A Self-Paddling Canoe Simulation with Deep Reinforcement Learning

by

Zi Cong Guo

Supervisor: Professor Tim D. Barfoot
April 2020

B.A.Sc. Thesis

Division of Engineering Science
UNIVERSITY OF TORONTO

A Self-Paddling Canoe Simulation with Deep Reinforcement Learning

by

Zi Cong Guo

Supervisor: Professor Tim D. Barfoot

April 2020

Abstract

Deep reinforcement learning (RL) techniques are powerful tools capable of training a neural net controller that is robust in complex, nonconvex environments. In this thesis, we investigate the validity of using Deep RL techniques to train a canoe to propel itself to a desired location in simulation. We first develop a 2D top-down simulation of a canoe-in-water simulation, where we designed the canoe-paddle model and modelled the solid-fluid interactions with simplified physics. We demonstrate the validity of this simulation by designing a handcrafted open-loop controller capable of propelling the canoe as desired. Then, we set up the RL problem and attempt to train it with Deep RL techniques (PPO2, A3C, A2C, TD3, HER), comparing the resulting agent to the open-loop controller as baseline. We found that by using PPO2, the agent performed slightly worse than the open-loop controller if the canoe was initialized at one consistent pose throughout. However, Deep RL failed to train an agent if the canoe was initialized at a random pose in the simulation. We outline possible reasons and future directions to investigate to resolve the training failure.

Acknowledgements

This project would not have been possible without the support of many generous individuals. I would like to show my sincere gratitude to my supervisor, Professor Tim Barfoot, for his invaluable guidance along the way. I was able to maintain my motivation due to his encouragement and patience, and he offered many valuable suggestions and feedback which kept the project from steering off track. I would like to thank my thesis course coordinator, Professor Alan Chong, for providing me with thesis workshops and resources, as well as all the background coordination. I would like to thank the University of Toronto Institute for Aerospace Studies (UTIAS) for allowing me to borrow the Nvidia DGX Station for training. Last but certainly not least, I would also like to thank my friends and family members for their gracious emotional support.

Contents

List of Figures

List of Tables

1 Introduction

Control is a extensive topic in robotics with a variety of applications, most of which involves controlling a complicated robotics system to meet a specific objective. Traditionally, classical optimal control aims to maximizes an objective function, either by manually designing a control policy for the system to follow or by optimizing for future controls with Model Predictive Control (MPC). For sensitive, non-linear systems involving fluid dynamics, designing control policies manually is very difficult. MPCs can solve a variety of control problems, but they more efficient when the problem can be formulated as convex [2]. If not, nonlinear MPCs (NMPC) can be used, but they are usually too slow for controlling such systems in real time, and sometimes even when offline [3]. This is the case with fluid dynamics where, although it can be considered convex once turbulence and other factors are ignored, the full governing equations are mostly nonconvex.

Thus, some researchers have turned to reinforcement learning (RL) techniques where instead of designing a controller to fit the system model, the controller learns an optimal control policy by running through a great number of trials ("episodes") and optimizing its policy appropriately based on their outcomes. In the case of classical RL, the agent builds and tunes a probability distribution of discrete control commands from the observations. In the realm of Deep RL, the agent estimates future rewards and generate actions, discrete or continuous, using neural networks. Although there have been some attempts to use RL for controlling a system involving fluids, RL is still widely considered as difficult to quickly converge to solutions [4], especially for fluids where simulations are computationally expensive. Nevertheless, if one can set up RL for a fluid system such that it can be trained efficiently, RL can be a useful method for control.

The objective of this thesis is to investigate using RL, specifically Deep RL techniques, to generate controls for efficiently paddling a canoe in simulation. As there are no such simulations that are fast enough for RL training, we will first create a simulation environment for a canoe in a fluid. The environment consists of a top-down view of a canoe floating within

a fluid velocity field. The canoe, modeled simply as a rigid elliptical body with a two-link manipulator at each end, would learn how to control its motions in the fluid by controlling the angular velocities of the paddles. The agent will then be trained with Deep RL to learn to actuate the paddles optimally to reach a target location while counteracting disturbances from the velocity field.

This thesis would provide further evidence for the feasibility of using RL to control a complex, nonlinear system. If successful, controlling with RL would no longer require accurate models of the system, which was the case of traditional optimal policy control. As well, after training is complete, inferencing the RL network for control would likely be faster than generating control commands from NMPC, potentially allowing for real-time control.

The remainder of this thesis is organized as follows. We briefly review previous works for fluid simulations and RL agents for control in Section 2. We discuss the creation of a low-fidelity 2D simulation of the canoe-in-water environment in Python in Section 3. We discuss the simulation physics and the design choices made such that the simulation is realistic and efficient enough for RL training. Then, in Section 4, we discuss using Deep RL to train a paddling controller. We set up the RL problem, justify the design choices for our RL algorithm, and discuss the training process involved to generate control inputs that achieves the required objective, and then present the results. After this, we attempt to use a different RL approach to solve the same problem, and we present this in Section 5. We describe possible future work in Section 6, outline a few lessons learned throughout this project in Section 7, and finally give concluding thoughts in Section 8.

2 Related works

In this section, we discuss related works for fluid simulators. We then discuss relevant works involving the use of RL agents, especially in the context of control.

2.1 Review of Fluid Simulators

We require an efficient fluid simulation capable of simulating fluids and solid-fluid interactions. The speed of the simulation can greatly affect the training time for RL. As an example, Ma et al.'s work bears very close resemblance to our system, using Deep RL to train a controller in simulation for controlling a rigid body (ball) with fluid propulsions [5]. Ma et al. used an adaptive grid-based fluid simulation environment and designed coupling physics that aims to minimize complexity. Still, the training process for the 2D case took 10 hours and used "four 2.5GHz Intel Xeon E5 CPUs (32 cores in all) and 1 GPU (Nvidia GTX1080)", even with parallelization [5]. The training time may be related to their choice of using TRPO, which is slower than other RL algorithms such as PPO2, but the training speed can be improved by reducing the complexity of the simulation.

There are a few Python fluid simulators that are immediately available to use, including Fluidity [6] and FluidSim [7]. Fluidity is a complex simulation system capable of accurately simulating multiple types of fluid flows and some solid-fluid interactions. However, it is not very computationally efficient and there is less control on the backend physics, making it difficult to be modified and simplified for our purposes. FluidSim is a fluid simulator designed for efficiency and easy development, but there are no physics implemented for solids nor solid-fluid interactions. We require a simulation that is efficient and easy to develop upon. As a tradeoff, the physics does not necessarily need to be the most accurate, but the simulation should appear realistic. For this purpose, we chose to implement the fluid field simulation from Joe Stam's *Stable Fluids* paper (described in Section 3.1) and develop our own solid-fluid interaction physics upon this work.

2.2 Review of Reinforcement Learning Agents

RL has been traditionally in games and other simulated environments. The classic examples of the successes of RL include the agents' super-human performances in Atari [8], Chess [9], and Go [10]. Recently, RL is beginning to be used for classical control problems [4]. One example is the work by Rahman et al., who used Deep Q-learning to make a balancing

robot in simulation (Gazebo) [11]. There have been a few attempts to use RL for controlling a system involving fluids. The work by Ma et al., which has a similar RL problem to ours, uses TRPO policy optimization, and is successful in balancing a ball and playing ball games with fluid propulsions [5]. Won et al. trained an imaginary dragon to do flapping flight using a novel learning algorithm on Deep Q-learning [12]. Most of the related works, however, are still training RL in simulation, and the majority of the safety-critical control problems are still solved with classical methods. Buşoniu et al. argues that this might be due to fundamental philosophical differences between AI and control, that AI researchers focus on performance based on a specific reward function, while control researchers focus on stability and guarantees [13]. In the end, more evidence is needed, including simulations of new control scenarios, for the general effectiveness of RL in a wide range of control formulations.

3 Canoe Simulation

3.1 Fluid Velocity Field Simulation

The physics of the fluid velocity field simulation is largely taken from Stam [14], which proposed a model for complex fluid flow animation that is unconditionally stable. This model is explained again in a follow up conference paper by Stam, using the simulations for fluid animations in games [15]. The fluid simulation is based on efficiently approximating solutions to the Navier-Stokes equations. The paper outlines the physics for both a density field (for animation effects such as smoke) and a velocity field (for propagating the density field). At each time step, the simulation add new forces, diffuse the velocities to their surroundings, propel the fluid parcels forward through time, and project the velocity field. The projection step uses Hodge decomposition to ensure that the fluid field conserves mass, and is the main contributor to fluid vortices and other complex behaviours. If a flow vector hits one of the four simulation boundaries (top, bottom, left, right), a boundary condition is applied by introducing a new reflected velocity vector at the boundary location, and solving the system of equations would then yield an orthogonal velocity of zero, so no flow "es-

capes". Unlike many other simulations in the realm of computational physics that focuses on simulation accuracy, the simulation model by Stam is stable for any diffusion rate J, time interval Δt, simulation space n_{sim}, and viscosity μ. There are some drawbacks in terms of accuracy, one of which is numerical dissipation, where the fluids dampen faster than they would in reality [14]. Nevertheless, the fluid animation appears fairly realistic and has complex emergent fluid behaviours such as eddies. Examples and figures of realistic fluid flows in various simulation scenarios can be seen at the end of [14] or [15].

To construct this fluid field for the canoe simulation, the initial fluid simulation code is taken from a GitHub repository by Alberto Santini [16], which implements the Stable Fluids simulation in Python. We set the parameters diffusion rate J, time interval Δt, simulation space n_{sim}, and viscosity (μ) such that the simulation is reasonably fast and appears realistic. To tune the parameters to get realistic interactions within the velocity field, we looked the velocity field directly and also examined its effects on smoke parcels in the density field (Figure 1). For viscosity μ (where $\mu \geqslant 0$ always), we notice that for any non-zero μ, the entire field eventually converges to a zero-flow state as long as no fluid cells emit a constant flow source. If there are constant flow sources, the velocity field would still not diverge but would converge to some steady-state flow (Figure 2). Then, it is possible to approximate a steady-state flow in the field by adding appropriate constant flow sources for a set of fluid cells. These non-zero steady flows can be used again later for training the canoe to paddle to goals within a constant flow field.

3.2 Canoe and paddle geometry

The canoe consists of the hull, the arms, and the paddles. The arms are attached to the canoe at one end, and attached to the paddle at the other end. The arms and the paddles can rotate at the attachment locations. Both the canoe and the paddles are considered "in water" and will affect force calculations, while the arms are considered "out of water" and will not affect forces. We define *limbs* as objects that are either arms or paddles, and we define *effective objects* as objects that affect force calculations, i.e. canoe hull and paddles.

Figure 1: Density field example

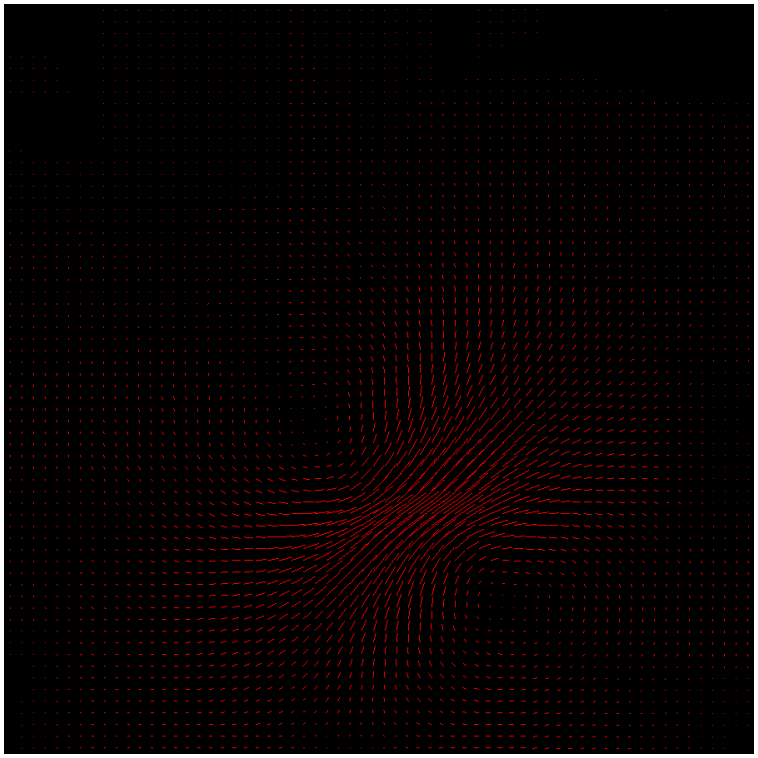


Figure 2: Constant velocity field example

The hull of the canoe is modelled as an ellipse with a semi-major axis of 5 units ($a = 5$) and a semi-minor axis of 1 unit ($b = 1$). The arms are modelled as a line segment with length of 2 units and are protruding from the canoe at 0.5 units from each end, roughly at where would be the seats of the canoe. They can rotate up to an angle limit of $5\pi/8$ in either direction from their neutral orientations. The paddles are also modelled as line segments, with length 1, and their centres are attached to the other end of the arm. The paddles can rotate at their attachment points up to an angle limit of $\pi/2$ in either direction from their neutral orientations. By setting these lengths and angle limits for the arms and paddles, we ensure that the paddles would never be in contact with the canoe hull for any possible orientation. See Figure 3 for a visualization of the canoe model geometry.

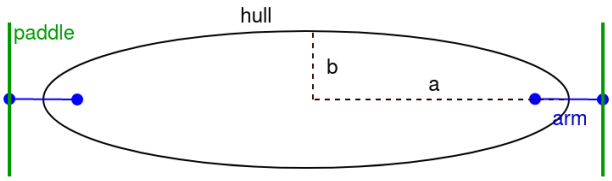


Figure 3: Canoe geometry

For drawing ellipse and the line segments and for calculating the forces upon them, the shapes are approximated by a series of small surfaces, represented by a series of points for positions and unit normal vectors for orientations. The ellipse surface positions are represented by a series of points along its outline, generated by:

$$x = a\cos\theta, \quad y = b\sin\theta \tag{1}$$

where a is the semi-major axis, b is the semi-minor axis, θ is the polar angle, $0 \leqslant \theta \leqslant 2\pi$. The polar angles ($\theta$) ranging from 0 to 2π are generated using numerical method from [17] such that the resulting points are equally spaced around the circumference. The normal vectors for their respective surfaces are generated by taking finite differences between the

points, then normalizing its orthogonal (a CCW rotation by $\pi/2$):

$$\Delta \mathbf{r}^i = \mathbf{r}^{i+1} - \mathbf{r}^{i-1} \tag{2}$$

$$\mathbf{n}^i = \mathbf{C}\left(\frac{\pi}{2}\right)\Delta \mathbf{r}^i \hat{\mathbf{n}}^i = \frac{\mathbf{n}^i}{||\mathbf{n}^i||} \tag{3}$$

Figure 4 shows the points and unit vectors representing the hull surface.

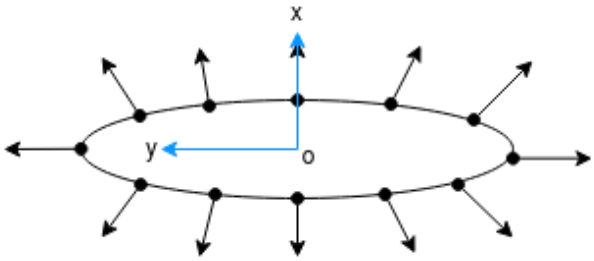


Figure 4: Canoe frame definition

For the limbs, the line segments are represented simply a series of equally spaced points along a straight segment from one end to the other, and the normal vectors are all upward unit vectors (Figures 5 and 6).

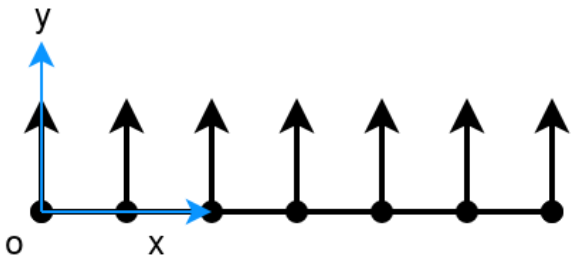


Figure 5: Arm frame definition

As a note, the direction of the normal vectors for all objects are chosen arbitrarily to be counterclockwise to the surfaces. These directions can be reversed with no effects on the resulting wrenches due to how the wrenches are calculated (we can see in this Section 3.4).

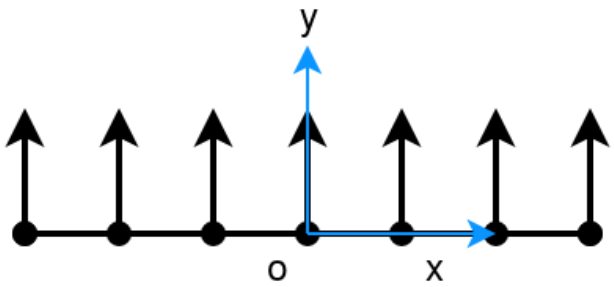


Figure 6: Paddle frame definition

3.3 Transformation Tree

To formalize definitions, we use the term *pose* to represent the position and orientation of an object about its frame origin. We use the term *twist* to represent the velocity and angular velocity about the frame origin of objects.

$$\vec{\mathbf{p}} = \begin{bmatrix} \vec{r} \\ \vec{\theta} \end{bmatrix}, \qquad \vec{\boldsymbol{\varpi}} = \begin{bmatrix} \vec{v} \\ \vec{\omega} \end{bmatrix} \tag{4}$$

where $\vec{r}$ is the position from the origin, $\vec{\theta}$ is the orientation about the origin, $\vec{v}$ is the velocity, and $\vec{\omega}$ is the angular velocity about the origin. Since the simulation is in 2D, orientation and angular velocity can also be written in scalar form (θ, ω), about the axis coming out of the page.

To calculate the force and velocity calculations of objects, we will need to calculate the positions and velocities of points within one object with respect to other objects. We set up a transformation tree to calculate these transformations systematically. Each frame contains the homogeneous transformation with respect to its parent frame, the velocities of this frame with respect to the parent frame written in the parent frame, and the frame's object surfaces (i.e. points and normal vectors) that are stationary within this frame. Since the objects within frame k are stationary with respect to frame k, the velocity of the frame k with respect to another frame i is in effect the velocities of all of the objects within frame k with respect to frame i. The frame centres are placed at the centre of rotation for all objects. For the canoe, the frame is set up so that the centre of the canoe is the frame origin

(see Figure 4). For the limbs, the frame is placed such that the segment are along the x-axis. The centre for the arms are placed at one end (Figure 5), and the centre of the paddles are placed at the midpoint (Figure 6). See Figure 7 for the resulting tree.

With the transformation tree, we can now find the position and velocity of objects of one frome with respect to another frame by recursively applying Equation 38 and 39 in Appendix A.1. This will be used in calculating wrenches in Section 3.4, where we will mostly inquire the position and velocity of objects in canoe, arm, or paddle frames with respect to the world frame.

3.4 Wrench calculation

We use the term *wrench* to represent the forces exerted on an object's frame origin and the torque about the origin:

$$\vec{\mathbf{F}} = \begin{bmatrix} \vec{f} \\ \vec{\tau} \end{bmatrix} \tag{5}$$

where $\vec{f}$ is the force on the frame origin and $\vec{\tau}$ is the torque about the frame origin. We assume that all effective objects contribute to the wrench onto the canoe's centre of mass, both by the velocity field surrounding the object and by the object's self movements, and that non-effective objects do not contribute to any wrenches directly. We also assume that the canoe frame's origin corresponds to the entire canoe's centre of mass.

We assume that the wrenches imparted onto the canoe are either direct forces imparted by the surrounding velocity field, or opposing forces from the movement of the canoe itself due to Newton's third law.

For an object in a velocity field, the drag force is proportional to the square of the velocity ($f \sim Cv^2$). Thus, the force is approximated as the velocity onto the normal vector, then squared. We define the force projection function as follows:

$$\vec{f}_{\text{proj}}(\vec{v}, \hat{n}) = \text{sign}(\vec{v} \cdot \hat{n})(\vec{v} \cdot \hat{n})^2 \hat{n} \tag{6}$$

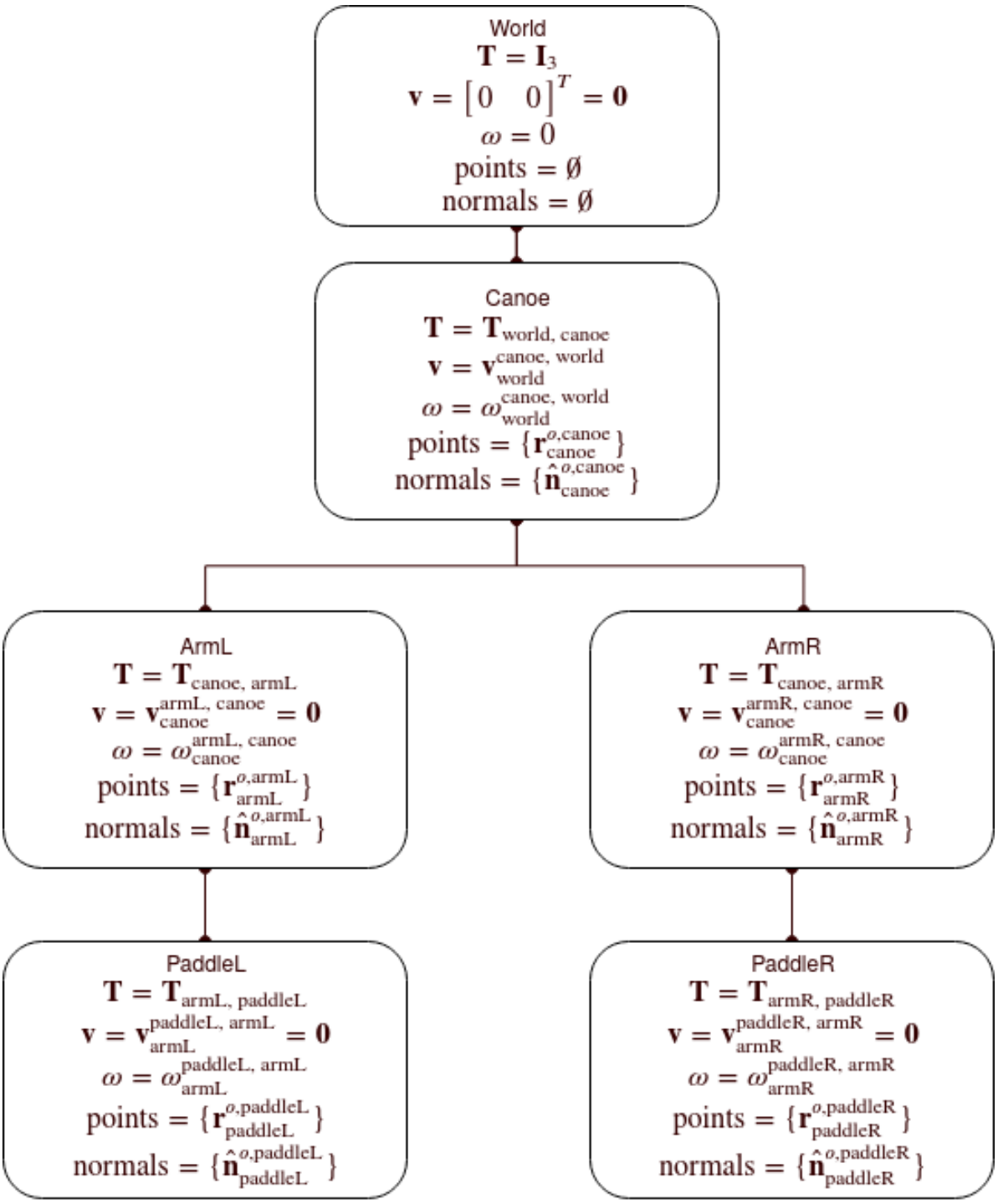


Figure 7: Transformation tree. "o" stands for objects (surfaces) within the frame, L stands for left, R stands for right. Since the same arms are used for the left and the right, $\{\mathbf{r}^{o,\text{armL}}_{\text{armL}}\} = \{\mathbf{r}^{o,\text{armR}}_{\text{armR}}\}$ and $\{\hat{\mathbf{n}}^{o,\text{armL}}_{\text{armL}}\} = \{\hat{\mathbf{n}}^{o,\text{armR}}_{\text{armR}}\}$, and similarly for the paddles. The points and normals for the canoe, arms, and paddles correspond to Figures 4, 5, 6, respectively.

Or, putting vectors into any frame:

$$\mathbf{f}_{\text{proj}}(\mathbf{v}, \hat{\mathbf{n}}) = \text{sign}(\mathbf{v} \cdot \hat{\mathbf{n}})(\mathbf{v} \cdot \hat{\mathbf{n}})^2 \hat{\mathbf{n}} \tag{7}$$

Using this equation (6) for force projections, we can see that if the normal vector is flipped (i.e. using $\hat{n}' = -\hat{n}$), $\vec{f}_{\text{proj}}$ is the same due to the sign operator at the front.

For an object in frame i, we calculate its contribution to the wrench onto the canoe as follows: let $\mathbf{r}_w$ be the position of object in world frame, and let $\hat{\mathbf{n}}_w$ be the normal vector of the object in world frame, both of which we calculate from the transformation tree (see Appendix A.1). Then,

$$\vec{f} = \vec{f}_{\text{proj}}(\vec{v}^{\,\text{water}}(\vec{r}^{\,o,w}) - c_k(\vec{r}^{\,o,w})^{\bullet}, \hat{n}^o) \tag{8}$$

where c_k is a scaling factor related to the difference in mass between the water and the canoe. Putting all vectors into the world frame:

$$\mathbf{f}_w = \mathbf{f}_{\text{proj}}(\mathbf{v}_w^{\text{water}}(\mathbf{r}_w^{o,w}) - c_k \dot{\mathbf{r}}_w^{o,w}, \hat{\mathbf{n}}_w^o) \tag{9}$$

We can get the values of $\mathbf{r}_w^{o,w}, \dot{\mathbf{r}}_w^{o,w}, \hat{\mathbf{n}}_w^o$ for any object (null, arms, paddles) by querying the transform tree.

The torque about the canoe's centre of mass due to the forces experienced by object, $\vec{\tau}$, is calculated using $\vec{f}$:

$$\vec{\tau} = \vec{r}^{\,o,c} \times \vec{f} \tag{10}$$

where $\vec{r}^{\,o,c}$ is the pose of the object in the canoe frame. Since the simulation is 2D, $\vec{\tau}$ would always have the form $\begin{bmatrix} 0 & 0 & \tau^k \end{bmatrix}^T$, and torque would have the same representation regardless of frame since all frames are right-handed. So, we simply write torque as the scalar form τ. Using the cross product rule, we get:

$$\tau = \|\vec{r}^{\,o,c}\| \left\| \vec{f} \right\| \sin(\theta^{r,f}) \tag{11}$$

Putting this into the world frame:

$$\tau = \|\mathbf{r}_w^{o,c}\| \, \|\mathbf{f}_w\| \sin(\theta_w^f - \theta_w^r) \tag{12}$$

We can get $\mathbf{f}_w$ from Equation 9 above, $\mathbf{r}_w^{o,c}$ from the transformation tree, and θ_w^f, θ_w^r from taking arctans of $\mathbf{f}_w, \mathbf{r}_w^{o,c}$:

$$\theta_w^f = \text{arctan2}((\mathbf{f}_w)_y, (\mathbf{f}_w)_x), \quad \theta_w^r = \text{arctan2}((\mathbf{r}_w^{o,c})_y, (\mathbf{r}_w^{o,c})_x) \tag{13}$$

We get the total forces and torques on the canoe from summing up the wrenches from all effective objects:

$$\vec{f'} = c_f \sum_k^{\text{all objects}} \vec{f^k}, \quad \tau' = c_\tau \sum_k^{\text{all objects}} \tau^k \tag{14}$$

where c_f is a scaling factor related to the mass of the canoe, and c_τ is a scaling factor related to the moment of inertia of the canoe. Both factors are tuned such that canoe moves at a reasonable pace under forces within the simulation.

To prevent the canoe from experiencing very large wrenches that could cause the simulation to become unstable, the forces and torques are also saturated:

$$\vec{f} = \begin{cases} \vec{f'} & \text{if } \left\|\vec{f'}\right\| \leqslant f_{\text{sat}} \\ \frac{\vec{f'}}{\|\vec{f'}\|} f_{\text{sat}} & \text{if } \left\|\vec{f'}\right\| > f_{\text{sat}} \end{cases} \tag{15}$$

$$\tau = \begin{cases} \tau' & \text{if } |\tau'| \leqslant \tau_{\text{sat}} \\ \text{sign}(\tau')\tau_{\text{sat}} & \text{if } |\tau'| > \tau_{\text{sat}} \end{cases} \tag{16}$$

The saturation levels are set such that in a velocity field with no additional sources or sinks, the canoe would not be able to reach the saturation wrenches simply by driving its paddles. It could only happen if artificial sources or sinks in the velocity field somehow induce huge wrenches onto the canoe.

3.5 Movement of objects

In this section, we use the wrenches calculated in Section 3.4 to update the twists and poses of objects.

3.5.1 Propulsion of canoe

After calculating the wrenches, the canoe's velocity with respect to the world frame is updated in the transformation tree:

$$\vec{v}^{\,c,w} \leftarrow \vec{v}^{\,c,w} + c_t \vec{f}, \quad \omega^{c,w} \leftarrow \omega^{c,w} + c_t \tau \tag{17}$$

where $\vec{f}, \tau$ is the total force and torque experienced by the canoe, and c_t is a scaling factor related to the time step of the simulation. Putting this into world frame:

$$\mathbf{v}_w^{c,w} \leftarrow \mathbf{v}_w^{c,w} + c_t \mathbf{f}_w, \quad \omega_w^{c,w} \leftarrow \omega_w^{c,w} + c_t \tau \tag{18}$$

The canoe's pose with respect to the world frame in the transformation tree is updated using the new velocities:

$$\vec{r}^{\,c,w} \leftarrow \vec{r}^{\,c,w} + d_t \vec{v}^{\,c,w}, \quad \theta_w^{c,w} \leftarrow \theta_w^{c,w} + d_t \omega^{c,w} \tag{19}$$

where d_t is the simulation time step. Or, in world frame:

$$\mathbf{r}_w^{c,w} \leftarrow \mathbf{r}_w^{c,w} + d_t \mathbf{v}_w^{c,w}, \quad \theta_w^{c,w} \leftarrow \theta_w^{c,w} + d_t \omega_w^{c,w} \tag{20}$$

3.5.2 Propulsion of limbs

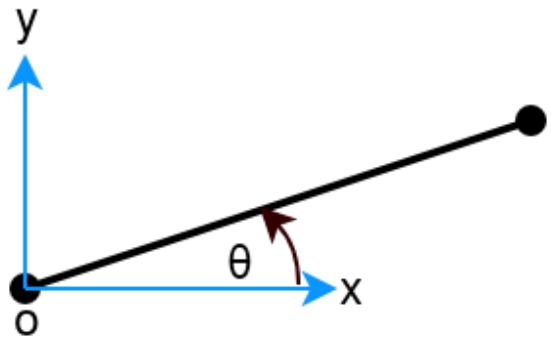


Figure 8: Definition of θ for arms

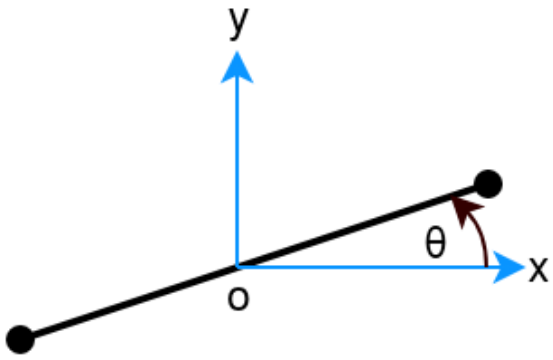


Figure 9: Definition of θ for paddles

We use a simple model for the movement of the limbs around their centres of rotation. Since their frames are rigidly attached to their parent frames at a fixed joint, they can only rotate about their frame origin (i.e. change θ in Figures 8 and 9). We assume that a limb rotate at a constant angular velocity, and assume that there is a lower-level motor controller that can instantaneously set the angular velocities of the limbs to the desired values. Letting ω_i' be the input angular velocities for limb i from a controller, the angular velocity of limb i is:

$$\omega_i = \begin{cases} \omega_i' & \text{if } |\omega'| \leqslant \omega^{\text{sat}} \\ \text{sign}(\omega')\omega^{\text{sat}} & \text{if } |\omega'| > \omega^{\text{sat}} \end{cases} \tag{21}$$

where ω^{sat} is the saturation angular velocity. We have the assumption that ω_i can change instantaneously as desired, which is reasonable as the control loop for motor speeds is usually much faster than the control loop for determining ω_i.

The orientation of limbs can be updated in a similar manner as in Section 3.5.1. Since they can only rotate with respect to their parent frames, the linear velocities of the limb *frames* with respect to their parent frames are always zero ($\vec{v}^{\,\text{arm, canoe}} = \vec{0}, \vec{v}^{\,\text{paddle, arm}} = \vec{0}$), so only their orientations need to be updated. The difference is that the limbs have a maximum angle of rotation from their neutral orientation:

$$\theta \leftarrow \begin{cases} \theta + d_t\omega_i & \text{if } |\theta + d_t\omega| \leqslant \theta_{\text{sat}} \\ \theta & \text{if } |\theta + d_t\omega| > \theta_{\text{sat}} \end{cases} \tag{22}$$

where d_t is the simulation time step.

3.6 Flow feedback from canoe to velocity field

We also assume that the movement of the canoe and submerged paddles has a feedback effect onto the velocity field the objects move through. This velocity effect is due to the force of the canoe onto the water, and it has a similar form to force onto the canoe as calculated above. For all effective objects, the velocity field underneath each object of coordinate in world frame, $\mathbf{r}_w^{o,w}$, is updated as the following:

$$\mathbf{v}_w^{\text{water}}(\text{round}(\mathbf{r}_w^{o,w})) \leftarrow \mathbf{v}_w^{\text{water}}(\text{round}(\mathbf{r}_w^{o,w})) - c_s\mathbf{f}_{proj}(\mathbf{v}_w^{\text{water}}(\text{round}(\mathbf{r}_w^{o,w})) - c_k\dot{\mathbf{r}}_w^{o,w}, \hat{\mathbf{n}}_w) \tag{23}$$

where the "round" operator rounds the (x_w, y_w) coordinates of $\mathbf{r}_w^{o,w}$ to nearest integers within the simulation bounds, $\hat{\mathbf{n}}_w$ is the normal vector corresponding to $\mathbf{r}_w^{o,w}$, and c_s is a scaling factor related to the how much the movement of the objects moves the water beneath.

We note that although velocity flow vectors are affected by the objects' feedback veloci-

ties, they are permitted to travel through objects such as canoes and paddles, and so there are in general non-zero flow vectors underneath the canoe. This is because the boundary conditions for the simulation boundaries (in Section 3.1) cannot be directly applied to the canoe's ellipse. If they are, all the flow vectors around the canoe would have an orthogonal component of zero and no forces would be imparted onto the canoe. Thus, the boundary conditions for the canoe would need to account for many factors (mass of water, mass of canoe, etc.) instead of simply treating the canoe as an immovable object, and the integration between the fluid field simulation and the canoe simulation would need to be more complex. As well, having flow fields underneath the canoe may not be completely inaccurate, as water is flowing beneath the canoe in real-life scenarios.

3.7 Efficiency Improvements

After the simulation is functional, a timing analysis is done using cProfile to find any major inefficiencies [18]. This is because after comparing this environment to other environments in OpenAI gym (described later in Section 4.2), it is found that the environment is approximately 10 times slower than the other environments in terms of simulation steps per second. It can run at about 15 steps/s on a standard i5 laptop with 4 cores (though the simulation is single-threaded, so the number of cores has no major effects), whereas the other environments can run at 100 steps/s. The cProfile profiler analyzed, among other metrics, the total (including sub-functions) and cumulative (including sub-functions) time spent in each function. By looking at the results, we found that a few functions in the fluid simulation code written by [16] was taking the majority of the time. The biggest culprit was `bilinear_interpolation`, which was run for every fluid cell in the velocity field and ended up taking approximately 8 times the time as other functions. So, we improved the efficiency of these functions mainly by taking advantage of numpy vectorization and adding a few algorithmic improvements, including finding a way to set up the function such that bilinear interpolation can be vectorized to run over the entire velocity field matrix. With these improvements, the resulting program now runs at about 57 steps/s on the laptop. or about half the time as most of the standard gym environments.

3.8 Program flow

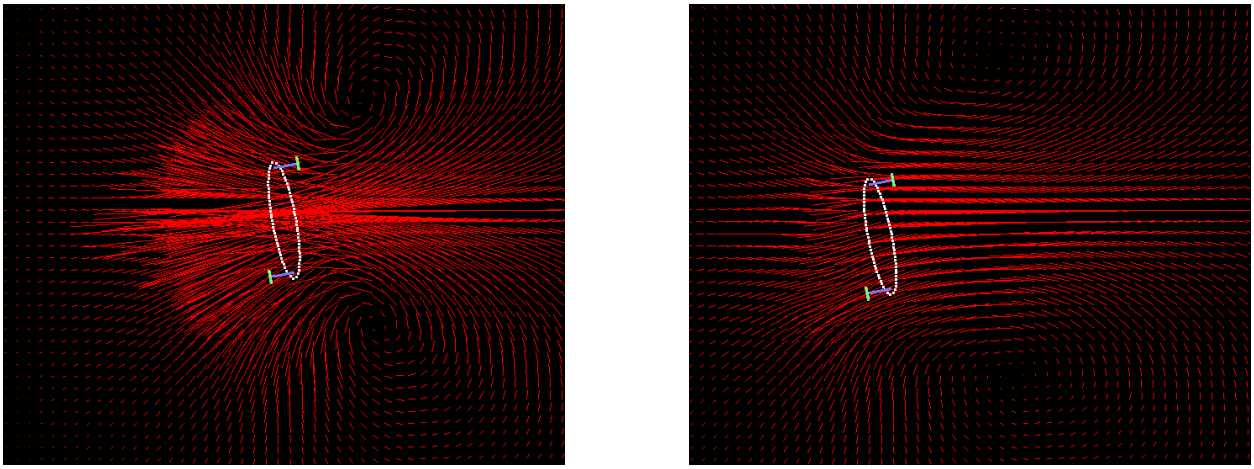
To summarize this section, the program flow of the simulation is as follows:

1. Update velocity field
   (a) Add field sources/sinks from user or in program. This could be to construct a steady-state flow (Section 3.1)
   (b) Add flow based on canoe movement as feedback flow (Section 3.6)
   (c) Propel the velocity field forward: diffuse flow, project flow, solve Navier Stokes (Section 3.1)
2. Controller computes a control command $(\omega_1', \omega_2', \omega_3', \omega_4')$, the angular velocities of the four limbs, using the necessary environment information.
3. Update canoe and limb states
   (a) Calculate wrenches (forces and torques) on the canoe's centre of mass based on the surrounding velocity field and the canoe's movement (Section 3.4)
   (b) Update canoe velocities and positions based on the calculated wrenches (Section 3.5.1)
   (c) Update paddle locations based on their new angular velocities set by the controller (Section 3.5.2)

3.9 Simulation Reults

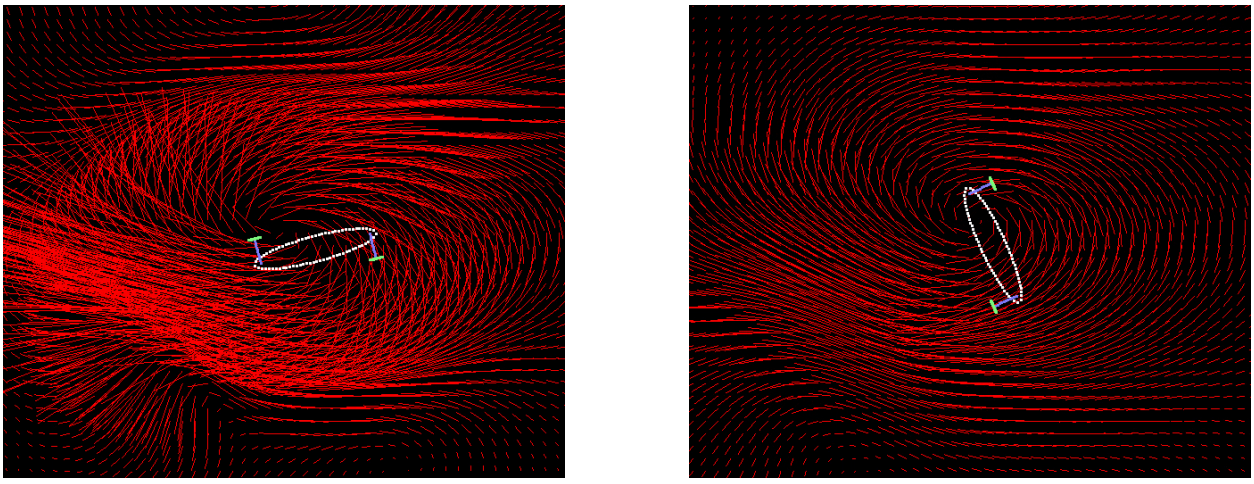
3.9.1 Motion verification

We verify the physics of the simulation by subjecting the canoe to flow currents and see whether the canoe movements are reasonable. The instantaneous flow currents are manually added using mouse inputs. As expected, the canoe exhibits a pure translation when a current is applied to the canoe's centre of mass (Figure 10), and exhibits a pure rotation when opposing currents are applied to the two ends of the canoe (Figure 11). We can see this in action in the supplement video of this thesis: `https://youtu.be/INZPWaw19tU`.



(a) $k = 1$ (b) $k = 2$

Figure 10: Screenshots of the canoe translating due to a flow current applied at its centre of mass.



(a) $k = 1$ (b) $k = 2$

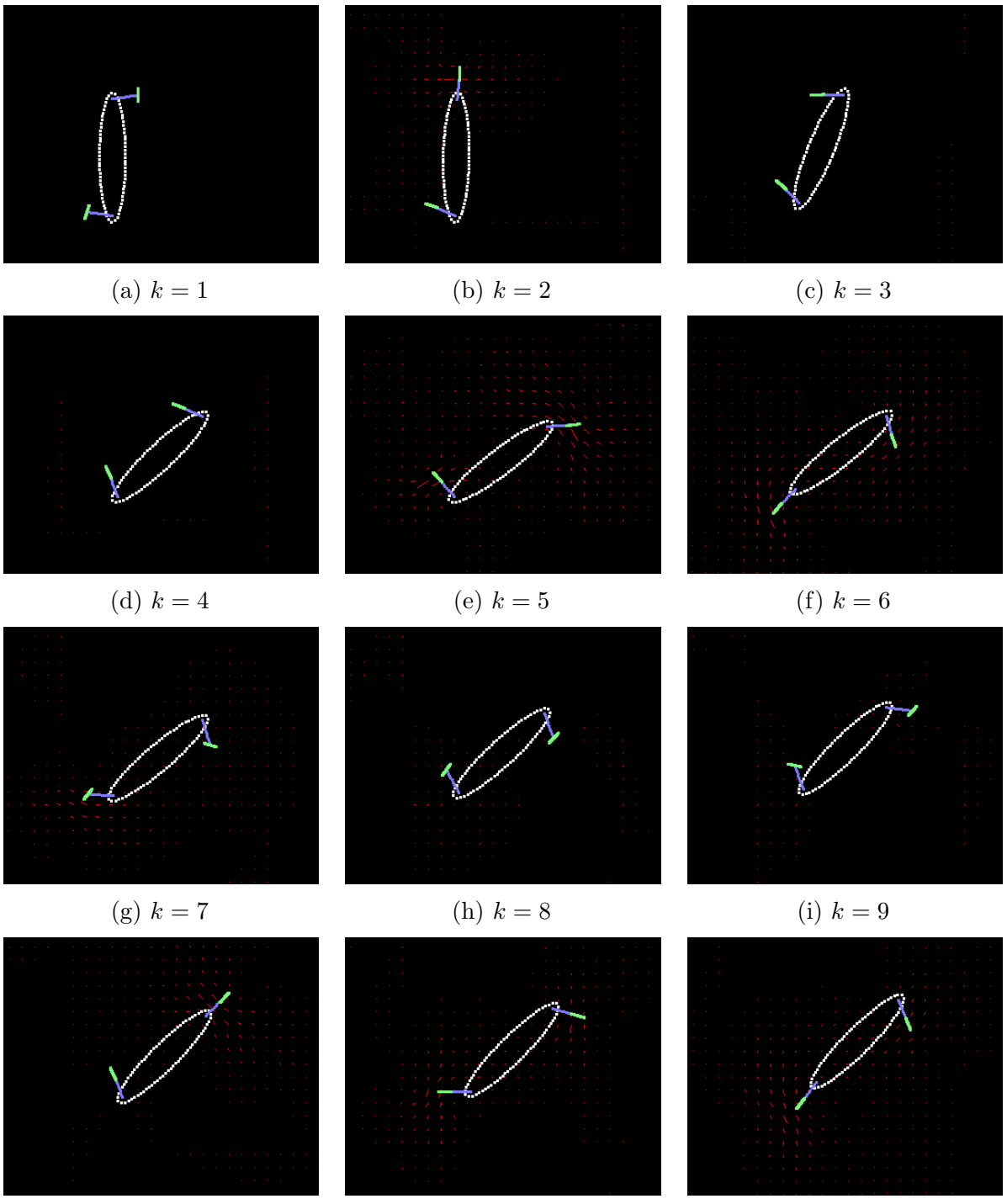
Figure 11: Screenshots of the canoe rotating due to flow currents applied at opposing currents at ends of the canoe

As well, the scaling factors $(c_k, c_f, c_\tau, c_t, c_s)$ are tuned such that the motions generated by currents and paddles are of reasonable speeds within the simulation space.

3.9.2 Open-Loop Control

We test the validity of the paddling physics, seeing if our reliance on the opposing forces from Newton's third law can propel the canoe in a reasonable manner. We designed an open-loop controller where the canoe starts at a set pose and paddles to a goal location under a flow field with no constant sources. We program a sequence of paddle movements that somewhat resembles the expected paddling motion for a canoe, and we can see the controller in action in Figure 12. The canoe first turns itself clockwise to roughly face the goal, which makes paddling forward easier. Then, both paddles start going through their periodic paddling motions to propel itself forward. The controller takes advantage of the paddle's angle with respect to its motion, angling the paddles such that they are orthogonal to the motions to maximize propulsion forces during paddle strokes, and parallel to the motions to minimize resistance forces during stroke resets. By repeating this periodic motion, the canoe is able to successfully reach the goal location (see Figure 13).

We then tested the same controller under a field with non-zero fluid flow (Figure 2). As expected, the controller does not account to the external influence of the field and the canoe drifts to one side. We concluded that as we have predicted, hand-crafting a controller that accounts for all scenarios would be difficult, and learning techniques may be more effective in overcoming disturbances. The open-loop controller would also act as a rough baseline controller for the RL controller for the next section.



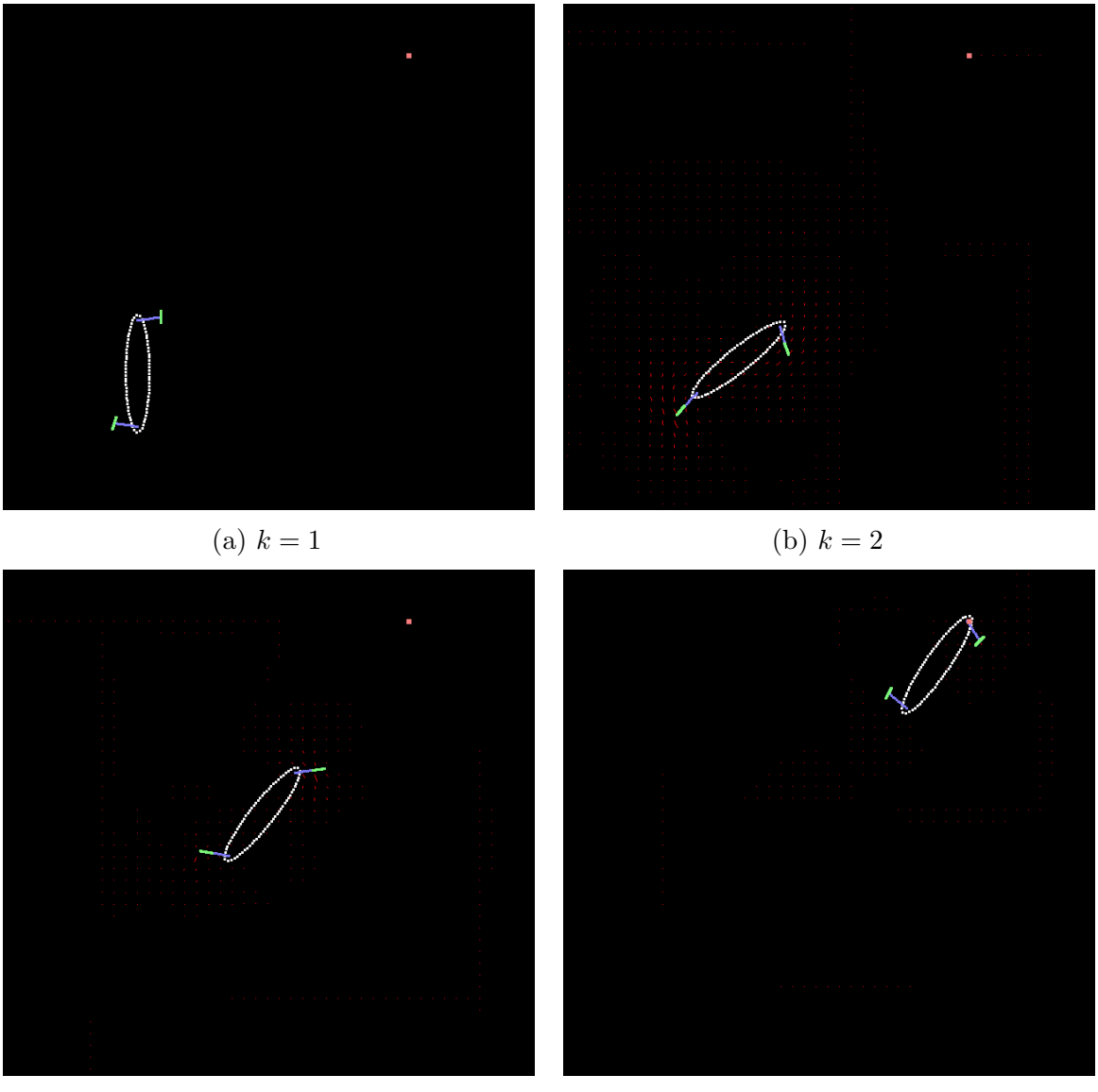
(a) $k = 1$ (b) $k = 2$ (c) $k = 3$

(d) $k = 4$ (e) $k = 5$ (f) $k = 6$

(g) $k = 7$ (h) $k = 8$ (i) $k = 9$

(j) $k = 10$ (k) $k = 11$ (l) $k = 12$

Figure 12: Screenshots of agent paddling with handcrafted open-loop controller. We can see it in action in the supplement video



(a) $k = 1$ (b) $k = 2$

(c) $k = 3$ (d) $k = 4$

Figure 13: Screenshots of agent reaching the goal with the open-loop controller

4 Reinforcement Learning (RL)

After constructing the simulation environment, the following sections of the thesis deals with controlling the canoe using reinforcement learning (RL). The goal is to train the agent to control the paddles such that the canoe propels itself to a desired goal location in the simulation (see Figure 14).

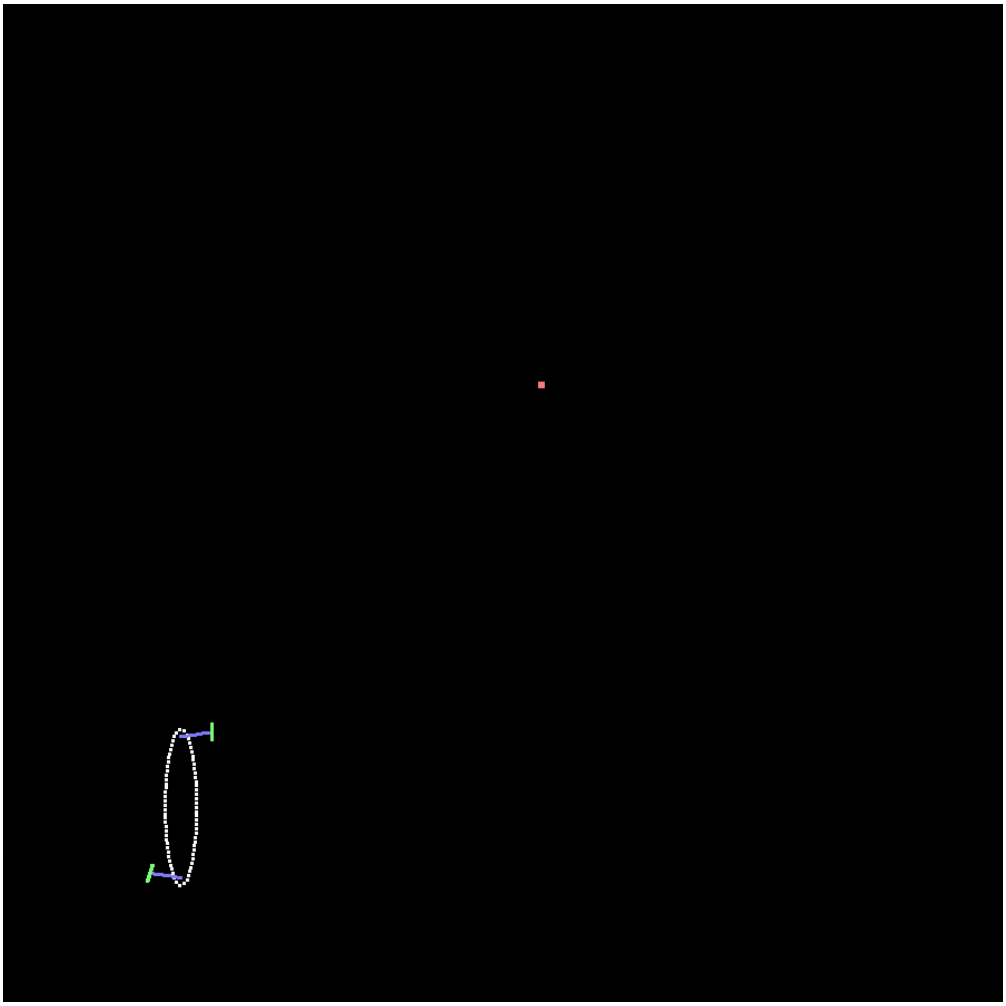


Figure 14: Screenshot of the entire simulation space. The pink dot represents the goal position to be reached by the canoe. The canoe in this screenshot represents the initializing pose and configuration for the Single Initialization Problem (defined in Section 4.1.5).

4.1 RL Problem Definition

We now formalize the definitions for the state, action, and reward of the RL problem that will encourage the controller to efficiently drive the canoe towards the goal. Appendix A.2 summarizes key concepts for RL and Deep RL.

4.1.1 State and Observation Definition

The state of the system is all the information that we need to predict the next state of the system. In other words, the state must be Markov. For our simulation, a reasonable state could be the position, velocity, and configuration of the canoe, and the values of the entire velocity field. The state is mainly a theoretical concept as the full state is never observed by the agent, but defining it is helpful for constructing the observations.

The observation is information available to the agent at each step to generate actions, and can be implicitely used in the process of estimating the state. We define our observation to be the subset of the state that excludes the velocity field information. Specifically:

$$\mathbf{s} = \begin{bmatrix} x & y & \theta & u & v & \omega & \theta_1 & \theta_2 & \theta_3 & \theta_4 \end{bmatrix}^T \tag{24}$$

We choose the observation to be the current pose (x, y, θ) and current twist (u, v, ω) of the canoe frame with respect to the inertial frame, as well as the four angles of arms and paddles $(\theta_1, \theta_2, \theta_3, \theta_4)$. This is a natural choice for the observation as the motion of the canoe is directly related to the canoe's twist, and the goal (and thus, the reward) is directly related the canoe's pose. As well, if the controller is to be used on a physical canoe, $x, y, \theta, u, v, \omega$ could found using standard localization techniques which could be done before feeding the results into the RL controller, and $\theta_1, \theta_2, \theta_3, \theta_4$ could be found with odometers on the joints.

Another advantage of having pose information (x, y, θ) within the observation is that the controller may be able to account for a non-zero steady state velocity field. If the field has constant sources, such as in the case of Figure 2, the controller may indirecly learn this information by having knowledge of the behaviour of the canoe in certain locations within

the field. This would likely yields better performance than treating such fields as random disturbances.

The four angles of the arms and paddles completely defines the configuration of the limbs. We excluded the angular velocities of the arms and paddles as we assumed they can be instantaneously changed by the controller.

We did not add velocity field information because to incorporate all the flow information of the velocity field, the number of inputs would be at least an order of magnitude larger, likely leading to much longer training times. We did consider adding to the observation information about the velocity field just surrounding the canoe. Since the interaction forces between the canoe and the surrounding velocity field is coupled, the controller could potentially account for the future forces onto the canoe if it can access the velocity field information. However, getting this information is likely difficult for a physical canoe, and the number of inputs would still be a few times larger. Thus, velocity field information was not added to the observation. If the controller is still robust to disturbances within the field without this information, it would further reinforce the effectiveness of model-free RL controllers.

Since the state space is continuous, using classical RL is intractable and we must use Deep RL techniques (see Appendix A.2).

4.1.2 Action Definition

The action, or the output of the controller, is defined to be:

$$\mathbf{a} = \begin{bmatrix} \omega_1 & \omega_2 & \omega_3 & \omega_4 \end{bmatrix}^T \tag{25}$$

which is simply the angular velocities of the arms and paddles. Since the limbs can change their angular velocities instantaneously, these four variables are all that is required to drive the canoe as desired.

We now consider the appropriate action space to use. Although the control inputs to the limbs are continuous ($\omega_i\prime \in \mathbb{R}$), it is possible to control the limbs with discrete action outputs from the RL controller by binning $\omega_i\prime$ to a set number of outputs. The minimal case would be two possible outputs for each limb: $\omega_i\prime = -\omega_{\text{sat}}$ to rotate the limb CCW and $\omega_i\prime = \omega_{\text{sat}}$ to rotate the limb CW. One level above could be to have three possible actions: rotate CCW ($\omega_i\prime = -\omega_{\text{sat}}$), stay stationary ($\omega_i\prime = 0$), and rotate CW ($\omega_i\prime = \omega_{\text{sat}}$). In general, let there be n possible outputs that discretizes $\omega_i\prime$ to values between between $-\omega_{\text{sat}}$ and ω_{sat}. Since the open-loop controller in Section 3.9.2 was able to achieve basic movements with the two-output discrete case, it is plausible that the RL agent would achieve the same or better performance with a discretized action space.

However, using a large discrete action space is likely to be expensive and difficult. For discrete actions, we must have Q-values for every possible action, and the optimal action would be the one with the highest Q-value. So, the number of output nodes of the Q-network would be all possible actions at each time step. For the canoe, there are 4 limbs, each with n possible actions. So, the number of output nodes is n^4, which scales poorly as n increases, and especially poorly if the number of limbs were to increase as well. Thus, to avoid training for a massive number of outputs nodes, we chose to output continuous actions.

4.1.3 End of Episode

We briefly discuss conditions for the end of episodes at training time. First, we end the episode when the agent reaches the goal, with a large positive terminal reward. We also wish to encourage the agent to stay within bounds, so we end the episode with a large negative terminal reward if the canoe drifts past the boundaries of the velocity field. These terminal reward would then need to be tuned, which we do in the next section (4.1.4).

As well, we decide to end the episode if the canoe floats for too long without reaching the goal. This is termed early stopping, and is helpful in diversifying training experience and preventing overfitting [19]. Although there are complicated early stopping criteria based

on gradient updates [19], for simplicity we use a fixed maximum episode length, which is a common practice (e.g. [20]). We set the maximum episode length to be 1024 steps, which is about 2 times longer than the open loop controller solution (described in Section 3.9.2), so the agent should have enough time to learn the optimal behaviour.

4.1.4 Reward Definition

We now engineer a reward function to encourage the canoe to "reach" the goal, where "reach" is defined as the canoe centre being within a small radius (4.5) of the goal. First, we give a large positive terminal reward for reaching the goal, and we give a large negative terminal reward for going out of bounds. Then, to encourage the canoe to paddle towards the goal, we engineer the intermediate rewards such that the canoe is encouraged to face and drive towards the goal. Specifically, we reward the agent for driving closer towards the goal, so we reward it for decreasing the distance-to-goal compared to the last timestep. We also reward it for adjusting its orientation such that its y-axis is closer to the line connecting the canoe's current position and the goal, compared to the last timestep.

The reward is chosen to be:

$$R_t = \begin{cases} 300 & \text{if } d_t \leqslant 4.5 \\ -100 & \text{if out of bounds} \\ -c_r \Delta d_t - c_\theta \Delta \theta_t & \text{otherwise} \end{cases} \tag{26}$$

where c_r, c_θ are weighting factors, d_t is the distance-to-goal, Δd_t is the change in distance-to-goal, and θ_t is the orientation offset, and $\Delta\theta_t$ is the change in orientation offset:

$$d_t = ||\mathbf{r}_t - \mathbf{r}_g||_2, \quad \Delta d_t = d_t - d_{t-1}, \quad \Delta\theta_t = \theta_t - \theta_{t-1} \tag{27}$$

We define $\theta_t = \min(|\alpha_t|, |\pi - |\alpha_t||)$, where α_t is the angle deviation of the y-axis from the line connecting the canoe's CoM and the goal position (see Figure 15).

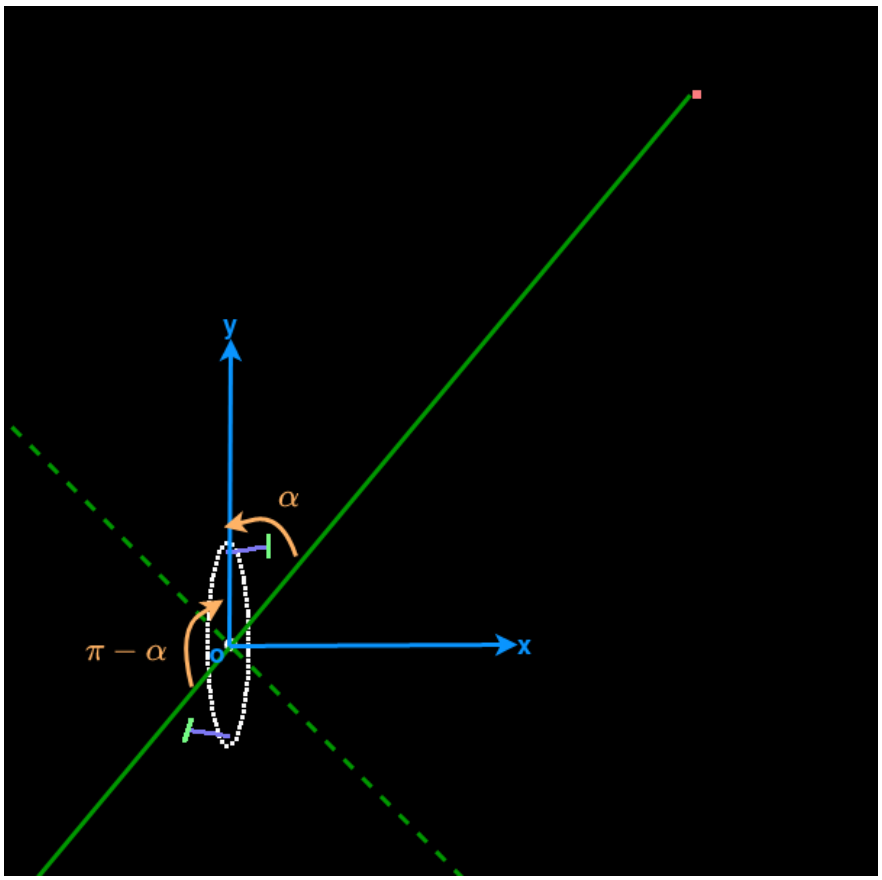


Figure 15: Definition of α_t for defining $\theta_t = \min(|\alpha_t|, |\pi - |\alpha_t||)$ for the reward function. Here, θ_t would be minimum (0) if the y-axis is aligned with the solid green line, and maximum ($\pi/2$) if the y-axis is aligned with the dotted green line.

The specific values of the weighting factors are tuned such that both terminal rewards are always fairly large compared to the total intermediate rewards. They are also tuned experimentally to weigh the trade-off between distance and angle adjustment. After tuning, the factors are set to be $c_r = 5, c_\theta = 80$. With these values, the largest possible total intermediate reward for distance-to-goal in an episode (achieved when the canoe starts at the corner farthest from the goal) is about 250, which is about 3 times larger than the largest possible reward of 90 for adjusting its orientation offset (starting at an offset of π).

4.1.5 Agent Initialization

The eventual goal of this project is to train the agent to the goal starting from any pose. We will start off by training the agent to solve a simpler problem: paddle to the goal from initializing at a fixed pose of $(x_0 = 22, y_0 = 23, \theta_0 = 0)$ (see Figure 14). We will call this the Single Initialization Problem. Then, we will attempt to generalize by picking the initial pose randomly each episode, done by uniform sampling:

$$X_0 \sim U(0 + p, N - p), \quad Y_0 \sim U(0 + p, N - p), \quad \Theta_0 \sim U(-\pi, \pi) \tag{28}$$

where N is the size of simulation, and p is a padding parameter to ensure the canoe is not initialized out of bounds too often. p is experimentally set to be 12. We will call this the General Initialization Problem.

4.2 Choice of RL library

There are a number of RL libraries to choose from. The major ones include OpenAI's Baselines [21], Stable Baselines [22], Tensorflow RL Agents [23], Tensorforce [24], and KerasRL [25], and there are a number of smaller libraries (e.g. pyqlearning [26]). There are many articles comparing the major libraries in terms of criteria such as implemented state-of-the-art algorithms, ease of use, and using customizable environments, such as this article [27]. We have selected to use Stable Baselines, a fork of OpenAI's official Baselines library, due to its documentation, easy use of custom environments, tensorboard support, and many other useful features such as gym environment checkers. It also enables us to use RL Baselines

Zoo [28], which contains a collection of trained RL agents on standard Gym environments and the hyperparameters used. Since the library's main RL algorithms are ported from the robust Baselines library, it is also fairly up to date in terms of the implementation of state-of-the-art algorithms. Both Baselines and Stable Baselines uses OpenAI gym for environments, and allows training on several pre-existing environments in the gym library [29]. For using custom environments, they simply needs to be packaged into a gym environment class, and we will do this next for our canoe simulation.

4.3 OpenAI Gym Environment Setup

To train the environment with Stable Baselines (or Baselines), the simulation needs to be set up as a custom environment from the library Gym from OpenAI [29]. It is set up as an installable pip package (`gym_canoe_sim0`) which registers the custom environment (`canoe_sim-v0`) into the gym registry upon installation. This was done by following tutorials such as [30]. Note that if later running the Stable Baselines training scripts fails, adding the line "import gym_canoe_env" to the source code fixed most of the problems.

The simulation also needs to be rewritten into an inherited class of Gym, which require definitions for specific variables and methods. Most of this is simply transferring information from Section 4.1 with some additional restrictions on variable scope. For completeness we define the required variables and methods below:

- `action_space = gym.spaces.Box(low=-1, high=1, shape=(4,))`
  The action space is set up as a 4-vector continuous space ranging from -1 to 1, the default range for the algorithms. (-1, 1) is the default range for Baselines and Stable Baselines, so all of the four paddle velocities are scaled by the maximum angular velocity to be within this range. Also, some algorithms such as DDPG can only output symmetric actions, and luckily our actions are already symmetric.

- `observation_space = gym.spaces.Box(low=-5, high=5, shape=(6,))`
  Similar as the action space, the observation space is set up so the observations are scaled to be between of -5 to 5 for all values. The x,y positions of the canoe are scaled

and translated such that 0 corresponds to the centre, and -5 and 5 corresponds to the edge of the simulation. The orientation is scaled from its range of $(-\pi/2, \pi/2)$ to $(-5, 5)$ as well. The linear and angular velocities are scaled by the maximum (saturation) velocities, and the paddle orientations are scaled by the paddle angle limits.

- `step(action)`: `return (observation, reward, done, debug)`
  - `action`: the new action to be taken in this step, in a variable type corresponding to `action_space` (see Section 4.1.2)
  - `observation`: the new observation after taking this step, in a variable type corresponding to `observation_space`. (see Section 4.1.1)
  - `reward`: the reward received from taking this step. (see Section 4.1.4)
  - `done`: whether the episode has ended after taking this step (see Section 4.1.3)
  - `debug`: left as {} as it is not used for actual training.
- `reset()`: `return observation`
  Resets the environment to a new initial state, and returns the observation from that state. Depending on what we want, the new initial state can be the same pose or a new randomly-sampled pose (see Section 4.1.5).
- `render(mode)`: `return None`
  For rendering the environment during testing.

4.4 Choice of RL Algorithms

With the RL problem set up, we see which RL algorithms are appropriate to train our canoe-paddling agent. Deep RL is a natural choice over the classical RL based on dynamic programming as deep techniques can handle continuous state spaces (see Appendix A.2). As we also have a continuous action space, a natural choice is to use an actor-critic framework (see Appendix A.3). Normally, though, using actor-critic by itself is unstable, so we use more advanced RL algorithms provided in Stable Baselines. We discuss the general types of agents or algorithms, then summarize the algorithms we attempted for the project.

4.4.1 Agent Type

For Deep RL, we can use either a policy-based (on-policy), value-based (off-policy), or model-based RL agent [31]. Model-based agent learns a transition model of the environment to predict the future states of the agent. The algorithm learns a transition model of the agent state by constructing and estimating the relevant hidden environment states, and then uses this transition model to "looks ahead" and find optimal actions with planning algorithms. Since the objective of the thesis is to develop and RL controller for a nonlinear, highly complex environment, we choose not to use model-free RL agent. It is also unlikely to be successful as the simulation model is already rather complex.

In contrast, both policy-based and value-based RL are model-free. Policy-based RL is on-policy, meaning that it directly optimizes its policy by learning the value function (Q-value) for the current policy and improving it via ascending the policy gradient [32]. For on-policy methods, the agent uses the current policy to both generate behaviour and evaluate the Q-value of states. Value-based RL such as Q-learning is off-policy, meaning that it optimizes the policy indirectly by learning the Q-value for the states under any policy ([33], [34]). When evaluating a state's Q-value, the agent greedily uses the Q-value of all possible actions instead of taking the expected value based on actions from the current policy. In comparison, policy-based methods tend to be more stable and have less failure modes, while value-based methods can be more sample-efficient [1].

In Stable Baselines, there are numerous algorithms available for model-free RL. See Figure 16 for a classification of algorithms. Only a few of the available algorithms in Stable Baselines support continuous action space: DDPG, A3C, A2C, ACKTR, PPO, SAC, TRPO, TD3. We describe a few of these algorithms that we attempted to use below, though the blog [35] by Lilian Weng offers great summaries of these and other algorithms available in Stable Baselines.

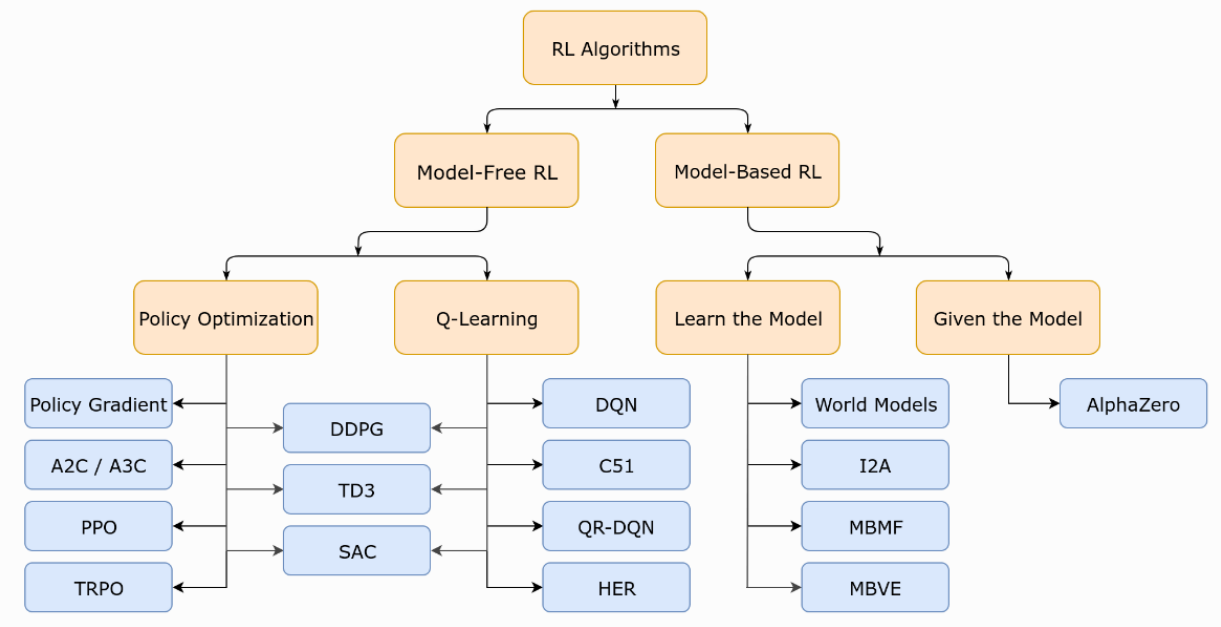


Figure 16: A taxonomy of a few representative algorithms in RL, including those available in Stable Baselines [1]. Note that DDPG, TD3, and SAC are placed in the middle for they have on-policy characteristics, but they are closer to the off-policy classification.

4.4.2 Deep Deterministic Policy Gradient (DDPG)

DDPG solves the stability issue of the vanilla actor-critic by introducing a replay buffer to store current interactions. The agent trains from interactions sampled from the buffer instead of the current interactions, thus breaking the dependence between interaction samples. If interested, please refer to the original paper [36], or a slightly longer explanation in Appendix A.4.

4.4.3 Twin-Delayed DDPG (TD3)

TD3 is a direct successor to DDPG which aims to help with some of the stability problems of DDPG [37]. The most significant change is clipped Double-Q learning: TD3 uses and updates two critic networks, and uses the smaller of the two Q-values for evaluation. Other changes include using delayed policy updates, and adding noise to the target action. These changes help address the failure mode of DDPG due to Q-value approximation errors.

4.4.4 Asynchronous Advantage Actor-Critic (A3C)

A3C is a classic policy gradient technique with modifications for asynchronous parallel training [38]. A3C contains one critic network to learn the value function, and several actor networks to explore asynchronously, using gradient ascent to directly optimize the policy. Each actor network's parameters are synchnronized with a set of global parameters when a condition is met. Due to the simultaneous exploration by multiple agents, the actor policy is able to converge without having a replay buffer.

4.4.5 A2C

A2C is a simplification of A3C where asynchronous updates are replaced by synchronous updates resolved by a global coordinator, henced the reduction to two A's for the acronym [39]. A2C is more efficient in GPU usage than A3C, and the empirical results show that the noise from asynchronous updates in A3C did not yield improvements [39], contrary to the predictions in the A3C paper. Thus, the synchronous version of the algorithm has been gaining popularity.

4.4.6 Proximal Policy Optimization (PPO)

PPO further improves training stability upon A3C and A2C by ensuring that the policy does not change too much [40]. With this, PPO also reduces a bit of reliance on finding the optimal learning rate and other hyperparamters. It does this by maximizing a clipped surrogate function instead of just doing gradient ascent on the policy. The authors showed that PPO achieved similar as TRPO, another RL algorithm which enforces a KL divergence constraint for a similar purpose, while PPO is much simpler to implement and tune.

In Baselines and Stable Baselines, there are two versions of PPO: PPO1 (the original version from the paper) and PPO2 (a new GPU-enabled implementation) [41]. PPO2 also have other minor modifications, such as normalized advantages and clipped value functions [42]. PPO1 is mainly kept for historical purposes and it is recommended to use PPO2, so we will use PPO2 for training.

4.5 Finding hyperparameters

In general, Deep RL techniques tend to be quite sensitive to hyperparameters [43], so we needed a suitable set of hyperparameters to start with. We developed intuitions for tuning hyperparamters from looking at pre-tuned hyperparameters for Gym environments in RL Baselines Zoo. The most similar environment may be the BipedalWalker-v2, an environment where a bipedal robot must learn walk along a flat path [29]. As well, we noticed that a linear schedule is used for learning rates and other parameters, meaning that selected parameter start at set values and linearly decrease to 0 as training steps goes from 0 to "max_steps". After attempting to tune for a while, in the end fixing other problems in network architecture (described in Results in Section 4.8.1) resulted in a much better agent for the single-initialization case. More tuning attempts led to visible but minor performance increases. The following parameters for PPO2 were the ones changed from default to produce a functional agent for the single-initialization case.

- `gamma=0.99`

  Discount factor (γ). A higher γ would represent less time pressure on receiving, and thus also represents higher certainty in future rewards [31].

- `n_steps=512`

  The number of steps to run each environment per update. From looking at this value in the Zoo for the Gym environments, `n_steps` tends to be higher for more complicated environments, possibly because the reward function is more complicated to learn.

- `nminibatches=32`

  The number of minibatches per update. Like `n_steps`, this tends to be higher for more complicated environments.

- `learning_rate=linear_schedule(5e-4)`

  Learning rate for gradient descent/ascent. The same learning rate is used by both actor and critic networks. More complicated environments tend to have higher learning rates, possibly to push the agent out of local minimas.

- `cliprange=linear_schedule(0.2)`

  The range for which the surrogate objective function can be updated without being clipped. Higher values would allow for larger deviations from the current policy. A linear schedule is used in the RL Zoo, likely to allow for larger updates at the start of training and smaller updates at the end.

There were other parameters that were less intuitive to tune, and they were either directly transferred from similar environments or left as default values: `ent_coeff=0.001`, `lam=0.95`, `vf_coef=0.5`, `max_grad_norm=0.5`, `noptepochs=4`. Luckily, the agent was able to achieve its goal with these values, likely because PPO is less sensitive to hyperparameters than other algorithms.

4.6 Network Architecture

In Stable Baselines, all of the algorithms uses the same network architecture for both the actor and the critic. There are 6 network types to choose from:

- `MlpPolicy`: Multi-layer perceptron for feature extraction
- `MlpLstmPolicy`: MLP with a layer of long-short-term memory
- `MlpLnLstmPolicy`: `MlpLstmPolicy` with the LSTM layer normalized
- `CnnPolicy`: Convolutional neural network (the Nature CNN)
- `CnnLstmPolicy`: CNN with a layer of long-short-term memory
- `CnnLnLstmPolicy`: `CnnLstmPolicy` with the LSTM layer normalized

We choose a simple `MlpPolicy`. LSTMs are likely not useful as the observations are approximately Markov, so having past observations would not yield more information. We also do not use convolutional layers since the observations have low dimensionality and are not spatially invariant. For the network architecture, we experimented with different sizes and activations (see Section 4.8.1 in Results) and ended up using 4 hidden layers of $64 \times 256 \times 64 \times 16$ neurons with ReLU activations.

4.7 Training Environment Setup

We train the agent on an NVIDIA DGX Station, which has a 20-Core Intel Xeon E5-2698 v4 2.2 GHz CPU, 256 GB of system memory, and 4 Tesla V100 GPUs (128 GB of GPU memory). It is set up with 40 virtual cores. Under these conditions, training parallelizable algorithms (PPO, A3C, A2C) with 32 cores is very fast, taking about half an hour for 1 million timesteps. As well, we found that the training speed was not reduced even when the networks are made orders of magnitude larger, likely due to the large GPU memory.

4.8 Training and Results

With the problem set up, we now attempt to train a functional agent solving the two problems in Section 4.1.5, varying up algorithms, environment settings, and other parameters when necessary. Note that this section is less methodical as it is more focused on finding a training method which is successful, and it is not as focused on the evaluation and comparison of different methods.

4.8.1 Solving the Single Initialization Problem

We found that training with parallelizable algorithms (A3C, A2C, PPO2) is much faster than training with the other algorithms. It takes about two hours to train these for 3 million steps using 32 cores, while it takes multiple days to train for 500,000 steps with the other algorithms. Of these parallelizable algorithms, PPO2 is the most advanced and the least sensitive to hyperparameters. For these reasons, our first algorithm to try is the PPO2. In fact, if PPO2 did not work, our attempts with using the other parallelizable algorithms had similar outcomes: if PPO2 failed, the other algorithms also failed.

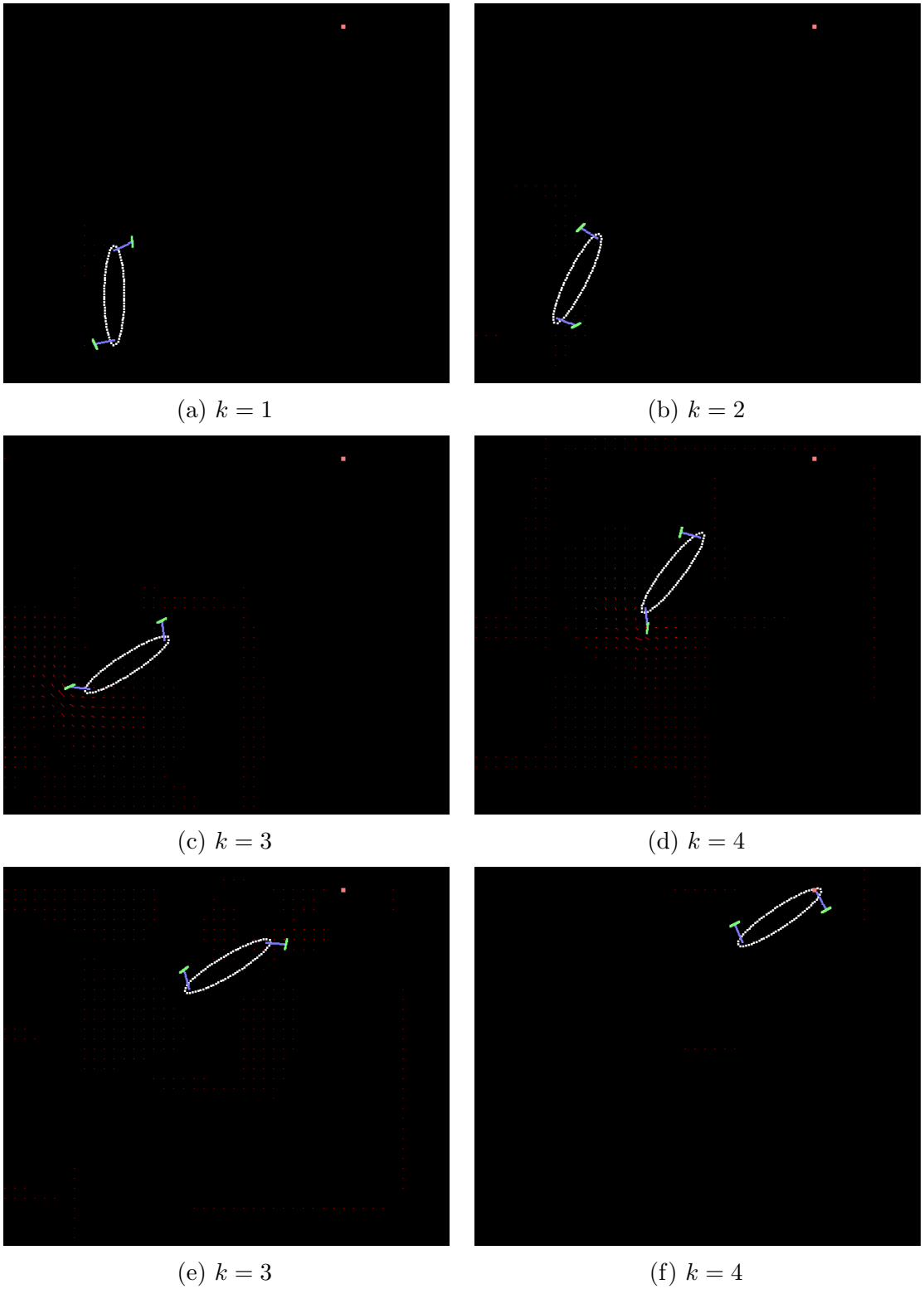
We started off using the default network for `MlpPolicy`, which has 2 layers of 64×64 with tanh activations, and trained for 1.18 million timesteps. From looking at the tensorboard graphs, we see that the value loss was converging to 0 but the policy loss saturated at around -0.004 (see Figure 20 in Appendix A.5). We ran the agent and found that, as expected, its behaviour is also not ideal. The canoe jitters its paddles and exhibits little of the

periodic motions that we expected.

After trying out multiple changes, we found that using a much larger network greatly helped with agent quality with practically no increase in training time on the DGX Station. We also found that using ReLU activations is much more efficient in reducing the losses. In the end, we used a MLP with ReLU activations and 3 hidden layers of neuron numbers 64, 256, 64. After training for 1.09 million timesteps, both the value and the policy losses converged to 0 (see Figure 21). We tested the agent and found that it is visibly paddling to the goal. Also, there are interesting similarities between the agent's behaviour and the open-loop controller. Like the hand-crafted solution, the agent first turns to roughly face the goal, then uses a periodic motion to propel itself forward with its back paddle, or the paddle farther away from the goal (see Figure 17). The paddling motion is similar to that of the open-loop controller in that it attempts to maximize forces by keeping the paddle orthogonal to the motion while doing a paddling stroke, and minimize resistances by keeping the paddle parallel to the motion while resetting the paddles. The difference is that the agent managed to learn a much simpler stroke that uses the entire range of motion of the paddles instead of just half (see supplement video).

One flaw, however, is that the agent only uses the back paddle to propel itself and just jitters its front paddle. This could be because the front paddle requires a more complicated paddling motion where the paddle needs to stop at the half-way point, as in the case for the open-loop controller. After running it a few times, we also see that the agent actually often misses the goal. It often recovers by turning towards the goal again, but sometimes it fails and drifts out of bounds (see video). We would expect that solving the General Initialization Problem would help with this issue, as then the agent would have the ability to reach the goal from any pose.

We now compare the discounted cumulative reward achieved by the agent to that of the baseline open-loop controller, using the same reward function as before (Equation 26 in Section 4.1.4). At each step, we calculate the undiscounted reward and discount it with γ,



(a) $k = 1$ (b) $k = 2$

(c) $k = 3$ (d) $k = 4$

(e) $k = 3$ (f) $k = 4$

Figure 17: Screenshots of the PPO2 agent reaching the goal for Single Initialization. This can also be seen in the supplement video

which starts at 0.99 and continuously decays as time goes on. The results for the open-loop controller and 10 trials with the trained PPO2 agent are shown in Table 1.

Run #	**Cumulative Reward**	**Episode Length**	**Reached Goal?**
Open-loop	63	343	Y
1	44	875	Y
2	51	1025	N
3	48	651	Y
4	45	497	Y
5	49	593	Y
6	51	721	Y
7	50	497	Y
8	53	528	Y
9	47	1025	N
10	44	1025	N

Table 1: Cumulative rewards for 10 runs with PPO2 agent and the open-loop controller for the Single Initialization Problem. These runs are in the same order as the 10 runs in the supplement video

We see that the open-loop controller achieves a higher reward than the trained PPO2 agent, although the difference is not too big. However, the trained agent often reaches the goal much slower than the open-loop. This shows that even though the behaviours between the two are somewhat similar, the agent could still be more efficient.

One interesting observation is that for the trained agent, the success or failure in reaching the goal in Table 1 seems to be only loosely correlated with the cumulative reward. That is, even if the agent fails to reach the goal, its cumulative reward is not much lower than that of the other trials where the goal has been reached. This is because by the time the agent reaches the goal in the "successful" trials, γ_t has decayed to be making any incoming rewards have negligible impacts on the cumulative reward. Therefore, the success of an agent in reaching the goal is not well reflected in its cumulative reward. On the other hand, open-loop controller reaches the goal much sooner, and the terminal reward notably increased its cumulative reward from 56 to 63. This suggests that although the discounted cumulative reward can be used for rough comparisons of agent performances, it is perhaps

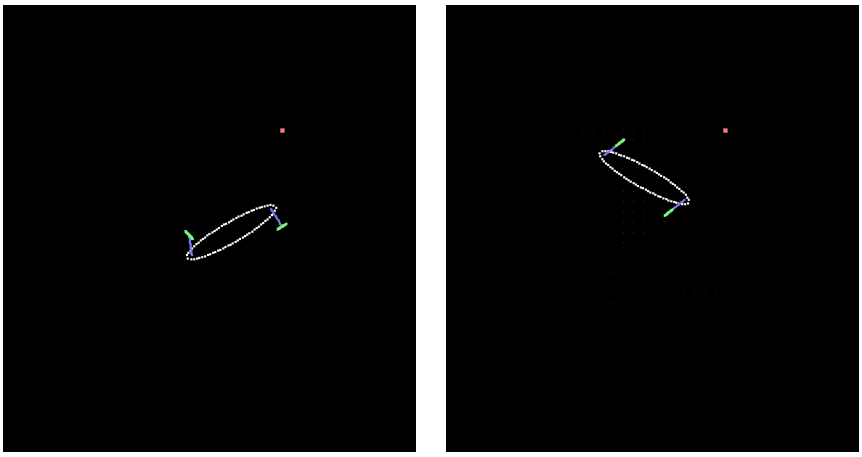
not the best metric for evaluating the learning progress of suboptimal agents.

4.8.2 Solving the General Initialization Problem

Unfortunately, transferring the success from the Single Initialization Problem to the General Initialization Problem proves to be quite difficult. We tried training for 5.42 million timestep, which is 5 times steps as the Single Initialization case. Still, the value loss does not converged to 0 (see Figure 22). The policy loss did converge to 0, but that does not guarantee success as the actor network cannot be trained with an erroneous critic network. When testing the agent, we see that the canoe mostly just freezes up and occasionally swings and jitters its paddles randomly. Even when initialized very close to the goal, the canoe just floats until episode termination, which occurs either by reaching max_steps or by drifting out of bounds (see Figure 18).

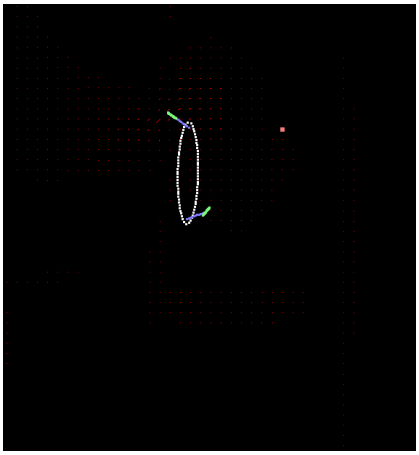
We are not sure of the exact reason for this training failure. The agent is able to learn to paddle to the goal from a far pose unaligned to the goal in the Single Initialization case, so paddling from other initial poses should be easier. It is possible that the networks are still too small to capture the general policy, but a network of layers $64 \times 256 \times 64 \times 16$ is already very large, much larger than the networks used to train default Gym environments in RL zoo. Also, it is the value function that is not converging, but the reward definition is unchanged from the Single Initialization Case. Of course, there is always possibility that the failure is simply due to a bug in setting up the RL problem, though we have not discovered bugs as of yet.

We tried making small changes, such as changing the max_steps for early stopping, using different algorithms, tuning hyperparameters, and minor tuning of reward function values. For trying out different algorithms, we were only able to try out using one non-parallelizable algorithm due to time constraints, and could train for a smaller number of timesteps. We chose to use TD3 due to its superior performance compared to DDPG. After training for 178 thousand timesteps, however, neither the two value networks nor the policy network converged (see Figure 23). This is not too surprising, we spent much less time on tuning

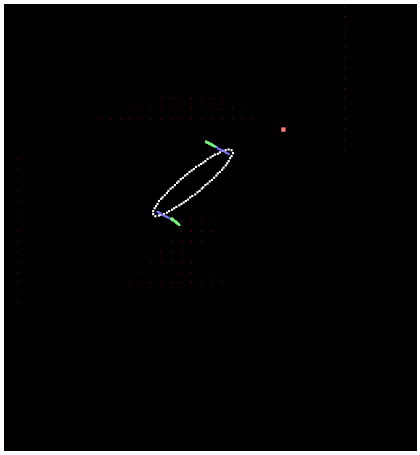


(a) $k = 1$

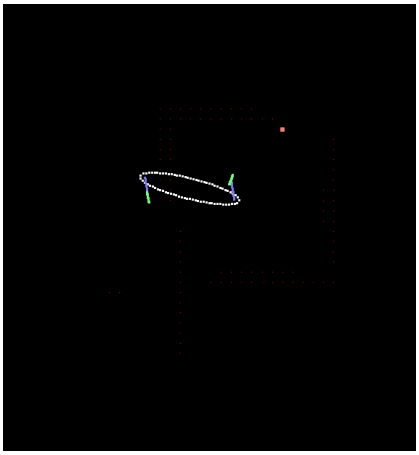
(b) $k = 2$



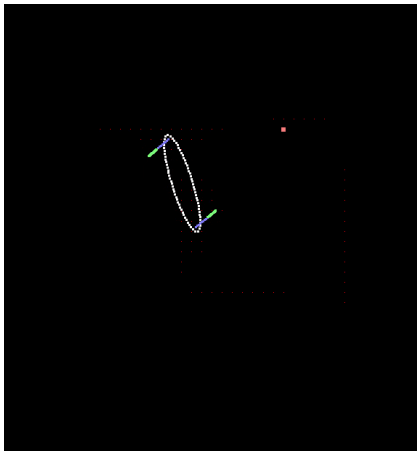
(c) $k = 3$



(d) $k = 4$



(e) $k = 5$



(f) $k = 6$

Figure 18: Screenshots of the failed PPO2 agent for the General Initialization Problem. Even when initialized near the goal, the agent seemingly just rotates randomly.

TD3 as we did on PPO2 due to the long training time. The resulting behaviour is not much better than the agent trained with PPO2.

There were other changes we wanted try with these RL algorithms, but due to time constraints they are instead outlined in the "Future Work" section (6.1). Before we moved on, however, we tried using Hindsight Experience Replay, a different kind of RL algorithm which requires some explanation. We describe the process in the following section.

5 Experimenting with Hindsight Experience Replay (HER)

In attempt to train a generalizable agent, we tried using an extension algorithm called Hindsight Experience Replay (HER), which should be theoretically beneficial for the training problem. HER requires a slight modification to the agent environment, so we describe the process separately here. Unfortunately, our experiences with HER did also not help with the generalization problem so far, but we still outline the process below as it may be expanded upon for future work. We first give an overview of the algorithm, then describe our set up and the results.

5.1 HER Overview

Hindsight Experience Replay is extension technique on top of off-policy RL methods, and offers a method to train an agent to reach goals without the use of reward engineering. It assumes that there are specific goal states in an environment is trying to reach, e.g. a manipulator pushing a box to a specific location. It also assumes that each agent state has an implicit goal state that can be easily found from the state or observation. For the manipulator example, the current x,y coordinate of the box would be the current goal state.

Without reward engineering, the rewards would be very sparse: zero or negative rewards for all states that are not the goal, and a positive value for goal states. Trying to train an

agent would be very difficult with standard RL methods, as the agent would only be learning from whether the goal is reached or not reached for each episode. On the other hand, HER attempts to learn from the failed episodes with the following mentality: although the agent has failed to reach the desired goal, it has "succeeded" in reaching the current goal, and can learn from this episode for reaching the current goal again. It can also use this experience to learn about reaching any other goal, which we could sample from the replay buffer. So, at each step, instead of storing just one transition of $(s_t||g, a_t, R_t, s_{t+1}||g)$ into the replay buffer, where g is the desired goal, it samples a number of goals, g', with a sampling strategy and stores all the transitions $(s_t||g', a_t, R_t, s_{t+1}||g')$ into the buffer as well. If the goal states are continuous, the agent can extrapolate among goal states with the neural network, and the outcomes on reaching the desired or sampled goals become continuous "degrees of success". The agent learns how traverse between goal states so that even if it reaches the desired goal a few times, it can still navigate through a series of goal states to reach the desired goal. See the original paper [44] for more information.

5.2 Using HER for Canoe Simulation

The main advantage of HER is that it eliminates the need for complicated reward engineering by connecting value function and goal states, and the authors of HER showed that HER with sparse rewards outperformed DDPG with an engineered value function [44]. In our environment where a change in the reward function led to large behavioural changes, HER could eliminate the need to find the optimal reward function. HER also allows the agent to learn how to reach any intermediary goal states from this training process, and can be trained to reach multiple goals without relearning the value function each time. This yields great potential if the canoe needs to reach multiple or random goals in the simulation. Even for the single-goal case, the authors of the paper showed that HER+DDPG outperformed DDPG with an engineered reward function [44]. However, the quality of the engineered reward in this evaluation could be questionable, and it is unknown whether these results generalize to different types of environments. Nevertheless, HER appears to be a suitable choice for our environment for these reasons.

Also, I feel that HER is especially useful if the mapping from observation to goal requires limited information from the observation. In our case, the goal is simply a small subset of the observation, and using HER in a way should explicitly tell the network to only care about manipulating the canoe's position to reach the goal state, which should make training easier. However, this is much more speculative.

5.3 Goal environment setup

In Stable Baselines, HER can be wrapped around the off-policy methods (DQN, DDPG, SAC, TD3). The environment must first be an inherited class of gym.GoalEnv. Compared to gym.Env, the main modifications are the observation and the observation space, which are augmented with an explicit definition of current and desired goal. For us, we defined the goal as the x,y position of the canoe centre

```
observation_space = gym.spaces.Dict({
    'observation': spaces.Box(low=-5, high=5, shape=(10,)),
    'achieved_goal': spaces.Box(low=-5, high=5, shape=(2,)),
    'desired_goal': spaces.Box(low=-5, high=5, shape=(2,))
})}
```

and the observation at each step is similarly a dictionary with the three fields.

The reward function is also changed to reflect the sparse reward environment used by the authors. The new reward is defined as:

$$R_t = \begin{cases} 300 & \text{if } |\mathbf{r} - \mathbf{r_g}| \leqslant 4.5 \\ -100 & \text{if out of bounds} \\ -0.1 & \text{otherwise} \end{cases} \tag{29}$$

The discount factor is kept at $\gamma = 0.99$. Using this reward function, even if the agent believes it will never reach the goal, its total discounted future reward for staying inside simu-

lation boundaries is at the minimum:

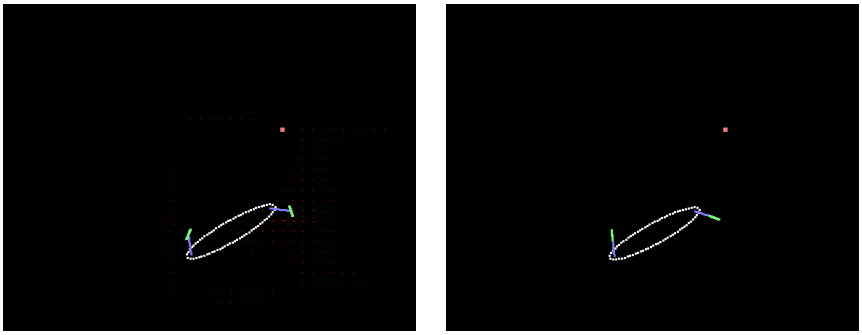
$$R = \sum_{t=0}^{\infty} \gamma^t(-0.1) = (-0.1)\frac{1}{1-\gamma} = (-0.1)(100) = -10 \tag{30}$$

This reward is still much higher than the reward of going out of bounds. Thus, the agent should always try to stay within the boundaries, and move towards the goal if possible.

5.4 Training and Results

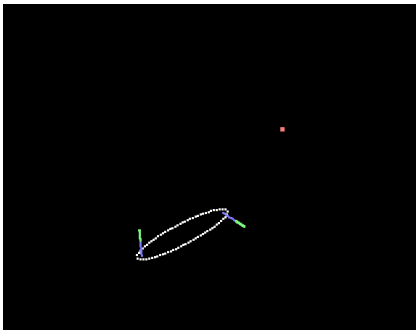
As HER is also not a parallelized algorithm, the training process is very long, and we were only able to try training HER once due to time constraints. We tried using HER on top of TD3 and using random pose initialization to see if it can solve the Generalization Initialization Problem. HER+TD3 takes even longer to train than TD3 alone, so we tried training for only 63 thousand timesteps. Unfortunately, the training process was also not successful. We see in the Tensorflow graphs that the both the policy and the value losses are actually diverging (Figure 24). Similar to with the other algorithms (results presented in Section 4.8.2), during test time the agent simply locks up and stays in one configuration for the rest of the episode, often drifting out of bounds (see Figure 19).

Similar to before, it is unknown exactly why the agent fails to learn. It is likely that the same reason for causing training to fail with the other algorithms, whatever it may be, is causing the training failure with HER.

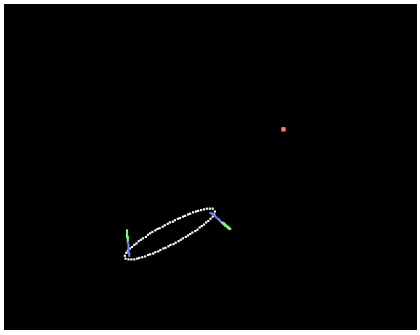


(a) $k = 1$

(b) $k = 2$



(c) $k = 3$



(d) $k = 4$

Figure 19: Screenshots of the failed HER+TD3 agent for the General Initialization Problem. The agent barely moves the paddles and just drifts.

6 Future Work

This section outlines some suggestions for future work. We first suggest a few methods to help achieve functionality under any initialization in Section 6.1. We then ouline a number of meaningful extensions after general functionality is achieved in Section 6.2. The methods in each section can be combined among each other as needed.

6.1 Improvements for general functionality

Since the canoe has not yet achieved the ability to reach the goal from any initial state, we would focus on getting that first. Below are a few possible methods to try in achieving functionality for the general case.

6.1.1 Simplifying the problem.

We can create a simpler problem for the RL agent to solve by reducing the size or complexity of the required solution. For this problem, it would mean that the canoe can paddle for a shorter duration or use less complex strokes to reach the goal. If the simplier problem can be solved by RL algorithms, we can perhaps train the canoe to gradually expand to a more general solution. Some possible ways are:

- *Initialize the canoe closer to the goal.* We can set the initial position of the canoe to be within a smaller radius of the goal. Since the space covered by the agent should be smaller, its value function for that area can be easier to learn. As well, the exploration space for the policy should be reduced due to the shorter episodes, so it should be easier to learn the optimal policy. If this works, we can perhaps gradually increase the initialization radius over the course of training. We can in hopes that the agent would use what it learned from the easier problems to eventually solve the original problem.

- *Initialize the canoe with it facing the goal.* We can initialize the canoe's position randomly and intialize its orientation such that the y-axis of the frame is pointing towards the goal. That is, initialize the canoe such that $\alpha = 0$ in Figure 15. This way, the canoe only needs to learn how to paddle forward to reach the goal without need-

ing to adjust its orientation to be facing the goal first. Note that we can also initialize the canoe with the flipped orientation, i.e. $\alpha = \pi$ such that the y-axis is pointing away from the goal, and the expected canoe strokes to reach the goal would be the same. After we know that training with ones of these orientations is successful, we can also train while randomly selecting from these two possible initializations, which would be more difficult as the stroke symmetry is not inherent in its action space, but could be a good way for the agent to learn the symmetry. Also, similar to the previous case, we can attempt to generalize it to the original any-orientation problem by gradually expanding the sample space from 0 to $\pi/2$ over the course of training, in hopes that the agent can learn to do larger and larger orientation adjustments.

- *Calculate the canoe's pose and velocity with respect to a different frame.* Currently, the given observation includes the canoe's pose written with respect to the world frame, and the canoe's velocity is written with respect to the canoe frame. It can be helpful to write both of these observations with respect to the goal. We can define a new goal frame as simply the world frame with the origin translated (without rotation) to the goal position, and then write the canoe's pose respect to the goal frame. The problem would then become a stabilization problem where the agent must control the two position coordinates in the observation to 0, which could be a easier problem to solve. We can also try using the canoe's velocity with respect to the world frame, noting that since the goal frame is simply the world frame translated, the velocity with respect to the world frame and the goal frame are the same. This could simplify the connection between velocity and pose as they are both within the same frame, but may complicate the connection between paddle actions and velocity as orientation is now a factor.

6.1.2 Adding more observations

Currently, the canoe can observe its own pose and velocity and the configuration of its paddles, but has no information on the state of the velocity field. Since the surrounding velocity field imparts a force onto the canoe, it is possible that adding the flow state of the

surrounding velocity field may yield improvements for training. However, we believe that the effect of the velocity field is not as major as the opposing force produced by the movement of the paddles. Also, although the velocity field introduces a factor of nonlinearity and randomness, the actor and the critic networks should be able to handle some of the nonlinearities, and so the canoe's movement from moving its paddles should be somewhat predictable without knowing the velocity field. Nevertheless, it is possible to try adding velocity field states, say for all fluid cells lying on on the outline of the canoe, to the observation to see whether the training process is improved.

6.1.3 More tuning of hyperparamters

RL algorithms tend to be quite sensitive to changes in hyperparamters [43]. In finding a functional set of hyperparameters, I attempted to use Optuna, a framework for automatic hyperparameter search [45]. The RL Baselines Zoo includes the functionality to use Optuna out of the box for any environment [28]. When I tried using it, however, the resulting values for all sets of hyperparameters are NaN, meaning Optuna could not evaluate the hyperparamter sets. As I eventually found hyperparameters which worked for the single-initialiation case, I did not investigate further into using Optuna. However, finding better hyperparamters could help in the general case, or at least potentially speed up the training process. One could also look into other hyperparameter optimizers, such as those listed in [46] or the one described in [43].

6.1.4 Experiment with more algorithms

If there is more time available, one can experiment with more of the available algorithms in Stable Baselines. Algorithms such as TD3 and HER take a long time to train due to its lack of parallelization, but they takes very little resources during training, allowing one to run experiments with little disruption to other processes. Also, although our experiments with HER did not work out, I still believe using HER is a good direction to investigate due to its applicability to our problem and its potential to generalize to goals at any location. These investigations can also be done in combination with other suggested methods above.

6.2 Extending the project

Once the canoe can reach the goal under any initialization, below are a few ideas for relatively simple but interesting extensions.

6.2.1 Maneuvering with constant velocity sources

In Section 3.1, we see that it is possible to have constant flow fields in the fluid simulation by adding constant velocity sources. An extension could be to train the canoe with these flow fields and see if the canoe does any special maneuvers to overcome the flow fields. For example, if the flow field is as in Figure 2, it would be interesting if the canoe predicts and compensenates for the flow by paddling against the flow field. Once this is successful, a further extension could perhaps be to train in a static flow field and test with small constant velocity sources, seeing if the canoe can counteract small "random" disturbances it has not seen before.

6.2.2 Minimum energy paddling

Currently, the action inputs are angular velocities of the four joints for the arms and paddles. The current cost function encourages the canoe to point and move towards the goal as fast as possible, but does not consider the energy input of moving the paddles. An extension could be to model the energy input required to move the paddles at their desired angular velocities. As an example, the energy could be modelled as the absolute sum of the resulting torques experienced by the paddles in velocity field with respect to the joint locations, using equations described in Section 3.4. We can add this energy to the cost function and tune the scalar factor such that the agent finds a reasonable solution, one which uses the minimal energy to paddle to the goal in a reasonable time. We can then compare the learned simulated strokes to paddling strokes in real life and see if there are significant resemblances, or perhaps even discover a new paddling stroke that is energy-efficient for paddlers.

6.2.3 Acceleration-based paddle joints

At the moment, the angular velocities of the joints can be set to any value within the valid range of ω, while in reality motors cannot instantaneously change speed. An extension for more realism would be to instead change the paddle velocities with acceleration values, which could be changed instantaneously. The action space could be the angular acceleration itself, but it may be difficult to control the canoe with accelerations. An alternative is to add a low-level PID controller for each joint which controls the acceleration to achieve desired velocities, and we keep the action space as angular velocities. Compared to before, this feature basically introduces a lag in the system and makes it a more difficult problem. However, the physics would be more accurate, and it can be combined with the extension above to perhaps learn paddling strokes that are closer to reality.

7 Lessons Learned

Throughout this project, there were times where the first attempt was immediately successful, by intuition or by luck. On the other hand, there were times where it took a few different approaches to get back on track. Though I did not experience any major dead-ends, there were still places where I could have saved time if I had the experience I have now. Below are some advice I would give if another student were to do the same research topic.

7.1 Start with a more general solution for frame transformations

When I was started developing pose and velocity transformations for the hull and the paddles, I felt uncertain about the modelling of the canoe-fluid interactions. So, I developed the transformations for each part individually to just see if the simulation prototype would be functional. The development process involved a long period of bug-fixing, as these transformations are very error-prone and the bugs are usually difficult to debug. For example, I had to debug the linear and angular component of the velocity transformations separately to find out why the velocities are as expected. After the simulation appeared functional,

we needed to add an arm between the paddle and the canoe hull instead of having just the paddle. Hardcoding another linked frame would be very messy and even more difficult to debug, so I decided to start developing a general transformation tree to calculate the transformed poses and velocities. This means basically redeveloping the transformation logic for a general case, and again involved a long period of bug-fixing. Afterwards, however, the transformations are completely bug-free and more frames can be added without fear of introducing bugs. In hindsight, developing the transform tree was a good decision, but I could have saved more time by developing the general solution from the beginning.

7.2 Don't use Cython without adding inline C/C++ code

Cython is an optimising static compiler that compiles Python code into efficient C/C++ code and allows interfacing between Python and C/C++ methods [47]. From the beginning of my simulation development, I used Cython in attempt to speed up the simulation. This means that every time a Python file is modified, I needed to build them using a setup script for Cython, and then the main Python files can import the optimized libraries from the built C/C++ files. However, when my simulation is complete, it ran at almost exactly the same speed (in steps/s) as just using the Python source files. I attempted to add inline C++ code, but found that to be difficult without causing compatibility issues with existing Python libraries. In the end, the simulation performance issues were solved by doing a timing analysis and optimizing the program itself, as described in Section 3.7. Using Cython is still possible if one spends more time in developing inline C++ code, but I now believe it should be lower on the list compared to other possible optimizations.

7.3 Multiprocessing is hard without asynchronous processing

Again in attempt to speed up the simulation, I tried using multiprocessing to take advantage of multiple CPU/GPU cores. The simulation has a single-threaded program logic and has no obvious place to asynchronously execute parts for significant periods of time. I tried experimenting with Pool [48] to create and execute multiple processes at its most expensive function call. The program ended up used all 4 cores on the laptop, but the simulation

became so slow that it is almost unrunnable. This is likely because time it takes to create, synchronize, and destroy the processes greatly outweighs the time spent within each function callback. If the frequency of function calls is lower compared to the function callback time, multiprocessing could have yielded improvements. However, the current simulation is very procedural with each step depending on the output from previous steps, so it is unlikely to have improvements from multiprocessing anywhere in the program.

7.4 Use the environment checker in Stable Baselines

I was testing my new Gym environment by training with algorithms from OpenAI's Baselines library. The training losses quickly became NaNs and it was difficult to debug the problem source. I then discovered Stable Baselines and used its environment checker, which correctly identified that the observations were faulty upon environment reset. In general, the environment checker in Stable Baselines is a great tool for pinpointing exactly where the NaNs or Infs are coming from for a custom Gym environment. It also supports the next lesson:

7.5 Use Stable Baselines instead of Baselines

At first, Stable Baselines looks to be not as reliable as Baselines, since the library appears to be less popular, has less contributors, and the contributors are mainly PhD students. However, the actual algorithms in Stable Baselines are based off of Baselines, so, most of the contributors' efforts are towards the library's ease of use. There are a number of other benefits outlined in Section 4.2. Therefore, I now believe that unless there is a new algorithm in Baselines that has not yet been ported into Stable Baselines, there is no reason to use Baselines over Stable Baselines for training custom environments.

8 Conclusion

Overall, the project is partially successful as we have achieved some of the objectives. We have successfully designed and developed a canoe fluid-field, where we see the canoe re-

sponding appropriately to flow currents, and we were able to design an open-loop controller which paddles the canoe as desired under the simulation physics. Training a controller with deep reinforcement learning was mostly successful for the case where the canoe initializes at a specific pose, though there is still room for improvement when compared to the open-loop controller. When the canoe was initialized randomly, training was not successful. We attempted to address this problem by briefly experimenting with Hindsight Experience Replay, but the results were similar. For future work, we described possible methods for improving the training process, and outlined small extension projects once the training difficulties are resolved. There are other limitations that we have not addressed. For example, the simulation is 2D, models the canoe hull as an ellipse, and only accounts for forces on the outline of the canoe hull instead of the entire hull.

However, through this process, we have demonstrated the process of defining a research problem from scratch. We have described both the theory and the practical tools required for developing a physics simulation and for training an agent with deep reinforcement learning. And we have compiled a series of lessons should one plan to do a similar project.

In the end, although RL is an exciting field, there is still plentiful room for growth, and a seemingly simple problem could be unexpectedly difficult to solve using the current RL algorithms. Nevertheless, we have gained valuable knowledge and experience along the way of this project, and it is our hopes that future readers, whoever they may be, would find a few lessons to take away as well.

References

[1] OpenAI, "Part 2: Kinds of RL Algorithms — Spinning Up documentation." https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[2] R. Rockafellar, "Convex analysis," *SERBIULA (sistema Librum 2.0)*, Jan. 1971.

[3] S. Gros, M. Zanon, R. Quirynen, A. Bemporad, and M. Diehl, "From linear to nonlinear mpc: bridging the gap via the real-time iteration," *International Journal of Control*, pp. 1–19, Sept. 2016.

[4] J. Kober, J. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, Sept. 2013.

[5] P. Ma, Y. Tian, Z. Pan, B. Ren, and D. Manocha, "Fluid directed rigid body control using deep reinforcement learning," *ACM Transactions on Graphics*, vol. 37, pp. 1–11, July 2018.

[6] M. Piggott, "Fluidity." http://fluidityproject.github.io/, 2017.

[7] A. V. Mohanan, C. Bonamy, M. C. Linares, and P. Augier, "FluidSim: Modular, Object-Oriented Python Package for High-Performance CFD Simulations," *Journal of Open Research Software*, vol. 7, 2019.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[9] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," Dec. 2017.

[10] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, Oct. 2017.

[11] M. Rahman, S. H. Rashid, and M. Hossain, "Implementation of q learning and deep q network for controlling a self balancing robot model," *Robotics and Biomimetics*, vol. 5, Dec. 2018.

[12] J. Won, J. Park, K. Kim, and J. Lee, "How to train your dragon: example-guided control of flapping flight," *ACM Transactions on Graphics*, vol. 36, pp. 1–13, Nov. 2017.

[13] L. Busoniu, T. de Bruin, D. Tolić, J. Kober, and I. Palunko, "Reinforcement learning for control: Performance, stability, and deep approximators," *Annual Reviews in Control*, Oct. 2018.

[14] J. Stam, "Stable fluids," *ACM SIGGRAPH 99*, vol. 1999, Nov. 2001.

[15] J. Stam, "Real-time fluid dynamics for games," *Proceedings of the Game Developer Conference*, May 2003.

[16] A. Santini,"Real-time fluiddynamics forgames." https://github.com/albertosantini/python-fluid, 2019.

[17] R. Metere, "Flyingcircus." https://pypi.org/project/flyingcircus/, Dec. 2019.

[18] "The Python Profilers — Python 3.8.2 documentation." https://docs.python.org/3/library/profile.html, Apr. 2020.

[19] J. Fu, A. Kumar, M. Soh, and S. Levine, "Diagnosing bottlenecks in deep q-learning algorithms," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, (Long Beach, California, USA), pp. 2021–2030, PMLR, June 2019.

[20] S. Phaniteja, P. Dewangan, P. Guhan, A. Sarkar, and K. M. Krishna, "A deep reinforcement learning approach for dynamically stable inverse kinematics of humanoid

robots," in *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 1818–1823, 2017.

[21] "OpenAI Baselines." https://github.com/openai/baselines, May 2017.

[22] A. Hill, "Stable Baselines." https://github.com/hill-a/stable-baselines, July 2018.

[23] "Tensorflow RL," Nov. 2018.

[24] "Tensorforce." https://github.com/tensorforce/tensorforce, Mar. 2017.

[25] "Keras RL." https://github.com/keras-rl/keras-rl, July 2016.

[26] chimera0, "pyqlearning: pyqlearning is Python library to implement Reinforcement Learning and Deep Reinforcement Learning, especially for Q-Learning, Deep Q-Network, and Multi-agent Deep Q-Network which can be optimized by Annealing models such as Simulated Annealing, Adaptive Simulated Annealing, and Quantum Monte Carlo Method.." https://github.com/chimera0/accel-brain-code/tree/master/Reinforcement-Learning.

[27] T. Simonini,"Choosing a Deep Reinforcement Learning Library." https://medium.com/data-from-the-trenches/choosing-a-deep-reinforcement-learning-library-890fb0307092, June 2019.

[28] A. RAFFIN, "RL Baselines Zoo: a Collection of Pre-Trained Reinforcement Learning Agents." https://github.com/araffin/rl-baselines-zoo, Oct. 2018.

[29] OpenAI, "Gym: A toolkit for developing and comparing reinforcement learning algorithms." https://gym.openai.com, Apr. 2020.

[30] A. King, "Creating a Custom OpenAI Gym Environment for Stock Trading." https://towardsdatascience.com/creating-a-custom-openai-gym-environment-for-stock-trading-be532be3910e, Oct. 2019.

[31] D. Silver,"Lectures on deep reinforcement learning." http://videolectures.net/rldm2015_silver_reinforcement_learning, June 2015.

[32] G. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," *Technical Report CUED/F-INFENG/TR 166*, Nov. 1994.

[33] C. J. C. H. "Watkins and P. Dayan, ""q-learning"," *"Machine Learning"*, vol. "8", pp. "279–292", May "1992".

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," Dec. 2013.

[35] L. Wang, "Lectures on deep reinforcement learning." https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html, Dec. 2019.

[36] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. M. O. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[37] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *CoRR*, vol. abs/1802.09477, 2018.

[38] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, Feb. 2016.

[39] "OpenAI Baselines: ACKTR & A2C." https://openai.com/blog/baselines-acktr-a2c/, Aug. 2017.

[40] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, July 2017.

[41] "Proximal Policy Optimization." https://openai.com/blog/openai-baselines-ppo/, July 2017.

[42] "PPO2 — Stable Baselines 2.10.1a0 documentation." https://stable-baselines.readthedocs.io/en/master/modules/ppo2.html.

[43] S. Paul, V. Kurin, and S. Whiteson, "Fast efficient hyperparameter tuning for policy gradients," *ArXiv*, vol. abs/1902.06583, 2019.

[44] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," July 2017.

[45] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[46] A. Bissuel, "Hyper-parameter optimization algorithms: a short review." https://medium.com/criteo-labs/hyper-parameter-optimization-algorithms-2fe447525903, Apr. 2019.

[47] S. Behnel, R. Bradshaw, L. Dalcín, M. Florisson, V. Makarov, and D. S. Seljebotn, "Cython: C-Extensions for Python." https://cython.org/.

[48] "multiprocessing — Process-based parallelism — Python 3.8.2 documentation." https://docs.python.org/3/library/multiprocessing.html.

[49] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction; 2nd Edition*. The MIT Press, second ed., 2017.

A Appendices

A.1 Frame Transformation

The 2D homogeneous transformation from frame i to frame j is denoted as

$$\mathbf{T}_{ij} = \begin{bmatrix} \mathbf{C}_{ij} & \mathbf{r}^{ji} \\ \mathbf{0}^T & 1 \end{bmatrix} \in SE(2) \subset \mathbb{R}^{3\times 3} \tag{31}$$

where $SE(2)$ is the special Euclidean group for 2D, and $\mathbf{C}$ represents the 2D rotation matrix in the 2D special orthogonal group:

$$\mathbf{C} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \in SO(2) \tag{32}$$

As well, $\mathbf{r}$ represents the 2D translation vector:

$$\mathbf{r} = \begin{bmatrix} r_x \\ r_y \end{bmatrix} \in \mathbb{R}^{2\times 1} \tag{33}$$

Suppose we desire the points within frame $i+j$ expressed in frame i, where frame $i+j$ is a descendent of frame i. The transformation would simply be to multiply by the points in frame $i+j$ by a chain of homogeneous transformations up to frame i:

$$\mathbf{R}_i = \mathbf{T}_{i,i+1} \cdots \mathbf{T}_{i+j-2,i+j-1}\mathbf{T}_{i+j-1,i+j}\mathbf{R}_{i+j} \tag{34}$$

where $\mathbf{R} = \begin{bmatrix} \mathbf{r} \\ 1 \end{bmatrix}$ is the augmented position vector for a point.

The normal vectors are transformed similarly, except only the rotation matrices are used:

$$\mathbf{n}_i = \mathbf{C}_{i,i+1} \cdots \mathbf{C}_{i+j-2,i+j-1}\mathbf{C}_{i+j-1,i+j}\mathbf{n}_{i+j} \tag{35}$$

where $\mathbf{n} = \begin{bmatrix} n_x \\ n_y \end{bmatrix}$ is a normal vector with no augmentation.

The velocities of the frames are transformed as follows: For an object k written in reference frame 1 whose position is $\mathbf{r}_1^{1,k}$, its position relative to frame 0 is:

$$\mathbf{r}_0^{k,0} = \mathbf{r}_0^{1,0} + \mathbf{C}_{0,1}\mathbf{r}_1^{k,1} \tag{36}$$

where we can extract $\mathbf{C}_{0,1}, \mathbf{r}_0^{1,0}$ from $\mathbf{T}_{0,1}$. We get the velocity transformation by differentiating this. The velocity of object k relative to frame 0 is:

$$\dot{\mathbf{r}}_0^{k,0} = \dot{\mathbf{r}}_0^{1,0} + \mathbf{C}_{0,1}\dot{\mathbf{r}}_1^{k,1} + (\boldsymbol{\omega}_0^{1,0\times}\mathbf{C}_{0,1})\mathbf{r}_1^{k,1} \tag{37}$$

We can then get a recursive formula for finding the transformation of the velocity of object k with respect to frame $i-1$ ($\dot{\mathbf{r}}_{i-1}^{k,i-1}$), given its position and velocity with respect to frame i ($\mathbf{r}_i^{k,i}, \dot{\mathbf{r}}_i^{k,i}$), the homogeneous transformation $\mathbf{T}_{i-1,i}$ (yielding $\mathbf{C}_{i-1,i}, \dot{\mathbf{r}}_{i-1}^{i,i-1}$), and the angular velocity $\boldsymbol{\omega}_{i-1}^{i,i-1}$:

$$\dot{\mathbf{r}}_{i-1}^{k,i-1} = \dot{\mathbf{r}}_{i-1}^{i,i-1} + \mathbf{C}_{i-1,i}\dot{\mathbf{r}}_i^{k,i} + (\boldsymbol{\omega}_{i-1}^{i,i-1\times}\mathbf{C}_{i-1,i})\mathbf{r}_i^{k,i} \tag{38}$$

where $\mathbf{r}_i^{k,i}$ above can be found by concurrently doing the frame transformation of position vectors from frame $i+1$ to i:

$$\mathbf{R}_i^{k,i} = \mathbf{T}_{i,i+1}\mathbf{R}_{i+1}^{k,i+1} \tag{39}$$

A.2 Reinforcement Learning (RL) Overview

At each time step, an RL agent generates an action a_t from state s_t, then takes the action to transition its state from s_t to s_{t+1}, and receives reward R_t. We define this sequence of (s_t, a_t, R_t, s_{t+1}) as a transition. The goal of the agent is to take actions a_t that maximize the return G_t, the total discounted future reward from time t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{40}$$

where γ is a discount factor, $0 < \gamma \leqslant 1$. To do this, the agent learns to follow a policy π which chooses an action based on the current state at each time step. The policy can be deterministic ,where π is simply a function of the state:

$$a = \pi(s) \tag{41}$$

or stochastic, where π at a given state s is a probability distribution of actions over states, and the action a_t is sampled from this distribution (A_t is the random variable for actions, and S_t is the random variable for states):

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{42}$$

The agent learns to generate actions which yield the maximum total future rewards by optimizing its policy via RL algorithms. It typically does this either by estimating the value function $v_\pi(s)$, the expected return from state s for following a policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \ldots|S_t = s] \tag{43}$$

or by estimating the action-value function $q_\pi(s, a)$, the expected return of starting from state s, taking action a, then following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{44}$$

Classical RL agents typically construct and learn a table of discrete value functions for all states under the optimal policy (π^*) using dynamic programming, then extract actions which yields the maximum $v_\pi(s)$ (see left table below). Or it could construct a table of action-value functions for all state-action pairs (see right table below):

state (s)	$v_{\pi*}(s)$
...	...

state (s)	action (a)	$q_{\pi*}(s, a)$
...	...	...

This classical method becomes ineffective when the state-space is very large or continuous, where it becomes unfeasible to encounter often enough every possible state or state-action pair to learn their values. Deep RL overcomes this by using neural networks to generalize value and action-value functions to state or state-action pairs that are previously infrequent or unseen. Neural networks are used for Q-value approximators instead of tables. Acting as nonlinear function approximators, these networks can extrapolate to new states and state-actions using training data of seen Q-values. The most common example is the Deep Q-Network (DQN), where the inputs are observations and the outputs are Q-values for all possible discrete actions.

A.3 Actor-Critic RL Framework

The Actor-Critic framework for reinforcement learning is a common framework for a continuous state-space and action-space. Although the vanilla actor-critic rarely works by itself, many of the advanced RL algorithms are based off of this framework. We summarize the framework very briefly here. For more information, see [49].

We need to estimate the Q-value for a continuous state and output a continuous action, and optimize both the action and the Q-value. The base Deep Q-Network (DQN) involving a single network is only suitable for a discrete action space. Instead, we have two networks: the actor network which takes in the state and output a candidate action, and the critic network which takes in the state and the candidate action and outputs a Q-value. Then, at each time step, we can optimize both networks. We optimize the actor network by following the policy gradient in the direction that improves Q:

$$\frac{\partial J(u)}{\partial w} = \mathbb{E}_s \left[\frac{\partial Q(s,a,w)}{\partial a} \frac{\partial \pi(s,u)}{\partial u} \right] \tag{45}$$

And we optimize the critic network by minimizing the quadratic cost of Q-values:

$$L(w) = \mathbb{E}[(r + \gamma Q(s', \pi(s'), w) - Q(s,a,w))^2] \tag{46}$$

$$\frac{\partial L(w)}{\partial w} = \mathbb{E} \left[(r + \gamma Q(s', \pi(s'), w) - Q(s,a,w)) \frac{\partial Q(s,a,w)}{\partial w} \right] \tag{47}$$

Note that since the output of the actor network is part of the input of the critic network, the gradient flows from the critic's output (Q-values) reaches both networks.

The vanilla actor-critic framework can be used directly to solve our RL problem. We may initialize the weights of both networks randomly, start an episode with the canoe at an arbitrary pose, and train the two networks concurrently. Once both the actor and the critic network converges, the agent is trained. However, training can diverge and fail very easily if we are using an off-policy algorithm with bootstrapping and a nonlinear function approximator, and this occurs so commonly that it is nicknamed the "deadly triad problem" [49].

This applies to Q-learning with actor-critic as it is off-policy and the neural networks are nonlinear function approximators. As well, the framework uses bootstrapping, meaning it estimates Q-value of the next state with the same network as the one used to evaluate and improve the Q-value of the current state.

A.4 DDPG Explanation

For this section, we discuss the theoretical motivations of the Deep Deterministic Policy Gradient (DDPG). I chose to write a longer explanation for DDPG as I previously could not understand how experience replay can be applied to actor-critic while still along the flow of gradients, and I eventually found DDPG to be the satisfying solution. Later, I discovered Lilian Weng's blog [35] which had useful summaries for algorithms including DDPG, and readers can refer to there. My original explanation is still kept here in the Appendices as supplement.

To address the stability issues outlined at the end of Appendix A.3, one general solution for this divergence is to use experience replay. In the vanilla actor-critic algorithm, we would always train both networks on the current transition the agent has just experienced throughout an episode. This method is certainly convenient, but the training data fed into the networks would not be independent of the previous data, which is a major reason for training divergence [49]. In experience replay, instead of immediately training with the current transition, we store away this interation in a replay memory, and we train both networks on a fresh set of transitions randomly sampled from the replay memory, breaking the dependence between training data [31].

However, experience-replay cannot be added immediately to our framework. In order to train the actor network with the policy gradient, the agent current must follow its current policy for Equation 45 to be valid. The actor network's policy gradient is dependent on the output ($Q(s, a)$) of the critic network, which is dependent on the action (a). Thus, if the agent chooses a different action than the output of the actor network, the formula $\frac{\partial Q(s,a,w)}{\partial a}\frac{\partial \pi(s,u)}{\partial u}$ is not meaningful for different a's. So, the agent cannot use past experiences where different actions were chosen (since the actor network had different weights), and also cannot explore off-policy actions while training.

The main contribution of DDPG is that it overcomes both of the problems above by provid-

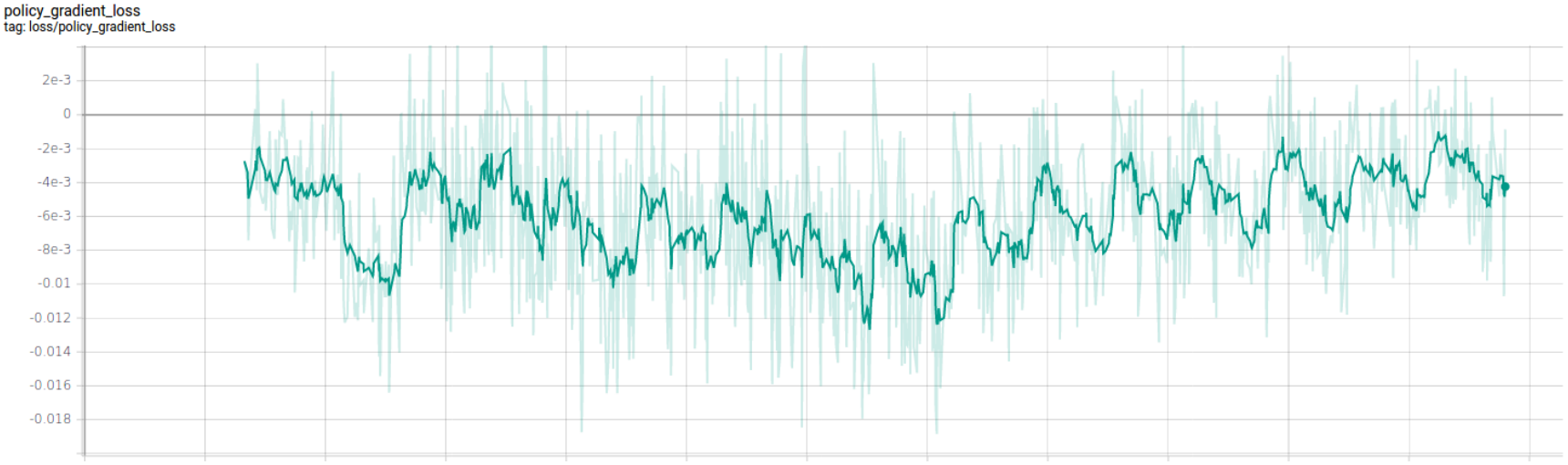
ing a formula for calculating policy gradients for off-policy actions [36].

Now, we may train both the actor and critic network on any action, allowing for experience replay and for exploration. The agent can reuse past transitions produced by networks of differing states, and can explores an action space around the action suggested by the actor network by adding a Gaussian noise:

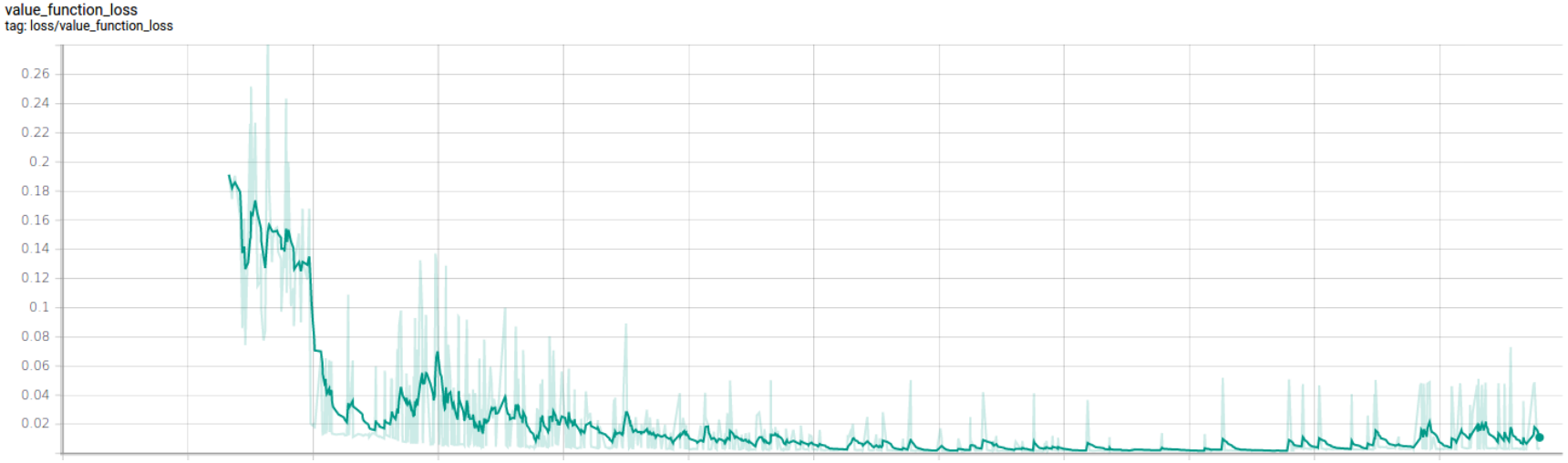
$$\mu'(s) = \mu_\pi(s) + \theta, \quad \theta \sim N(0, \sigma^2) \tag{48}$$

A.5 Tensorboard Training Graphs

The tensorboard graphs referenced by Section 4.8 and 5.4 are large and difficult to manipulate, so they are placed on their separate pages for clarity. To avoid disrupting the flow of the thesis body, they are placed in the following pages. The horizontal axis represents the timesteps, and the vertical axis represents the respective function loss. The smoothness factor for the graphs is is 0.85, meaning that the solid data lines are much smoother versions of the original data lines, though traces of the original data can still be seen as the faded data lines. Note that sometimes the RL algorithm from Stable Baselines does not output losses for a small number of iterations at the beginning of training.

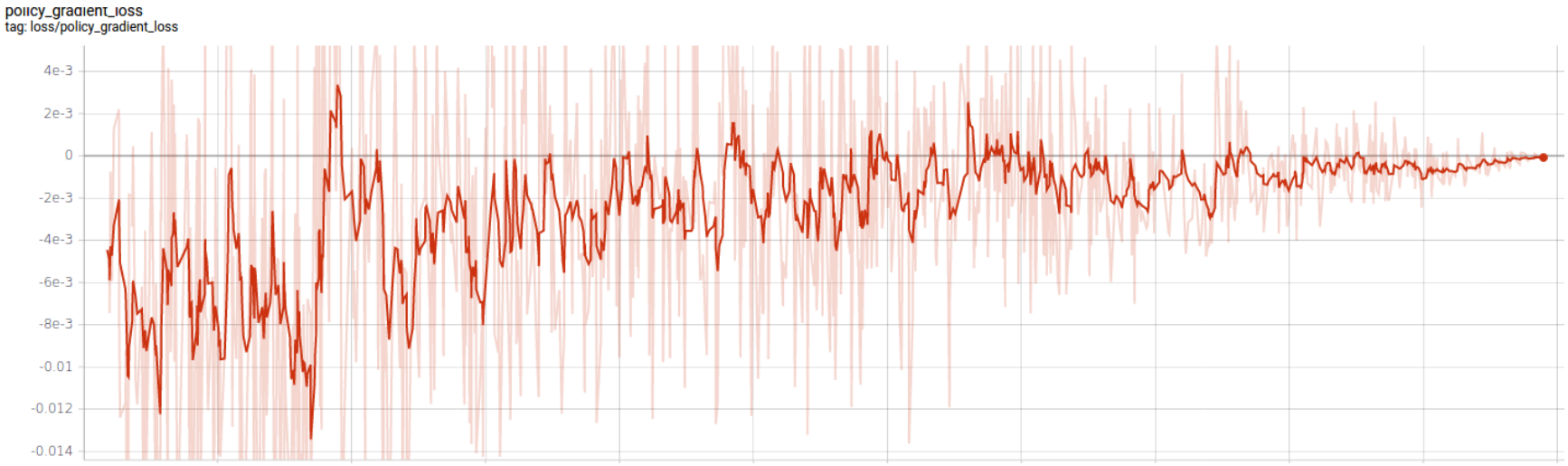


(a) Policy function loss with PPO2

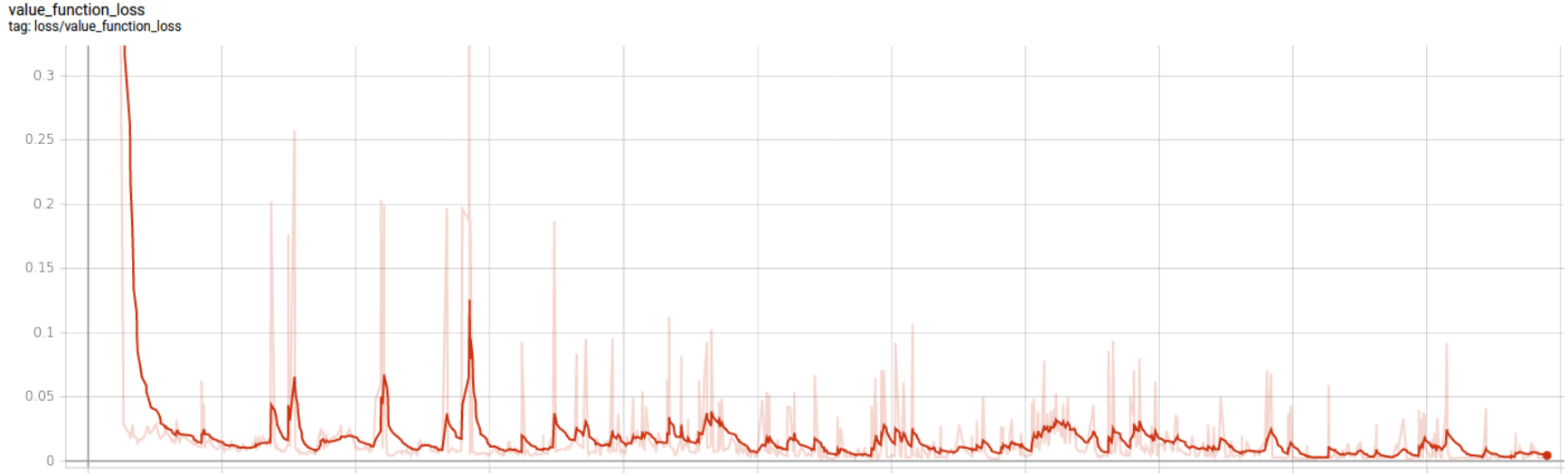


(b) Value function loss with PPO2

Figure 20: Tensorboard graphs for training PPO2 with the default network architecture (2 hidden layers of 64×64) over 1.18 million timesteps for the Single Initialization Problem

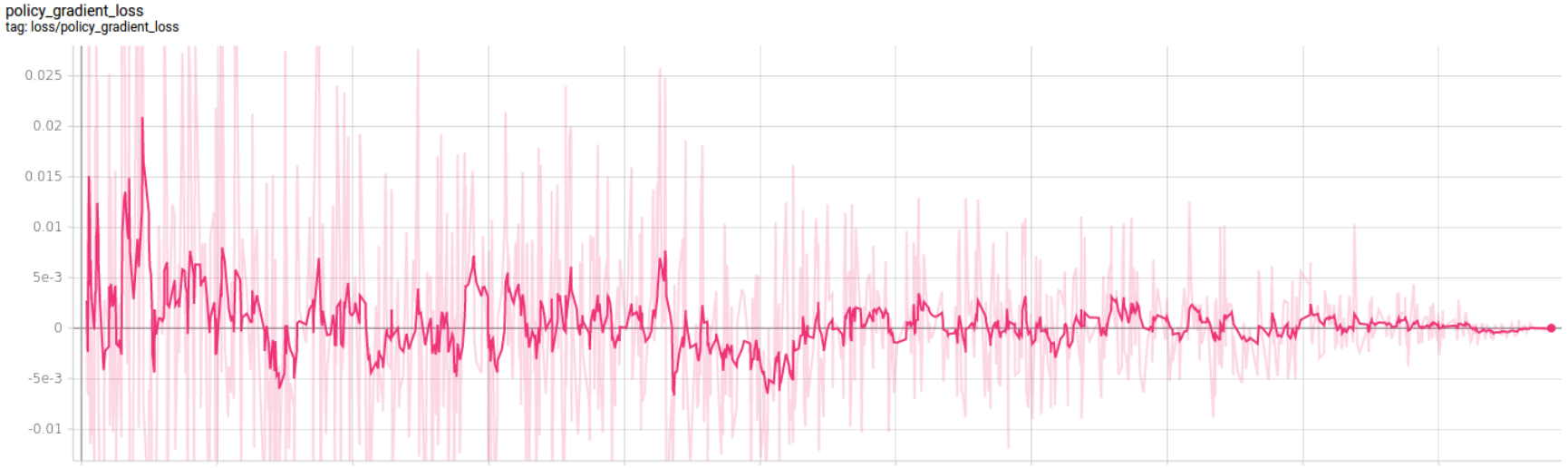


(a) Policy function loss of PPO2 with custom network architecture

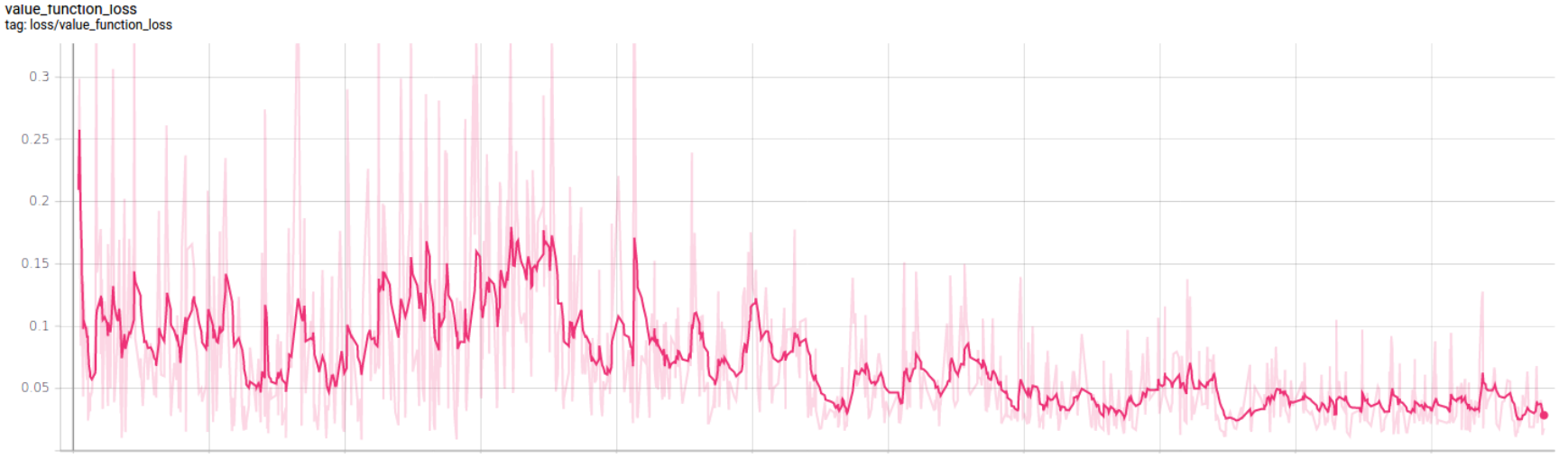


(b) Value function loss of PPO2 with custom network architecture

Figure 21: Tensorboard graphs for training PPO2 with the custom network architecture (4 hidden layers of $64 \times 256 \times 64 \times 16$) over 1.09 million timesteps for the Single Initialization Problem

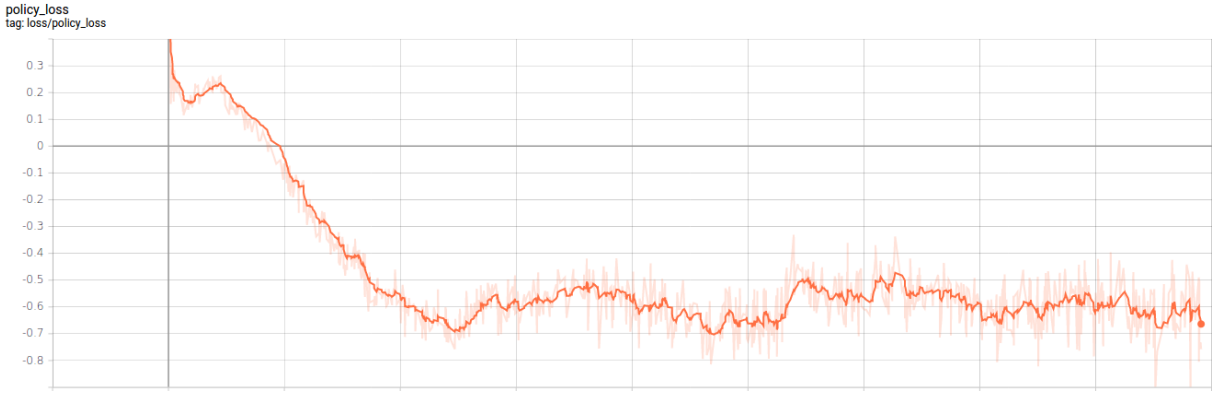


(a) Policy function loss of PPO2 with custom network architecture

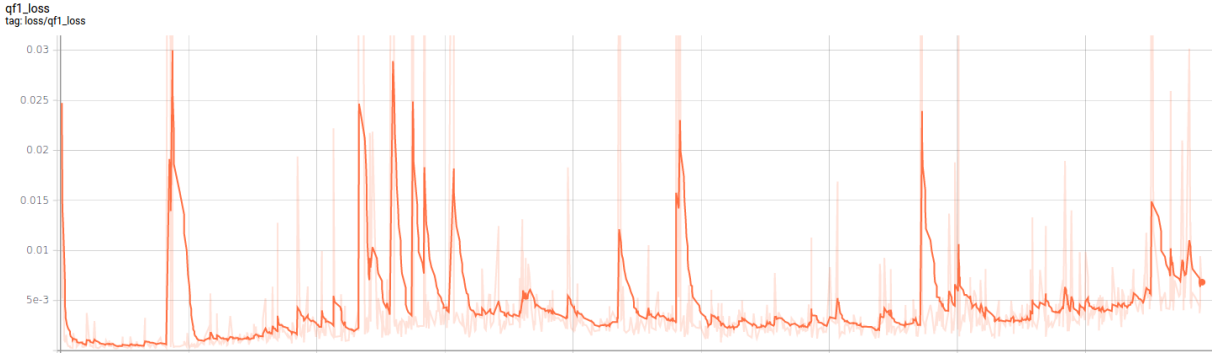


(b) Value function loss of PPO2 with custom network architecture

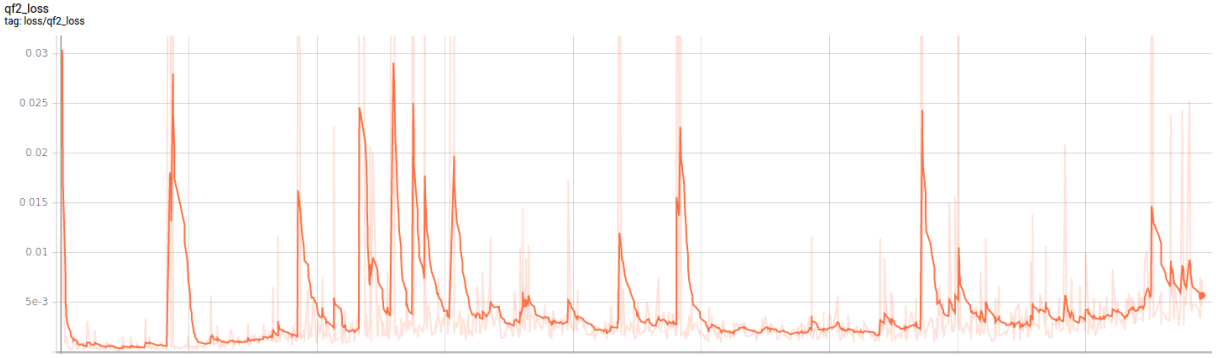
Figure 22: Tensorboard graphs for training PPO2 with the custom network architecture (4 hidden layers of $64 \times 256 \times 64 \times 16$) over 5.42 million timesteps for the General Initialization Problem



(a) Policy function loss of TD3 with custom network architecture

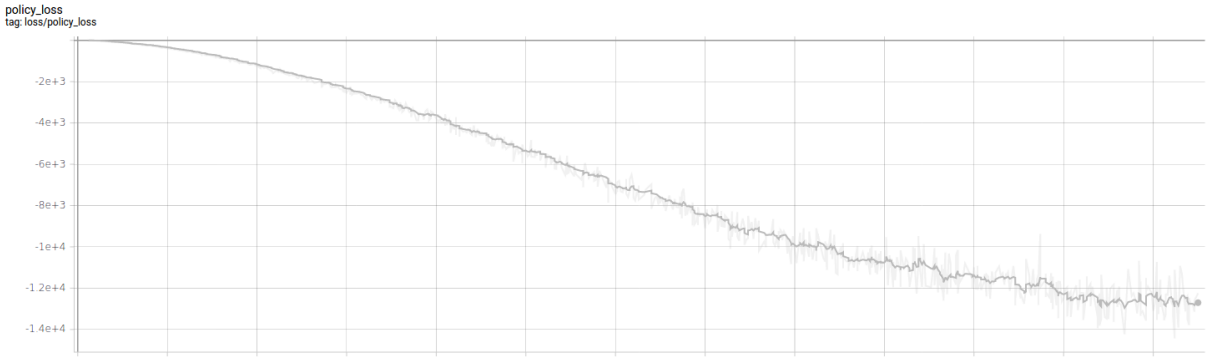


(b) QF1 value function loss of TD3 with custom network architecture

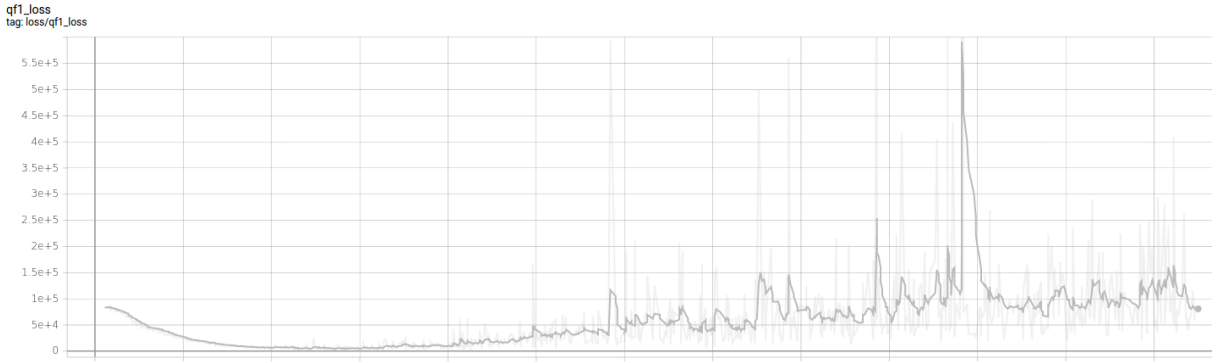


(c) QF2 value function loss of TD3 with custom network architecture

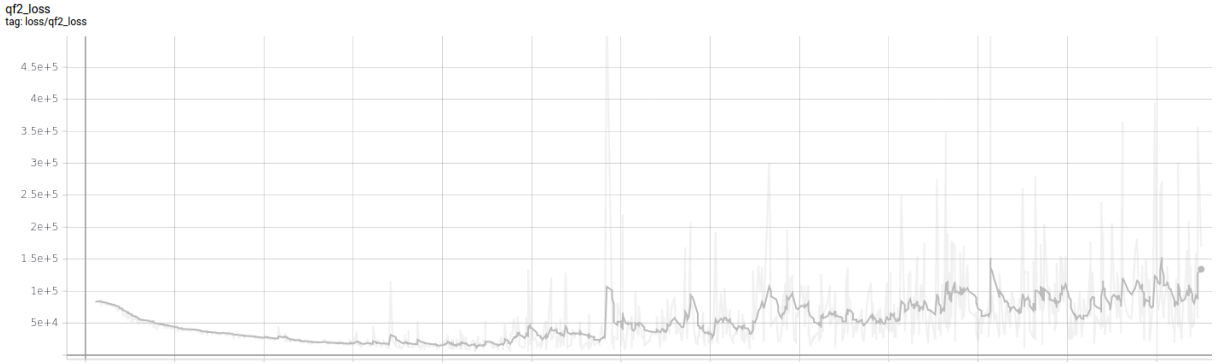
Figure 23: Tensorboard graphs for training TD3 with the custom network architecture (4 hidden layers of $64 \times 256 \times 64 \times 16$) over 178 thousand timesteps for the General Initialization Problem



(a) Policy function loss of HER on top of TD3 with custom network architecture



(b) QF1 value function loss of HER on top of TD3 with custom network architecture



(c) QF2 value function loss of HER on top of TD3 with custom network architecture

Figure 24: Tensorboard graphs for training TD3 on top of TD3 with the custom network architecture (4 hidden layers of $64 \times 256 \times 64 \times 16$) over 63 thousand timesteps for the General Initialization Problem