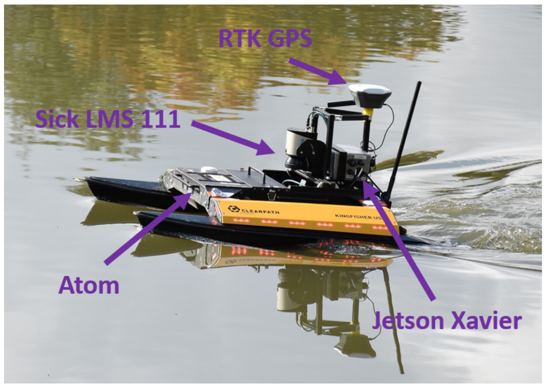
A Robotic systems

A.1 Heron



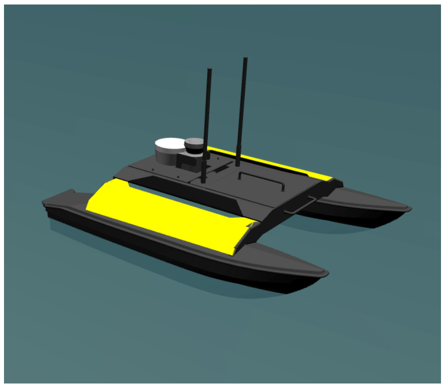


Figure 7: Left the real Kingfisher, right the simulated Heron.

The simulation and the real experiments were performed using similar systems. The real robot is a Clearpath Robotics Kingfisher, while the simulated robot is a Clearpath Robotics Heron (the Kingfisher's new version). In both simulated and real experiments our USV is equipped with a SICK LMS111, a 20 meter-range and 270° field of view 2D-LiDAR running at 50Hz. To acquire the pose of our robot we use a REACH RS+, an RTK GPS from Emlid coupled to a pair of IMU that we use to get the heading of our robot. These informations are then fused inside an EKF that provides pose and velocity estimates. The weight distribution of the real and simulated systems are different: our real system was adapted to carry an NVIDIA Jetson Xavier and the RTK GPS. On the real system, an Intel Atom is used for low-level computations, while the Jetson Xavier at base clocks is used to compute and apply the agent policy. Both computers are running ROS with a single master.

The main challenge of this system is its inertia. With its current configuration, our Kingfisher's weight is around 35 kg, and it only has two 400 W motors (one left, one right). Hence, if an agent wants to take turns correctly, it needs to anticipate.

On the real system, the RL agent runs on Ubuntu 18.04 with ROS melodic, CUDA 10.0, python 2.7 and TensorFlow 2.1 [28]. We are using a custom compiled tensorflow wheel on the Xavier, if you would like to get our tensorflow wheel, feel free to reach out to us. The agent model has not been converted using Tensor RT or any other DNN compilers and can run easily at 12Hz on the Xavier.

A.2 Husky

The second system we tested our method on is an Unmanned Ground Vehicle (UGV): a Clearpath Robotics Husky. This UGV is a four driving wheels robot designed for outdoor applications. Because our real robot is not equipped with any GPU yet, we only tested it in simulation. The Husky in simulation is equipped with the same SICK LMS111 than the Heron.

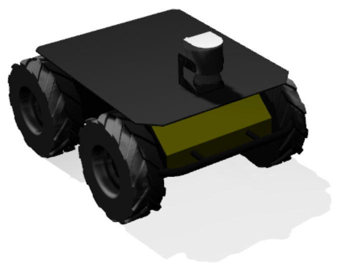


Figure 8: The simulated Husky.

B Task definition

This appendix gives more details on the task the reinforcement learning agent were trained to do. The task that we solved is a sensor-based navigation task. More precisely, we teached a robot to follow lake and river shores at a fixed distance and a fixed velocity. This is motivated by applications involving long-term monitoring of natural environments (for example, to assess that the shores are not moving over time). To follow the shore, our agent relies on a laser scanner and has access to its velocities (forward velocity, lateral velocity, angular velocity). Figure 9 shows the robot and the task it has to accomplish.

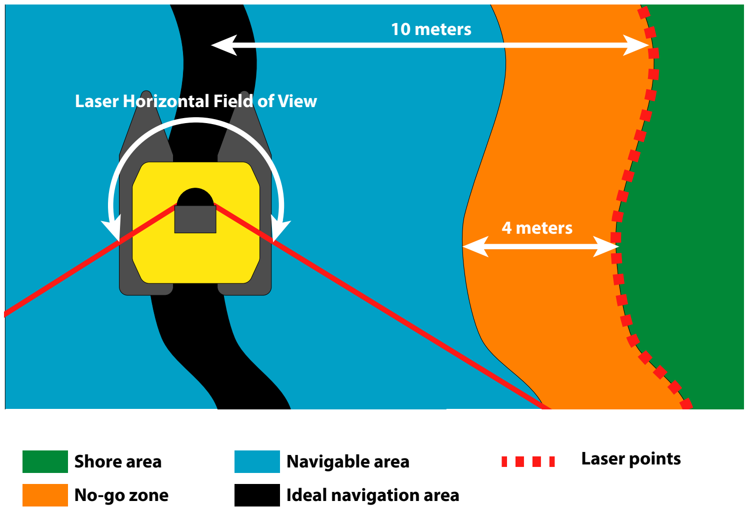


Figure 9: The USV and its shore-following task. (Colors are illustrative.)

The reward of our agent is based on two independent metrics: Δ_d, the distance of the agent from the target shore distance d_t, and Δ_v, the difference between the agent's velocity v and the target linear velocity v_t. In all our experiments, we set $v_t = 1.0$ and $d_t = 10$. Furthermore, we penalize the agent if it gets too close to the shore or if it goes backward. The reward R is defined as follows:

$$R = 1.0 \times R_v + 2.5 \times R_d$$

with R_v and R_d given by:

$$R_d = \max(-20, 1 - 0.5(d_t - d)^2)$$

$$R_v = \begin{cases} \frac{1-|v_t-v|}{v_t}, & \text{if } v \geq 0.05 \\ -0.625, & \text{otherwise} \end{cases}$$

This reward is computed when training the model, and learned as part of it. The agent is trained on the learned reward (estimated solely from the laser-scan measurements, through the learned embedding).

C Algorithms

C.1 Online Learning Algorithm

Algorithm 1: Physics Driven Dreamer

```
Fill dataset $\mathcal{D}$ with $N$ random actions episodes.
Initialize neural network parameters $\theta, \eta, \psi$ randomly.
if load dynamics then
    Load network parameters $\phi$.
else
    Initialize neural network parameters $\phi$ randomly.
while not converged do
    if refine dynamics then
        for update step $i = 1..I$ do
            // Dynamics learning
            Draw $B$ data sequences $\{(a_t, x_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
            Compute dynamic states
            $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, x_t)$.
            Update $\phi$ using representation learning.
    for update step $c = 1..C$ do
        // Environment learning
        Draw $B$ data sequences $\{(x_t, o_t, r_t, a_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
        Compute dyn states
        $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, x_t)$.
        Compute env states
        $S_t^{env} \sim p_\eta(S_t^{env} \mid S_{t-1}^{env}, x_{t-1}, o_t)$.
        Update $\eta$ using representation learning.
        // Behavior learning
        Imagine trajectories
        $\{(S_\tau^{env}, S_\tau^{dyn}, a_\tau, x_\tau)\}_{\tau=t}^{t+H}$ from each $(S_t^{env}, S_t^{dyn})$.
        Predict rewards and values
        $\mathrm{E}\big(q_\eta(r_\tau \mid S_\tau^{env}, S_\tau^{dyn})\big)$, $v_\psi(S_\tau^{env}, S_\tau^{dyn})$.
        Update $\theta$ and $\psi$ using behavior learning.
    // Environment interaction
    $o_1 \leftarrow$ env.reset()
    for time step $t = 1..T$ do
        Compute $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, o_t)$ from
          history.
        Compute $S_t^{env} \sim p_\theta(S_t^{env} \mid S_{t-1}^{env}, x_{t-1}, o_t)$ from
          history.
        Compute $a_t \sim q_\phi(a_t \mid S_t^{env}, S_t^{dyn})$ with the action
          model.
        Add exploration noise to action.
        $r_t, o_{t+1} \leftarrow$ env.step($a_t$).
    Add experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$.
```

Model components

Dynamics	$p_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1}, x_t)$
D-Transition	$q_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1})$
Environment	$p_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1}, o_t)$
E-Transition	$q_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, a_{t\text{-}1})$
Reward	$q_\eta(r_t \mid S_t^{env}, S_t^{dyn})$
Action	$q_\theta(a_t \mid S_t^{env}, S_t^{dyn})$
Value	$v_\psi(S_t^{env}, S_t^{dyn})$

Hyper parameters

Number of random episodes	N
Collect interval	C
Physics train step	I
Batch size	B
Sequence length	L
Imagination horizon	H
Learning rate	α

C.2 Offline Learning Algorithm

Algorithm 2: Offline Learning, and Environment Transfer

```
Fill dataset $\mathcal{D}$ with all the episodes collected on Robot B.
Initialize neural network parameters $\theta, \psi$ randomly.
Load neural network parameters $\eta$ from robot B.
Load neural network parameters $\phi$ from robot A.
for training step $c = 1..C$ do
    // Reward Finetuning
    Draw $B$ data sequences $\{(x_t, o_t, r_t, a_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
    Compute dyn states
    $S_t^{dyn} \sim p_\phi(S_t^{dyn} \mid S_{t-1}^{dyn}, a_{t-1}, x_t)$.
    Compute env states
    $S_t^{env} \sim p_\eta(S_t^{env} \mid S_{t-1}^{env}, x_{t-1}, o_t)$.
    $r_t \sim q_\eta(r_t \mid S_t^{env}, S_t^{dyn})$
    Update reward $q_\eta$ using representation learning.

    // Behavior learning
    Imagine trajectories
    $\{(S_\tau^{env}, S_\tau^{dyn}, a_\tau, x_\tau)\}_{\tau=t}^{t+H}$ from each $(S_t^{env}, S_t^{dyn})$.
    Predict rewards and values
    $\mathrm{E}\big(q_\eta(r_\tau \mid se_\tau, S_\tau^{dyn})\big), v_\psi(S_\tau^{env}, S_\tau^{dyn})$.
    Update $\theta$ and $\psi$ using behavior learning.
```

Model components

Dynamics	$p_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1}, x_t)$
D-Transition	$q_\phi(S_t^{dyn} \mid S_{t\text{-}1}^{dyn}, a_{t\text{-}1})$
Environment	$p_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, x_{t\text{-}1}, o_t)$
E-Transition	$q_\eta(S_t^{env} \mid S_{t\text{-}1}^{env}, a_{t\text{-}1})$
Reward	$q_\eta(r_t \mid S_t^{env}, S_t^{dyn})$
Action	$q_\theta(a_t \mid S_t^{env}, S_t^{dyn})$
Value	$v_\psi(S_t^{env}, S_t^{dyn})$

Hyper parameters

training steps	C
Batch size	B
Sequence length	L
Imagination horizon	H

D Training

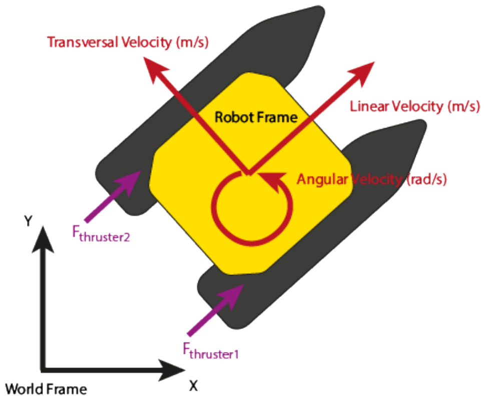
To train our robots, we used Gazebo: a robotic simulator, coupled to ROS, a robotic middleware. The USV dynamics simulation is handled by the `uuv-simulator` package. Each agent was trained in TensorFlow [28] for 1000 episodes of 500 steps amounting to 0.5 million interactions with the simulated environment. All our agents were trained purely in simulation, and were never fine-tuned on real-data. During the training, we applied a simple fixed-step curriculum learning. At the beginning of the training, the robot spawned close from the requested distance to the shore. As the training progressed, the agent spawned farther from that position with different headings. More details about the hyper-parameters, our training setup, the curriculum and others can be found bellow.

D.1 Code

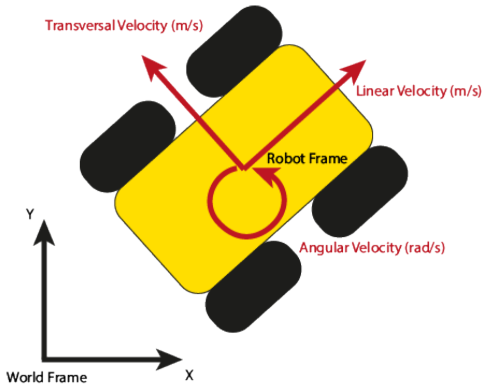
Training: `https://github.com/anon556677/Dreamer-Phy.git`
ROS nodes: `https://github.com/anon556677/ROS-Dreamer-Phy.git`
Simulation ROS: `https://github.com/anon556677/Simulation-Dreamer-Phy.git`

D.2 State-Space

In our experiments, our robots are given access to three proprioceptive variables: their linear velocity, their transversal velocity, and their angular velocity. All these velocities are given in the robot frame, as illustrated in Fig.10. Regarding the action-space of the robots, the commands are values in the range $[-1, 1]$. The Kingfisher has a two-dimensional action space, where each dimension controls the amount of current sent to the turbine in each of its floats. As for the Husky, the first component of its action space requests a linear velocity in m/s while the second one requests an angular velocity rad/s.



(a) Kingfisher/Heron



(b) Husky

Figure 10: State-space/Action-space illustration.

D.3 Hyper-Parameters

	Parameters	Values
Training settings	Initial Dataset Size	2500 steps
	Update steps	100
	Sequence lenght	50
	Batch size	50
	Action repeat	4.2
	Discount	0.97
	Imagination Horizon	15
	Interaction steps	0.5M
	Episode lenght	500
	λ	0.95
Environment RSSM	Deterministic state size	300
	Stochastic state size	30
	Dense size	400
	Free_nats	3.0
	KL scale	1.0
	Learning rate	6e-4
Dynamics RSSM	Physics deterministic state size	50
	Stochastic state size	5
	Free_nats	0.5
	KL scale	1.0
	Dense size	60
	Learning rate	2e-4
Laser Embedding (1D CNN)	Number of layers	6
	Depth	16,32,64,128,256,512
	Activation	Elu
Actor (MLP)	Number of layers	4
	Neurons	400,400,400,2
	Activation	Elu
Value (MLP)	Number of layers	4
	Neurons	400,400,400,2
	Activation	Elu
Reward (MLP)	Number of layers	3
	Neurons	400,400,2
	Activation	Elu
Physics Decoder (MLP)	Number of layers	3
	Depth	60,60,3
	Activation	Elu
Image Decoder (2D CNN)	Number of layers	6
	Depth	512,256,128,64,32,3
	Activation	Elu

Table 1: agent parameters.

D.4 ROS & Learning interactions

The way we interact with gazebo can be described as follows:

Algorithm 3: Training Interaction

```
Start the Gazebo simulation $SIM$.
Start the ROS synchronization node $RS$.
Start the ROS agent node $RA$.
Start the training code $TR$.
while not converged do
    $TR$.request_new_episode($RS$).
    $RS$.reset($SIM$).
    $RS$.refresh_agent($RA$).
    $RA$.fetch_new_weight($TR$).
    $RA$.start_interaction().
    $RA$.done($RS$).
    $RS$.done($TR$).
    $TR$.fetch_last_episode($RA$).
    $TR$.train().
```

D.5 Curriculum

The curriculum learning of the robot is done by spawning the robot in increasingly difficult positions. To do so, we use a normal density probability function $f(x)$ with $x \in [0, 1]$. The mean of the probability density function μ, starts at 0 and shifts towards 1 as the training progresses. The equation is given in Eq. 4 and examples of the probability function can be seen in Fig. 11, where $\sigma = 0.25$.

$$f(x) = e^{\frac{-(x-mu)^2}{2\sigma^2}} \tag{4}$$

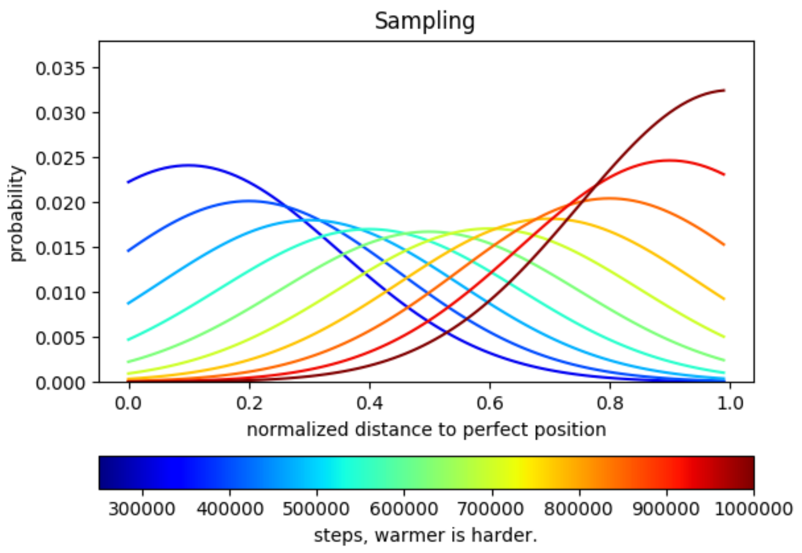


Figure 11: Probability density function depending on the number of steps. Each color corresponds to a step.

To spawn the robot, we pick a random position $(p_{\text{ideal\_x}}, p_{\text{ideal\_y}}$ on the ideal trajectory. The heading $p_{\text{ideal\_}\theta}$ associated with this position is set to be tangent to this trajectory. Please note that the ideal distance is known beforehand as the environment was generated using a cad model of known geometry. Using the probability density function f we draw d and ϵ, numbers between $[0, 1]$, and then apply the equations given in eq. 5 with, max_distance $= \pm 4$, max_angle $= \pm\frac{3\pi}{4}$.

$$\begin{aligned}
p_{\text{spawn\_x}} =& p_{\text{ideal\_x}} + cos(p_{\text{ideal\_}\theta} + \pi/2) \times d \times \text{max\_distance} \\
p_{\text{spawn\_y}} =& p_{\text{ideal\_y}} + sin(p_{\text{ideal\_}\theta} + \pi/2) \times d \times \text{max\_distance} \\
p_{\text{spawn\_}\theta} =& p_{\text{ideal\_}\theta} + \text{arctan2}(-d, 5) \pm \epsilon \times \text{max\_angle}
\end{aligned} \tag{5}$$

Eq. 5 moves the spawning position along the normal to the ideal trajectory. Ideally, the boat would spawn on the perfect trajectory, and its angular position would be computed using the tangent to that trajectory. However, as the distance to the ideal trajectory becomes large, this angle is no longer ideal. This is why we introduce the term $\text{arctan2}(-d, 5)$, which sets the boat heading such that it reaches the ideal trajectory after five meters.

Examples of spawn positions generated by the curriculum can be seen in figure 12

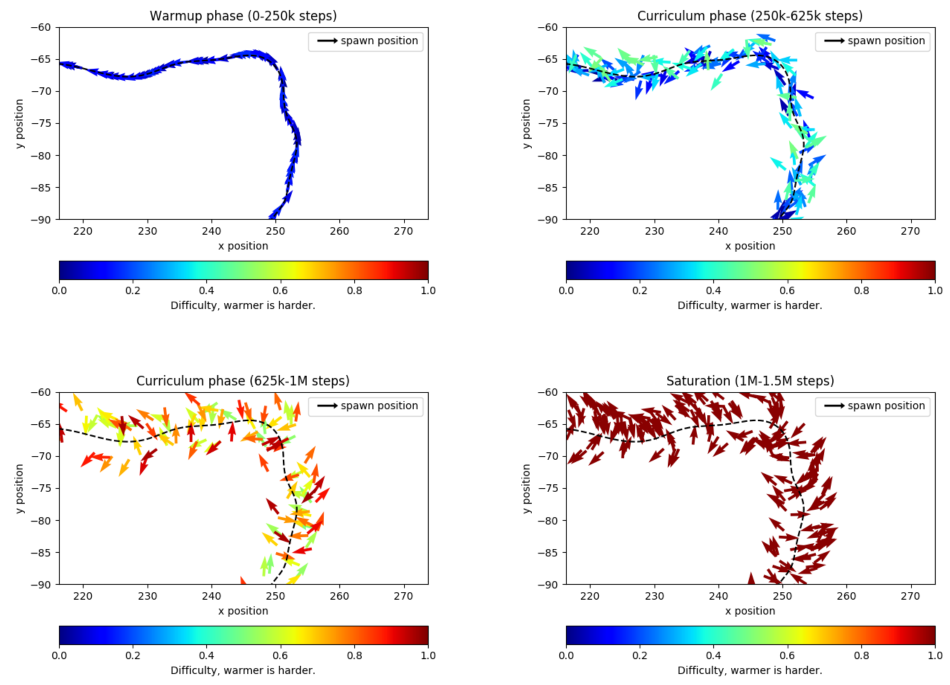


Figure 12: Possible spawn positions generated by the curriculum-based spawner. The training was cut into 4 phases for better readability. The warmer the color of the arrow, the higher the difficulty setting.

Our curriculum setup is designed such that the agent learns to drive around the lake in a single direction. To do so, we start by a warm-up phase of 250,000 steps, where the agent always spawns in the ideal position with the shore of the lake on its right. This creates a lot of samples on which the shore of the lake is on the right side of the agent. Naturally, this leads to the agent being more comfortable with the shore on its right side. We observed during field experiments that an agent trained this way would always fall back to this configuration. Counterintuitively, this behavior is highly desirable as an agent trained this way will never do u-turns and retrace its steps. During the rest of the curriculum, between the step 250,000 and 1,000,000, we gradually increase the value of mu. During this phase, the boat faces increasingly difficult situations. And finally, for the remaining steps we leave the curriculum at the maximum difficulty setting.

D.6 Learning Hardware

In simulation, the RL runs on Ubuntu 18.04 with ROS Melodic, CUDA 11.2, python 2.7 and custom compiled TensorFlow 2.3 binaries for python 2.7 and CUDA 11.2. If you would like to get access to our tensorflow wheel, feel free to reach out to us.

To run the simulation, infer (in simulation) and train the RL agents we use servers equipped with 2 NVIDIA RTX 3090, a 12 cores 24 threads Ryzen 9 3900X CPU and 128Gb of RAM.

E Real-Results

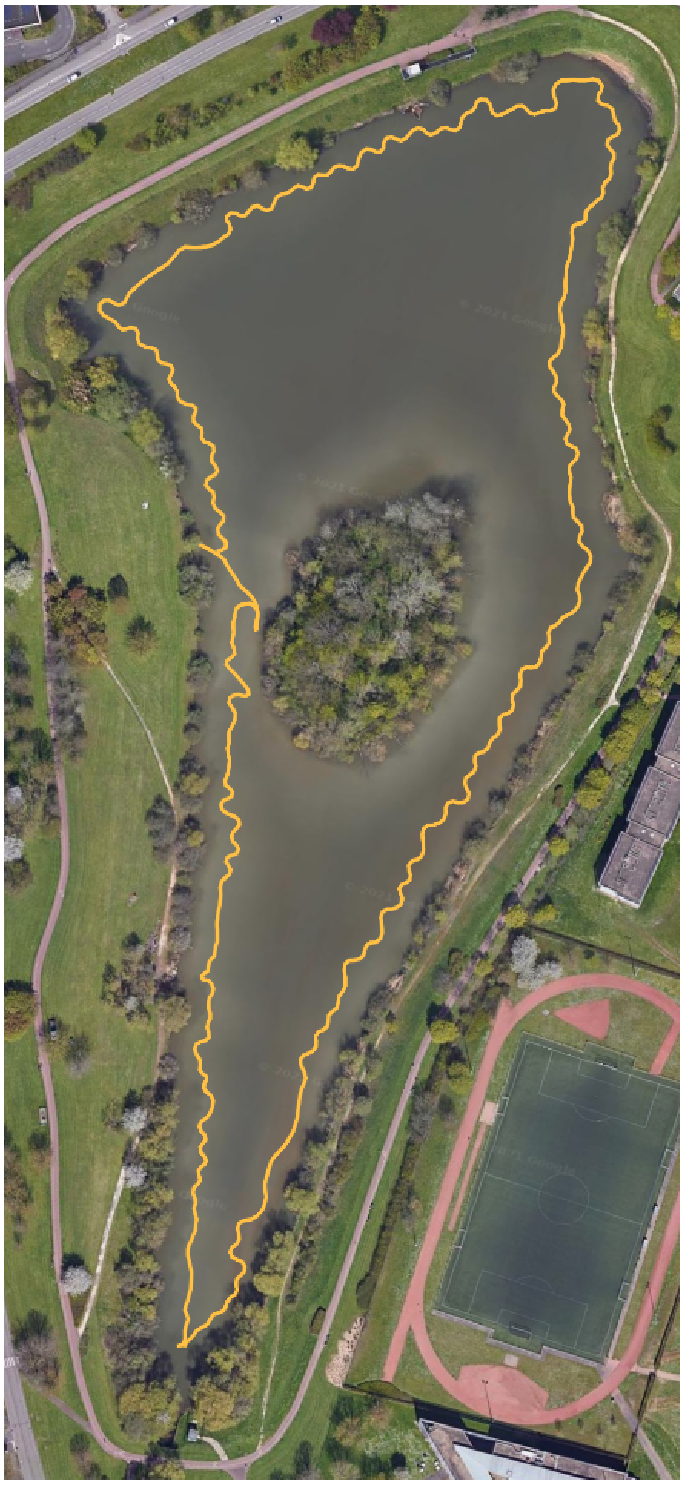


Figure 13: Original Dreamer trajectory around the lake.

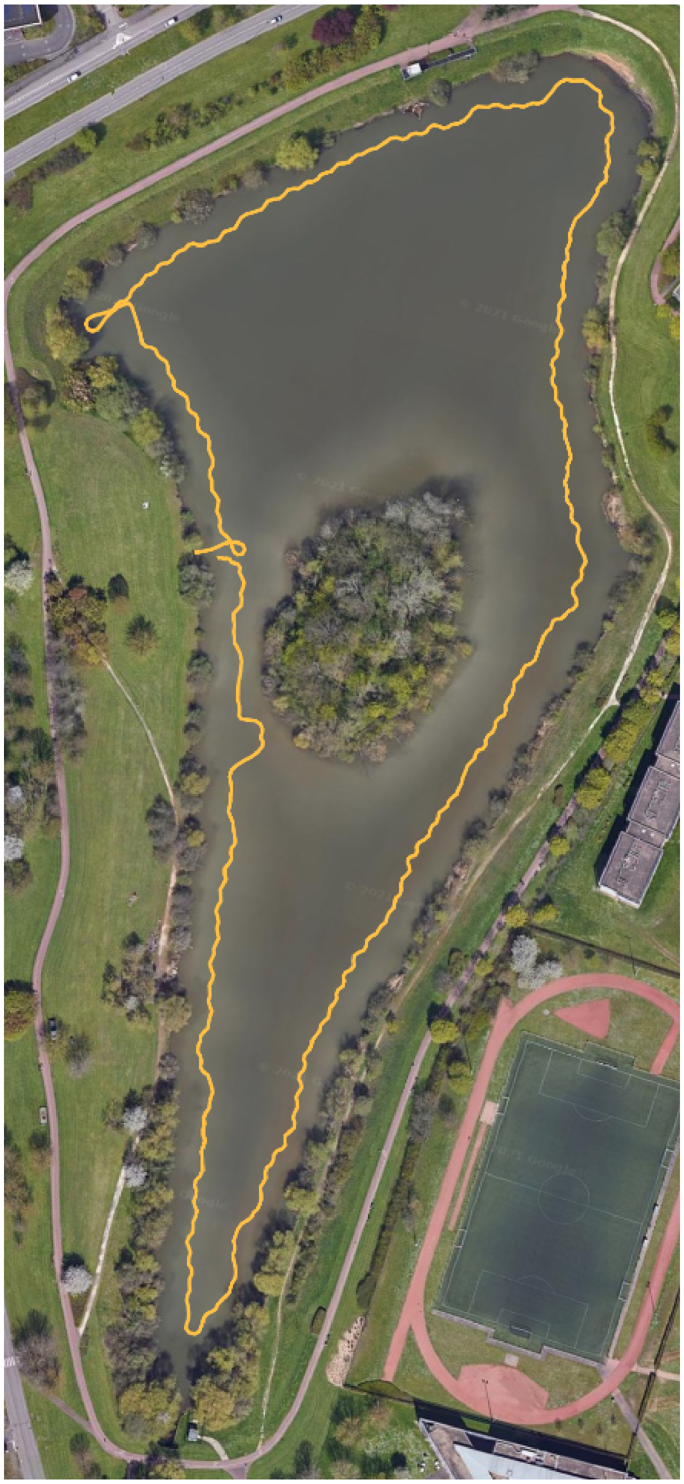


Figure 14: Our method trajectory around the lake.