CS/ME/EE 134 Project Report

Qingzhuo Aw Young, Timur Kuzhagaliyev

1 Introduction

In this project, we focused on autonomous exploration and mapping of 3D environments using drones and planning tasks in swarm robotics. We begin by experimenting with some basic exploration concepts involving drones, as well as multi-agent task solving in a cooperative setting.

We explored both areas in parallel, limiting the scope of each to focus on solving the main problems of each as defined in the next section, Project Goals.

As our first goal, we'd like to tackle autonomous exploration and mapping of a 3D environment using a single drone, either using RGB-D data or optical imagery in combination with photogrammetry. We will only consider closed spaces - that is, the goal for the drone will be to map out the current room (e.g. a lecture hall) until some map uncertainty threshold is achieved. This process will involve not only performing SLAM while the drone travelling around the environment, but also the problem of intelligent exploration.

In our second goal, we'd like to explore solving tasks involving multiple goals (e.g. reach all checkpoints, move an object, preserve robot integrity, etc) with multiple agents in cooperative settings. Goals might require multiple agents to accomplish, e.g. an object too heavy for a single agent to move, which requires planning for solving individual goals whilst also solving multiple goals in parallel by splitting the swarm into smaller units. One additional notable challenge is collision avoidance swarm navigation.

2 Project Goals

This section briefly describes the goals for each part of the project, and nice to have things that we plan to work on if time permits. We also limit the scope of each task, as described in the non-goal sections in order to focus on solving the essential problems and not waste time on infrastructural and etc tasks.

Exploration of unknown 3D environments

Goals

- Quadcopter control & sensing in emulated environment.
- Simultaneous localization & mapping in 3D environment.
- Navigation in environment using internal map of surroundings.

Stretch Goals

- Optimal policy for exploration

Non-Goals

- Hardware implementation. Due to budget constraints, we will only be conducting our experiments using drones in a simulated environment.

Multi-agent multi-goal task solving in cooperative swarm setting

Goals

- Finding policy to achieve multiple goals under constraints
- Collision-free navigation within swarm
- Robust solution that works with different goals

Stretch Goals

- Collision-free swarm navigation with no communication (allow individual agents/units to perform tasks including navigation in isolation).

Non-goals

- Communication / networking between agents. We'll assume that a message broadcasting interface is available to the entire swarm.

Overall stretch goal

An optimistic stretch goal for the project is to combine our methods in multi-agent co-operative task solving with multiple goals (mapping entire environment, conserving fuel, no collision between agents) with our 3D mapping using drones, culminating in a solution for simultaneous mapping of unknown environments by multiple drones. Such a solution will have interesting applications in search & rescue and disaster recovery scenarios. We were unable to complete this stretch goal due to time constraints and limited manpower.

3 Approach and Technical Details

3.1 Autonomous exploration and 3D SLAM of unknown environments

This part of the project used Gazebo for simulations and ROS for all navigation, SLAM, and other functions necessary for autonomous exploration. All of the source code, scripts, launch files and configuration files can be found in the `caltech_samaritan` repository on GitHub, available at https://github.com/TimboKZ/caltech_samaritan. The repository also contains a README file with installation instructions.

3.1.1 Setting up of drone simulator & environment

We heavily relied on Gazebo simulations, including both the test environment (indoor, outdoor) and the drone itself. We used the hector_gazebo [1] framework to emulate the drone, namely a quadrotor drone with a Kinect attached to it, pointing directly forward. Kinect camera provided the depth data as a 3D point cloud. Figure 1 shows the drone model and an example depth cloud as received from the simulated Kinect, visualised in rviz.

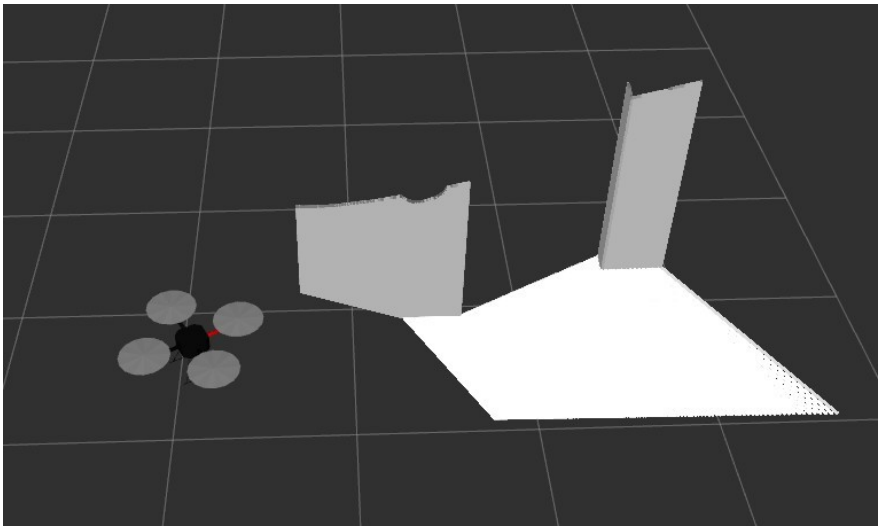

Figure 1: Simulated `hector_quadrotor` drone with a visualised depth cloud from the Kinect.

For testing purposes we used two different environments: an outdoor setup and an indoor setup. As expected outdoor environment provided more free space and was generally safer to navigate in, whe indoor environment contained a lot of narrow corridors which posed a challenge to our planner. Figure 2 shows the visualisations of both environment.

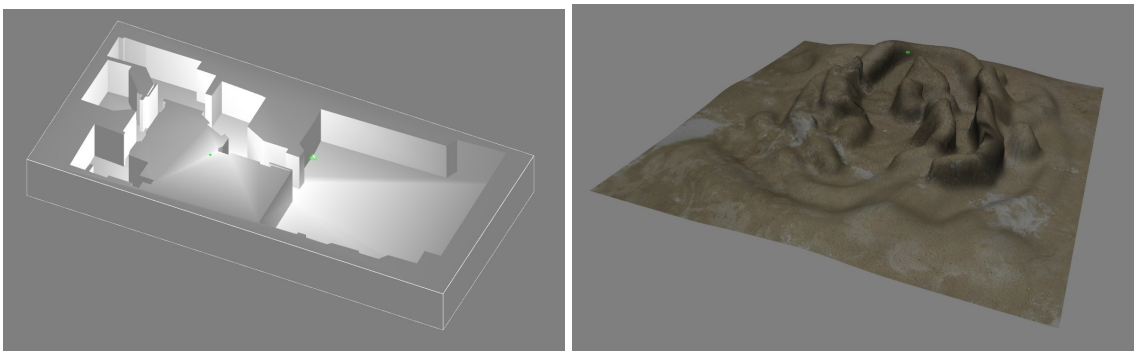

Figure 2: Gazebo visualisation of the indoor environment (left) and the outdoor environment (right) used in our experiments.

3.1.2 SLAM in 3D

The first thing to note is that since we were working in a simulated environment, we had perfect knowledge of drone's true transform relative the world frame. We used that information to localise the robot. For mapping, we produced a 3D occupancy grid using the OctoMap [2] framework. OctoMap uses depth cloud data to produce an occupancy grid of arbitrary resolution, efficiently storing the final result using octree representation.

Intuitively, using lower resolutions for the OctoMap occupancy grid gives a significant performance boost. In our setup, we ended up using the resolution of 15 centimeters per cell (note that this is much lower than the resolution we used in our ME 134 turtlebot labs). Not only did this resolution allowed for quick generation of occupancy grid as the drone is flying around, it also improved the performance of the navigation stack (which is described in section 3.1.3). Figure 3 shows an example of an occupancy grid produced by flying the drone around and scanning surroundings with Kinect.

For reasons explained in section 3.1.3, we decided to use a 2D map for navigation purposes. To save time on generation of the 2D map, we simply project the 3D occupancy grid down to the floor plane. We specify a certain height cutoff for the occupancy grid, below which all occupied cells are ignored. An example of the resultant 2D occupancy grid can be seen on the right panel of figure 3. Specifying a cutoff height threshold lets us incorporate the fact that the drone is hovering into our navigation logic - since our drone is maintaining an altitude of 1m above ground, we can ignore all obstacles that appear below that altitude, even if our occupancy grid contains them.

This approach is pretty natural for the main goal of this part of the project - indoor exploration. In indoor environments, the walls span the entire height of the room and are always perpendicular to the ground which makes them distinct on the projected 2D map, while the cutoff height threshold lets us fly over objects like tables and chairs. In outdoors environment, this approach only works well for flat surfaces or surfaces where the incline is not too steep - our drone can deal with that since it uses local sonar (pointing downwards) data to maintain the hover height. The height cutoff threshold for outdoor environments must be set to be high enough for the drone to explore the area, otherwise the drone won't see any free space after following the incline for long enough.

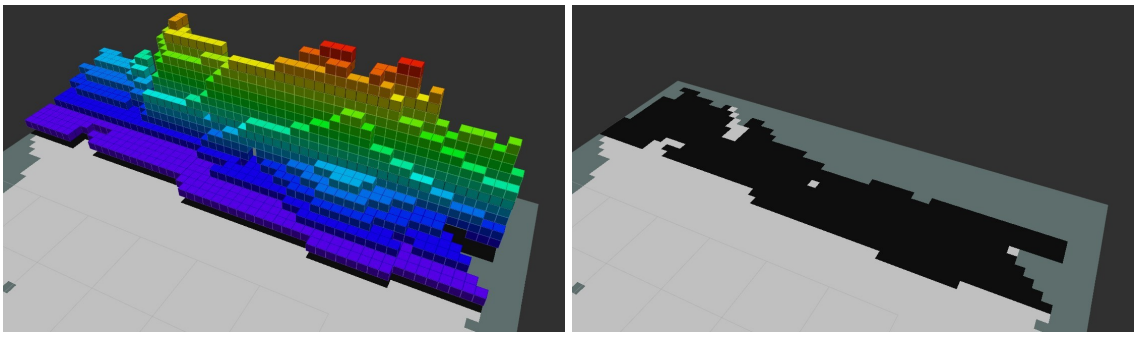

Figure 3: The 3D occupancy grid generated by OctoMap (left) and its 2D projection on the floor (right).

3.1.3 Drone navigation stack

For the most part we used the standard ROS navigation stack, move_base, with the exception of non-standard local planner. Most of the existing, mainstream solutions working with `move_base` were designed for 2D navigation, which imposed a dilemma on us - we had to choose between adapting a 2D navigation stack to our drone, creating `move_base` planners to work with 3D navigation, or using a completely different solution (such as MoveIt, mentioned below). Since our main goal was indoor exploration, we stuck with adapting the 2D navigation stack to our drone.

Aside: MoveIt as an alternative. MoveIt! is a state-of-the-art ROS framework for motion planning in 3D space. It supports complex robots with multiple arms and grippers, which can easily be defined using MoveIt Setup Assistant. MoveIt was designed to support a wide range of robots thanks to a robust, extendable planning API. That said, native support for drones (which are essentially a robot with a single, floating joint) seems to be lacking and requires a lot of additional setup and scripting to work correctly. Despite this initial complexity, once the setup is complete, MoveIt performs extremely well with drones - it plans non-trivial 3D trajectories and ensures obstacle avoidance. After some initial experimentation with MoveIt, we decided that the standard `move_base` navigation stack provides a better trade-off between complexity and useful payoff.

In our repository, the relevant ROS launch file is navigation.launch, and the parameters for different components of `move_base` navigation stack can be found in the param/ directory. One of the most important things to note here is that our drone is hovering at the altitude of 1 metre (more on this in section 3.1.4) - which largely affected how our navigation is configured. For example, in costmap_common_params.yaml, we used a voxel map for our obstacle layer, with the following parameters:

```
# ...
obstacle_layer:
    # ...
    origin_z: 0.8
    z_resolution: 0.4
    z_voxels: 1
    # ...
    observation_sources: scan
    scan:
        data_type: LaserScan
        topic: scan
        marking: true
        clearing: true
        inf_is_valid: true
        min_obstacle_height: 0.8
        max_obstacle_height: 1.2
```

The first 3 parameters essentially define a 2D obstacle layer that starts at 0.8m above ground and ends 1.2m above ground. This matches the hover altitude of our drone precisely, which is essential for the local costmap to be updated correctly. Without these settings, `move_base` will incorrectly interpret the depth cloud data the originates from drone hover level (i.e. the most important data for local planner to avoid collisions).

As seen in the YAML snippet above, we're telling our navigation stack to use the laser scan to detect some obstacles in real time - this is mostly done to make sure that the drone doesn't fly into a wall if it's close to unknown regions of the map. The reader can recall that our drone does not have a LIDAR attached to it - the laser scan comes from converting the depth point cloud into a laser scan using a dedicated node. This conversion migth seem weird at first since we could just use the depth point cloud in the first place, but in reality, this is a necessary step because `move_base` doesn't clear the local costmap correctly when you use a point cloud (see this similar issue to get a context).

Although some other parameters also had to be tweaked to work with drones, the changes are pretty straightforward and should be apparent to anyone setting up a similar system. It is worth noting that we configured our navigation stack to maximise safety without compromising the speed of exploration. This means we increased the tolerance of planners, so they are not required to exactly match the position and bearing of the goal, but rather they just need to get close enough to it. We also increased the danger zone around the walls by adding an `inflation_layer` to both the global and local costmaps - this was necessary to make sure we don't bump into walls while still allowing the drone to move at high speeds.

Now for the local planner: First of all, it's important that appropriate recovery behaviours are specified - in our case they can be found at the bottom of move_base_params.yaml. Sometimes obstacles on the local costmap "linger" around if the planner cannot process (or does not receive) relevant depth cloud information - if this prevents further movement, our recovery behaviour clears the local costmap and allows robot to proceed.

Secondly, tuning local planner parameters to get best performance with a drone can be a challenging task. In our experiments, we tried dwa_local_planner, base_local_planner and teb_local_planner. After some fine-tuning we found that `dwa_local_planner` works better with car-like robots and doesn't guarantee enough safety for drones on high speeds; `base_local_planner` gives a decent speed but has the same issue with safety - avoiding walls required us to bring down the speed; finally, `teb_local_planner` gave a very good trade-off between safety and speed, so we decided to stick with in the end. The relevant configuration for the planner can be found in teb_local_planner_params.yaml.

The main thing to note with local planners for our drone was that, most of the time, we had to disallow any movement backwards and with some planners movement sideways. It is obvious that drones can travel in any direction, so, potentially, these types of movement could very easily be carried out by the drone. That said, the Kinect camera on our drone was facing forward, which meant that local planner would get no information from the local costmap if an obstacle would suddenly appear behind the drone or on its side (i.e. when entering an unknown environment). To make sure we don't bump into walls, we had to constrain the drone to only move in the direction in which Kinect is pointing.

Thanks to the way our ROS setup is laid out, it is possible to test the navigation stack separately from the exploration, relevant instructions can be found in the repository for our project.

3.1.4 Exploration script

The source code for the exploration logic can be found in scripts/exploration/ directory, and the relevant script to start exploration is start_exploration.py. Those familiar with Caltech's me134_explorer ROS node will recognise some of the concepts in our script, since our solution is roughly based on ideas from `me134_explorer`. Otherwise, the paragraph below provides a high-level overview of how the script works.

High-level overview. Our exploration logic is based on a control loop that runs at the rate of 10 Hz. It operates using states - the initial state is `Takeoff`, from which the drone moves into `FullRadialScan` to analyse its immediate surroundings and record them in a 2D/3D occupancy grid. After that, the robot moves into `FindGoal` state during which no movement occurs, but the script analyses the occupancy grids generated so far and picks a goal for the robot. Next state is `TravelToGoal`, during which the drone follows commands

from the navigation stack (if any). Once the robot reaches the goal and does a full radial scan again. At this point, if no other goals can be found, the drone completes exploration and enters `Land` state.

The basic idea for our exploration script is that we're trying to fully explore the 2D projection of the 3D occupancy grid generated by OctoMap. In doing this, the drone will fly around using Kinect camera to scan not only the floor, but also the walls (and sometimes ceilings), all of which is picked up by `octomap_server` and incoroporated into the 3D occupancy grid. As a result, even though our exploration logic uses a mere 2D projection of the actual 3D occupancy grid, the end result is still a decent 3D model of the environment.

We also try to make intelligent decisions about what goal we want to travel to next, blacklisting some goals if necessary to avoid infinite loops. The main philosophy is that our script should be able to deal with all runtime errors and issues automatically, and terminate only when it is sure that no additional useful exploration can be carried out.

Maintaining hover altitude. Our script maintains a constant hover altitude of 1m by centralising the control of the velocity commands (i.e. `Twist` messages published to `/cmd_vel` topic, which the drone consumes). Note that **all** velocity commands are controlled by our exploration logic - even velocity messages published by the navigation stack go through pre-processing in our script before being sent to the drone.

To achieve this, we expose a method called `request_velocity()` to all components of the exploration script. Using this method, a component can specify a particular linear or angular velocity that it needs to be carried out in the end of the current iteration of the control loop. Finally, in the end of the iteration, movement handler (destructively) combines all of the requested velocities into a single `Twist` message, adds an appropriate Z-linear velocity needed to maintain altitude to said message, and finally publishes it to `/cmd_vel`. Since in `Takeoff` and `Land` our logic provides custom Z-linear velocities, movement handler doesn't maintain hover altitude when these states are active.

Full radial scan. This operation is pretty self-explanatory, the drone just spins around to make sure that Kinect camera can scan all of its immediate surroundings. `octomap_server` uses this data to produce a 3D occupancy grid in real time, which is also projected to floor level to produce a 2D occupancy grid. It is very important to choose the right angular velocity - if the rotation is too fast, OctoMap will not be able to keep up, leaving wide gaps in the occupancy grid. We found that angular velocity of 0.4 radians per second is as fast as you can go while still generating a decent occupancy grid. As it stands, the full radial scan is somewhat inefficient - it often scans areas in which there is no uncertainty (e.g. all cells in 180 degrees sector behind the drone are known to be free, but radial scan will cover them anyway).

Finding a suitable goal. Our logic for finding a suitable goal incorporates several optimizations for speed of exploration, drone safety and computational efficiency. All of the relevant code can be seen in planner.py (this is only for the exploration script - not to be confused with `move_base` planners). Note that we're using a fairly low-resolution 2D occupancy grid in our exploration script, which makes certain grid-search algorithms pretty efficient.

We start the goal finding process by identifying exploration frontiers. Note that we're actually using the global costmap to find frontiers, not the occupancy grid published to the `/map` topic. The advantage of using global costmap is that we can make use of the `inflation_layer` we specified earlier - this inflation layer adds a danger zone around all-known obstacles, which makes sure we don't pick a frontier that is directly next to a (known) wall.

We define a frontier as a cell in the costmap that has at least three free neighbours and at least three unknown neighbours (diagonal neighbours are also counted). All of the other cells should either be free or unknown, we do not allow any non-zero cells from the costmap as neighbours because this might result in us picking a goal that is too close to an obstacle.

At this point, we have a list of potential frontiers. Next, we filter out all undesirable frontiers using several heuristics. First of all, we remove all frontiers that are in blacklisted regions (more on them below) - this helps us avoid infinite loops where we could pick a goal, navigation stack would reject it, and then we would pick it again. After that we make sure there is at least a 1 meter gap between all potential frontiers, which is achieved by dropping frontiers that are too close to each other. This is done mostly for computational efficiency in the next step.

Now we have a list of sparse potential frontiers, all of which are not in blacklisted regions. We calculate the expected information gain (EIG) for each potential frontier. Since we always perform a full radial scan once we get to a goal, to calculate EIG we consider the occupancy values of cells around the goal in a radius approximately equal to the range of OctoMap processing, which is effectively around 3 meters.

Finally, we pick the frontier with the best EIG and try to find a safe spot to observe the frontier from. This is done by considering 8 cells in a square of side-length 3 (cells) around the frontier, then rating each of the 8 cells by safety. Finally, we pick the cell with highest safety rating and set it as our next goal. If the frontier we chose cannot be observed from any safe spots, we simply move to the frontier with the second best EIG and so on.

Note that the goal finding logic can exhaust all of its option and return `None` as the goal. At this point, the exploration script assumes we cannot perform any more useful exploration, so the drone lands and the exploration script terminates.

Travelling to goal. The actual velocity commands for travelling to the goal are pretty straightforward. Once the drone publishes a goal to the navigation stack and moves into the `TravelToGoal` state, our script begins listening to velocity commands published to the `/planned_cmd_vel` topic. Each velocity command, i.e. `Twist` message, is pre-processed according to the logic described earlier and published to `/cmd_vel`. As mentioned earlier, this part is pretty straightforward.

The main feature of our logic when it comes to travelling to a goal is using the full capability of ROS actionlib. `actionlib` is a framework that allows different ROS nodes to issue and carry out complex actions. Most importantly, it is possible to query the state of actions issued via `actionlib` - this means that while our drone is travelling to a goal, we can continuously query the navigation stack to get the state of our goal. When the navigation stack cannot plan a trajectory to the goal (e.g. there is no safe path due to obstacles) it will abort the goal we specified. Our exploration logic notes when a goal was aborted, and adds the goal to an array of blacklisted goals. As a result, all potential frontiers that are in 0.6m radius of the blacklisted goal will get ignored in future exploration.

Of course, just blacklisting a region like that is a bold move which can sometimes hurt the efficiency (and completeness) of exploration. We assume that the configuration that we used for the navigation stack only ever aborts goals that are truly unreachable.

3.1.5 End result and demonstration

See section 4.

3.2 Multi-agent multi-goal task solving in cooperative swarm setting

3.2.1 Background and Simulation Environment

The environment chosen for this subproject is the Halite II game environment. The Halite II game environment features coorperative-competitive gameplay between 2-4 AI controllers, each controlling multiple agents (game units).

The game is a turn based strategy game with a finite horizon of 300 timesteps per game (or episode). It is an extremely simplified version of a game like Starcraft, where additional units can be built to battle the opponent. An overview of the game rules can be found here and detailed documentation is available here. Like Starcraft, it shares many common challenges, such as a varying number of units as the game progresses, limited time for computation between actions, and a continuous state and action space. The Halite II game environment allows us to tackle a some complex challenges involving control and constrained optimization, while still being simple enough for us to make good progress in a short period of 4 weeks.

This section of the project is done in conjunction with CS 159 [3], where we explore solutions from an machine learning perspective, and in ME 134, from a control perspective. We find that a hybrid solution consisting both a learned and model-based controller worked best. The rationale driving us to take a learning approach is that finding the best strategy in such a game is a very hard task, since the problem can be framed as a constrained combinatorial optimization problem, which is NP hard as shown in [4]. Machine learning

techniques such as reinforcement learning can be used to learn a policy that performs optimally in a subset of the state space (that we care the most about).

3.2.2 Markov Decision Processes

The problem can be formalized as a one-player game against an opponent with an unknown (and possibly non-stationary) strategy profile. The game can thus be formulated as a discounted MDP represented by the tuple $(S, A(s), p, r, \gamma)$, where S is the state space, $A(s)$ is the set of actions that the player can take in state s, an action $a \in A(s)$ is vector of commands for each player controlled ship in state s. The state transition probability $p(s'|s,a)$ is an unknown probability distribution conditional on the opponent strategy profile. The reward $r(s,a)$ is the change in the difference between the player and the opponent's score, and γ is the discount factor.

3.2.3 Value Functions

We've discussed in class the notion of a state-value function in the recursive form of:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[r + \gamma V^{\pi}(s')]$$

In words, the value of a given state s under a policy π is the expectation of the discounted reward over actions drawn from the policy distribution π conditioned on the state s. P is the state transition probability, which accounts for the stochastic outcome of performing action a in state s.

We've also derived in class the optimal state-value function

$$V^{*}(s) = \max_{\pi} V^{\pi}(s)$$

which is the maximum value function over all policies. There are various ways of finding the optimal value function, such as value iteration which was also covered in class. Given the optimal value function V^{*}, we can derive the optimal policy

$$\pi^{*}(s) = \arg\max_{a}[r(s,a) + \gamma P(s'|s,a)V^{*}(s')]$$

However, the solution involving state-value function involves knowing P, the state transition probabilities. The problem is defined as a game against an arbitrary opponent, and thus the opponent's strategy is an unknown distribution, and thus do not know the dynamics of the MDP, P.

An alternative to the state-value function is the action-value function, defined as

$$Q^{\pi}(s,a) = \sum_{a} \pi(a|s)[r + \gamma Q^{\pi}(s',a)]$$

Note that the action-value function is no longer dependent on P, the state transition probability, which means that we can find the optimal policy π^{*} without knowing the dynamics of the MDP, also known as *model-free* control.

$$Q^{*}(s) = \max_{\pi} Q^{\pi}(s)$$

$$\pi^{*}(s) = \arg\max_{a}[r(s,a) + \gamma Q^{*}(s')]$$

There are various methods for finding Q^{*}, such as Monte Carlo policy iteration [5], SARSA [6], and Q-learning [7]. For the purposes of this project, we'll focus on Q-learning

3.2.4 Q-learning & DQN

Q-learning [7] is an off-policy algorithm for learning of action values $Q(s,a)$. It's an off-policy algorithm because the behavior policy, which is the policy used during exploration, is differnt from the target policy.

The target policy is greedy w.r.t Q and the behavior policy is sampled ϵ-greedy w.r.t Q. That is,

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon, & \text{if } a = \arg\max_{a \in A} Q(s,a) \\ \epsilon/m, & \text{otherwise} \end{cases}$$

The Q-learning update rule is given below, where α is the learning rate. The portion highlighted is the *Q-learning target*.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a') - Q(S_t, A_t))$$

The Q-learning algorithm is shown below. It is shown in [8] that Q-learning converges to the optimal action-value function.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

Up till now, it's been assumed that our environments are discrete, that is V or Q is simply a lookup table that maps a finite domain to it's image. This is not practical in the case where the state space is large, or in our case, continuous. The issue with the naive representation using a lookup table is that it requires memory proportional to the size of the state space, and is unable to generalize to unseen subsets of the problem domain from seen experiences.

Deep-Q-Learning [9] proposes a solution to the above problem, by introducing the idea of a *value function approximator* $\hat{Q}$ parameterized by θ, such that:

$$\hat{Q}(s,a;\theta) \approx Q^{\pi}(s,a)$$

Neural networks are chosen as the class of function approximators for $\hat{Q}$ in DQN as it is differentiable which allows for iterative updates such as stochastic gradient descent, while being able to model complex non-linearities of the target function.

We want to approximate target $Q_{\pi}(s,a)$ using $Q(s,a;\theta)$. The loss function is given by the mean squared error

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(Q_{\pi}(s,a) - Q(s,a;\theta))^2\right]$$

We can derive the average gradient update for weights θ by differentiating the loss function w.r.t θ.

$$\Delta\theta = \alpha \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(Q_{\pi}(s,a) - Q(s,a;\theta))\nabla_{\theta} Q(s,a;\theta)\right]$$

Alternatively, we can perform stochastic gradient updates by sampling from the average gradient

$$\Delta\theta = \alpha\,(Q_{\pi}(s,a) - Q(s,a;\theta))\nabla_{\theta} Q(s,a;\theta)$$

In the case of Q-learning, we replace the oracle $Q_{\pi}(s,a)$ with TD target:

$$\Delta\theta = \alpha\left(r + \gamma \max_{a'} Q(s',a',\theta) - Q(s,a;\theta)\right)\nabla_{\theta} Q(s,a;\theta)$$

We now have an iterative update rule for the parameters of our function approximator. However, there are still some issues which could lead to instabilities during updates. For one, since we are updating our parameters by sampling outcomes from the environment, sequential states sampled are highly correlated with each other which means that naive SGD is sample inefficient and prone to hitting local minimas. In

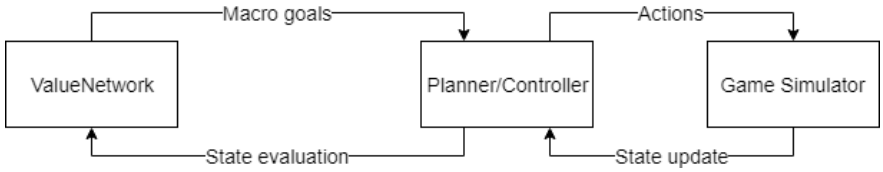

Figure 4: Graphical representation of control flow between the value network, controller, and game simulator

addition, the policy is prone to oscillations due to the fact that small changes in θ may result in rapid changes of the policy.

The solution to the first issue is to use an experience replay buffer, which stores previously seen states, actions and rewards. By randomly sampling from the buffer during gradient updates, we are able to decorrelate the data while also reusing past experiences which is much more sample efficient. For our purposes, we use a fixed size cyclical replay buffer to store a history of past experiences.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

3.2.5 Goal Planning

We adopted a hierarchical approach to planning, with long-term (low frequency) goals known as **strategies** and short-term (high frequency) goals known as **commands**.

Via experimentation, we've discovered that a completely model-free approach was not a good solution, as the high frequency / low level goals are difficult to learn. Examples of such low level actions include pathfinding and collision avoidance, which there exist algorithmically efficient solutions for. Instead, we'll focus the learning objective on learning the long-term goals / strategies for each agent, which is a combinatorial optimization problem and hard to solve by brute force.

The strategies for each agent is given by a value network (Q-network) trained using Double-DQN [10], which is just a 1 line of code change from the original DQN algorithm. The weights of the network is shared amongst all agents. We implement a low level controller to convert the high-level / long-term strategies from the value network into actions which are taken by the game simulator. Figure 4 shows the control flow between the value network, low-level controller, and the game environment.

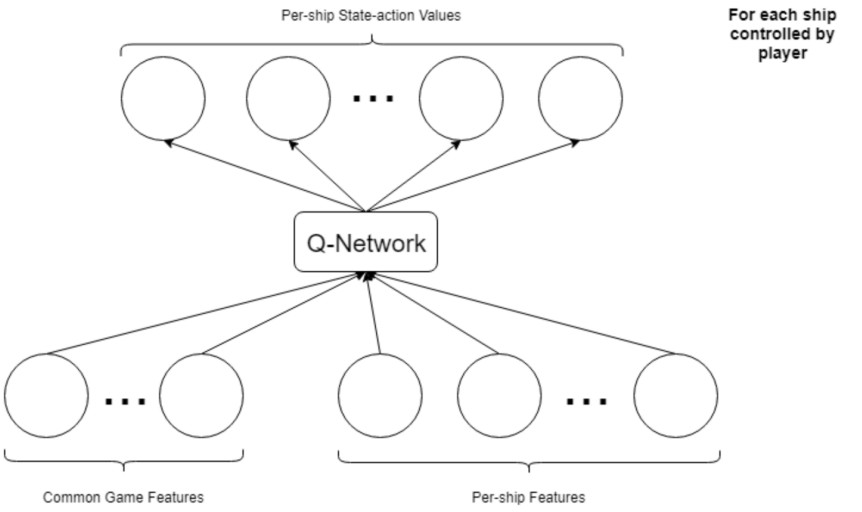

Figure 5: Graphical representation of DQN implementation. At each timestep, a set of common game features are extracted. For each ship controlled by the player, a set of per-ship features are extracted and concatenated with the set of common game features to form a feature vector which serves as the input to the DQN. The output of the DQN is the softmax normalized values of each strategy in the discretized action space. Network weights and parameters are shared amongst all ships.

3.2.6 Low-level planner / controller

Artificial potential fields One approach to global pathfinding and collision avoidance is the use of artificial potential fields [11, 12]. Attractive potentials represent targets or goals, and repulsive potentials represent obstacles. The potential U at a point q is the sum of attractive and repulsive potentials:

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

Motion planning is done by minimizing the potential energy of the agent (i.e. moving to a level set of lower potential). This is accomplished by following the negative gradient of the potential function:

$$\nabla U(q) = DU(q)^T = \left[\frac{\partial U}{\partial q_1}(q), ..., \frac{\partial U}{\partial q_m}(q) \right]^T$$

There are several issues with this approach. Firstly, our environment is non static (in fact, most of our obstacles are non static), and that necessitates re-computation of the potential function for each agent at every timestep.

Secondly, the velocity is dependent on the gradient of the potential field, which means that movement might be sub-optimal near to the goal where the gradient is shallow. Hacky (domain specific) fixes for this include using different potentials e.g. quadratic vs conic, or forcing higher velocities.

We assign arbitrary priorities to robots for multi-agent path planning.

Velocity obstacles Another approach to motion planning and collision avoidance is using Velocity Obstacles (VO) [13]. At each timestep, the position and velocities of obstacles (i.e. other ships, planets) is extrapolated from the current state and velocity obstacles are computed for each unit under our control. The velocity obstacle of a unit represents the set of velocities that will result in a collision with some other entity at a future timestep.

Collision avoidance is done by selecting velocities outside of the velocity obstacle of each unit. By computing velocity obstacles at each timestep, we can perform motion planning in the dynamic environment.

Reciprocal velocity obstacles Reciprocal Velocity Obstacle (RVO) [14] is an extension to VOs where path planning and collision avoidance can be performed in a multi-agent setting **without any communication** between agents. The requirement is that all agents follow the same logic for collision avoidance, hence "Reciprocal" in the name. Hybrid Reciprocal Velocity Obstacle [15] (HRVO) introduces asymetry to VOs of individual agents to assign a bias for choosing a side to pass on to avoid "reciprocal dance" where agents cannot decide which side to pass. This was an optimistic stretch goal for us, and in the end there was insufficient time to implement either RVO or HRVO. This can be an interesting area for future work.

4 Deliverables / Demonstrations

We have completed all goals identified in Section 2, as well as some of the stretch goals.

4.1 Autonomous exploration and 3D SLAM using drones

The code, launch files and installation instructions can be found at https://github.com/TimboKZ/caltech_samaritan. We've made two iterations of demonstrations for autonomous exploration - first iteration used a somewhat sub-optimal configuration for the local planner and a very naive goal-finding heuristic (essentially random frontier, not described in this report). The links to the videos of the first iteration can be found below under the label "Naive exploration". The resultant maps for naive exploration can be seen on figure 6.

The EIG-based algorithm described in this report was a part of the second iterations. The videos of the EIG exploration algorithm in action can be seen below under the "EIG-vased exploration" label. The final 3D occupancy grid can be seen on figure 7.

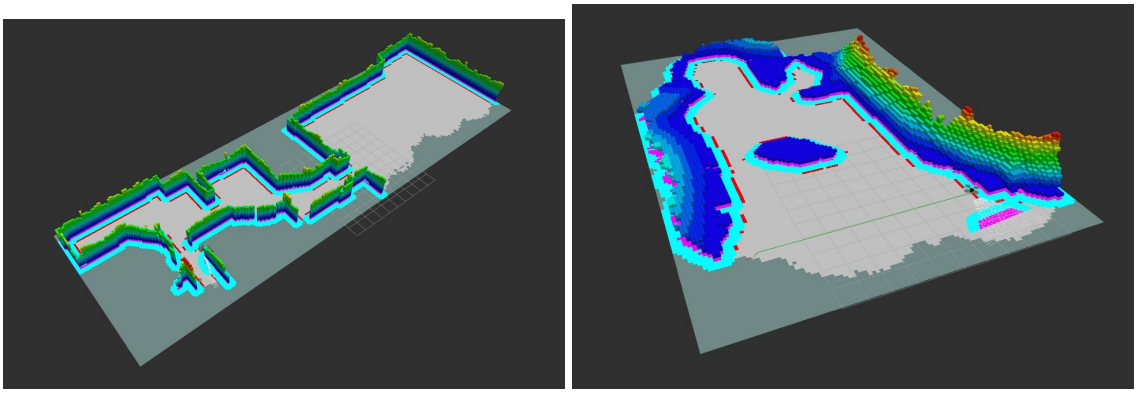

Figure 6: The final 3D occupancy grid produced by the naive exploration algorithm on termination (or crash), in an indoor environment (left) and outdoor environment (right).

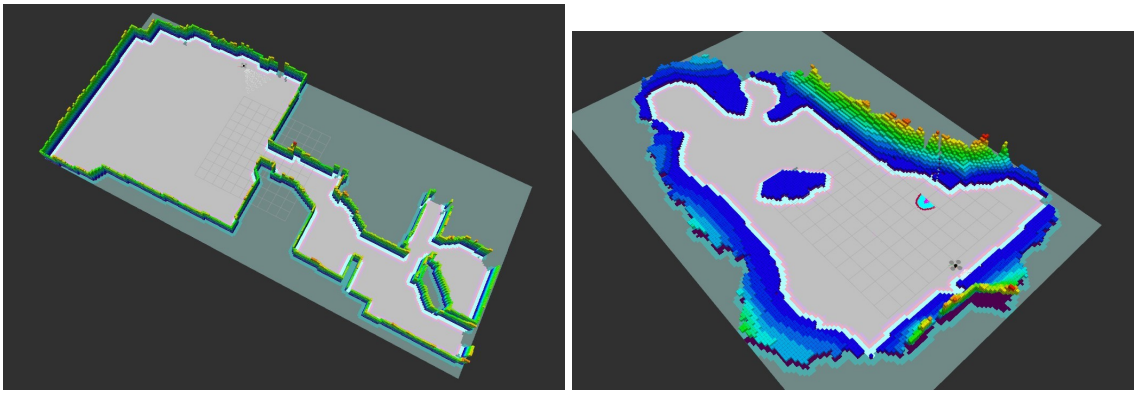

Figure 7: The final 3D occupancy grid produced by the EIG-based exploration algorithm on termination, in an indoor environment (left) and outdoor environment (right). Note that these maps are much more complete then those generated by naive exploration.

(Naive exploration) Indoor autonomous exploration https://www.youtube.com/watch?v=jkk4TcFPYxM

(Naive exploration) Outdoor autonomous exploration https://www.youtube.com/watch?v=Co9L9Dkdvdc

(EIG exploration) Indoor autonomous exploration https://www.youtube.com/watch?v=5fQLEVJwjwE

(EIG exploration) Outdoor autonomous exploration https://www.youtube.com/watch?v=hfpO-X9Q2aM

4.2 Multi-agent multi-goal task solving in cooperative swarm setting

GitHub repository: https://github.com/veniversum/HaliteII/

Here are video recordings of various replays in the Halite II game environment.

Hybrid controller vs naive opponent https://www.youtube.com/watch?v=2MIgXMCEObI

Hybrid controller vs itself https://www.youtube.com/watch?v=CnDhyspBNKY

Hybrid controller vs skilled human opponent https://www.youtube.com/watch?v=KUDI5xNF3uM

5 Discussion / Debriefing

Overall, we're reasonably happy with the performance of the EIG-based exploration algorithm, but motion planning remains an issue. The global planner in our ROS navigation stack did a good job, but the local planner didn't always produce a safe path. A good vector for future work would be experimenting with different local planners to increase safety and average movement speed. As mentioned in section 3.1.3, we decided to sacrifice the ability to plan complex trajectories in favour of a more straightforward navigation setup. Using MoveIt navigation framework would make these complex 3D trajectories possible, and, given more time, it would be good to adapt MoveIt for our needs.

Since we were working in a simulated environment, accurate localization was a non-issue, which is of course not the case with real drones. A vector for future work would involve adding an artificial source of uncertainty to our setup and trying out different localization solution. Note that adding this feature requires minimum changes to our existing setup - the only difference is the transform that `octomap_server` will use, which will now come from the localization tool and not Gazebo. Since our robot constantly hovers at 1m altitude using the on-board sonar for measurements, 2D localization solution will work just as well with our setup, no need for dedicated 3D localization tools.

As a final remark for the exploration script - right now, most of the logic is carried out in the main thread. This poses an issue when the code "hangs" on one of the external API calls - for example, when communicating with `move_base`'s action server. Since the main thread is blocked, the script isn't publishing

velocity commands for the drone in real-time which can result in undefined behaviour. Given more time, we could rewrite the whole exploration framework to use a dedicated thread for each component.

For the multi-agent planning task, initial work on model-free approaches was performed in [3] as part of CS 159. The model-free approaches were able to learn the basic controls and dynamics of the environment, but were unable to act with purpose. In our experiments, the model-free approaches learned to produce more units, but did not learn to identify and perform other tasks such as attacking or defending against enemies. The challenge was that 'tasks' were implicit, and behavior was quite heavily influenced by the reward function chosen. One approach that could be taken in future work is to explore reward shaping to incentivize learning of diverse behaviors.

We identified early on that splitting the planning task into levels of hierarchy based on macro and micro goals would combine the best of both worlds, the flexibility of reinforcement learning algorithms and the robustness and speed of model based approaches. Our approach of a hybrid controller performed very well in practice, and is capable of producing complex strategies that works well against multiple opponents.

As mentioned earlier, we could explore more sophisticated algorithms for motion planning (such as HRVO) if given more time. From a learning perspective, we've included a list of interesting areas for future work in Section 7 of [3]

References

[1] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar von Stryk. Comprehensive simulation of quadrotor uavs using ros and gazebo. In *3rd Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, page to appear, 2012.

[2] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013. Software available at http://octomap.github.com.

[3] Qingzhuo Aw Young, Navid Azizan, and Ola Kalisz. Multi-agent imitation & reinforcement learning in a turn-based strategy game. *CS 159 Caltech*, 2018. Available at https://goo.gl/qcB22u.

[4] John Canny and John Reif. New lower bound techniques for robot motion planning problems. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 49–60. IEEE, 1987.

[5] Gerald Tesauro and Gregory R Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems*, pages 1068–1074, 1997.

[6] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering, 1994.

[7] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[8] Francisco S Melo. Convergence of q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep*, pages 1–4, 2001.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[10] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

[11] Charles W Warren. Global path planning using artificial potential fields. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 316–321. IEEE, 1989.

[12] Charles W Warren. Multiple robot path coordination using artificial potential fields. In *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pages 500–505. IEEE, 1990.

[13] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772, 1998.

[14] Jur Van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1928–1935. IEEE, 2008.

[15] Jamie Snape, Jur Van Den Berg, Stephen J Guy, and Dinesh Manocha. The hybrid reciprocal velocity obstacle. *IEEE Transactions on Robotics*, 27(4):696–706, 2011.