Quadcopter_Project

December 8, 2018

1 Project: Train a Quadcopter How to Fly

Design an agent to fly a quadcopter, and then train it using a reinforcement learning algorithm of your choice!

Try to apply the techniques you have learnt, but also feel free to come up with innovative ideas and test them.

1.1 Instructions

Take a look at the files in the directory to better understand the structure of the project.

- `task.py`: Define your task (environment) in this file.
- `agents/`: Folder containing reinforcement learning agents.
  - `policy_search.py`: A sample agent has been provided here.
  - `agent.py`: Develop your agent here.
- `physics_sim.py`: This file contains the simulator for the quadcopter. **DO NOT MODIFY THIS FILE**.

For this project, you will define your own task in `task.py`. Although we have provided a example task to get you started, you are encouraged to change it. Later in this notebook, you will learn more about how to amend this file.

You will also design a reinforcement learning agent in `agent.py` to complete your chosen task.

You are welcome to create any additional files to help you to organize your code. For instance, you may find it useful to define a `model.py` file defining any needed neural network architectures.

1.2 Controlling the Quadcopter

We provide a sample agent in the code cell below to show you how to use the sim to control the quadcopter. This agent is even simpler than the sample agent that you'll examine (in `agents/policy_search.py`) later in this notebook!

The agent controls the quadcopter by setting the revolutions per second on each of its four rotors. The provided agent in the `Basic_Agent` class below always selects a random action for each of the four rotors. These four speeds are returned by the `act` method as a list of four floating-point numbers.

For this project, the agent that you will implement in `agents/agent.py` will have a far more intelligent method for selecting actions!

```
In [35]: import random

         class Basic_Agent():
             def __init__(self, task):
                 self.task = task

             def act(self):
                 new_thrust = random.gauss(450., 25.)
                 return [new_thrust + random.gauss(0., 1.) for x in range(4)]
```

Run the code cell below to have the agent select actions to control the quadcopter.

Feel free to change the provided values of `runtime`, `init_pose`, `init_velocities`, and `init_angle_velocities` below to change the starting conditions of the quadcopter.

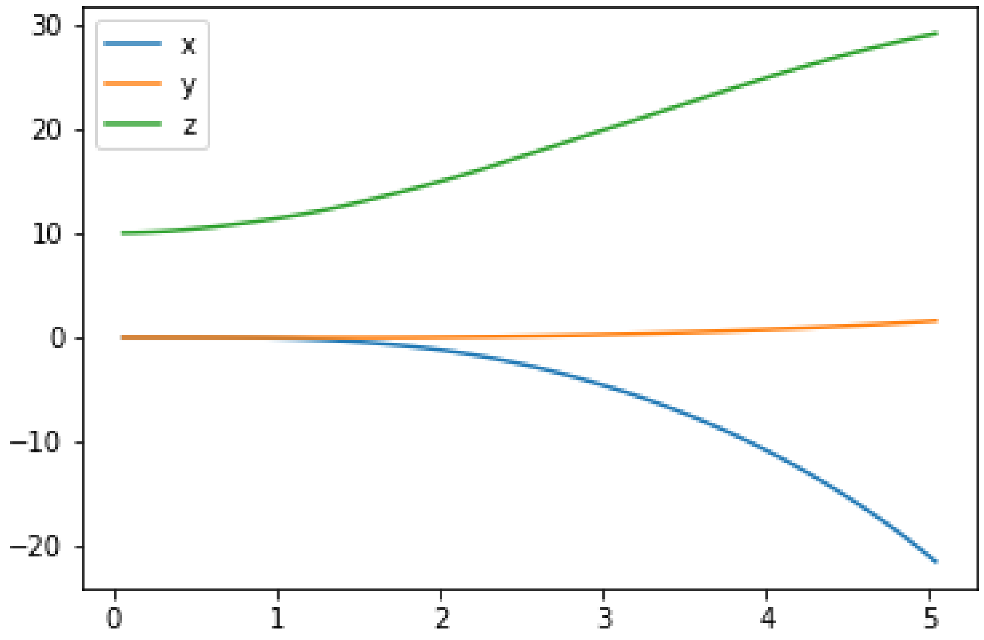
The `labels` list below annotates statistics that are saved while running the simulation. All of this information is saved in a text file `data.txt` and stored in the dictionary `results`.

```
In [36]: %load_ext autoreload
         %autoreload 2

         import csv
         import numpy as np
         from task import Task

         # Modify the values below to give the quadcopter a different starting position.
         runtime = 5.                                     # time limit of the episode
         init_pose = np.array([0., 0., 10., 0., 0., 0.])  # initial pose
         init_velocities = np.array([0., 0., 0.])         # initial velocities
         init_angle_velocities = np.array([0., 0., 0.])   # initial angle velocities
         file_output = 'data.txt'                         # file name for saved results

         # Setup
         task = Task(init_pose, init_velocities, init_angle_velocities, runtime)
         agent = Basic_Agent(task)
         done = False
         labels = ['time', 'x', 'y', 'z', 'phi', 'theta', 'psi', 'x_velocity',
                   'y_velocity', 'z_velocity', 'phi_velocity', 'theta_velocity',
                   'psi_velocity', 'rotor_speed1', 'rotor_speed2', 'rotor_speed3', 'rotor_speed4
         results = {x : [] for x in labels}

         # Run the simulation, and save the results.
         with open(file_output, 'w') as csvfile:
             writer = csv.writer(csvfile)
             writer.writerow(labels)
             while True:
                 rotor_speeds = agent.act()
                 _, _, done = task.step(rotor_speeds)
                 to_write = [task.sim.time] + list(task.sim.pose) + list(task.sim.v) + list(task
                 for ii in range(len(labels)):
                    results[labels[ii]].append(to_write[ii])
                writer.writerow(to_write)
                if done:
                    break
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

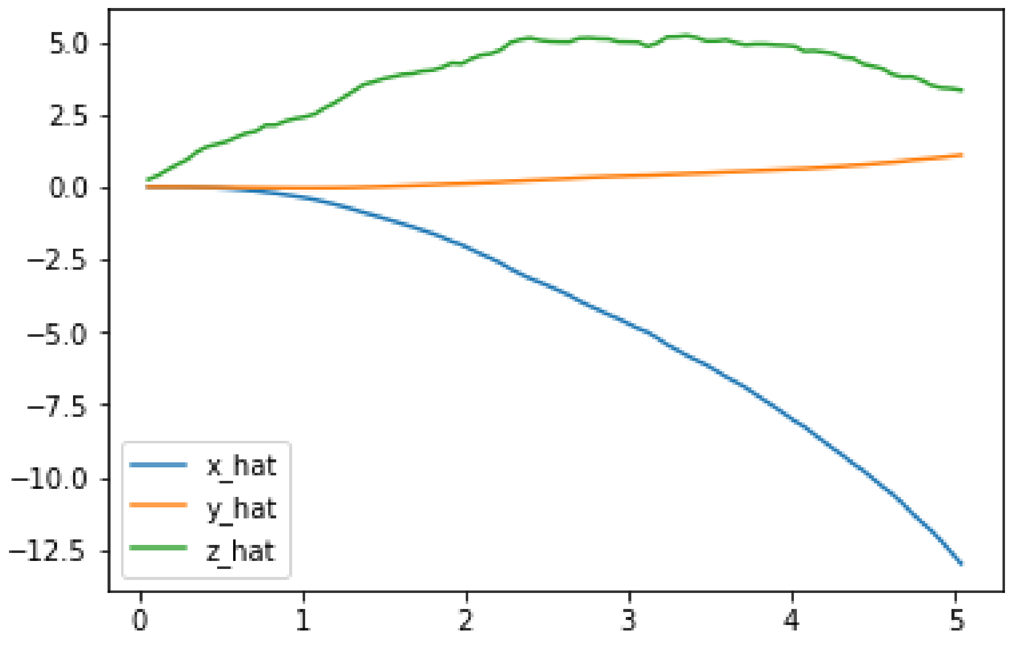
Run the code cell below to visualize how the position of the quadcopter evolved during the simulation.

```
In [37]: import matplotlib.pyplot as plt
         %matplotlib inline

         plt.plot(results['time'], results['x'], label='x')
         plt.plot(results['time'], results['y'], label='y')
         plt.plot(results['time'], results['z'], label='z')
         plt.legend()
         _ = plt.ylim()
```



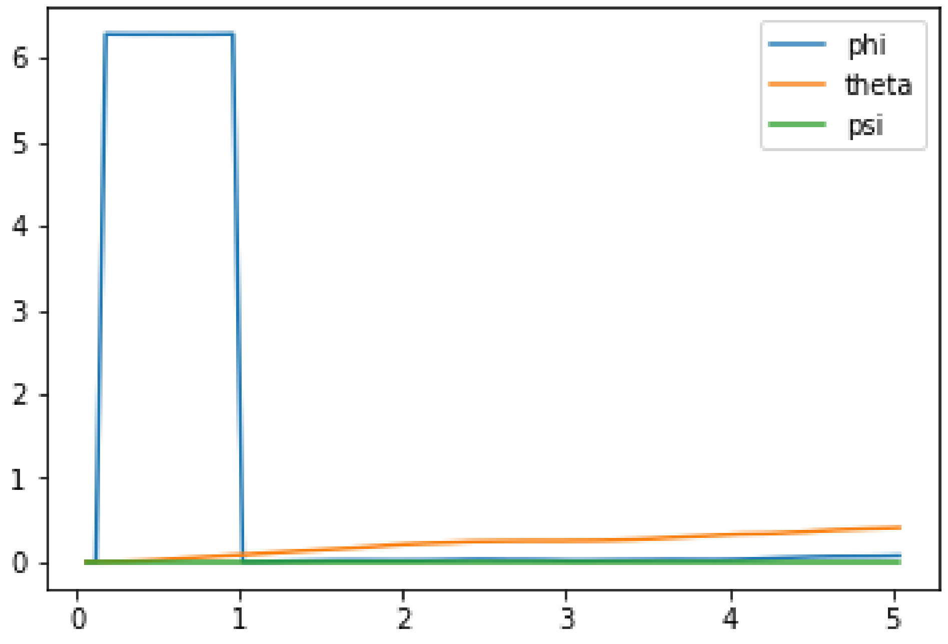
The next code cell visualizes the velocity of the quadcopter.

```
In [38]: plt.plot(results['time'], results['x_velocity'], label='x_hat')
         plt.plot(results['time'], results['y_velocity'], label='y_hat')
         plt.plot(results['time'], results['z_velocity'], label='z_hat')
         plt.legend()
         _ = plt.ylim()
```



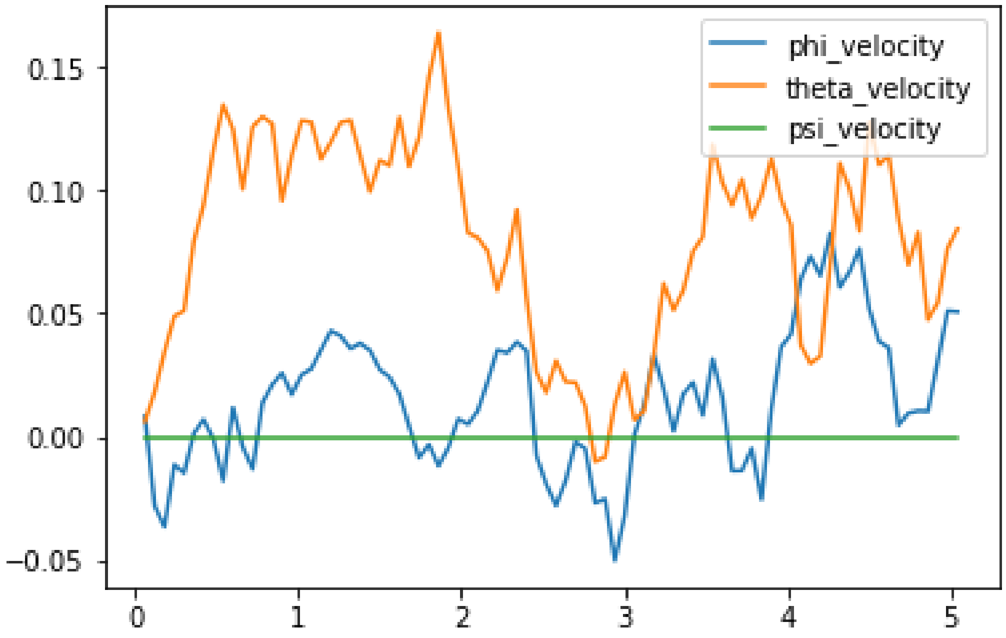
Next, you can plot the Euler angles (the rotation of the quadcopter over the *x*-, *y*-, and *z*-axes),

```
In [39]: plt.plot(results['time'], results['phi'], label='phi')
         plt.plot(results['time'], results['theta'], label='theta')
         plt.plot(results['time'], results['psi'], label='psi')
         plt.legend()
         _ = plt.ylim()
```



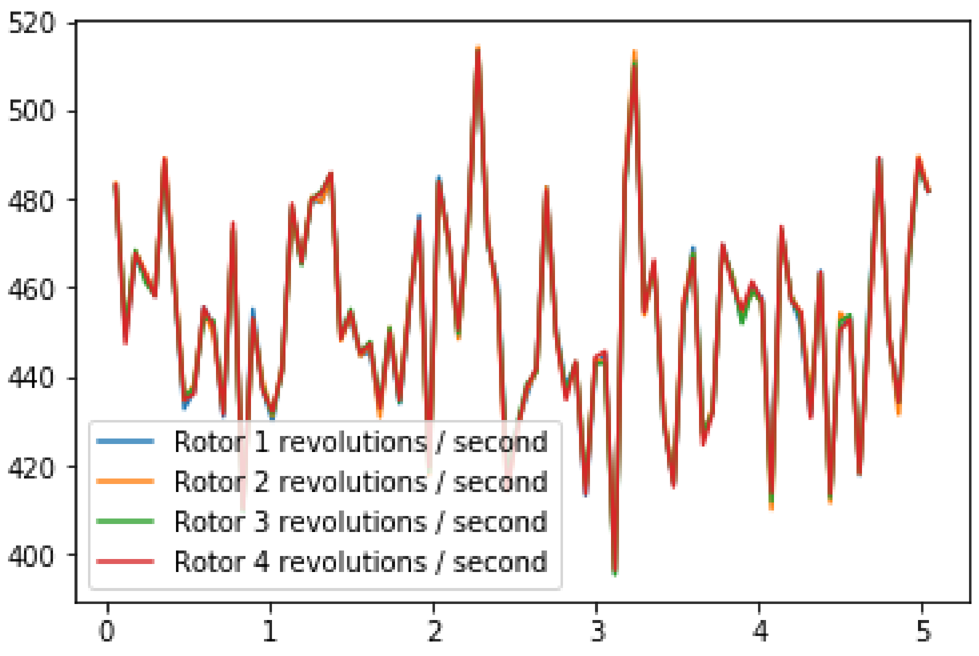
before plotting the velocities (in radians per second) corresponding to each of the Euler angles.

```
In [40]: plt.plot(results['time'], results['phi_velocity'], label='phi_velocity')
         plt.plot(results['time'], results['theta_velocity'], label='theta_velocity')
         plt.plot(results['time'], results['psi_velocity'], label='psi_velocity')
         plt.legend()
         _ = plt.ylim()
```



Finally, you can use the code cell below to print the agent's choice of actions.

```
In [41]: plt.plot(results['time'], results['rotor_speed1'], label='Rotor 1 revolutions / second'
         plt.plot(results['time'], results['rotor_speed2'], label='Rotor 2 revolutions / second'
         plt.plot(results['time'], results['rotor_speed3'], label='Rotor 3 revolutions / second'
         plt.plot(results['time'], results['rotor_speed4'], label='Rotor 4 revolutions / second'
         plt.legend()
         _ = plt.ylim()
```



When specifying a task, you will derive the environment state from the simulator. Run the code cell below to print the values of the following variables at the end of the simulation: - `task.sim.pose` (the position of the quadcopter in (x, y, z) dimensions and the Euler angles), - `task.sim.v` (the velocity of the quadcopter in (x, y, z) dimensions), and - `task.sim.angular_v` (radians/second for each of the three Euler angles).

```
In [42]: # the pose, velocity, and angular velocity of the quadcopter at the end of the episode
         print(task.sim.pose)
         print(task.sim.v)
         print(task.sim.angular_v)
```

```
[-21.48325841   1.59223531  29.10852569   0.07687619   0.40804527   0.        ]
[-12.96578682   1.09563617   3.32770273]
[ 0.05072074  0.08436597  0.        ]
```

In the sample task in `task.py`, we use the 6-dimensional pose of the quadcopter to construct the state of the environment at each timestep. However, when amending the task for your purposes, you are welcome to expand the size of the state vector by including the velocity information. You can use any combination of the pose, velocity, and angular velocity - feel free to tinker here, and construct the state to suit your task.

1.3 The Task

A sample task has been provided for you in `task.py`. Open this file in a new window now.

The `__init__()` method is used to initialize several variables that are needed to specify the task.

- The simulator is initialized as an instance of the `PhysicsSim` class (from `physics_sim.py`).
- Inspired by the methodology in the original DDPG paper, we make use of action repeats. For each timestep of the agent, we step the simulation `action_repeats` timesteps. If you are not familiar with action repeats, please read the **Results** section in the DDPG paper. - We set the number of elements in the state vector. For the sample task, we only work with the 6-dimensional pose information. To set the size of the state (`state_size`), we must take action repeats into account.
- The environment will always have a 4-dimensional action space, with one entry for each rotor (`action_size=4`). You can set the minimum (`action_low`) and maximum (`action_high`) values of each entry here. - The sample task in this provided file is for the agent to reach a target position. We specify that target position as a variable.

The `reset()` method resets the simulator. The agent should call this method every time the episode ends. You can see an example of this in the code cell below.

The `step()` method is perhaps the most important. It accepts the agent's choice of action `rotor_speeds`, which is used to prepare the next state to pass on to the agent. Then, the reward is computed from `get_reward()`. The episode is considered done if the time limit has been exceeded, or the quadcopter has travelled outside of the bounds of the simulation.

In the next section, you will learn how to test the performance of an agent on this task.

1.4 The Agent

The sample agent given in `agents/policy_search.py` uses a very simplistic linear policy to directly compute the action vector as a dot product of the state vector and a matrix of weights. Then, it randomly perturbs the parameters by adding some Gaussian noise, to produce a different policy. Based on the average reward obtained in each episode (`score`), it keeps track of the best set of parameters found so far, how the score is changing, and accordingly tweaks a scaling factor to widen or tighten the noise.

Run the code cell below to see how the agent performs on the sample task.

```
In [43]: import sys
         import pandas as pd
         from agents.policy_search import PolicySearch_Agent
         from task import Task

         num_episodes = 1000
         target_pos = np.array([0., 0., 10.])
         task = Task(target_pos=target_pos)
         agent = PolicySearch_Agent(task)

         for i_episode in range(1, num_episodes+1):
             state = agent.reset_episode() # start a new episode
             while True:
                 action = agent.act(state)
                 next_state, reward, done = task.step(action)
                 agent.step(reward, done)
                 state = next_state
                 if done:
                     print("\rEpisode = {:4d}, score = {:7.3f} (best = {:7.3f}), noise_scale = {
                         i_episode, agent.score, agent.best_score, agent.noise_scale), end="")
```

```
                break
            sys.stdout.flush()
```

```
Episode = 1000, score =   6.834 (best =   6.842), noise_scale = 3.2625
```

This agent should perform very poorly on this task. And that's where you come in!

1.5 Define the Task, Design the Agent, and Train Your Agent!

Amend `task.py` to specify a task of your choosing. If you're unsure what kind of task to specify, you may like to teach your quadcopter to takeoff, hover in place, land softly, or reach a target pose.

After specifying your task, use the sample agent in `agents/policy_search.py` as a template to define your own agent in `agents/agent.py`. You can borrow whatever you need from the sample agent, including ideas on how you might modularize your code (using helper methods like `act()`, `learn()`, `reset_episode()`, etc.).

Note that it is **highly unlikely** that the first agent and task that you specify will learn well. You will likely have to tweak various hyperparameters and the reward function for your task until you arrive at reasonably good behavior.

As you develop your agent, it's important to keep an eye on how it's performing. Use the code above as inspiration to build in a mechanism to log/save the total rewards obtained in each episode to file. If the episode rewards are gradually increasing, this is an indication that your agent is learning.

```
In [44]: %load_ext autoreload
         %reload_ext autoreload
         %autoreload 2

         from agents.agent import DDPG_Agent

         import sys
         import pandas as pd

         from task import Task

         num_episodes = 1000
         r2 = []

         target_pos = np.array([0., 0., 10.])
         task = Task(target_pos=target_pos)
         agent = DDPG_Agent(task)
         for i_episode in range(1, num_episodes+1):
             state = agent.reset_episode() # start a new episode
             while True:
                 action = agent.act(state)
                 next_state, reward, done = task.step(action)

                 agent.step(action, reward, next_state, done)
                 state = next_state
                 if done:
```

```
                print("\rEpisode = {:4d}, score = {:7.3f} (best = {:7.3f})".format(
                    i_episode, agent.score, agent.best_score), end="")
                r2.append(reward)
                break
        sys.stdout.flush()


    # Testing performance

    file_output = 'data.txt'

    labels = ['time', 'x', 'y', 'z', 'phi', 'theta', 'psi', 'x_velocity',
              'y_velocity', 'z_velocity', 'phi_velocity', 'theta_velocity',
              'psi_velocity', 'rotor_speed1', 'rotor_speed2', 'rotor_speed3', 'rotor_speed4
    results_ddpg = {x : [] for x in labels}
    n_episodes = 20
    # Run the simulation, and save the results.
    with open(file_output, 'w') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(labels)
        state = agent.reset_episode() # start a new episode
        for i in range(n_episodes):

            while True:
                rotor_speeds= agent.act(state)
                next_state, reward, done = task.step(rotor_speeds)
                agent.step(action, reward, next_state, done)
                state = next_state
                to_write = [task.sim.time] + list(task.sim.pose) + list(task.sim.v) + list(
                for ii in range(len(labels)):
                    results_ddpg[labels[ii]].append(to_write[ii])
                writer.writerow(to_write)

                if done:
                    break
```

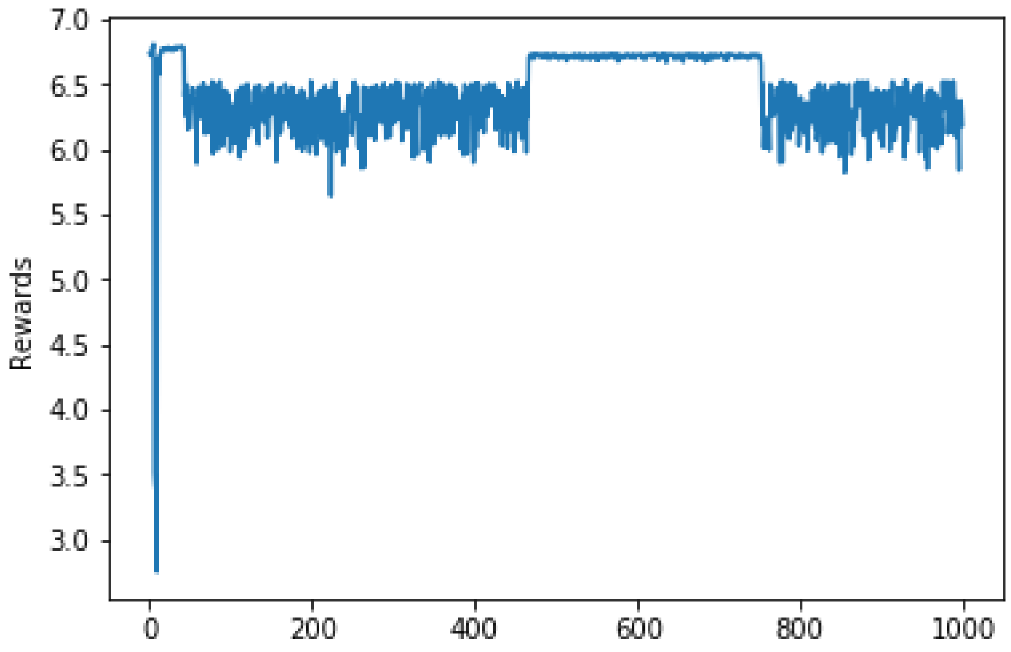
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
Episode = 1000, score =   6.647 (best =   6.854)
```

1.6 Plot the Rewards

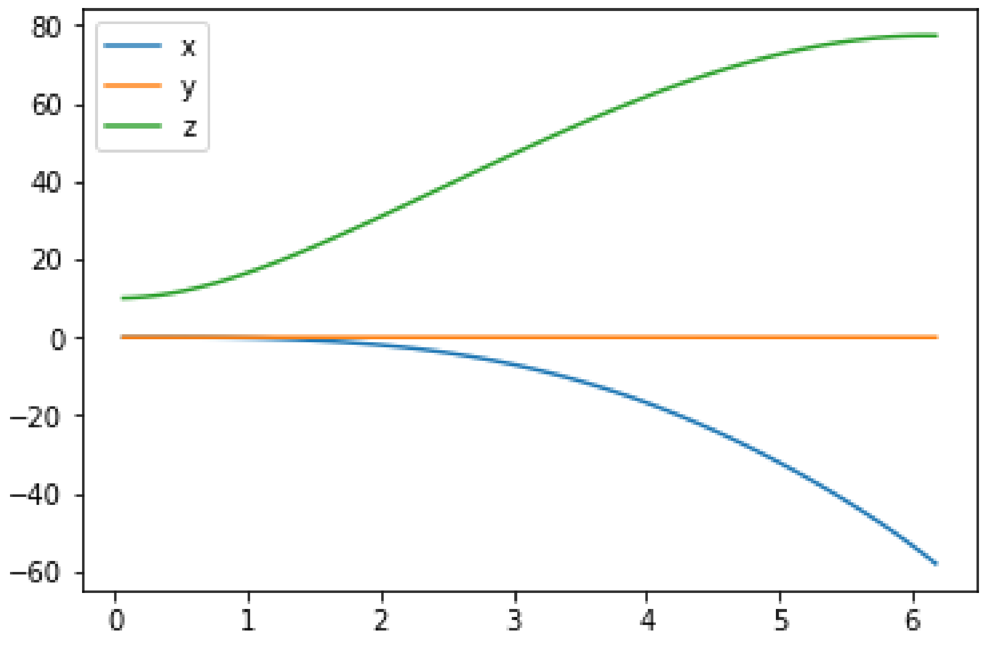
Once you are satisfied with your performance, plot the episode rewards, either from a single run, or averaged over multiple runs.

```
In [45]: #Plot documentation reference:
         #https://matplotlib.org/users/pyplot_tutorial.html
         # Position of the quadcopter evolved during the simulation
```

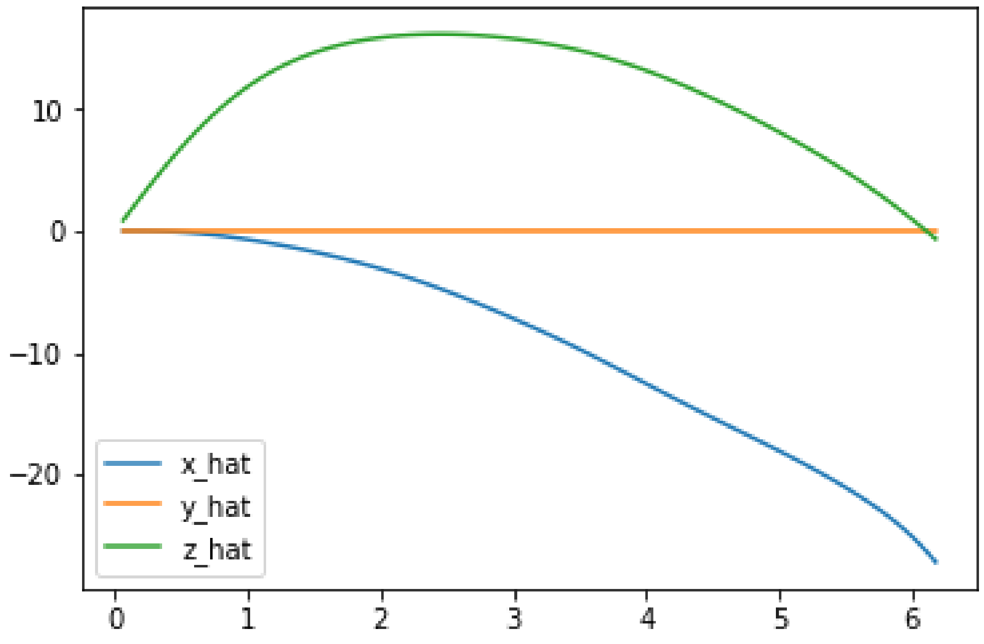
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(r2)
plt.ylabel('Rewards')
plt.show()
```



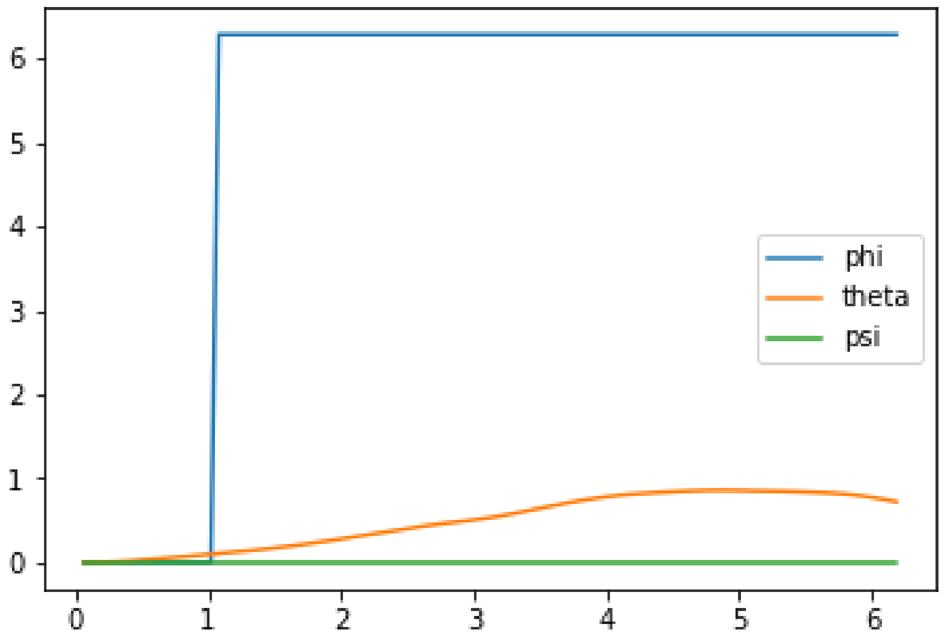
```
In [46]: #Visualizing of how the position of the quadcopter evolved during the simulation.
         plt.plot(results_ddpg['time'], results_ddpg['x'], label='x')
         plt.plot(results_ddpg['time'], results_ddpg['y'], label='y')
         plt.plot(results_ddpg['time'], results_ddpg['z'], label='z')
         plt.legend()
         _ = plt.ylim()
```



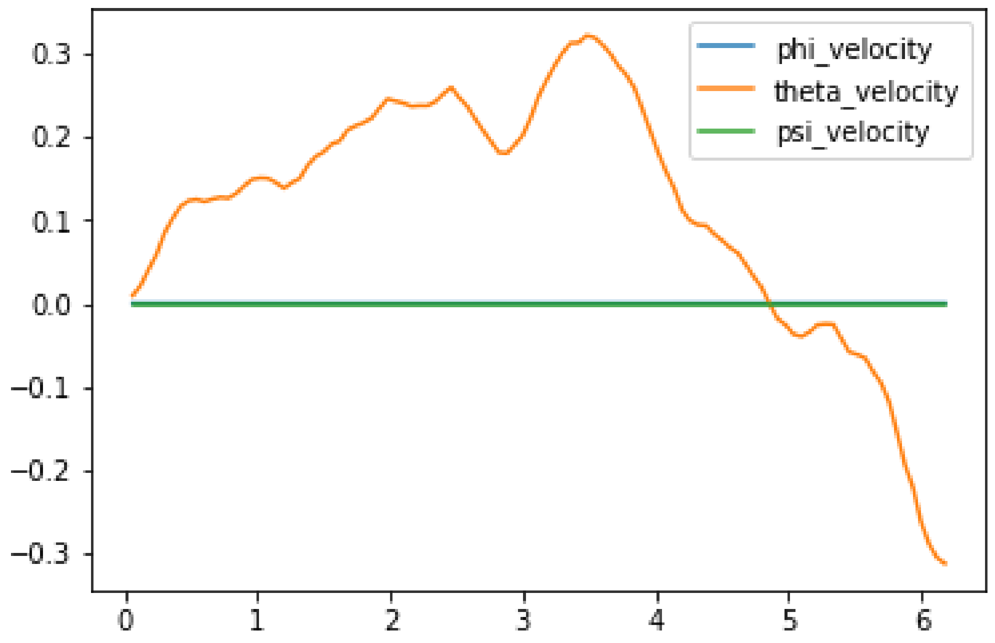
```
In [47]: # Velocity of quadcopter during simulation
         plt.plot(results_ddpg['time'], results_ddpg['x_velocity'], label='x_hat')
         plt.plot(results_ddpg['time'], results_ddpg['y_velocity'], label='y_hat')
         plt.plot(results_ddpg['time'], results_ddpg['z_velocity'], label='z_hat')
         plt.legend()
         _ = plt.ylim()
```



```
In [48]: # Euler angles
         plt.plot(results_ddpg['time'], results_ddpg['phi'], label='phi')
         plt.plot(results_ddpg['time'], results_ddpg['theta'], label='theta')
         plt.plot(results_ddpg['time'], results_ddpg['psi'], label='psi')
         plt.legend()
         _ = plt.ylim()
```



```
In [49]: # plotting the velocities (in radians per second) corresponding to each of the Euler an
         plt.plot(results_ddpg['time'], results_ddpg['phi_velocity'], label='phi_velocity')
         plt.plot(results_ddpg['time'], results_ddpg['theta_velocity'], label='theta_velocity')
         plt.plot(results_ddpg['time'], results_ddpg['psi_velocity'], label='psi_velocity')
         plt.legend()
         _ = plt.ylim()
```



1.7 Reflections

Question 1: Describe the task that you specified in `task.py`. How did you design the reward function?

Answer: Take off from point 0 in x, y and z and reach 0, 0, 10. I tried to add more complexity to improve velocity so as to reach quicker but could not get a good way to design the rewards along with the positioning problem.

Using tanh to assign reward to the copter based on the position.

Question 2: Discuss your agent briefly, using the following questions as a guide:

- What learning algorithm(s) did you try? What worked best for you?
- What was your final choice of hyperparameters (such as α, γ, ϵ, etc.)?
- What neural network architecture did you use (if any)? Specify layers, sizes, activation functions, etc.

Answer:

We have in this solution continuous action and state space problem, so using an Actor Critic agent. The agent actor uses three hidden layers sized 32, 64 and 32 with relu activations. The final layer uses a sigmoid activation to scale the output between 0 and 1. The critic uses two branches initially to process the state and actions. The state branch uses two layers sized 32 and 64 with relu activations. The action branch also uses two layers sized 32 and 64 with relu activations. The two branches are merged together by adding and followed relu activation function.

Essentially, used the recommended and default actor-critique algorithm as suggested. Used the default hyperparameters, with plotting functions similar to the sample solution so as to compare modified agent's learning with initial given.

Question 3: Using the episode rewards plot, discuss how the agent learned over time.

- Was it an easy task to learn or hard?
- Was there a gradual learning curve, or an aha moment?
- How good was the final performance of the agent? (e.g. mean rewards over the last 10 episodes)

Answer: - Definitely not an easy task to learn. - Agent performs great initially and then diminishes and then again better and causing the spike and then again reduces. However, it never is able to reaches the maximium reward that it initially reaches.

Question 4: Briefly summarize your experience working on this project. You can use the following prompts for ideas.

- What was the hardest part of the project? (e.g. getting started, plotting, specifying the task, etc.)
- Did you find anything interesting in how the quadcopter or your agent behaved?

Answer:

Definitely one of the hardest projects to complete. I really struggled with beginning to define the task and probably got really ambitious. My initial intent was to get the quadcopter to reach target and be rewarded for quickness as well. Unfortunately, my reward design for both cases did not work well together, making me revert to the simpler task of just reaching the target position. An interesting point was that for specified target, regardless of number of episodes, maximum reward was almost always reached at the beginning episodes and followed the same pattern of best score in early episodes, lesser values, hovering around a local minima, improving at the halfway mark, and then again dipping in last sessions.

References: - https://keras.io/getting-started/faq/ - https://keras.io/backend/ - https://keras.io/layers/writing-your-own-keras-layers/ - https://www.tensorflow.org/api_docs/python/tf/k - https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam - https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287

```
In [ ]:
```