Introduction to ROS

Group Project: Quadrotor Reinforcement-Learning

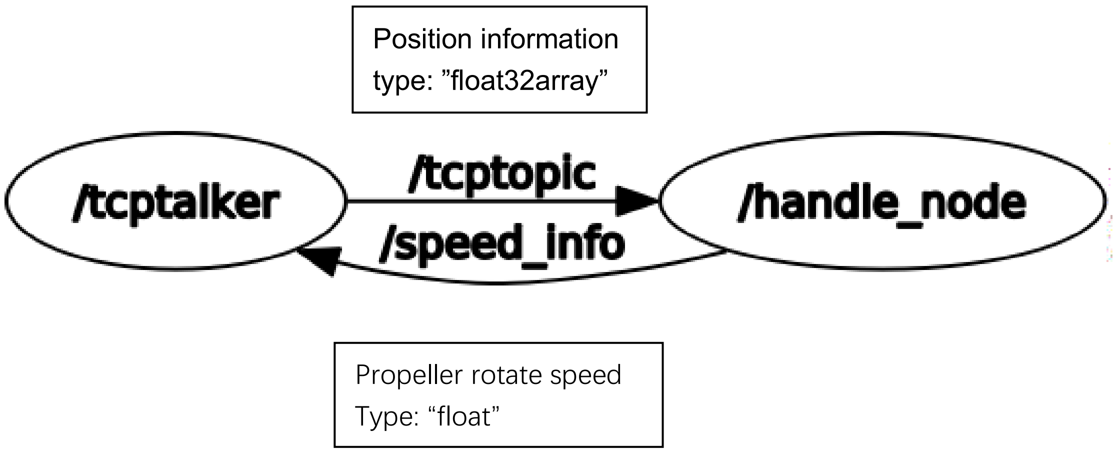
Tao Ma, Camilo Andrés Rodríguez Salamanca

Abstract - This report details the code used for the project. The operation of the nodes created is explained, as well as the topics and the information they contain. The results will also be showed, both numerically and graphically, for analysis.

Ros graph:

Node “**tcptalker**” is in file “connect”. It publishes the position information of quadrotor with topic “tcptopic” and subscribes rotate speed information with topic “speed_info”

Node “**handle_node**” is in file “handleNode”, it subscribes the position information from topic “tcptopic” and publishes the propeller rotate speed information to topic “speed_info”.



Project structure:

- orchestrator: unity model
- project_rl/src/orchestrator/
    - msg
        - float32array.msg: data type, transport position information
    - launch
        - run.launch: launch file
    - scripts
        - connect: executable file, connect with unity model
        - handleNode: executable file, associate task, agent and ros node
        - ddpg/agents
            - actor.py: actor model
            - critic.py: critic model
            - ddpg.py: ddpg agent with Replay buffer and OU Noise
        - ddpg/tasks

- takoff.py: quadcopter agent reward formulation

Function explanation:

The class connect has four functions and one purposes:

1. It initializes the connection with the environment. It publishes the position information of the simulation and subscribes to the RL speed.

Functions:

    __init__:

        It allows the class to initialize the attributes of a class. For this case, the function initializes the connection between ROS and the simulation environment, which is an environment based on Unity.

    start:

        This instance to send the speed to the propellers of the quadrotor that is in simulation environment, it also receives the position of the simulation environment of the quadrotor and publishes it.

    callback:

        This function is used to subscribe to the information that is being published from another node.

    empty_socket:

        This function clears the information stored in the socket.

The code on which this code was based was the configuration code provided by the teacher in charge [1]. For the correct configuration of the work environment, subscribers, publishers and nodes were created. In addition, the script was given the class structure and instances.

The infrastructure of class HandleNode and MyTask is from another project [2].
The class HandleNode has five functions and it has two purposes:

1. Subscribes the quadrotor position information from topic “tcptopic” and publishes the propeller rotate speed to topic “speed_info”
2. Initializes task and agent instance and start to run algorithm.

Functions:

    __init__:

        initializes ros node, publisher and subscriber. Sets up necessary parameters for algorithm and initializes the task and agent instance

    start:

        starts the program, at first resets the episode and then starts to run the loop

    loop:

        transmits the from callback function obtained position information and timestamp into task instance in order to get calculated propeller rotate speed and flag. According to the flag decides if the current episode is finished, publishes current propeller rotate speed to topic “speed_info”

reset:

resets the state and propeller rotate speed. Initializes new episode parameters and decides if the current episode number more than maximal number of episodes.

callback:

uses a mutex to guarantee that every time only one message would be processed. Calculates the current timestamp and extracts the quadrotor position information from message.

In Reinforcement learning, there are two main parts:

1. Task part, initializes an environment so that agent can interact with it.
2. Agent part, according to the current information such as reward, propeller rotate speed, current position to make decision.

The main class of task part is “MyTask”:

__init__:

Initializes the observation space and action space as well as task specific parameters. Logs the current time as start time which is aiming to calculate of the whole run time of program

reset:

resets the position and timestamp

update:

receives the position and timestep information from “handleNode” class to calculate the current propeller rotate speed and reward. Uses last state, last action, last reward, current state and done to train the RL network. At the mean time output total run time of program.

get_reward:

according to the current position of quadrotor to calculate the current reward.

set_agent:

at the initialization set an agent to carry out this task

The second part is agent, which is used the external code [3] to realize DDPG algorithm. It receives current position of quadrotor as input and through algorithm output propeller rotate speed.

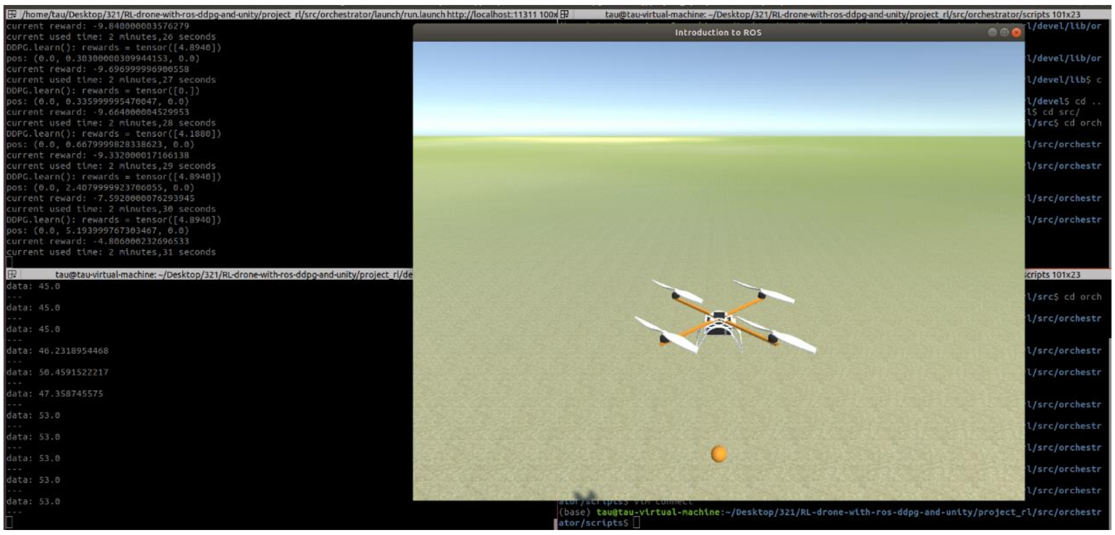
Result

As a result, the program can successfully receive the position information from unity and send it through ros topic to reinforcement learning algorithm. After the calculation of algorithm, the program would send the wished propeller rotate speed back to unity model. This project realized the connection between unity model, ros node and reinforcement learning algorithm.

But because of limited time we still can’t let the quadrotor after 1 hour reinforcement

learning successfully stays in the certain height. Though the output information of algorithm showed that it has chosen the action with the best reward. The best reward it can reach is 5.
Possible reason is that ddpg algorithm uses deep learning network which need time to train and compute. it leads to a delay of information transmission. The output rotate speed is no more real-time.

Screen shot at the beginning



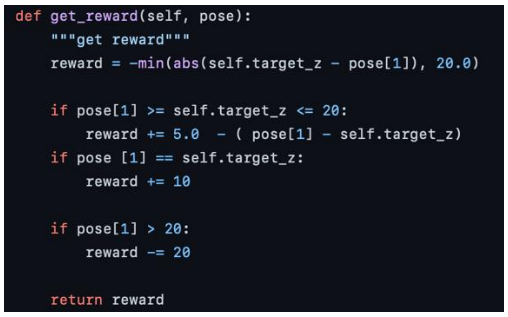
```
/home/tau/Desktop/321/RL-drone-with-ros-ddpg-and-unity/project_rl/src/orchestrator/launch/run.launch http://localhost:11311 100x
current reward: -9.840000003576279
current used time: 2 minutes,26 seconds
DDPG.learn(): rewards = tensor([4.8940])
pos: (0.0, 0.30300000309944153, 0.0)
current reward: -9.696999996900558
current used time: 2 minutes,27 seconds
DDPG.learn(): rewards = tensor([0.])
pos: (0.0, 0.335999995470047, 0.0)
current reward: -9.664000004529953
current used time: 2 minutes,28 seconds
DDPG.learn(): rewards = tensor([4.1880])
pos: (0.0, 0.6679999828338623, 0.0)
current reward: -9.332000017166138
current used time: 2 minutes,29 seconds
DDPG.learn(): rewards = tensor([4.8940])
pos: (0.0, 2.4079999923706055, 0.0)
current reward: -7.5920000076293945
current used time: 2 minutes,30 seconds
DDPG.learn(): rewards = tensor([4.8940])
pos: (0.0, 5.193999767303467, 0.0)
current reward: -4.806000232696533
current used time: 2 minutes,31 seconds
```

```
tau@tau-virtual-machine: ~/Desktop/321/RL-drone-with-ros-ddpg-and-unity/project_rl/de
data: 45.0
---
data: 45.0
---
data: 45.0
---
data: 46.2318954468
---
data: 50.4591522217
---
data: 47.358745575
---
data: 53.0
---
data: 53.0
---
data: 53.0
---
data: 53.0
---
data: 53.0
---
```

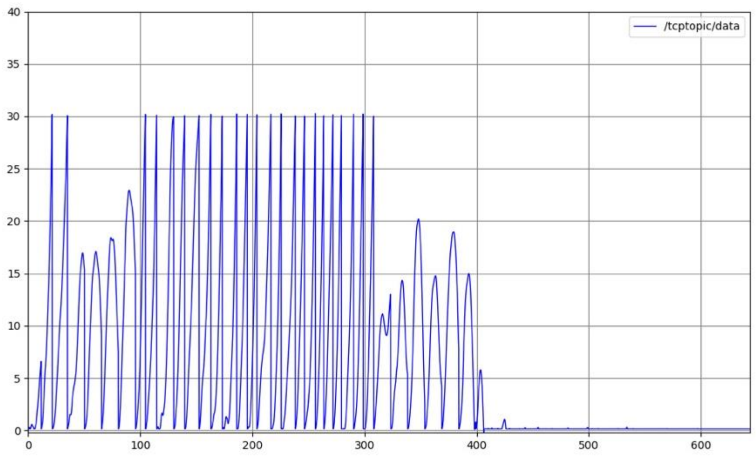
```
tau@tau-virtual-machine: ~/Desktop/321/RL-drone-with-ros-ddpg-and-unity/project_rl/src/orchestrator/scripts 101x23
l/devel/lib/or
l/devel/lib/or
l/devel/lib$ c
l/devel$ cd ..
l$ cd src/
l/src$ cd orch
l/src/orchestr
l/src/orchestr
l/src/orchestr
l/src/orchestr
cripts 101x23
l/src$ cd orch
l/src/orchestr
l/src/orchestr
l/src/orchestr
l/src/orchestr
l/src/orchestr
l/src/orchestr
l/src/orchestr
l/src/orchestr
(base) tau@tau-virtual-machine:~/Desktop/321/RL-drone-with-ros-ddpg-and-unity/project_rl/src/orchestr
ator/scripts$
```

Results Analysis

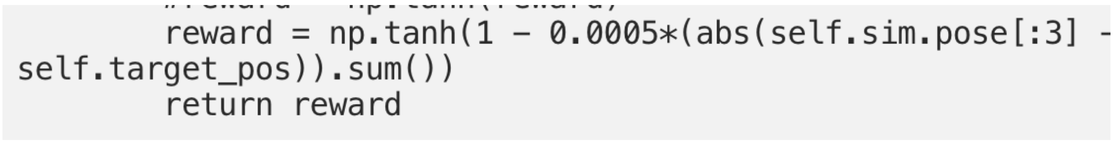
Even though the result was not what was expected, several solutions were tried using the reward. In all of them there are moments where it approaches the desired height. But nowhere is it possible to keep at this height.

1. 

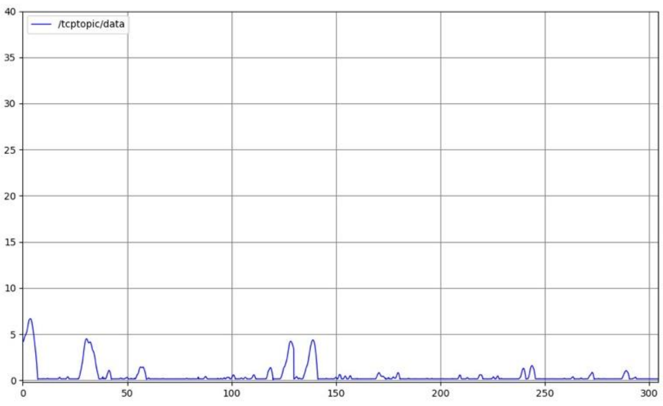
```python
def get_reward(self, pose):
    """get reward"""
    reward = -min(abs(self.target_z - pose[1]), 20.0)

    if pose[1] >= self.target_z <= 20:
        reward += 5.0  - ( pose[1] - self.target_z)
    if pose [1] == self.target_z:
        reward += 10

    if pose[1] > 20:
        reward -= 20

    return reward
```



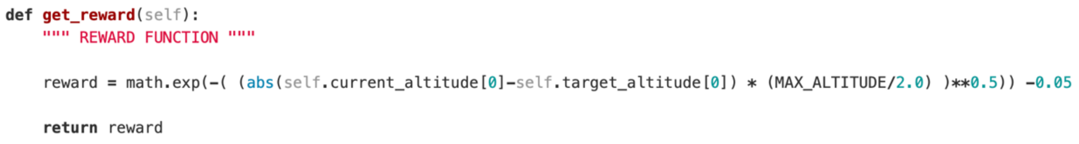
2.



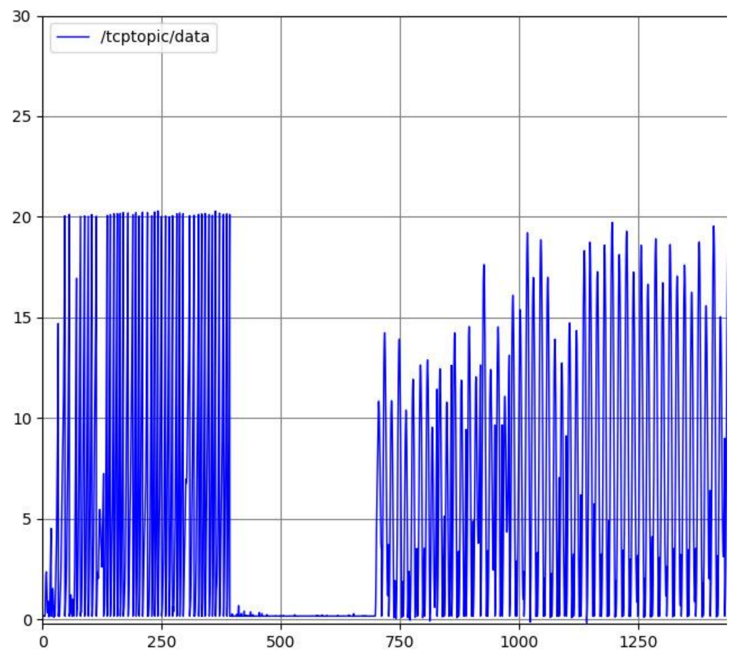
```
        #reward = np.tanh(reward)
        reward = np.tanh(1 - 0.0005*(abs(self.sim.pose[:3] -
self.target_pos)).sum())
        return reward
```



3.



```
def get_reward(self):
    """ REWARD FUNCTION """

    reward = math.exp(-( (abs(self.current_altitude[0]-self.target_altitude[0]) * (MAX_ALTITUDE/2.0) )**0.5)) -0.05

    return reward
```



The third reward was the function that gave the best results, because at one point it began to reach the desired value

work allocation

Camilo:

1. package build
2. write “connect” file and build node “tcptalker” to establish the connection between quadrotor and ros, it publishes position information as well as subscribes rotate speed information

Tao:

1. write “handleNode” file and build node “handleNode” to subscribe the position information and publish the rotate speed information. It will start the reinforcement learning algorithm
2. write “task” file which is inherited from gym library [4] to estimate the action from agent
3. search and read ddpg algorithm examples [2] and change the external code to fit with our project
4. write written summary

[1] Setup conexion, Setup
[2] Deep RL Quadcopter Controller, https://github.com/1jsingh/RL_quadcopter
[3] Deep RL Quadcopter Controller, https://github.com/1jsingh/RL_quadcopter
[4] How to create new environments for Gym
https://github.com/openai/gym/blob/master/docs/creating-environments.md