Final project report: Low-power Reinforcement learning on Visual Confined-Space Navigation for Insect-scale Robots

Pengcheng Chen
Department of Mechanical Engineering
pengcc@uw.edu

1 Introduction

1.1 Micromechanical Flying Insects (MFIs)

Micromechanical Flying insects (MFIs) also known as the "insect robot", are inspired by the mechanism of flapping wings of insects. Generally, a typical modern MFIs' main body structure is composed of two or more insect-inspired flapping wings, a piezoelectric ceramic actuator as the engine that drives the flapping wings, and a set of highly integrated micro-electro-mechanical system (MEMS) that provides the energy, sensor, and controller for the robot.(Figure 1) In 2008, RJ Wood fabricated the first modern MFIs in Harvard[7]. Its diminutive size and agility allow it to enter environments that regular unmanned aerial vehicles (UAVs) cannot perform.

However, the tiny and delicate structure brings a new challenge to further development. Most of the automatic navigation algorithms for UAVs cannot directly transfer into the MFIs because the processing unit inside the MFIs is too small to have a similar performance as regular UAVs, thus, for most tasks of insect robots, we need to develop algorithms with low computational power.

Figure 1: Profile of a typical micromechanical flying insect[3]

1.2 Tasks and Advancement

Corridor following and obstacle avoidance are two primary tasks that are widely applied in autonomous UAVs, while they the still absent in MFIs research, our project is attempting to fill in the blanks and provide a basic autonomous cruising ability to the MFIs. Note that in our course project, we only focus on the corridor following task due to time constraints.

Our task is to navigate the MFIs through a narrow corridor given only visual input. Visual navigation for MFIs is very challenging because in such a small scale, the size, weight, and power (SWaP) constraints do not appear to permit visual navigation techniques such as SLAM (Simultaneous Localization and Mapping). After all, they are likely to be too power-hungry.

In our project, we are going to try to use reinforcement learning algorithms to train a policy that needs low computational power to make the inference. Note that we don't care about the computational power for the training process because the training is offline, but once the training is done, we will have a policy for the robot to do the navigation, and we need this policy function to be simple.

And our work will be purely on simulation given the time constraint. We will use the multi-rotor quadcopter as our robot because we cannot find the MFIs developed in the simulation environments we selected so far, but our algorithm will aim for low computational power. There are two aspects of reducing power in our robotics task:

1. **Sensing.** It might be power-consuming for a robot to sense the environment. From the sensor perspective, laser range finder is too power-consuming. Global positioning system (GPS) is too heavy or power-consuming and it is not accurate in the indoor environments. Thus, to realize the low-power objective, we use a tiny omnidirectional camera which is inspired by the Flies' omnidirectional eye.

2. **Control.** Our problem becomes a visual flight control problem. To reduce the power further, instead of using the raw images as the inputs to the RL, we propose to use the **sparse** optic flow information as the inputs to the RL, which has a lower dimension than the raw image. By using the sparse optic flow, we can develop a simpler policy than using the raw images (will be many layers of neural network), then we achieve our low-power goal.

Note that we will develop this whole lower-power system aiming for a publication at a top conference in the near future. We will continue working on this project after this course. For this course project, due to the tight time constraint, we developed the following:

1. Build a 3D graphical simulation environment from scratch using UE4.

2. Figure out how to use Airsim to control the drone and how to get the optic flow information from the pure camera inputs.

3. Develop the components of the reinforcement learning algorithm such as action sets, the reward, the policy gradient method, etc.

4. Setup the overall framework of training the Reinforcement learning algorithm of Proximal policy gradient (PPO) using Airsim [6] and Stable-baseline3.

5. Finish training and evaluate the training process and the optimal policy.

For sparse optic flow extraction and the simpler policy in the actor model part, it is a bit time-consuming to tune parameters and try different implementations, we will continue after this class due to the time constraint.

2 Problem statement and formulation

In our project, we do corridor following along a non-straight corridor (Figure. (6)) using pure camera input.

We add rich textures on the walls such that the robot can perceive information to calculate the optic flow. Our robot has two cameras, one for each side to mimic the omnidirectional camera. We fix the altitude and only do navigation on x-y plane because the altitude is controlled by other controller which is beyond the scope of the project.

In the training process, the robot will be equipped with two cameras, and the sensor to know its ground truth positions (x, y, ϕ) to calculate the reward. And for the test process, the robot will be only equipped with two cameras because the sensor to measure positions is too heavy and too power-hungry for insect-scale robots.

The overall formulation is shown in Figure 2.

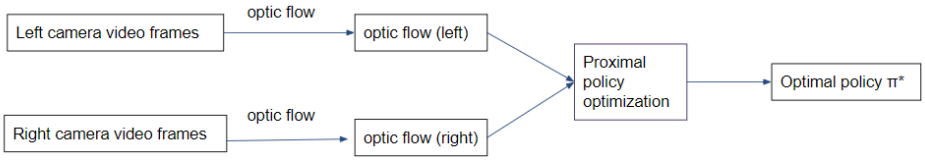


Figure 2: The overall formulation

Note that the optimal policy is a CNN policy for now. After the class, we will simplify the policy further. And the optic flow is the dense optic flow map for now, which needs to be sparse for publication later. For more details, please refer to section 4 : Methodology, such as the states, actions, observations, etc.

3 Platform

AirSim[6] (Aerial Informatics and Robotics Simulation) is an open-source, cross-platform simulator for multi-rotors, ground vehicles such as cars and various other objects, built on unreal engine 4 (UE4). We will use the AirSim plugin and UE4 to demonstrate the path planning of the multi-rotor.

OpenAI Gym[2] is an open-source python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments.

In this project, we will implement Reinforcement Learning algorithms in AirSim (which runs on UE) using the OpenAI gym wrapper around the AirSim API. The environment setup is developed in Unreal Engine. AirSim provides the multi-rotor and the API to control it and also to record images. In order to use AirSim as a gym environment, we extend and implement the base method such as `step`, `_get_obs`, `_compute_reward` and `reset` specific to AirSim and the problem statement.

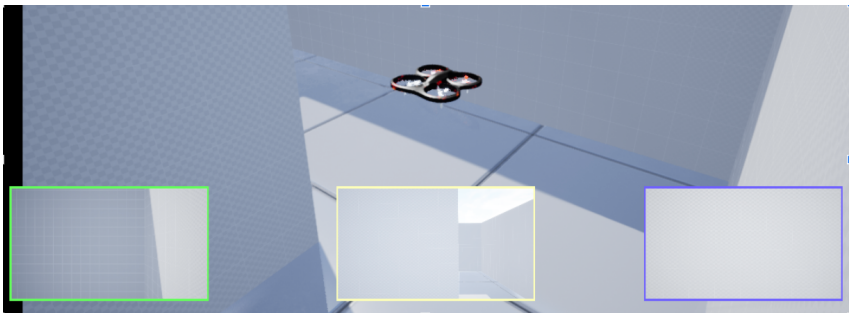


Figure 3: Drone with Front Camera View, Left Camera View and Right Camera(left->right)

3.1 Software Architecture

The software architecture is shown in Figure4, each path finding environment contains 2 actors (multi-rotorActor, TargetActor), they are used to detect the collision and whether the multi-rotor hits the target. The multi-rotor movement module contains 2 classes which are relevant to the UE actors.

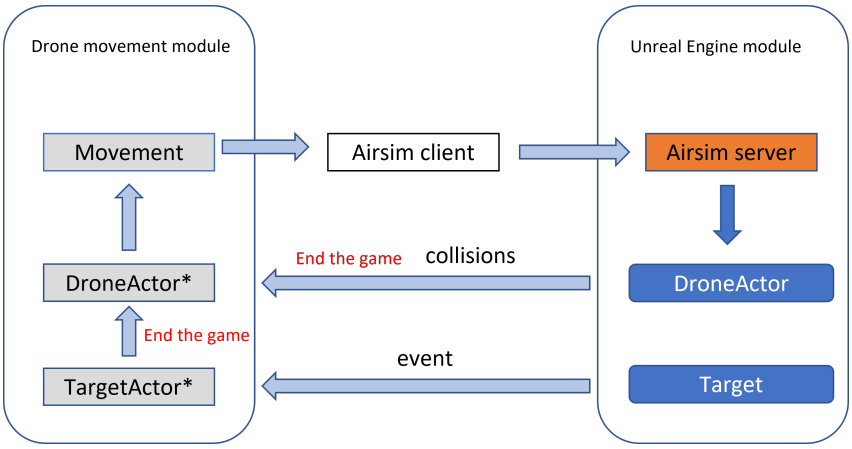


Figure 4: Software architecture

4 Methodology

4.1 Reinforcement Learning Algorithm

In this project, we train the multi-rotor using the **PPO (Proximal Policy Optimizations Algorithm)**. PPO [5] is a state of art reinforcement learning algorithm which belongs to the family of policy gradient methods. It can learn simple policies efficiently, which is aligned with the goal of our project. We will skip the details about the PPO because it is in paper [5] and we have a tight page limit. Here, we will focus on the analysis of why PPO will help solve the problem, which we think is worth knowing:

1. PPO is a policy gradient method and as we know, policy gradient method can handle the partially observable reinforcement learning problem. In fully observable RL problem, we have

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}))(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}),\mathbf{a}_{i,t})) \tag{1}$$

   ,which does not require the Markov assumption. Then for the partially observable RL problem (our problem), we have

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{o}_{i,t}))(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}),\mathbf{a}_{i,t})) \tag{2}$$

   Similarly in the Clipped Surrogate Objective in PPO, we can just replace $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ with $r_t(\theta) = \frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{old}}(a_t|o_t)}$. Thus, you can see, the formulation is same, except that we only need to modify the inputs to the Actor model $\pi_\theta(\mathbf{a}_{i,t}|\mathbf{o}_{i,t})$ to solve the partially observable RL problem.
2. Actor-critic style. As we know from above, the actor is the policy that proposes a set of possible actions given an observation. In PPO, the critic is the estimated advantage function. Thus, it will be easy for future implementation of the simpler policy by modifying the actor model.
3. Sample efficiency. PPO is sample efficient because it uses the surrogate objective to avoid the new policy changing too far from the old policy. In robotics, sometimes it is hard to collect samples, thus the sample efficiency of the RL algorithm is important.

4.2 State, Action and Observation

An agent's goal is to learn the best possible way to successfully complete the task at hand, which in our case, is to navigate through a corridor without colliding with the walls. This is achieved by taking

action according to a policy. After the action has been executed, the agent observes the state of the environment and the reward acquired. These feedback are used to update the policy to improve the future expected cumulative rewards.

For our project, we take the optic flow map from the left and the right camera as our observation (o_t). These images are converted to single channel image and then stacked column wise to create one image which is then fed to our PPO network as an observation. The multi-rotor's position and collision information are the state (s_t). We have discretized the action space, i.e., the multi-rotor can move in 3 modes, first: moving in forward direction with respect to multi-rotor's NED frame, second: rotate clockwise while moving forward and lastly, rotate anti-clockwise while moving forward. We want the multi-rotor to turn while moving forward because it results in a better optic flow map as the yaw rate alone is not enough to get a good optical flow map.

Note that our problem is actually a partially observable reinforcement learning shown in Figure. (5).

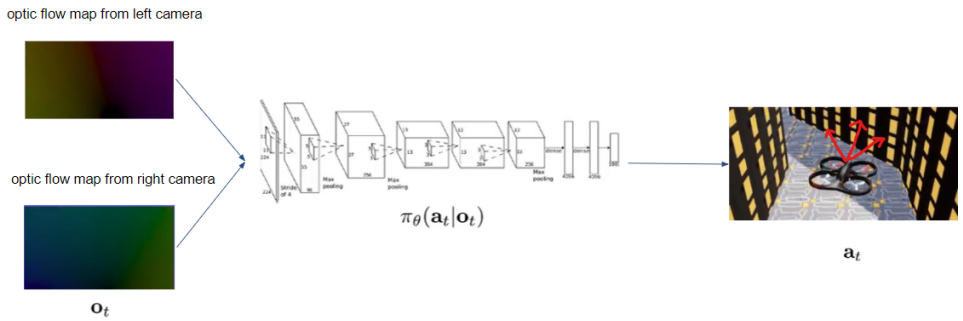


Figure 5: Illustration of partially observable reinforcement learning.

The policy is mapping the observations to the action. The states are used to calculate the reward rather than the policy.

4.3 Reward

We reward the agent in the following scenarios:

- **Distance from Center line**: The goal here is that the multi-rotor should reach the target location while avoiding collision with the walls and by following the center line, we can definitely avoid collision with the wall, but it is not always possible to track. For example, while going around the corner there is a high possibility the multi-rotor might drift and can hit the wall. Thus we reward the multi-rotor if it is within a threshold from the center line
- **Distance from Target Location**: The reward increases as the multi-rotor gets closer to the target location. This is done so that the drone won't try to circle about some point and will move closer to the target

$$\boxed{reward = \frac{1000}{e^d}}$$

- **Collision**: End the game if the multi-rotor collides with the wall with 0 reward

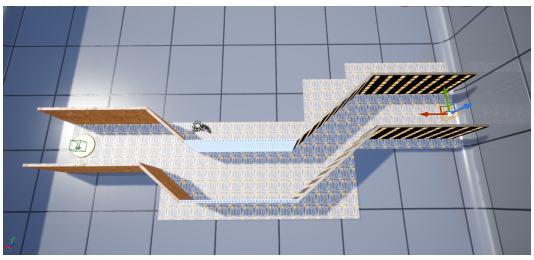
5 Simulation

5.1 3D simulation environment

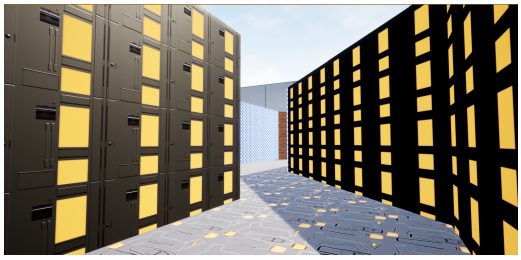
In this project, two different corridors were established by UE4 to act as work environments. The first is a narrower corridor with angled corners, as illustrated in figure6a and figure6b; we refer to this as corridor$_1$. In this simpler case, the re-spawn point is positioned on the rightmost of the corridor, while the objective is located at the golden panel on the leftmost. The multi-rotor must go along this corridor from right to left in order to achieve the goal.

Our second task for this undertaking is shown in figure6c; we refer to it as corridor$_2$; its structure is the same but wider than corridor$_1$. A wider environment enhances the difficulty of algorithm convergence since a wider corridor provides the multi-rotor with more space to explore, making it more difficult to discover an optimum policy in a given amount of time. Same as the corridor$_1$, the re-spawn point is positioned on the rightmost, whereas the finish point is on the left end. Figure6d depicts the inside of this corridor; compared to corridor$_1$, The broader construction of corridor$_2$ enables the multi-rotor to reach its destination with little or no direction change at the bend.

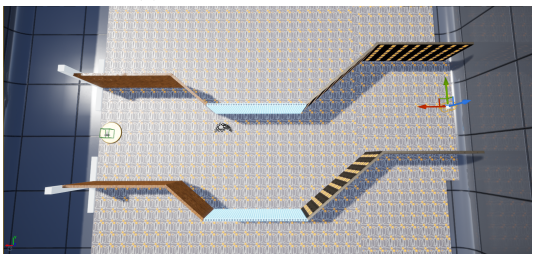
The details of these two corridors are shown in table1. To be more precise, we define "generic corners" as the location where the multi-rotor must alter its movement direction to follow the center-line, and "necessary corners" are the locations where the multi-rotor must change its movement direction to avoid a collision.



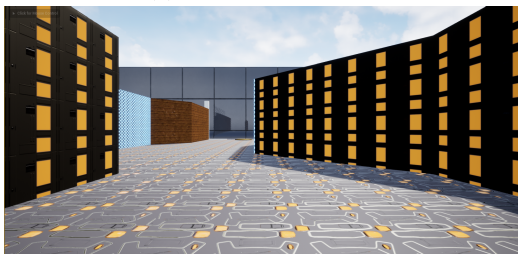
(a) Aerial view of corridor$_1$



(b) Interior of corridor$_1$



(c) Aerial view of corridor$_2$



(d) Interior of corridor$_2$

Figure 6: Illustration of experiment scenarios

	Length	Width	Generic corners	Necessary corners
Corridor$_1$	33.2m	4m	4	4
Corridor$_2$	33.2m	5.7m	4	1

Table 1: detailed data of corridor$_1$ and corridor$_2$

5.2 Optic Flow

Optic flow is the apparent motion of objects in an image between consecutive frames as the camera moves. Once computed, the measured image velocities can be used for wide variety of task, ranging from scene interpretation to autonomous, active explorations[1].

We are requesting the optical flow image data with the help of an AirSim[6] API, but this data still needs further processing. The API provides single channel pixel as float with values $\in [0, 1]$. So directly feeding this image as an observation to the PPO network would not help the agent learn about the results from its action as the image will be dark, as shown in Figure 7a. So we normalise the data and convert the pixel values $\in [0, 255]$ to capture the environment features better as shown in Figure 7b. Brighter pixel represents higher velocity component, implying the object is closer compared to the object with lower pixel values.

(a) Raw Optical Feed from Left Camera (b) Processed Optical Feed from Left Camera

Figure 7: Optical Flow Image data from Left Camera

5.3 PPO training procedure

Once the gym-styled wrapper is defined as in `drone_env.py`, we then use stable-baseline3 to run PPO training loop. A training environment and an evaluation environment is defined using `EvalCallback`[4].Callback are set of functions that allow us to access the internal state of the agent during training. `EvalCallback` evaluates the performance of the agent periodically, using a separate test environment and save the best model and the evaluation results in `evaluations.npz`. A tensorboard log directory is also defined as a parameter to PPO.

6 Results and Discussions

The tensorboard of the training process for the narrow corridor (corridor$_1$) is shown in Figure.(8). We can see the training process has a good shape. All the videos (recommend to see videos in the folder "recommended Videos"), maps, and more training plots are in the google drive `https://drive.google.com/drive/folders/1Rzxsl5mIMBZII9ydF4PUQPlivu8CN9dH?usp=sharing`.

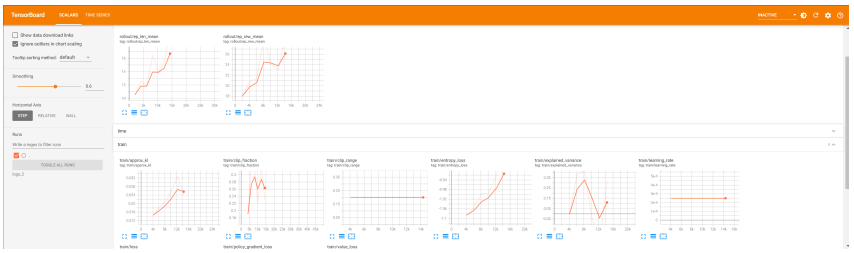


Figure 8: Plots for the training process.(You might need to enlarge to see the details or you can go to google drive to see each plot in more detail.)

Note that the next step is that we should build a test set to test our optimal policy. For example, we can build some new corridors with different shapes, textures, and light conditions for testing. And if the policy works in the test environments, we will work on Sim-to-Real for real robot testing. Due to time constraint, we will do those tests after class.

7 Conclusion

In our project, We successfully built and trained the RL algorithm (PPO) on the corridor following visual navigation task using optic flow information. We did not use any existing built environments and for most of the code, we write from scratch. Our approach shows that it is possible to use the optic flow information to navigate a UAV through a corridor with several turns, which is potential to deploy a low-power version on the insect-scale robots,

8 Issues

- While running simulations, after some 20,000 or 30,000 iterations, Airsim throws an error because of which the training stops. The error is thrown by the simGetImages() API, which is used to get the camera feed. For some reason, it throws an empty array, meaning it didn't capture any image data, which shouldn't happen. A same bug was reported on their github page (`https://github.com/microsoft/AirSim/issues/3623`), which they claim has been resolved, but we got this error a lot of times while training. Figure. 9shows the error.

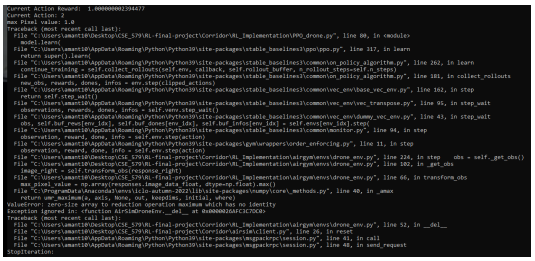


Figure 9: Camera returns empty array error

9 Future Work

- In this project we have used discrete action space with 3 modes: Moving forward, rotating clockwise about vehicle's z axis while moving forward and rotating anti-clockwise about vehicle's z axis while moving forward. Furthermore, these rates are currently constant, with forward velocity(v_x) = 2 m/s and Yaw Rate($\dot{\Phi}$) = 20°$/s$. Instead we can create a continuous action space of v_x and v_y and use those to control the multi-rotor
- In many cases, for different reward functions, the policy converged to a sequence of action that would prefer colliding with the wall to get an immediate reward. It stops exploring very early in the training process. This could be addressed by playing with the hyper-parameters of the PPO function in the stable-baseline3 [4]library
- In this project, we are directly using the optical flow map provided by AirSim[6] API, which may not be optimal for MFIs. Instead we will explore sparse optical flow extraction methods so that they can be used to help MFIs autonomously navigate.
- After the class, we will try a simpler policy by modifying the actor model to reduce the inference power further to be potential to implement on the insect-scale robots.

References

[1] J.L. Barron, D.J. Fleet, S.S. Beauchemin, and T.A. Burkitt. Performance of optical flow techniques. In *Proceedings 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 236–242, 1992.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[3] Yogesh M. Chukewad, Johannes M. James, Avinash T. Singh, and Sawyer B. Fuller. Robofly: An insect-sized robot with simplified fabrication that is capable of flight, ground, and water surface locomotion. *IEEE Transactions on Robotics*, 37:2025–2040, 2021.

[4] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[6] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[7] Robert J. Wood. The first takeoff of a biologically inspired at-scale robotic insect. *IEEE Transactions on Robotics*, 24(2):341–347, 2008.