QPlane: An Open-Source Reinforcement Learning Toolkit for Autonomous Fixed Wing Aircraft Simulation

David J. Richter
Purdue University Northwest
Hammond, Indiana, USA
djrichte@pnw.edu

Ricardo A. Calix
Purdue University Northwest
Hammond, Indiana, USA
rcalix@pnw.edu

ABSTRACT

Reinforcement Learning (RL) is a fast-growing field of research that is mostly applied in the realm of video games due to the compatibility of RL and game tasks. AI Gym has established itself as the gold standard toolkit for Reinforcement Learning research. Unfortunately, toolkits like AI Gym are very optimized for benchmark purposes and may not always be suitable for real world type problems. Additionally, fixed wing flight simulation has specific requirements and may need other solutions. In this paper, we propose QPlane as an alternative toolkit for RL training of fixed wing aircraft. QPlane was developed in an effort to create a RL toolkit for fixed wing aircraft simulation that is easily modifiable for different scenarios. QPlane is replicable and flexible for ease of implementation to high performance computing, and is modular for quick environment and algorithm replacement. In this paper we present and discuss details of QPlane, as well as proof of concept results.

CCS CONCEPTS

• Reinforcement Learning • Artificial Intelligence • Neural Networks

KEYWORDS

Reinforcement learning, deep learning, artificial intelligence, aviation

ACM Reference format:

David J. Richter and Ricardo A. Calix. 2021. QPlane: An Open-Source Reinforcement Learning Toolkit for Fixed Wing Aircraft Simulation. In *12th ACM Multimedia Systems Conference (MMSys '21)*, Sept. 28-Oct. 1, 2021, Istanbul, Turkey. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3458305.3478446>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

MMSys '21, Sept. 28-Oct. 1, 2021, Istanbul, Turkey

© 2021 Copyright held by the owner/author(s). 978-1-4503-8434-6/21/09...\$15.00
<https://doi.org/10.1145/3458305.3478446>

1 INTRODUCTION

With the ever-growing interest in Machine Learning and Artificial intelligence in recent times, the field as a whole has gained a lot of momentum. Reinforcement Learning (RL) has benefited greatly from these developments and has since incorporated deep learning as universal function approximators with much success. Mnih et al. [13, 14] developed an approach called Deep Q-Learning, which successfully learned how to play an assortment of ATARI 2600 games at and above human level skill. Since then, a lot of work has been conducted in Reinforcement Learning, most of which focuses on video games [22]. Due to their ease of implementation and the nature of them having built in scoring systems, games are a perfect match for RL. Games are designed in a way to reward players both short term as well as long term in order for the player to achieve a certain goal. This lends itself perfectly to the reinforcement learning approach. Games are also developed with specific control systems that should be easy to master for human players, again aiding the Reinforcement Learning approach. To this day many video games have been used for this purpose and mastered by the trained agents, even beating world class human players in some cases. Prominent examples are the games of Go [23] and Dota 2 [3].

Autonomous flight, advanced autopiloting systems, and UAVs (Unmanned Aerial Vehicles) are a very active field in research. UAVs are widely implemented for tasks where safety and efficiency and costs play a big factor. UAVs are therefore often used in tasks for inspections and monitoring in dangerous environments and environments that are not reachable for humans. Auto piloting systems are not just used for UAVs but also for manned aircraft. These aircraft, as well as UAVs, use autonomous systems for flight control, both for the outer and inner loop controls. Inner loop controls like attitude control are difficult to learn due to their nonlinear nature, since the values for attitude control depend on a large number of variables. This low level, inner loop task of attitude control is however essential to fixed wing flight, and employing RL to solve it is an idea worth investigating. For training RL models for tasks like attitude control, real life airplanes are unusable, due to the nature of RL behaving randomly initially. Having planes controlled by randomness for a while is not just dangerous but also very expensive, which is why simulators should be used. Testing and training on multiple simulated environments

can provide better generalization and testing capabilities; therefore, better preparing the model for the real-world test.

2 RELATED WORK

Reinforcement Learning has been a field of research for decades [24] before its big resurgence in 2015. Before Mnih et al. introduced the deep Q-network (DQN) approach, RL had only been capable of working in relatively simple environments using traditional Q-learning [28]. Traditional Q-learning, however, only allows for simple small-scale environments with small state and action spaces. All Q values are saved in a table, which may affect generalization. This means that if an agent needs to be fully operational, it would have to be successfully trained on every possible action and space combination. In larger environments this leads to very long training times and is therefore impractical. TD-Gammon [25] was one of the first games to be solved with RL using neural networks, and was long considered to be a special case, not to be replicated in other environments. With the publication of the DQN approach by Mnih et al. it was shown that RL is very much capable of achieving expert human performance in many different environments using high sensory inputs. Mnih et al. used convolutional neural networks (CNN) to estimate the Q values and fed the neural network with preprocessed screenshots of the game screen. To achieve these improved results Mnih et al. implemented two new techniques called memory replay and a second twin network (the target network). These two improvements allowed the model to master several ATARI 2600 games using trial and error with deep RL. This publication kickstarted a new wave of interest into the realm of Deep RL and a lot of work has been conducted in RL since. Algorithms and their hyperparameters have been optimized and new approaches have been proposed and been tested on the same ATARI games as DQN since then. Big achievements were marked by alpha go and Open AI's Dota 2 agent, for example. The field of reinforcement learning for aviation tasks is still rather new because of the challenges associated with flight. It also differs from most current deep RL research in the way pilot agents can observe their environment. While most RL approaches in the literature take screenshots as inputs, this method is not always applicable to flight. Especially because, in the current airplane fleet, most of the information is not in what is visible (via camera data) but in what is given by the flight instruments. In Flight, attitude control is essential. Koch et al. [12] developed a system for quadcopter attitude control using proximal policy optimization and achieved results at the level of a PID controller [12]. Fixed wing aircraft attitude control was also explored by Zhen et al. [29] and Bøhn et al. [4]. These works use proprietary and self-written environments for the agent to learn in, which work well but are difficult to be ported to test for generalization and are also very difficult to modify. Using publicly available flight simulators for reinforcement learning was established in the masters' dissertations by Rennie [17] and Eckstein [7]. Both of these projects follow the AI Gym [5] workflow and built on top of each other. But yet again, because of the way that some simulators like JSBSim [2] are implemented, there is no easy way for switching the model over to

another simulation environment. Both the works of [17] and [7] are also written as computer science thesis, which again makes it hard for researchers with a lesser background in software engineering to pick up the software to implement their own scenarios. Attempts such as in [1] have been made to incorporate Xplane [18] into the AI Gym framework which, just like the JSBSim environments, only work with the XPlane flight simulator. This is where this work extends the pool of available open-source platforms for reinforcement learning for aviation research. QPlane [21] is written in a way where many simulators as well as many reinforcement learning algorithms (or future additions) can be interchanged with a single line of code. This allows for ease of use, modularity, and great flexibility, for researchers of all backgrounds. However, our work is completely independent from AI Gym and, therefore, more portable even to hardware.

3 BACKGROUND

Reinforcement Learning, unlike supervised and unsupervised deep learning, does not require a pre-existing dataset for the agent to learn. Reinforcement Learning models start off in an unknown environment, of which the agent only sees a representation, called the observation or state. The agent then interacts with the environment by applying an action from the action space. This action will affect the environment in a certain way. The agent will receive the new state, the one after the action was completed, and a reward value (or penalty), which determines how well the agent is doing at any given time (see Figure 1).

Figure 1: Interaction of the RL agent and the environment.

Based on that state representation, the used action, and the reward alone, the agent will learn how to maximize its policy, meaning which action to take in which state to gain the highest reward. The Q values are updated after every step using the Bellman equation (Eq. 1). The action to be taken is either chosen by exploitation, meaning that the best action with the highest possible Q value will be picked, or by exploration, where a random action will be executed instead. For solving the exploration-exploitation problem, the ϵ -greedy approach is implemented. ϵ is a value between 0 and 1 that decays over time. At each time step, a random value between 0 and 1 is generated and compared to the ϵ value. If the random value is larger than ϵ , then the action will be chosen based on exploitation. If the random value is less than ϵ value, then the action will be chosen based on exploration.

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max Q(s_{t+1}, a_{t+1})) \quad (1)$$

$Q^{new}(s_t, a_t)$ is the new Q value at the given state-action pair at timestep t . The value α is the learning rate. $Q(s_t, a_t)$ is the previous Q Value. The value γ is the discount factor, r_t is the reward after taking an action in the current state. And $\max Q(s_{t+1}, a_{t+1})$ is the max future Q of the next state after taking the action.

Attitude control for fixed wing aircraft involves the roll, pitch, and yaw angles of an aircraft in flight, meaning it has 6 degrees of freedom (Figure 2). It is a fairly common exercise in fixed wing aviation RL. For fixed wing aircraft, the agent (much like a human pilot) takes control of the rudder, the ailerons, and the elevators to achieve the desired attitude as fast as possible, and is most commonly evaluated on the deviation of the current state to the desired state. The coordinate systems and flight dynamics are handled by the simulators, and therefore not implemented in QPlane itself.

Figure 2: The 6 degrees of freedom of a plane.

4 QPLANE

QPlane is a tool for RL that can implement multiple RL algorithms, multiple simulators, and multiple environments (Figures 3, 4), that can easily be interchanged. This allows for great modularity and also allows users to train a model in one simulation environment and then evaluate that trained model in another simulator.

Figure 3: Architecture and modules of QPlane.

This is a great way of testing for generalization ability of the trained agent without having to test the policy on a real plane or a radio

controlled (RC) model plane. If the generalization process does not succeed, this could get very expensive with real hardware. Using the two simulators, instead, is much more convenient, much faster to set up and repeat, much better to control and track, and also much less cost intensive. The code for QPlane is written with the intent of keeping it easy to read and easy to modify for anyone interested in conducting experiments in autonomous flight. QPlane aims to make it easy for anyone to read and understand the code, and for anyone to make the changes they would like, or even create completely new scenarios from the ground up. The modular design allows for changes in one file that do not affect the other files, which keeps the margin of error in check, and which also makes it much easier and faster to implement the necessary changes. New object classes can also be written easily and then used in a plug and play fashion. The default environments and algorithms can be changed by only modifying a single line in code.

Figure 4: The folder structure of the QPlane source code.

All results during a run can be logged (see Table 1) and then saved dynamically to file (see Figure 4) at a rate defined by the user to suit their needs. All results and saved states can be reloaded to allow the agent to continue learning without affecting any aspect of the experiment, by just setting a few parameters to True (see Table 3). This allows experiments that failed due to whatever reason to be restarted on the fly, from any given saved state, with no extra work to be done, and does not require the entire run to be redone.

Environments

QPlane can interface with many flight simulators that communicate via network packets. Both simulators implemented in this work are set up in a way where, no matter which one is picked, the rest of the code will function identically, and will not even notice which environment it is interacting with (Figure 5).

QPlane is built using external software sources (Table 2). QPlane can be run completely free of charge, but one module (XPlane)

which is optional, does require the purchase of a license. QPlane is however fully functional without the XPlane module, and XPlane is mostly used to test for generalization capabilities of a trained model. If XPlane is chosen as the environment model of choice, the XPlaneConnect [15] plugin needs to be added to the game. XPlaneConnect is an open-source repository by NASA that allows the user to communicate with XPlane through UDP. Other modules tested are JSBSim (the other flight dynamics module), which is open source, and also TensorFlow. TensorFlow can be exchanged for other neural network packages like PyTorch without any problems, but the deep neural network architectures need to be adjusted.

Table 1: Sample of the logging output provided by QPlane at runtime.

Game	870
Move	0
Time taken	1.0962927341461182
State	[10. -15. 0. 1. -12. -0. 33. -8.]
	[p+,p-,r+,r-]
actions_binary	[0 1 0 0]
Current Control:	[-0.1, 0, 0, 0.5, -998, -998]
Current Qs	[[5. 6. 6. 5.]]
Current Orientation:	[10., -15., 1.]
Current AVE of QTable:	5.
Explored (Random)	False
Current Epsilon	0.18741528066231183
Current Reward:	0.07257032044198711
Error Percentage & Count:	0.0, 0.0
Error Code:	{'reset': 0, 'update': 0, 'step': 0}

XPlane

Xplane is a commercial flight simulation game developed by Laminar research, which is highly regarded for its accurate flight dynamics and is used in numerous research projects [8, 19]. To use this environment, a copy of the game needs to be purchased. The XPlane environment is better suited for testing models than it is for training them, since in XPlane it is not possible to step the physics in code, which means that the training happens in real time, which is, therefore, very slow. The interfacing with the game happens through an open source github package developed by NASA called XPlaneConnect, which communicates with XPlane through UDP. This causes for packets to sometimes get lost, which in QPlane will trigger an attempt to resend them and if those attempts fail, the default values will be applied. XPlane also does not have a headless mode, a mode in which it could run without rendering the graphics, which again makes it slower, more hardware intensive, and also unable to perform on systems that run command line only. The game will be paused whenever no actions are being performed to preserve the game state between actions and to make the new state as representable as possible.

JSBSim

The JSBSim environment can be best suited for training since the physics can be stepped by hand. This means that the flight dynamics and flight physics are calculated faster than real time

which allows for the training process to be much quicker than it would be with XPlane.

Figure 5: JSBSim using FlightGear (left) and XPlane (right)

JSBSim also runs in headless mode by default, reducing the hardware cost to run this software, but if desired, JSBSim does offer rendering using FlightGear [16]. When having rendering enabled, the software can still be run at a faster than real-time speed but can also be set to run at real time to see an accurate representation of the results. JSBSim is an open source and free to use flight dynamics simulator developed by the JSBSim team. JSBSim can be downloaded free of charge on their github. FlightGear, the rendering software is also open source and free to use, making JSBSim a completely free choice for RL. Since XPlane was the first module to be implemented into QPlane, the orientation is centered around the XPlane coordinate system, meaning that for some values in JSBSim, they need to either be mirrored or converted (e.g. feet per second to meters per second). JSBSim has also been used for multiple research projects before [10, 27].

Table 2: Software packages and versions used to build QPlane.

SOFTWARE	VERSION
XPLANE11 VERSION:	11.50r3 (build 115033 64-bit, OpenGL)
JSBSIM VERSION:	1.1.5 (GitHub build 277)
FLIGHTGEAR VERSION:	2020.3.6
XPLANECONNECT VERSION:	1.3-rc.2
PYTHON VERSION:	3.8.2
NUMPY VERSION:	1.19.4
TENSORFLOW VERSION:	2.3.0
ANACONDA:	4.9.2
WINDOWS VERSION:	1909

Algorithms

The algorithms module in the QPlane toolkit is the part in charge of the learning process. These modules which can be Q-Learning, DQN, Double DQN, etc. use the same state, action, new state, and reward parameters to train an agent (Algorithm 1). The Bellman equation (Eq. 1) is used to optimize the policy by maximizing for the expected future reward.

QLearning

One of the currently implemented algorithms is the original QLearning [28] algorithm. This algorithm uses no deep learning mechanisms and works on the basis of a lookup table. Each state action pair is one table entry and is updated only when encountered during training using the Bellman equation (1). This approach allows for less generalization. Our module follows the original

implementation of Q Learning [6, 11]. This approach does not require TensorFlow and because of that, and due to the simple computation needed to update the table, runs the fastest.

Deep QLearning

Deep QLearning makes use of neural networks for the Q value estimations, by using the multi-layer perceptron as a universal function approximator. This allows the Deep Learning models to achieve generalization capabilities, meaning that they can perform in states that they have never been trained on. For tasks like attitude control, this can be an important feature, since having an aircraft stop functioning midair is catastrophic. The network architecture takes state variables as inputs for the neural network and then the neural net outputs the Q values for each possible action from the action space. Deep Q Learning, as proposed by Mnih et al. introduced two new improvements: Memory replay and twin target networks. The replay memory adds a list of past actions, that are stored for use later on. In this way, the network does not train on one datapoint at a time, but it can instead randomly pull previous interactions with the environment from the memory and learn from those as well, further improving the stability and generalization required by neural networks. The target network works as a clone of the neural network and is used to update the Q values in the memory replay for training. The target network is only updated every n episodes, to improve stability.

Algorithm 1: Workflow of QPlane

```

Initialize Q agent  $Q$ 
Initialize environment  $E$ 
for episode = 1,  $M$  do
    get state  $s$  from  $E$  after reset
    for step 1,  $N$  do
         $Q$  select action  $a$  based on  $s$ 
        apply  $a$  in  $E$  and get new state  $s'$  and reward  $r$ 
        update  $Q$  with  $s', s, r$ 
         $s = s'$ 
    log and archive

```

Double Deep QLearning

Double Deep QLearning [9, 26] works very similarly to Deep QLearning. It also implements replay memory and target networks and also uses neural networks for value function approximation. It does come with one addition to the way DQN works. Double Deep QLearning makes it so that the target network predicts the future Q values from the memory, but not based on its own previous predictions. It uses the other network to predict those, which according to [9, 26] increases performance, stability, and reduces the overestimation problems that DQNs can have. With Double Deep Q Learning, models could learn better policies, that are more stable and more reliable, by removing the overoptimization that DQNs could have.

Table 3: QPlane Parameters.

Parameter	Short Description	Default
experimentName	Name of folder for saving	Experiment
logPeriod	every so many epochs the metrics will be printed	10
savePeriod	every so many epochs the model will be saved to a file	25
pauseDelay	time an action is being applied to the environment	0.01
logDecimals	Decimals to be printed in log	10
n_epochs	Number of inputs per episode	5000
n_steps	No. neurons for hidden layer 2	1000
n_actions	Number of possible inputs to choose from	4
n_states	Number of states for non-Deep QLearning	4
gamma	The discount rate	0.95
lr	Learning rate	0.0001
epsilon	Starting Epsilon Rate	1
decayRate	Epsilon decay rate per step	0.00001
epsilonMin	Minimum value at which epsilon will stop decaying	0.1
n_epochsBeforeDecay	number of episodes before epsilon starts to decay	0.1
numOfInputs	Number of inputs fed to the model for Deep RL	10
minReplayMemSize	min size determines when the replay will start being used	1000
replayMemSize	Max size for the replay buffer	100,000
batchSize	Batch size for the model	256
updateRate	update target model every so many episodes	5
loadModel	will load model from file	False
loadMemory	Loads replayMemory from file	False
loadResults	Loads results from file	False
jsbRender	Enables rendering for JSBSim	False
jsbRealTime	JSBSim will render in realtime	False
usePredefinedSeeds	Sets random seeds to fix values	False
saveForAutoReload	Saves and overrides models, results and memory to the root	False

Proof of Concept

As proof of concept, experiments were run using the regular Q Learning approach with a reward function that very much aides the learning process. The task was attitude control (stabilization), and the plane was spawned in 9 different starting positions to train it on as many states as possible. A total of 729 states were derived from pitch, roll, and yaw readings. Each of the 3 ranges which generally go from -180 to +180 (pitch, roll, yaw) is divided into 9 bins (e.g. bin 1 is for 0-10 degrees, bin 2 is for 10-30 degrees, and so on). Then, using cube encoding, we obtain 729 states ($9 \times 9 \times 9 = 729$) to represent every possible combination of states. This is the same state encoding approach used in the classic taxi cab RL problem. The reward is simple and can be modified by the users. Ours is based on the goal of airplane stabilization. The reward function uses information derived from reading the gauges that a pilot would see on the cockpit dash board. In general aviation, the main gauges are sometimes referred to as the six pack and present the aircraft current information about pitch, roll, yaw, etc. Any flight

characteristics (landings, weather, etc.) could be modeled as long as the researcher changes the reward function. In RL, reward functions depend on the problem and are a very active field of research. The reward is calculated from the delta between the desired state (0° pitch and 0° roll) and current state. The closer the aircraft is to this state, the higher the reward. The hyperparameters were kept as default for the QLearning approach. The pauseDelay was set to 0.1, to make actions have more impact. As can be seen in Figure 6, the model converges to a maximum score after around 2000 played games (200,000 updates). A video of the model in action can be found online [20]. The minimum rewards (Fig. 6) remain somewhat low due to the fact that 4 of the 9 starting positions are spawning the plane in a very unbalanced position, where it is tough to gain a very high score, due to the fact that it takes a considerable number of steps to reach the desired state.

Figure 6: Rewards converging to the optimal policy (QLearn).

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented and discussed details of QPlane, as well as proof of concept results. We believe the toolkit will be very useful for research purposes and benefit the UAV research community. Some future work for QPlane includes adding additional Reinforcement Learning algorithms, as well as potentially adding more environments.

ACKNOWLEDGEMENTS

The authors would like to thank Mirko Rosenthal M.Sc. for his help and advice in writing the code for this project.

REFERENCES

- [1] adderbyte. Github. *Gym XPlane*. Retrieved May 31, 2021 from https://github.com/adderbyte/GYM_XPLANE_ML.
- [2] Jon Berndt. 2004. JSBSim: An open source flight dynamics model in C++. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*. 4923.
- [3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębniak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. 2019. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* (2019).
- [4] Eivind Bohn, Erlend M Coates, Signe Moe, and Tor Ame Johansen. 2019. Deep reinforcement learning attitude control of fixed-wing uavs using proximal policy optimization. In *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 523–533.
- [5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).
- [6] DeepLizard. DeepLizard.com. *Q-Reinforcement Learning - Goal Oriented Intelligence*. Retrieved May 31, 2021 from https://deeplizard.com/learn/playlist/PLZbbT5o_s2xoWNVdDudn51XM8IouZ_Njv.
- [7] Felix Eckstein. 2020. *Learning to Fly – Building an Autopilot System based on Neural Networks and Reinforcement Learning*. Master's thesis. FernUniversität Hagen, Hagen, Germany.
- [8] Richard Garcia and Laura Barnes. 2009. Multi-UAV simulator utilizing X-Plane. In *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*. Springer, 393–406.
- [9] Hado Hasselt. 2010. Double Q-learning. *Advances in neural information processing systems* 23 (2010), 2613–2621.
- [10] Joshua P Kim. 2016. Evaluation of Unmanned Aircraft Flying Qualities Using JSBSim. (2016).
- [11] Harrison Kinsley. Pythonprogramming.net. *Q-Learning introduction and Q Table – Reinforcement Learning w/ Python*. Retrieved May 31, 2021 from <https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/>.
- [12] William Koch, Renato Mancuso, Richard West, and Azer Bestavros. 2019. Reinforcement learning for UAV attitude control. *ACM Transactions on Cyber-Physical Systems* 3, 2 (2019), 1–21.
- [13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
- [14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.
- [15] NASA. Github. *XPlaneConnect*. Retrieved May 31, 2021 from <https://github.com/nasa/XPlaneConnect>.
- [16] Alexander R Perry. 2004. The flightgear flight simulator. In *Proceedings of the USENIX Annual Technical Conference*, Vol. 686.
- [17] Gordon Rennie. 2018. *Autonomous Control of Simulated Fixed Wing Aircraft using Deep Reinforcement Learning*. Master's thesis. The University of Bath, Bath, United Kingdom.
- [18] Laminar Research. XPlane.com. *XPlane*. Retrieved May 31, 2021 from <https://www.xplane.com/>.
- [19] Lucio R Ribeiro and Neusa Maria F Oliveira. 2010. UAV autopilot controllers testplatform using Matlab/Simulink and X-Plane. In *2010 IEEE Frontiers in Education Conference (FIE)*. IEEE, S2H–1.
- [20] David J. Richter, Ricardo A. Calix. 2020. QPlane result video. Video. Retrieved May 31, 2021 from <https://www.youtube.com/watch?v=zd5ldaWiCjI>.
- [21] David J. Richter. Github. *QPlane Fixed Wing Flight Simulation Environment for Reinforcement Learning*. Retrieved May 31, 2021 from <https://github.com/JDatPNW/QPlane>.
- [22] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. 2019. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944* (2019).
- [23] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.
- [24] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
- [25] Gerald Tesauro. 1995. Temporal difference learning and TD-Gammon. *Commun. ACM* 38, 3 (1995), 58–68.
- [26] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.
- [27] Tomáš Vogeltanz and Roman Jašek. 2015. JSBSim library for flight dynamics modelling of a mini-UAV. In *AIP Conference Proceedings*, Vol. 1648. AIP Publishing LLC, 550015.
- [28] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3–4 (1992), 279–292.
- [29] Yan Zhen, Mingrui Hao, and Wendi Sun. 2020. Deep Reinforcement Learning Attitude Control of Fixed-Wing UAVs. In *2020 3rd International Conference on Unmanned Systems (ICUS)*. IEEE, 239–244.