IBOAT RL Documentation

Release 1.0.0

Tristan Karch

Jan 23, 2018

Contents

CHAPTER 1

A brief context

This project presents **Reinforcement Learning** as a solution to control systems with a **large hysteresis**. We consider an autonomous sailing robot (IBOAT) which sails upwind. In this configuration, the wingsail is almost aligned with the upcoming wind. It thus operates like a classical wing to push the boat forward. If the angle of attack of the wind coming on the wingsail is too great, the flow around the wing detaches leading to a **marked decrease of the boat's speed**.

Hysteresis such as stall are hard to model. We therefore proposes an **end-to-end controller** which learns the stall behavior and builds a policy that avoids it. Learning is performed on a simplified transition model representing the stochastic environment and the dynamic of the boat.

On this page, you will find the documentation of the simplified simulator of the boat as well as the documentation of the reinforcement learning tools. Each package contains tutorials to better understand how the code can be used

CHAPTER 2

Requirements

The project depends on the following extensions :

1. NumPy for the data structures (http://www.numpy.org)
2. Matplotlib for the visualisation (https://matplotlib.org)
3. Keras for the convolutional neural network models (https://keras.io)

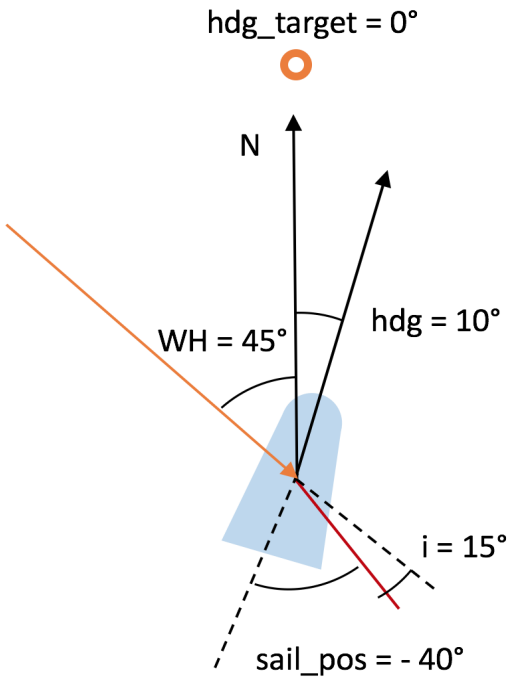
CHAPTER 3

Contents

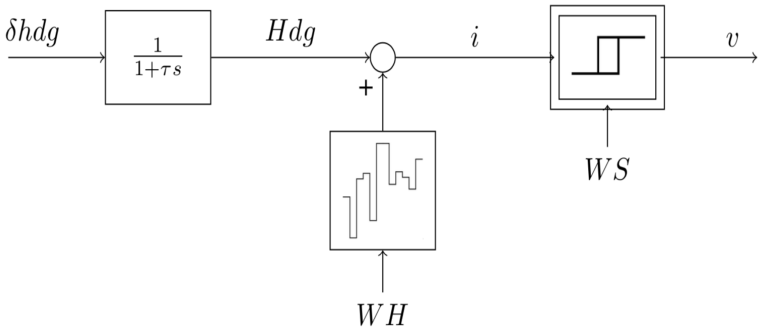
3.1 Package Sim

This package contains all the classes required to build a simulation for the learning. In this small paragraph, the physic of the simulator is described so that the reader can better understand the implementation.

We need the boat to be in a configuration when it sails upwind so that the flow around the sail is attached and the sail works as a wing. To generate the configuration we first assume that the boat as a target heading *hdg_target* = 0. The boat as a certain heading *hdg* with respect to the north and faces an upcoming wind of heading *WH*. To lower the number of parameters at stake we consider that the wind has a **constant speed** of 15 knts. The sail is oriented with respect to the boat heading with an angle *sail_pos* = -40°. The angle of attack of the wind on the sail is therefore equal to $i = hdg + WH + sail\_pos$. This angle equation can be well understood thanks to the following image.



The action taken to change the angle of attack are changes of boat heading *delta_hdg*. We therefore assume that *sail_pos* is constant and equal to -40°. The wind heading is fixed to $WH = 45°$. Finally, there is a delay between the command and the change of heading of $\tau = 0.5$ seconds. The simulator can be represented with the following block diagram. It contains a delay and an hysteresis block that are variables of the simulator class.



3.1.1 Simulator

Warning: Be careful, the delay is expressed has an offset of index. the delay in s is equal to delay*time_step

3.1.2 Hysteresis

3.1.3 Markov Decision Process (MDP)

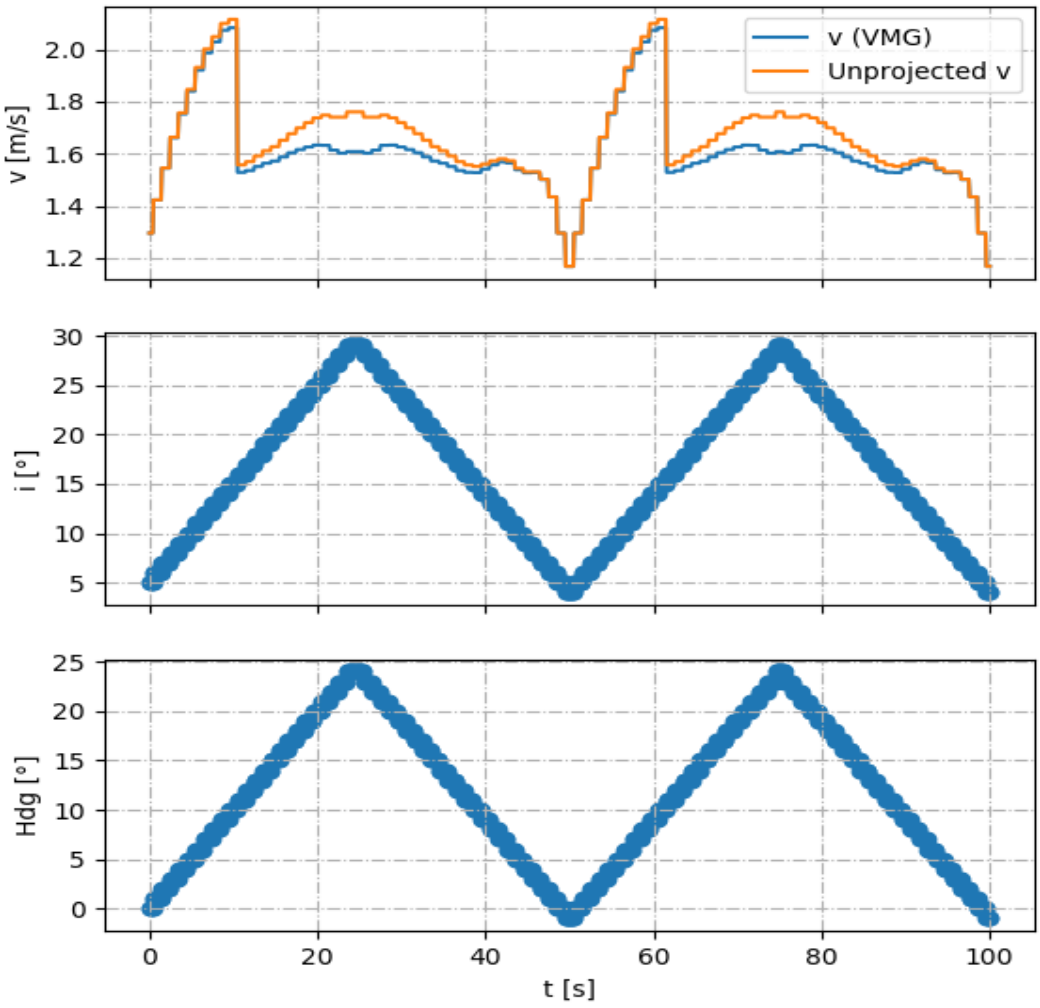
Note: The class variable simulation_duration defines the frequency of action taking. The reward is the average of the new velocities computed after each transition.

3.1.4 Tutorial

To visualize how a simulation can be generated we provide a file MDPmain.py that creates a simulation where the heading is first increase and then decrease.

```
TORAD = math.pi / 180

history_duration = 3
mdp_step = 1
time_step = 0.1
SP = -40 * TORAD
mdp = mdp.MDP(history_duration, mdp_step, time_step)

mean = 45 * TORAD
std = 0 * TORAD
wind_samples = 10
WH = np.random.uniform(mean - std, mean + std, size=10)

hdg0 = 0 * TORAD * np.ones(wind_samples)
state = mdp.initializeMDP(hdg0, WH)

SIMULATION_TIME = 100

i = np.ones(0)
vmg = np.ones(0)
wind_heading = np.ones(0)

for time in range(SIMULATION_TIME):
    print('t = {0} s'.format(time))
    action = 0
    WH = np.random.uniform(mean - std, mean + std, size=wind_samples)
    if time < SIMULATION_TIME / 4:
        action = 0
    elif time < SIMULATION_TIME / 2:
        action = 1
    elif time < 3 * SIMULATION_TIME / 4:
        action = 0
    else:
        action = 1

    nex_state, reward = mdp.transition(action, WH)
    next_state = state
    i = np.concatenate([i, mdp.extractSimulationData()[0, :]])
    vmg = np.concatenate([vmg, mdp.extractSimulationData()[1, :]])
    wind_heading = np.concatenate([wind_heading, WH])

time_vec = np.linspace(0, SIMULATION_TIME, int((SIMULATION_TIME) / time_step))
hdg = i - wind_heading - SP
```

This results in the following value for the velocity, angle of attack and heading.



3.2 Package RL

3.2.1 Policy Learner

3.2.2 Tutorial

```
history_duration = 3  # Duration of state history [s]
mdp_step = 1  # Step between each state transition [s]
time_step = 0.1  # time step [s] <-> 10Hz frequency of data acquisition
mdp = MDP(history_duration, mdp_step, time_step)

mean = 45 * TORAD
std = 0 * TORAD
```

```
wind_samples = 10
WH = np.random.uniform(mean - std, mean + std, size=10)

hdg0=0*np.ones(10)
mdp.initializeMDP(hdg0,WH)

hdg0_rand_vec=(-4,0,2,4,6,8,18,20,21,22,24)

action_size = 2
policy_angle = 18
agent = PolicyLearner(mdp.size, action_size, policy_angle)
#agent.load("policy_learning_i18_test_long_history")
batch_size = 120

EPISODES = 500

loss_of_episode = []
for e in range(EPISODES):
    WH = np.random.uniform(mean - std, mean + std, size=10)
    hdg0_rand = random.sample(hdg0_rand_vec, 1)[0]
    hdg0 = hdg0_rand * TORAD * np.ones(10)
    # initialize the incidence randomly
    mdp.simulator.hyst.reset()  #
    #  We reinitialize the memory of the flow
    state = mdp.initializeMDP(hdg0, WH)
    loss_sim_list = []
    for time in range(40):
        # print(time)
        WH = np.random.uniform(mean - std, mean + std, size=wind_samples)
        action = agent.actDeterministicallyUnderPolicy(state)
        next_state, reward = mdp.transition(action, WH)
        agent.remember(state, action, reward, next_state)
        state = next_state
        if len(agent.memory) > batch_size:
            loss_sim_list.append(agent.replay(batch_size))
    loss_over_simulation_time = np.sum(np.array([loss_sim_list])[0]) / len(np.
↪array([loss_sim_list])[0])
    loss_of_episode.append(loss_over_simulation_time)
    print("Initial Heading : {}".format(hdg0_rand))
    print("episode: {}/{}, Mean Loss = {}"
          .format(e, EPISODES, loss_over_simulation_time))
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search