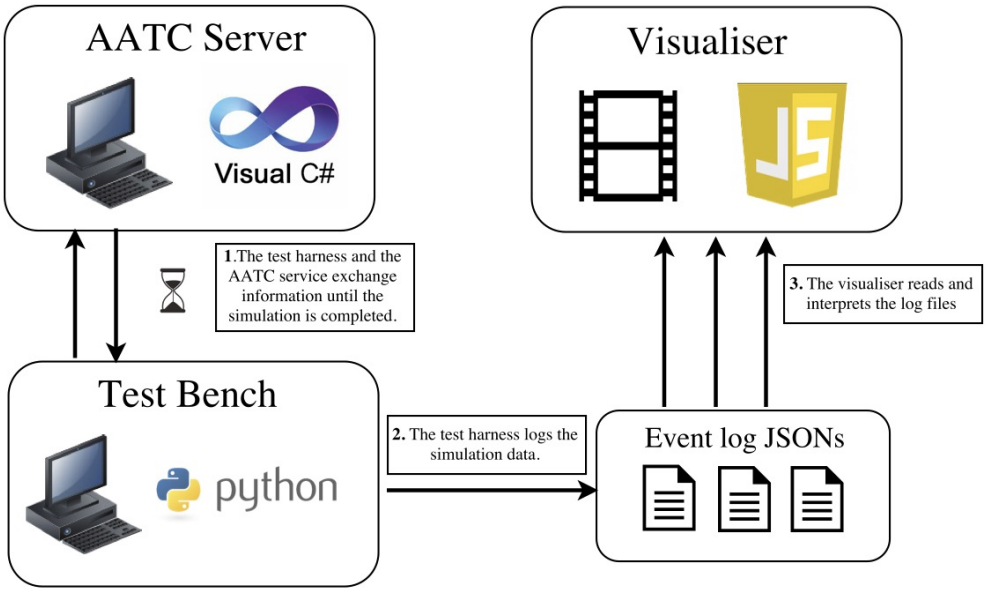

Figure 2: The Testing System Architecture.

The fact that discrete time is being used in the test harness enables us to run the simulation at the fastest speed possible. This allows us to run extensive simulations that would potentially take a very long time if executed in real-time.

4.2.3 Test Case Description Format

The test case environments are read by the test harness and simulated. These environments are described in a JSON file and are made up of three main components:

1. *Drone Data*: describing each of all the drones in an environment.
   For each drone, the data contains the drone name, its starting position and a list of its desired goal waypoints, to be visited in order. Moreover, each drone also has an associated start time, which is the time at which it will send its first position update to the server.
2. *No fly zones*: A set of static GeoJSON descriptions of all NFZ's in the environment.
3. *Manned aviation's*: A set of manned aviation descriptions, each with a start time and a set of waypoints which they travel through. Each manned aviation also has a speed at which it travels between the waypoints.

```
{
  "name" : "Example Test Case",
  "timeLimit" : 100,
  "drones" :
  [
    {
      "uid" : "d1",
      "startTime" : 1,
      "waypoints" :
      [
        { "lng" : 1.0, "lat" : 2.0, "alt" : 50.0 },
        { "lng" : 2.0, "lat" : 4.0, "alt" : 75.0 }
      ],
      "actions" :
      [
        {
          "action" : "move",
          "target" : { "lng" : 1.0, "lat" : 2.0, "alt" : 50.0 },
          "startTime" : 2,
          "ttl" : 4
        }
      ]
    }
  ],
  "noFlyZones" :
  [
    { "<Additional geoJSON data>" : "<Additional geoJSON data>" }
  ],
  "mannedAviation" :
  [
    {
      "startTime" : 1,
      "waypoints" : [
        { "lng" : 1.0, "lat" : 2.0, "alt" : 50.0 },
        { "lng" : 2.0, "lat" : 4.0, "alt" : 75.0 }
      ],
      "speed" : 10
    }
  ]
}
```

Figure 3: An example test case with the components described above.

Using this information represented in a JSON file, the test harness simulates the environment for a specified amount of time. At each time step, the positions of all drones and manned aviations are recorded to an events log.

4.2.4 Simulated Drone Model

Here, we model drones to be a simple unmanned aircraft of spherical shape with radius 1m. The drones can travel in the X, Y and Z directions and can change their velocities. A drone can move with a maximum speed of 5 $m.s^{-1}$ in any direction. At any given time, the total *acceleration* for a drone must not exceed 2 $m.s^{-2}$. See Figure 4 on the following page.

The drones are assumed to have the capabilities to send position updates to the server via some fast, low latency protocol and also react to recommendations sent back by the server.

4.2.5 Test Harness Output

For a given test case, the test harness outputs the following:

- A report summarizing any drones which crashed, if any, and the fuel consumption of each drone.
- An *events log* which is a record of everything that happened during the simulation, at each time-stamp. This most importantly includes any drone position updates and drone deaths (either from collisions or entering no fly zones).

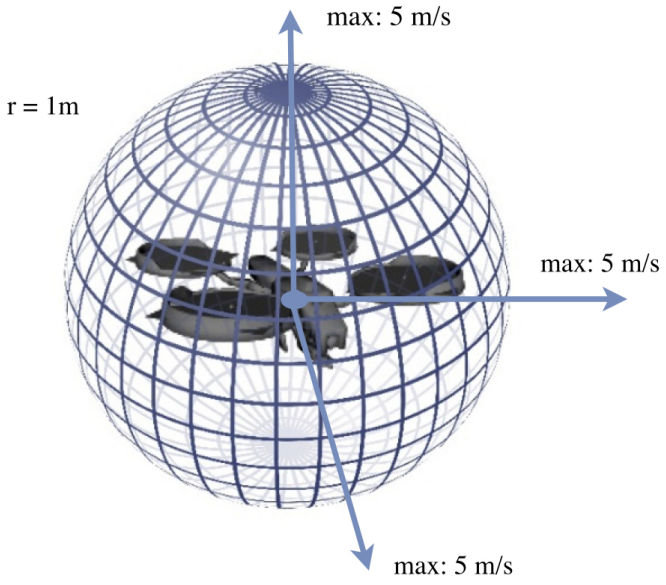

Figure 4: A drone model.

If all drones reach their goals, the fuel consumption metrics give a mathematical basis for comparison between a simulation run against different versions of the AATC. However, in testing and demonstration of the AATC it is often the case that we require more than just knowing that the drones reached their goals safely with a particular fuel consumption limit. While a drone may reach its goal and not exceed a fuel limit, this information alone will rarely indicate any unwanted anomalies in the drone flight paths generated by the AATC.
Examples of flight path anomalies we encountered during development and testing are:

- Drones that avoid nearby drones by taking paths which go back on themselves and unnecessarily loop - a byproduct of misconfigured APF constants (*c.f* subsection 4.4).
- Drones that reach their goal by flying a nonsensical route around a no fly zone - a byproduct of the global layer providing a non-optimal route to a waypoint (*cf* subsection 4.3).

Therefore, for the correct analysis of results we wish to view the simulated drones and their flight paths. For this, we developed a visualiser tool which interactively displays a graphical simulation of the events log generated by the test harness.

4.3 Global Layer

4.3.1 Purpose

The global layer has two main objectives:

1. Represent an area of land.
2. Allow the application of algorithms to find the information required by the other layers for the drones, such as *the shortest path from one point to another*, and the *shortest distance from a point to a no fly zone*.

However, one may ask themselves how we represented an area of land, in the most *compact* and *useful* way? In addition, our clients ultimately want our drone service to be as *responsive* and

scalable as possible for their drones. It is also important that the produced drone pathways avoid *No Fly Zones*, and that they be *smooth and realistic*. How did we attain that objective through our implementation? This will be the matter of discussion of this section.

4.3.2 Cartesian coordinates from GPS coordinates

The first challenge we were faced with for this part was to determine a suitable *coordinate system* to work with. Given that our client's drones send their locations in GPS coordinates, we had to find a way of converting those coordinates into another format, as GPS coordinates are, unfortunately, not suitable for mathematical operations. This is because there is no simple correlation between the GPS coordinates of two points and the distance between them. Thus, we decided that we would convert the GPS coordinates to coordinates in the *cartesian coordinate system*. At first, we wanted such a system to be centered in the middle of the earth and then simply convert GPS coordinates the same way we would convert polar coordinates.

However, the problem here was that no axis was actually representing altitude, so the mathematics in further operations would have been unnecessarily complex.

Therefore, after a bit of research, we decided to use the **U**niversal **T**ransverse **M**ercator coordinate system. This allowed to simply translate GPS coordinates to 2-dimensional cartesian coordinates: that is, by quantifying the number of meters East and North a point is from a specified reference point in each of the 60 zones spanning the whole globe. We found libraries implementing the translation formulas for C# (Proj.net)[2] and for Python (utm 0.4.1)[3]. No conversion was necessary for altitude.

4.3.3 Representing Space: The Global Layer Grid

An important objective of the global layer is to find a path, *i.e*, a list of GPS waypoints, from one point to another. Below, we provide a summary of some techniques we researched and considered to attain our objective, along with their advantages and drawbacks with respect to our constraints, to illustrate how, and why, we have chosen our method.

- **R**apidly exploring **R**andom **T**rees[6]. This method creates a tree like spatial structure at the start point that is randomly expanded by iteratively sampling a point in the surrounding space and connecting it to the tree if such a connection is possible, *i.e*, the connection does not cross a No Fly Zone. It is possible to improve this method by using certain heuristics, such as sampling with *voronoi diagrams* to prefer large unexplored areas, or expanding the tree towards the goal.

  Advantages:

  - *Uniqueness*: Paths found for each drone are unique. Therefore the drones are less likely to form crowded paths
  - *Memory Efficiency*: Memory usage is relatively small in sparsely populated areas: there is no need for persistent area representation.
  - *Flexibility*: It is easy to incorporate holonomic limitations.

Drawbacks:

- *Reusability*: The computed paths are not reusable: we would need to rebuild the whole data structure from scratch for each query.
- *Termination*: This technique will not terminate for unreachable endpoints, as the tree would expand forever.
- *Sub-optimality*: The computed paths tend to be up to twice as long as the optimal path, according to original paper.

- Probabilistic roadmaps[5]. This method creates a spatial graph in the surrounding space by first obtaining a random sample of points and then connecting the points that can be directly connected. The start and goal are then connected to the closest reachable node in the roadmap and the final path is created by a pathfinding algorithm, usually A*. This method can be significantly improved by biasing the sampling towards places in space that are densely populated with obstacles. We can take normally distributed points around no fly zone edges for example. In addition, it can also be improved by enforcing the nodes to connect using some heuristics, such as the *n-closest neighbours*.

  Advantages:

  - *Reusability*: The roadmap is built once and it is reused to answer all future queries, until a new static obstacle is added.
  - *Memory efficiency*: in sparsely populated areas.

  Drawbacks

  - *Crowded paths*: This method can create bottleneck paths that are used by all drones flying through the same area. This can cause unnecessarily heavy load on the reactive layer and an unnecessary high risk of collisions.
  - *No warranty*: The map is only probabilistically complete, which means that we would need to take infinitely many samples to ensure that we can route from each start point to each endpoint.
  - *Suboptimality*: The paths generated by this method tend to be worse than that of RRTs. However, they improve with number of samples taken.

- A Grid. This method splits the space into a regular square grid and uses it to represent the space as a graph where we can apply pathfinding algorithms.

  Advantages:

  - *Reusability*: The grid is computed only once, and is used for all queries. Moreover, the grid space representation can be reused for reactive layer computations.
  - *Optimality*: The grid-graph can produces near optimal paths.
  - *Detects Infeasibility*: The grid works with impossible end goals. Indeed it can detect infeasibility in *finite time*.
  - *Non-crowded paths*: The paths produced for each drone are different.
  - *Documentation*: There is a lot of literature written on pathfinding on grids, mainly since this method is widely used in the gaming industry.

Disadvantages

- *Memory*: The grid has a high memory consumption.

In summary:

- The cost of recomputing RRT's for each drone query was prohibitively high in an environment with many drones.
- The probabilistic roadmaps were problematic because of the risk of crowded paths: Even in a large open space, all drones would need to unnecessarily fly to the closest entry point in the map, which could potentially cause a high risk of collision.

Hence, we decided to compromise higher memory usage and use a Grid.

Thus, in our code, we represent an area of land as a `byte[][]` matrix. The idea of having this matrix is to divide the total area into many different sub-areas, through which a drone can or cannot fly. A viable sub-area will have a value of 0, while a *No Fly Zone* will have a non-zero value. We can then use the grid as a graph, where each zero value represents a node in the center of the sub-area with edges interconnecting each node (*c.f* figure 1). The graph can be used to apply the algorithms that we discuss later on.

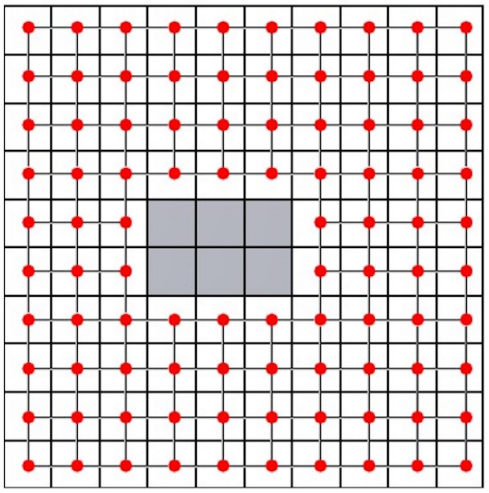

Figure 5: An example grid with a no fly zone, and how it can be interpreted as a graph. [4]

We can thus represent a $40 * 40$ km area with a sub-area grid size of 10 m for example. An advantage to this approach is that the actual grid and sub-grid sizes are modular. That is, the grid itself could also be a rectangle and can be of any dimensions. Moreover, the modularity of the sub-grid size allows us to divide the entire area into bigger chunks, thus decreasing memory usage, but compromising the precision of paths found using the graph, since the smaller the sub-area sizes, the more fine-grained the path. These parameters can be adapted depending on the area the client is dealing with, and the client's wishes.

4.3.4 Grid Algorithms

In this section, we discuss the crux of the global layer, *i.e* the algorithms that it uses.
First of all, let us detail how the no fly zones are placed on the grid, then the findings we have made concerning specialized graph algorithms we used for our application, such as the ones for

ray-tracing, A, Theta** (and its improvement, *Lazy Theta**), and why they are effective and relevant. We shall then benchmark some of these algorithms and conclude by discussing their respective advantages and drawbacks, as well as potential improvements. In particular, we will mention *Block A**[13] which has been the subject of research by *Peter Yap, Neil Burch, Rob Holte* and *Jonathan Schaeffer* from the University of Alberta.

4.3.5 Algorithm: Placing No Fly Zones

The first step to initialising the bitmap with the correct information is to put the No Fly Zones (that we shall abbreviate NFZ) onto the grid. In our code, the `updateWithNoFlyZones(List<NoFlyZone> zones)` is responsible for positioning the NFZ's onto the grid. The way NFZ's are represented is as *polygons*, or a set of vectors (two endpoints). For each vector in the polygon, the algorithm samples points at equal distance along the line that it represents, and sets the subgrid's containing those points to the value 1 (in red on the figure), and sets all neighbouring subgrids to value 2 (in orange on the figure), to indicate that those grids are viable, but dangerously close to an NFZ. As a general rule, no algorithm will find a path that makes a drone fly through a non-zero sub-grid.

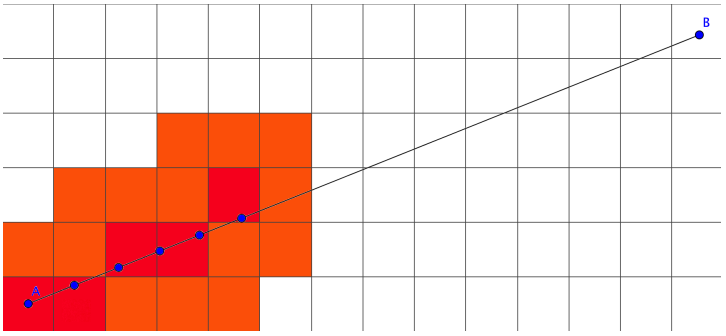

Figure 6: A line sampling example.

This process is then done iteratively for all vectors in the polygon. At the end of the process, the whole perimeter of the polygon will have been marked on the grid with a thick line.

4.3.6 Algorithm: Minimum distance to a NFZ

It is frequently the case that the upper layers of the system need to find out how close a drone is to a NFZ, to keep a minimum safety distance between the drone and the no fly zone when following a path. This algorithm is implemented by `distanceToNoFlyZone(Vector<double> point))` in our code. The `distanceToNoFlyZone(Vector<double> point))` takes the coordinates of a point, finds the sub-grid corresponding to that point and iteratively searches the square layers around the said sub-grid, since the closest NFZ can be anywhere around the given point. This process continues until a NFZ is encountered or until

1. The maximum number of layers is reached, in this case, the return value is positive infinity, there is no close NFZ.
2. An NFZ, i.e., a non-zero sub-grid is encountered, in this case, the algorithm will keep exploring the current layer to find and return the minimum distance to this said NFZ.

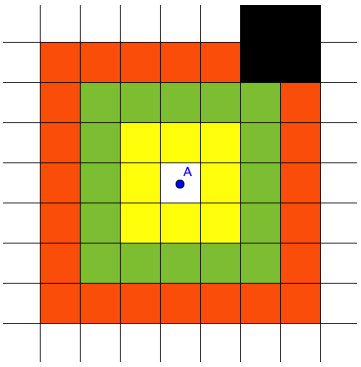

Figure 7: Exploring different layers to find the closest NFZ.

4.3.7 Algorithm: A*

As we know, the goal of the Global Layer is also to statically compute the best possible path for a drone to move from one location to another. In computer science, there exist multiple searching algorithms such as *Breadth-first search*, and one of the most famous perhaps, *Djisktra's algorithm*. A* is a combination of *Dijkstra's Algorithm* and of *Greedy best first search*, that is, it is optimized for a single destination, and prioritizes paths that seem to be leading closer to the goal using a heuristic. It combines the best of both[10]. Thus, we have implemented it in our code as one of the paths finding algorithms. It is relatively fast, as it only expands the nodes it has to [7].

Although A* has been proven to be *optimal* in a general graph, it is not always the case in grids because the graph overlay does not necessarily show all possible routes. Moreover, the computed paths are not always very *smooth* for a drone to follow. Consider the following scenario:

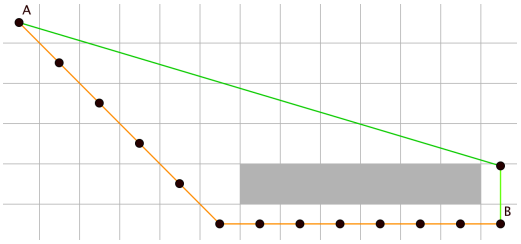

Figure 8: Pathological case for A*.

As one can notice, in this scenario, A* does not perform very well. Ideally, in real life, we would want our drone to avoid making a massive detour to get to its goal and make it fly straight in the same direction for as long as possible. There exist an algorithm to solve this problem.

Due to its simplicity and optimality guarantees, A* is almost always the first choice when it comes to path-finding. However, as we mentioned the shortest path on the underlying representation (i.e., the grid) is not necessarily shortest path in the continuous environment.[7]

One of the solutions to this problem is a technique called *post-processing*. What this does is apply smoothing techniques to the optimal paths found by A*. One can imagine this as 'straightening out' the paths wherever it is possible (i.e., when the paths found by A* are 'zig-zaggy'. The problem with such an approach is that it is very slow: Handling more difficult cases such as the one in figure 8 would indeed require a lot of effort, and the post-processing would most likely be ineffective [7]. Thus, one must find another solution to the *Any angle path planning*.

4.3.8 Theta*

A paper [8], that Alex Nash co-wrote with Sven Koenig, proposes a solution to the problem we have described above, called *Theta**. Theta* is a variant of A*, and builds upon it. It is similar to A* and finds optimal paths, without relying on any *post-processing* technique. The crux of this algorithm lies in *line of sight* checks. The concept is worth introducing here, as it will be required in further explanations. A cell, B is in the line of sight of cell A if there is a direct path from A to B with no NFZ in between. The two opposite cases are illustrated below:

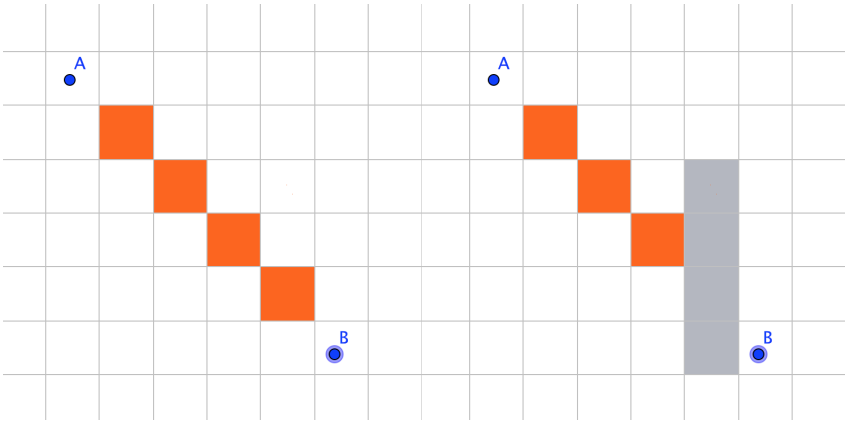

Figure 9: Line of sight checks in different scenarios.

In our code, we wrote a function called
`public bool lineOfSight (GridLocation a, GridLocation b)` which performs the appropriate calculations to determine this.

The difference between Theta* and A* is that Theta* does not restrict the parent of a node to one of its neighbours. During execution, Theta*, like A*, maintains two pieces of information per vertex: the length of the shortest path to it from the start vertex so far, and its parent. It also keeps track two data structures, a *maxHeap* which sorts the nodes to be expanded next by priority, and a *closed set*, the set of vertices already explored.[7]

The central part of the algorithm is the same, the only difference is that instead of only considering one possible path per expansion of a node, like A* does, Theta* considers two. When starting from a start node s_{start}, and expanding node s, Theta* evaluates:

1. The path from s to s', where s' is a neighbor of s.
2. The direct path from *parent(s)* to s', where s' is a neighbour of s, using a line of sight check.

It will then pick the shortest viable one, and update the $g.s$, that is, the cost to reach s from s_{start} and s's parent, and continue.[7]

The following picture illustrates this:

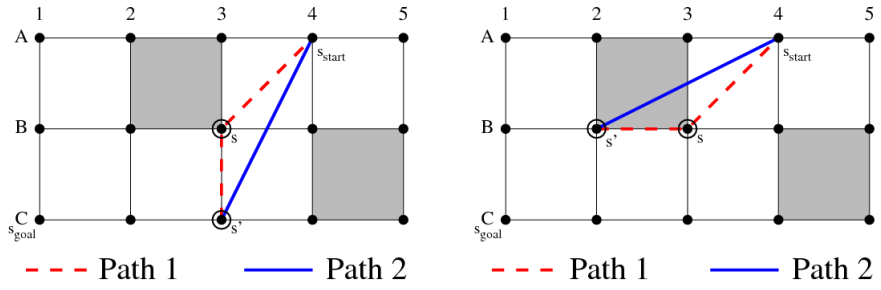

Figure 10: Theta* path choices [7].

4.3.9 An improvement to Theta*: Lazy Theta*

Continuing our research on Theta*, we realised that the authors of the previous paper on Theta* wrote another paper [9], about an algorithm called Lazy Theta*, an improvement to Theta*, which we subsequently implemented in our code. In this section we won't review many details, but rather we will present a brief overview.

The main idea behind Lazy Theta*, as presented in the paper, is to reduce *Line Of Sight* computations. If we consider Theta*, we realise that at each step, two paths are considered, the direct path, using a LOS check, and the path that A* would normally consider. However, it is not always the case that this node be expanded, which means that a LOS check would have been calculated for nothing. The key idea behind Lazy Theta* is too *always assume a node is in line of sight of its parent*, and only *lazily* check this using a line of sight check, once the child is expanded. If the assumption turns out to be false, then the algorithm defaults to considering the path that would initially be considered by A*.

This optimisation hence reduces the number of total LOS checks throughout computation, and improves the algorithm's overall performance.

4.3.10 Path-finding Algorithms Benchmark Study

Following our research into path-finding algorithms, we decided it would be interesting to measure various statistics for each path-finding algorithm on multiple test cases, to see if practice confirms the theory. We will comment and discuss the results that were found. All statistics were measured in a similar environment, that is, the same computer. When we had to measure execution time for an algorithm, we decided to record the given time three times to get the mean.

First, we decided to measure execution time. This yielded the following table:

Test Case	A*	Theta*	Lazy Theta*
Imperial Tunnel	118	141	124
	120	138	132
	121	137	129
Imperial Tunnel Mean	119.7	138.7	128.3
The Great London Beehive	859	1318	1166
	916	1486	1075
	894	1555	1201
The GLB Mean	889.7	1453	1147.4
The Nightmare of Hyde Park	9771	28436	21330
	9687	27970	24786
	10118	28273	21609
The NHP Mean	9858.7	28226.4	22575
The Great Wall Of Imperial College	394	913	838
	447	910	840
	416	912	841
The GWIC Mean	419	911.7	839.7

Table 3: Execution times for path-finding Algorithms on various test cases.

As we can notice, the *A** algorithm is by far the fastest. For smaller test cases, such as *The Imperial Tunnel* or *The Great Wall of Imperial College* the times are similar, however, the larger and the more complex the test case, the more significant the difference. For the *Nightmare of Hyde Park* test case, for example, *Theta** is up to 2x slower that *A**. What we can conclude from this is that *A** is the algorithm of choice when computational complexity is essential and AATC service needs to be extremely efficient. This might, however, as we will see, compromise the lengths of the paths. Another interesting result from this table is the difference in speed between *Theta** and *Lazy Theta**. Clearly, we can see an improvement between the two. The fact that *Lazy Theta* star is expanding fewer nodes and doing less CPU-intensive operations such as LOS checks is making a difference, as we can see in some cases, such as the *NHP* once again, it is 1.25x faster.

Let us now have a look at the differences in terms of the lengths of the paths computed.

Test Case	A*	Theta* and Lazy Theta Star
Queen Threat	6.11	5.98
The Imperial Tunnel	10.83	10.22
The Great London Beehive	52.63	49.77
The Nightmare of Hyde Park	118.4	115.32
The Great Wall of Imperial College	16.37	15.46
Multi Drone collision	15.27	14.82

Table 4: Total path distances for all drones (in km).

In this case, both Theta* and Lazy Theta* find the same path, so we include them in the same column. As expected, the paths found by A* are longer than that of those found by Theta*. Theta*'s extra computation paid off, by avoiding all the pathological cases like that of the one in figure x.x. Indeed, for the *Great London Beehive*, the results are quite significant: the total drone path plan lengths were 3km less, and thus, these paths closer to the optimal ones, and much better suited if the client's goal is to save drone flight time, and subsequently battery usage.

4.3.11 Conclusion and possible Extensions

Although our algorithms work well in practice, there still room for research on this topic. An interesting paper that we came across during our research was a paper entitled *Block A**[x]. The main idea behind *Block A** is to use a pre-computed local database to achieve high-performance pathfinding. The A* algorithm proposed would differ to the traditional one in the way that it explores $n * m$-sized blocks of cells as opposed to individual ones. Despite not having the time to implement Block A*, experiment with it and study its results, we believe this would be the next step to take, as this could potentially drastically improve overall performance if the author's claims are true.

4.4 Reactive layer

As outlined in the previous sections, the global layer will compute a list of waypoints that are guaranteed to be free of collisions with static obstacles (NFZ). The next objective is now to use this list of waypoints to generate real-time direction and speed recommendations based on the position of the drone with respect to not only static obstacles but also to dynamic obstacles (manned aviation and other drones).

4.4.1 Algorithm Overview

We decided to use an artificial potential fields algorithm, and we initially took this paper [14] as the main reference. The idea is to create an artificial potential field in the area where the drone works and associate a potential to every point in this area. We can then use this field to navigate the drone by attempting to minimize its potential over time.

For navigation to be successful, the potential field needs to follow those two constraints:

- It needs to decrease smoothly with distance to the goal, *i.e.*, the closer the drone is to its destination, the lower its potential.
- It needs to increase rapidly with distance to the obstacle *i.e.*, the closer the drone is to either another drone, manned aviation or no fly zone, the higher its potential.

If we restrict our attention to 2D spaces for a moment, we can use a simple physical analogy to show how this algorithm works. We can visualize this space as a landscape, where points with high potential are peaks and points with low potential are basins. We can then imagine a drone as a steel ball, that is rolling downhill (from places with high potential) to places with low potential.

In the simplest case, when there are no obstacles, the landscape is simply a cone centered on the goal (0,0) and the drone will fly directly towards it as in figure 11. If we add 3 point obstacles (these can be drones or manned aviation) at positions for example (5,-5), (-2,3), (0,8), the potential landscape would look like in figure 12. It is then very easy to imagine how this would work for arbitrary obstacle shape, for example, rectangular no fly zones would create rectangular potential peaks.

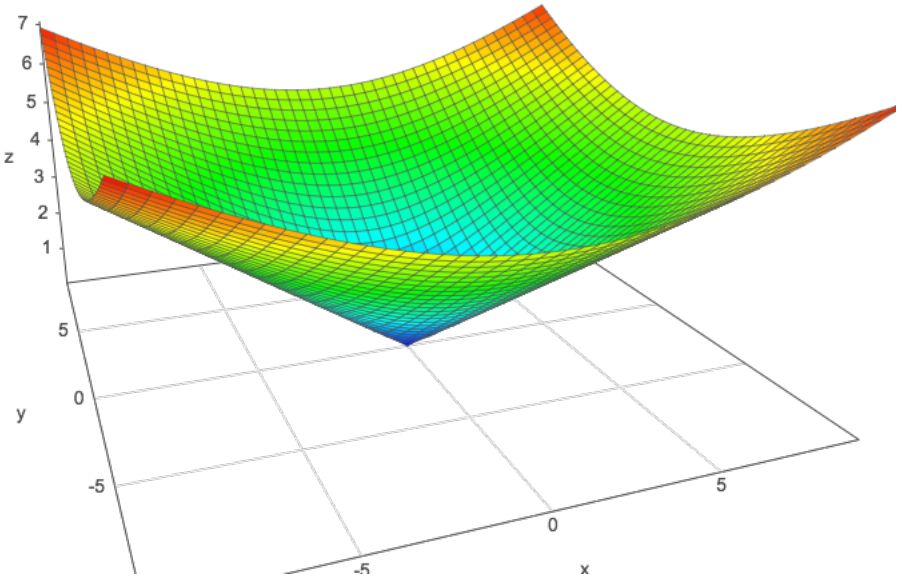

Figure 11: Potential landscape over 2D space with no obstacles.

However, the analogy of a steel ball rolling down the hill towards smaller potentials only works well if we restrict our attention to 2D space. In 3D, we can think of the drones as magnets with

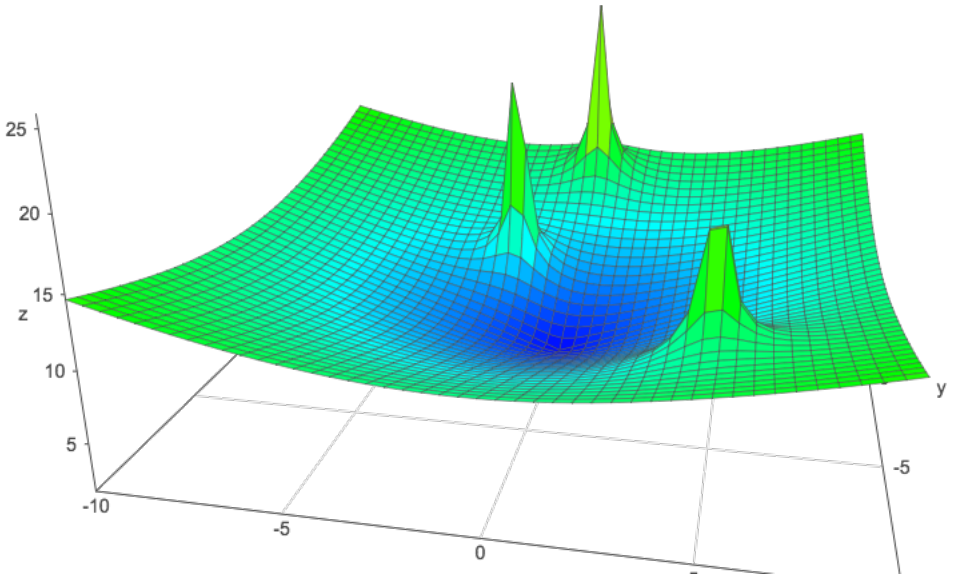

Figure 12: Potential landscape over 2D space with 3 point obstacles.

the same polarity. The drones that are close together are repelling each other strongly while drones that are far away have no effect on each other. The goal of the drones is modeled as a magnet with opposite polarity with respect to the drone trying to reach it.

One could argue that there is no need to have a global layer because the reactive layer seems to be solving the navigation problem completely. Unfortunately, this is not the case. Consider the example in figure 13. The drone on the bottom of the area is trying to reach the goal on top. There is a U-shaped obstacle in its way. If we use the classic APF approach, the first picture on the left, the drone would end up stuck in the local minima: the drone would just fly towards the goal until it starts sensing the presence of an obstacle and then it would start oscillating at the same spot forever. There exists a pure APF solution to this problem that introduces changes to the potential field, so that the drone rotates around the obstacle. This does not push the drones away perpendicularly, but it pushes them away with a 45 degree angle so they will eventually reach the destination. This approach is very difficult to implement mathematically and also produces highly sub-optimal paths (as shown in the center picture). Our solution avoids local minima problems thanks to the global layer, that already produces paths that are guaranteed to pass around static obstacles.

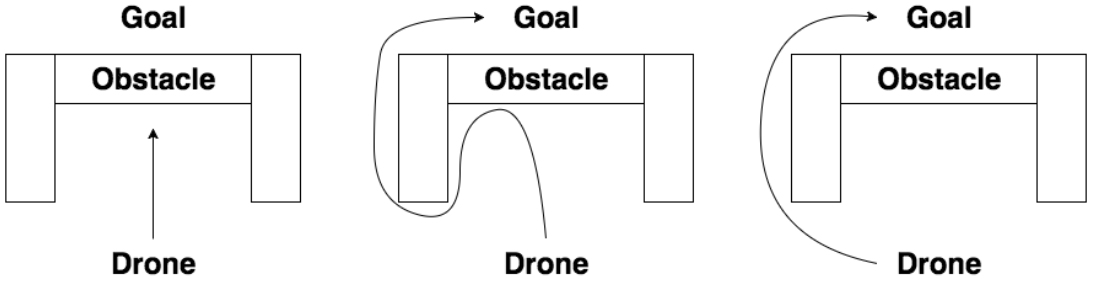

Figure 13: Comparison of classic APF, rotating field APF and our implementation with global layer.

4.4.2 Original mathematics and how we changed it

The magnitude of potential U at each point is generally given as the sum of attraction potential U_p and repelling potential U_r:

$$U = U_a + U_r$$

The velocity is then simply computed by finding the direction of the gradient of potential at the given point and taking the negative. This is a version of a very well-known gradient descent algorithm that is widely used in optimization research and deep learning.

The problem now is to find U_a and U_p. The original paper [14] as well as most other papers on the topic of artificial potential fields that we have read are inspired by physics and therefore their expressions of attracting and repelling potential are based on the equations of electromagnetic force and look like this:

$$U_a = \frac{1}{2}\rho_a d_{goal}^2$$

$$U_r = \begin{cases} \frac{1}{2}\rho_r(\frac{1}{d_{obst}} - \frac{1}{d_{influence}})^2 & d_{obst} \leq d_{influence} \\ 0 & otherwise \end{cases}$$

where d_{goal} is the current distance to the goal, d_{obst} is the current distance to the nearest obstacle and $d_{influence}$ is the maximal distance where the drones influence each other and ρ_a and ρ_r are attraction and repulsion constants respectively.

After implementing this method, we observed strange behavior where the drones were much more likely to hit obstacles if they were far away from the goal. This was happening because the gradient of attraction potential was growing linearly with the distance to goal, therefore the relative contribution of repelling potential to the total gradient was diminishing with distance to the goal i.e. the drones far from the destination were ignoring the obstacles.

$$\frac{\delta U_a}{\delta d_{goal}} = \rho_a d_{goal} \propto d_{goal}$$

We therefore decided to redefine U_a without squaring it so the gradient is constant regardless of the position of the drone to overcome the issue with ignored obstacles. The disadvantage of this

approach is that the drone does not naturally slow down when approaching the goal, so we had to post process the speed to make the drone stop.

Similarly, we removed the square from the repulsion potential U_p and removed $\frac{1}{d_{influence}}$ since it became a constant. However, we subtracted the safe distance from d_{obst} to make the drone react to obstacles sooner.

Finally, we added another custom potential to the system. We have noticed that if multiple drones meet in proximity to no flight zone, they occasionally tend to start oscillating (flying back and forth for a short period while they are stuck in a local minima). During this oscillation, the drone tend to quite quickly return to their positions at the previous time instant, so we needed to discourage this behavior. The new potential U_{ret} repels the drones from their previous position as well as from other obstacles. This can be visualized as a cat chasing a mouse - the cat is a strong motivation for the mouse not to sharply turn around unless it is going to hit a wall.

The full set of equations that we used to obtain recommended unnormalized velocity v_{raw} of the drone is then:

$$U_a = \rho_a d_{goal}$$

$$U_r = \begin{cases} \rho_r(\frac{1}{d_{obst} - d_{safe}} & d_{obst} \leq d_{influence} \\ 0 & otherwise \end{cases}$$

$$U_{ret} = \rho_{ret} d_{last}$$

$$U = U_a + U_r + U_{ret}$$

$$\boldsymbol{v_{raw}} = \nabla U$$

where ∇U is the gradient of U (partial derivative with respect to x, y and z).

4.4.3 Implementation

The implementation directly follows those mathematical equations. Whenever a drone requests a new command, we need to approximate the gradient of potential at its current position. We decided to compute this from first principles - we first compute the potential at the position of the drone and then we subtract this value from potentials computed after applying a small offset (1m) to each axis respectively.

$$\nabla U(x,y,z) = \begin{pmatrix} \frac{\delta U(x,y,z)}{\delta x} \\ \frac{\delta U(x,y,z)}{\delta y} \\ \frac{\delta U(x,y,z)}{\delta z} \end{pmatrix} \approx \begin{pmatrix} U(x+1,y,z) - U(x,y,z) \\ U(x,y+1,z) - U(x,y,z) \\ U(x,y,z+1) - U(x,y,z) \end{pmatrix}$$

Therefore we need to compute four point potentials to estimate the gradient. For any one point, the following algorithm is used to calculate potential:

1. Compute attracting potential by using euclidean distance from the point to the goal.
2. Compute repelling potential by using the euclidean distance to the nearest obstacle, provided by the global layer.
3. Compute return potential using the euclidean distance from the point to the place where the drone was in during the last time step (this is a variable saved in the drone object).

4. Sum the values to obtain the potential at the point.

As mentioned before, since we removed the square computation of attracting potential, drones do not slow down when they approach their destination. Therefore we used gradient only as an indication of direction of desired velocity vector and we computed its magnitude (speed) as follows:

$$speed = max(max\_speed, d_{goal})$$

This ensures that the drones slow down when approaching the destination. We then simply normalize the gradient, so it becomes a unit vector and multiply it by speed to compute the final velocity recommendation.

4.4.4 Genetic Algorithm to improve constants

So far, we haven't talked about the constants that are driving the APF algorithm: ρ_r, ρ_a, ρ_{ret} and $d_{influence}$. The interplay of these constants is crucial to the success of the algorithm. If ρ_a is too high relatively to ρ_r drones ignore the obstacles and die hitting them, on the other hand, if ρ_r is too high, the drones oscillate and never reach the goal. If ρ_{ret} is too high, drones are so reluctant to turn that they take extremely sub-optimal paths and sometimes never reach the goal.

Firstly, we tried to set those constants empirically by manually adjusting the values and looking at the results. This approach was too time-consuming but provided us with valuable insights on the sensible ranges those values live in.

We have then decided to try using a genetic algorithm to solve this optimization problem. Firstly, we defined a simple cost function for a run of our test suite which heavily penalizes drones dying or not reaching the destination:

$$cost = \sum_{testcase} \sum_{drone} \begin{cases} batteryUsed & if\ drone\ reached\ destination \\ 100000 & otherwise \end{cases}$$

Having the cost function, we have implemented a classic genetic algorithm where each entity represented some combination of the constants to optimize. Our initial population was based on our findings from manual testing as shown in figure 4.4.4. We ended up having 240 entities in our population and knowing that one run of our test suite takes 1 - 3 minutes, we could run one generation in approximately 8 hours, which was quite reasonable.

One generation run consisted of the following steps:

1. Run the test suite with the current population and record the cost of each run to the entity. The cost is later used to evaluate the fitness of the particle. This step is the most time-consuming.
2. Choose elites - choose 10 entities with the lowest cost to be the elites copied to the next generation without any change. This is called elitist selection and it guarantees that the results of the algorithm only improve over time, because the best entities are never lost.
3. Natural selection - eliminate weak entities using a probabilistic approach. We compute the discrete probability distribution based on the fitness of each particle. This distribution is then repeatedly sampled and corresponding entities are copied to the next step (the entity

is not removed from the distribution after it was picked, so strong entities will appear in new generation multiple times). The probability of each entity surviving is given by:

$$p_j = \sum_{i=1}^{230} \frac{1}{cost_i}$$

where $cost_i$ is found in the first step.

4. Crossover - the entities obtained in the previous steps are randomly paired to couples (corresponding to mother and father). The new entities are obtained by taking the average of each constant stored in the parents. Elites are used in this step.
5. Mutation - mutate the produced children to obtain new values of all constants. We replace each constant with a sample from a Gaussian distribution centered at the original value of the constant and with standard deviation equal to 5% of the original value.
6. Reassemble - in the last step, we copy over the 10 elites from step 2 and 230 mutated entities from step 5, so we finally end up with 240 entities in next generation. Now the algorithm starts over with step 1.

In the final generation, we took the best entity and used the constants it represented in our algorithm. Unfortunately, the results with the best entity were improved only by around 2%, but we hope the improvement would be more significant with more simulation scenarios, bigger population and more generations. One might argue that by doing so we have overfitted the constants on our test suite. It is a valid concern. We have tried to make the test suite quite general and make it contain some complex scenarios so it should be reasonably exhaustive. We have also added some tests later that seem to be working reasonably well (this can be considered to be a validation set). However, this was still only a proof of concept run and in real life, we would need to run it on more training scenarios with a proper validation set. It would be also possible to run this algorithm online on the production system to improve it over time (obviously, we are talking about fine tweaking rather than big changes that my crash the drones).

Also, when the system starts using multiple different types of drones with different constraints on speed and on acceleration, it will be useful to determine a different set of constants for each of them to achieve optimal performance. This is why it was crucial to design an automatic way to determine them.

Constant	Min	Max	Step	Total
Attraction	0.8	1.1	0.1	3
Repulsion	100	500	100	5
Return	0.1	0.7	0.2	4
Influence distance	300	600	100	4
Population				240

Table 5: Initial configuration of the genetic algorithm.

4.5 Drone service

This double layered model is implemented as a web service so it can be accessed by drones (these are simulated by the test bench) via an API as suggested by Altitude Angel on the first meeting. The API is simple (not intended to be RESTful) and provides only two endpoints:

1. `/api/drone` for actions related to drones
2. `/api/environment` for all actions related to the status of the system

When the service is started, it contains no data at all and therefore it waits for a POST request to `/api/environment`. This request is sent before the execution of each test case by the test bench and it contains information about the no fly zones and manned aviation. This information is then processed by the service and loaded into internal data structures.

The service in testing configuration does not have its own clock and it uses the time provided by the test bench. This allows us to run the test cases deterministically and also to run them faster than in real time (a drone journey that would take hours in reality, takes only seconds to run). The service advances its internal time upon receiving a GET request on `/api/environment` (tick event). The tick event is then propagated to each dynamic entity in the service(drones and manned aviations), which are then moved forward according to their current velocity.

After the movements are done, we execute the collision checks. We iterate over all drones and we compute the distances to other drones and manned aviations, checking that they are smaller than the safe distance (defined to be 10m by specification). Then we also check that the drones are not within no fly zones by counting the number of edges of no fly zone that are above the drone (number of edges intersecting the ray started at the point and going up vertically) . If the number is even, then a point is inside the no fly zone, otherwise, it is outside (this is known as even-odd rule algorithm [11]). If any collision check fails, the drone is set as dead.

The drones communicate with the service via `/api/drone` endpoint. When they first connect and also on each successive step, they send their telemetry data including position, speed, plan (the list of waypoints drone needs to visit) and status to the service using PUT action. The data is then saved in memory. The first connection is special because the system also asks global layer to generate a list of waypoints that is guaranteed to be free of collisions with static obstacles as described in the section on the global layer. This waypoint is then retained throughout the drone journey (the drone directly uses global layer only once on first connection) along with the index of next waypoint, which is 0 at the start.

On every time step, each drone also sends a GET request on `/api/drone` endpoint which returns the recommended velocity the drone should aim to achieve in the next second. This is done by querying the APF algorithm described in the section on the reactive layer. It also checks if the drone is at the next waypoint and if it is, it increments the next waypoint index. The destination is reached when the next waypoint index is equal to the number of waypoints - in this case, we just generate LAND command.

4.6 Visualiser

The visualiser was developed from the beginning of the project in order to give a graphic representation of the simulation results. The visualiser is a browser-based tool that can load and graphically playback an event log representing a simulation produced by the test harness. The simulation can be played as an interactive 3D animation within the browser, showing the Earth, drones, no fly zones and any manned aviations that were involved.

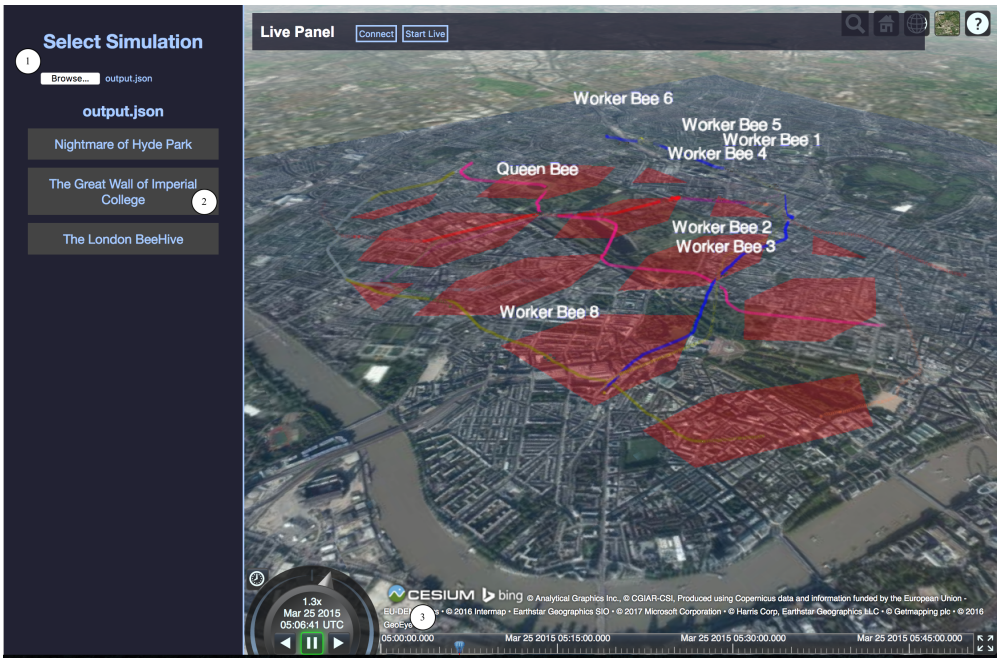

Figure 14: Visualiser tool in use.

4.6.1 Usage

The visualiser is used by first selecting an event log JSON file using the browse button (1) and then clicking on one of the contained simulations to view it (2). The simulation is then shown in the main viewing panel and playback can be controlled using the timeline (3). The camera's view can be adjusted by clicking and dragging on the Earth. In the simulation shown in the figure above, the drones are labeled with white text, their 3D flight paths shown by the colored lines and the no fly zones as red polygons on the surface of the Earth. Additionally, any flying entity (manned aviation or drone) can be double clicked to make the camera track the object close up as shown in figure 15.

4.6.2 Implementation

Being a browser based tool, the visualiser is written in HTML5, CSS, and JavaScript which makes it inherently platform independent.

The actual 3D visualization is created using a JavaScript library called Cesium JS[1]. It is an open-source library for 3D globe and map projects. The visualizer takes the event log JSON and makes calls to the Cesium library in order to display the playback of the simulation.

In designing the look and feel of the visualiser, we used online colour picker tools[12] to influence our design choices.

4.6.3 Reproducibility

The separation of the test harness from the visualiser tool contributes towards a productive development workflow of the AATC. The encapsulation of a simulation in an event log JSON file produced by any version of the test harness against any version of the AATC service, means that simulation results can be stored easily as files and viewed at any later point. It also makes

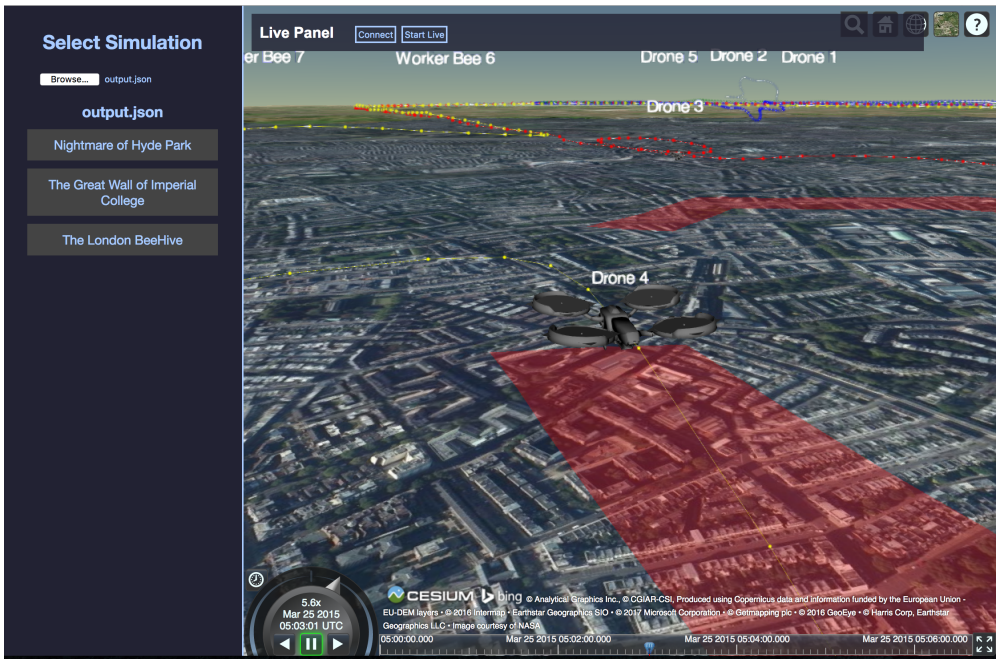

Figure 15: Double clicking on an entity causes the camera to track it as it moves.

it easier to share the simulations as it is just a case of transferring the event log file to someone else and then playing it back on their machine's browser. Finally, it is also simple to view the simulation as many times as necessary, stop it or slow it down at any point to clearly see each event that happened and be able to thoroughly analyse it.

4.6.4 Live Mode

As part of iteration four, we developed an extension to the visualiser and test harness system which allows multiple users to 'connect' to the test harness and interact with a simulation in real time. In this "live mode", users on different machines can choose to 'hijack' individual drones in the simulation and control them using the keyboard. This was suggested by Altitude Angels as a potential extension in one of our bi-weekly meet ups and is motivated by the need to interactively test how the AATC service responds to "rogue drones" - that is, drones which do not respond to the commands given by the AATC. It also allows the dynamic creation of test cases which would be too cumbersome to manually enter into a JSON file.

4.6.5 Live Mode Usage

Each user should press the "connect" button in the visualiser tool on their own machine to establish contact with the test harness. When all users are ready, the "Start Live" button can be pressed to begin the live simulation. Then, each user has the "hijack" button for each drone, which allows the user to take control of that drone using the keyboard. The test harness will then ignore the AATC commands for the hijacked drone and instead follow the commands specified by the user.

4.6.6 Live Mode Implementation

"Live Mode" was implemented by extending the existing visualiser and test harness so that they could bidirectionally communicate in real time. The real-time communication is achieved using WebSockets.

The messages sent from each visualiser client instance to the server are shown in appendix A.

Upon receiving the "start" signal from any of the connected visualiser clients, the test harness will send the manned aviation and no fly zone data for the running test case to all the clients. After this, the test harness begins the simulation and sends the position updates of all the simulated drones to the connected clients at regular intervals when the updates occur.

5 Conclusion and future extensions

From the beginning of the project all team members were willing to create a product that would bring something new and innovative to the table. The autonomous air traffic control for drones met all our expectations. It was an interesting and challenging project that attracts many research teams in well-known companies, therefore, we had a sensation we could create something that is really needed.

Autonomous air traffic control was an amazing opportunity to learn new programming languages such as C#, improve those we already had been familiar with e.g. JavaScript or Python and use tools we haven't used before like Rider and Azure. The project was often conceptually challenging and required more research than we expected. We managed to build the system that can not only detect when the drone collides but it is also able to take action to prevent that collision. Furthermore, our system is able to take into account static as well as dynamic entities in the environment and provide efficient routes for drones in a local airspace.

5.1 Deliverables

Our submission contains 3 main components:

- The drones service - web service is written in C# that implements the algorithms as described in previous sections and exposes an API for drones to connect to.
- Test Bench written in Python that contains the extensive set of test cases to evaluate our solution. Each test case contains the description of the environment and paths of the drones to be simulated.
- Visualiser that allows us to visualise the outputs of test cases produces by the test bench and allows us to interact with the system in real time by controlling the simulated drones.

5.2 Feedback and Results

This project aimed to create a system that would safely and securely integrate drones into the air space and our main goal was to satisfy the expectations of Altitude Angel. In each meeting with them we got constructive feedback mostly suggesting new edge cases for us to consider which they themselves had encountered issues with. These were taken into account for further improvement. Our last demo during the final project meeting was successful in that we received a lot of positive feedback . Our supervisor and project mentors were fully content with the final outcome and capability of the system. They valued highly the idea of artificial potential fields as well as its combination with other path-finding algorithms. Furthermore, they enjoyed the demonstration of the visualiser's live mode which was deemed to be a very useful and innovative addition to the whole project.

5.3 Future work

Even though we believe that our deliverables accomplish the specification given by our client, there is still a lot of scope for improvement. The biggest problem we see is the fact that the system currently works only in a small region and whilst we developed with scalability in mind,

we did not demonstrate it through our implementation. In our case, scaling our system would be to extend the AATC service to work across the whole planet. We do however present a short overview of how this could be implemented if we had more time.

5.3.1 Global Scale Navigation Design

Our current system is limited to the size of one global layer environment which is currently a square with a side of 40 *km*. It is enough to cover even big cities, such as London, but it would still be better if we could improve our system to cover potentially unlimited areas.

We propose adding another layer on top of our global layer that will be able to compute the static routes between the 40 *km* * 40 *km* patches. This layer will firstly statically preprocess each of those patches using the following algorithm:

1. Draw points in uniform intervals along the edges of the square patch. We propose a point on every 1 km, so there will be 156 points (a simplified version with 16 points is shown in figure 16)

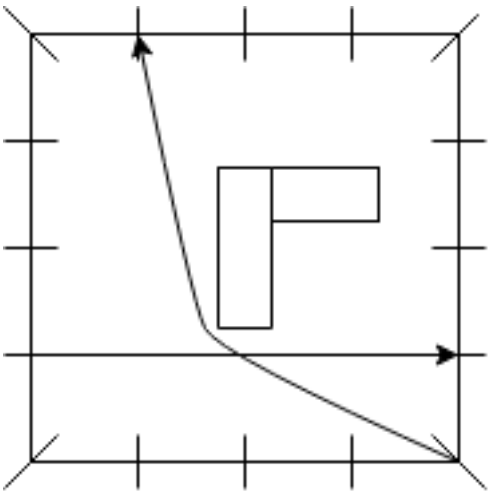

Figure 16: Potential landscape over 2D space with 3 point obstacles.

2. Load No Fly Zones into those patches and use global layer algorithm to compute the cost of flying from each endpoint to each endpoint (2 such distances are shown in figure 16)
3. Store those values as a triangular 2D matrix with 156*156 elements in a persistent data store.

The points on the edges should be overlapping with the similar points in the neighboring patch, therefore we can then use this preprocessed data and execute any path planning algorithm. We propose a modified version of A* that uses an LRU cache to store those patches in memory. Whenever A* needs to expand a given node, it will load the corresponding patch from the persistent database and store it in the LRU cache. In the best case (which is quite often) we will use each patch only once.

Because of the considerable amount of database queries, the performance of this solution might not allow real-time operation. This is not an issue because the drone can issue this request before it starts its journey and wait a few seconds for a reply.

It is also impossible to split the Earth into geometrically perfect 40km x 40km squares, so in real life, we would need to design a different geo partitioning scheme. One candidate would be the system of geo hashes, that splits the Earth into rectangular areas. One patch would then correspond to a single geo hash.

5.3.2 Scaling and Reliability

Our system is currently running on a single machine, which would not be acceptable for real deployment because it would be impossible to assure high availability and scalability. We propose a simple design of a larger deployment handling multiple (10 in the example) geo hash areas without single points of failure (assuming distributed database) in figure 17.

The query is first routed by DNS Round robin to one of Geo FrontEnd routers. Those machines read off the position of the drone from the query and route it to the backend responsible for the area where the drone is currently located. Each area containing a drone should be available in one of the backends and GeoFrontend routers should keep track of the area assignments. The backends are always duplicated, where one is active and the other one is a backup. All communication is sent to both machines on best effort basis (if a packet is lost, we don't wait) and the backup machine is maintaining the same model as the active machine so it can immediately take over should there be a failure. Failures are detected by sending periodic heartbeats. If even higher reliability is necessary, we can run n machines and active one can be elected using PAXOS.

The global planning backend machines should implement the algorithm outlined in the previous subsection. It should be able to compute the routes between concerned geo hashes in long distance journeys. They are not used for journeys within a geo hash. The machines are replicated as necessary and they communicate with a shared database. All machines would communicate via remote procedure calls.

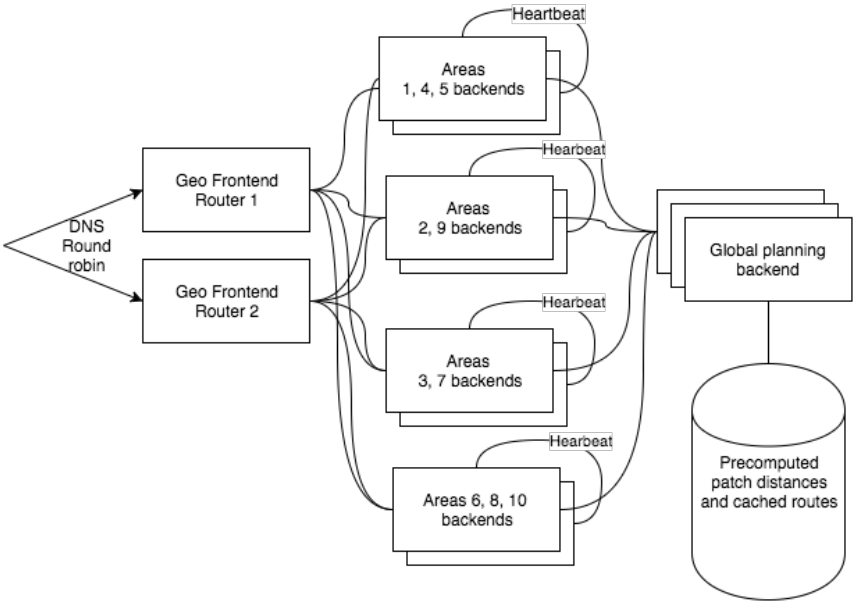

Figure 17: More reliable and scalable system design.

A Visualiser Live Mode WebSocket Communication

Message Type	Description	Parameters
Connect	Visualiser instance connects to the test harness.	None.
Start	Signals the test harness to start simulating the given test case.	None.
hijackInput	Takes manual control of a given drone, specifying a dx and dy direction for its velocity. The test harness will make the drone ignore any subsequent AATC commands and instead follow this input, until an "unhijack" message is sent.	*uid* - the id of the drone to send the input to. *dx* - the x component of the input. *dy* - the y component of the input.
unhijack	Yields manual control of a given drone, causing it to resume responding to AATC commands.	*uid* - the id of the drone to return AATC control.

B Bibliography

References

[1] Cesium, javascript webgl virtual globe and map engine.

[2] Proj.net, c# library for coordinate conversions, 2017.

[3] Utm: 0.4.1 python package index_2017, 2017.

[4] Wolfram Alpha, 2014.

[5] Lydia Kavraki, Petr Svestka, Jean claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION*, pages 566–580, 1996.

[6] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[7] Alex Nash. Theta*: Any-angle path planning for smoother trajectories in continuous environments — aigamedev.com, 2010.

[8] Alex Nash. Any-angle path planning. 2012.

[9] Alex Nash, Sven Koenig, and Craig Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. 2010.

[10] Amit Patel. Introduction to a*, 2014.

[11] M. Shimrat. Algorithm 112: Position of point relative to polygon. *Communications of the ACM*, 5(8):434, 1962.

[12] Awwwards Team. Trendy web color palettes and material design color schemes & tools, 2015.

[13] Peter Yap, Neil Burch, Rob Holt, and Jonathan Schaeffer. Block a*: Database-driven search with applications in any-angle path-planning. 2011.

[14] Lihua Zhu, Xianghong Cheng, and Fuh-Gwo Yuan. A 3d collision avoidance strategy for uav with physical constraints. *Measurement*, 77:40–49, 2016.