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.

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.

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 [1]:

```
%load_ext autoreload
%autoreload 2

#Imports
import random
import pandas as pd
import csv
import numpy as np
import sys
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('fivethirtyeight')
import gym

# Import supplementary visualization code visuals.py
import visuals as vs

# Import tasks
from sample_task import Sample_Task
from task import Custom_Task

# Import agents
from agents.policy_search import PolicySearch_Agent
from agents.ddpg_agent_mountain_car_continuous import DDPG_Agent_Mountain_Car_Co
ntinuous
from agents.ddpg_agent_sample_task import DDPG_Agent_Sample_Task
from agents.ddpg_agent_improved_sample_task import DDPG_Agent_Improved_Sample_Ta
sk
from agents.agent import DDPG_Agent_Custom_Task
```

```
Using TensorFlow backend.
```

In [2]:

```
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 `basic_agent_sample_task_data.txt` and stored in the dictionary `results`.

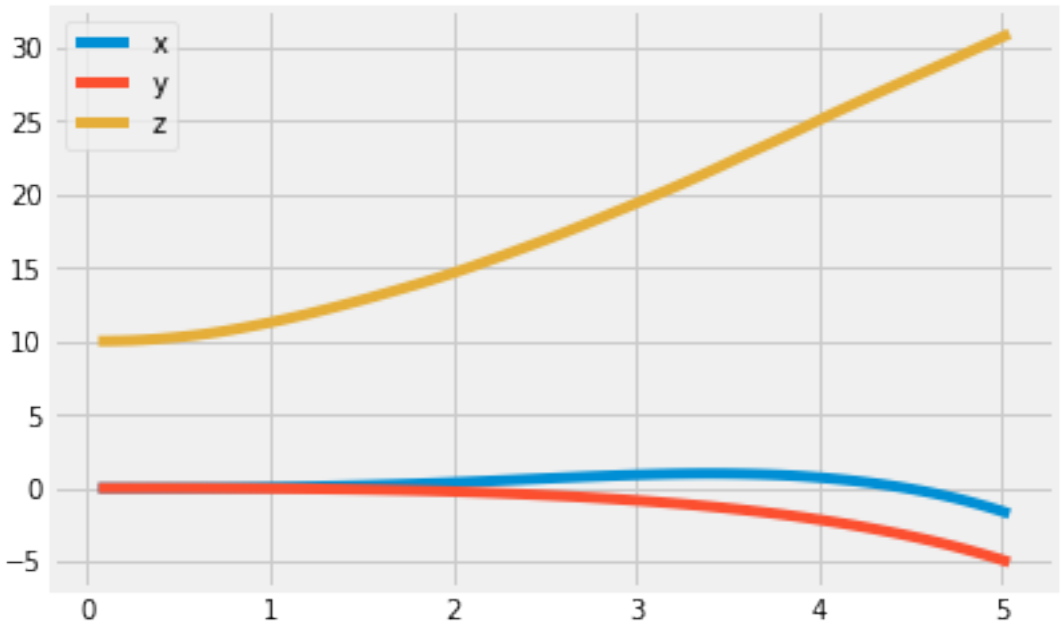
In [3]:

```
# 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 = 'basic_agent_sample_task_data.txt'  # file name for saved results

# Setup
task = Sample_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) + li
st(task.sim.angular_v) + list(rotor_speeds)
        for ii in range(len(labels)):
            results[labels[ii]].append(to_write[ii])
        writer.writerow(to_write)
        if done:
            break
```

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

In [4]:

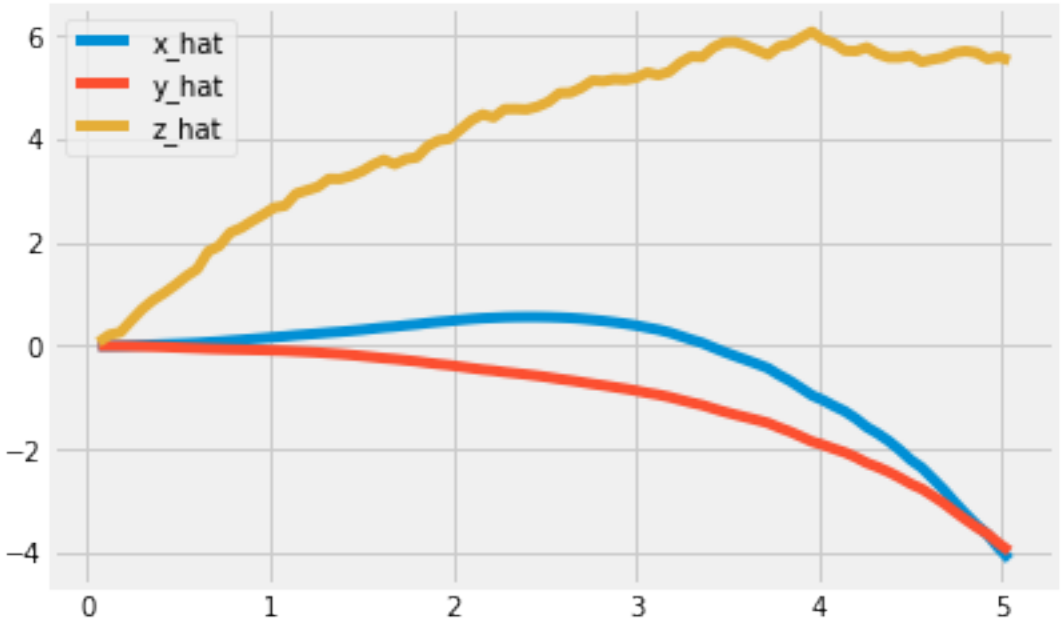
```
plt.plot(results['time'], results['x'], label='x')
plt.plot(results['time'], results['y'], label='y')
plt.plot(results['time'], results['z'], label='z')
plt.legend()
_ = plt.ylim()
```



The next code cell visualizes the velocity of the quadcopter.

In [5]:

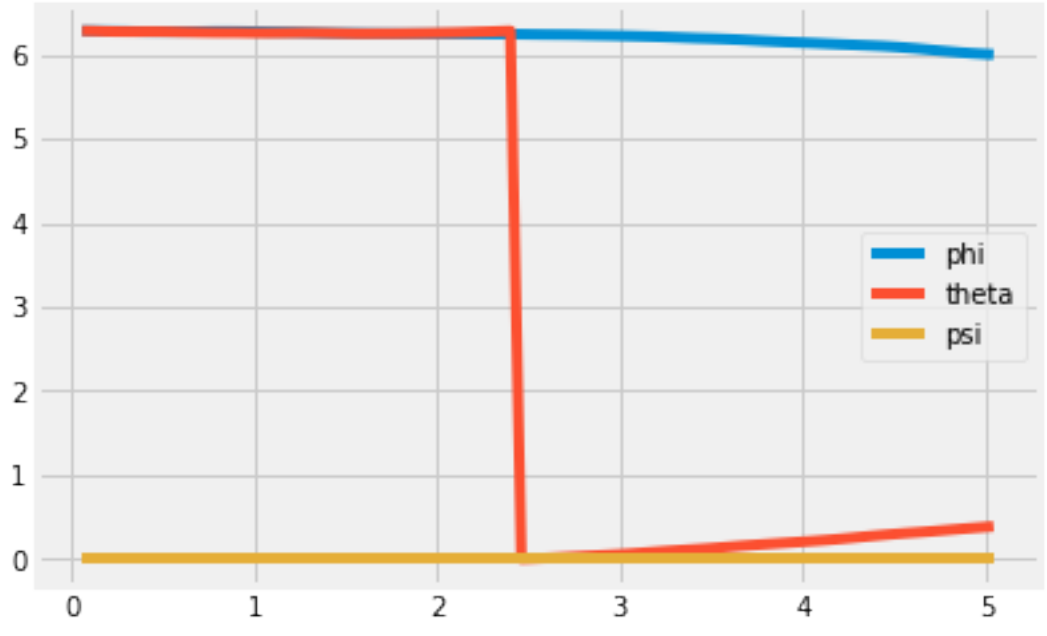
```
plt.plot(results['time'], results['x_velocity'], label='x_hat')
plt.plot(results['time'], results['y_velocity'], label='y_hat')
plt.plot(results['time'], results['z_velocity'], label='z_hat')
plt.legend()
_ = plt.ylim()
```



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

In [6]:

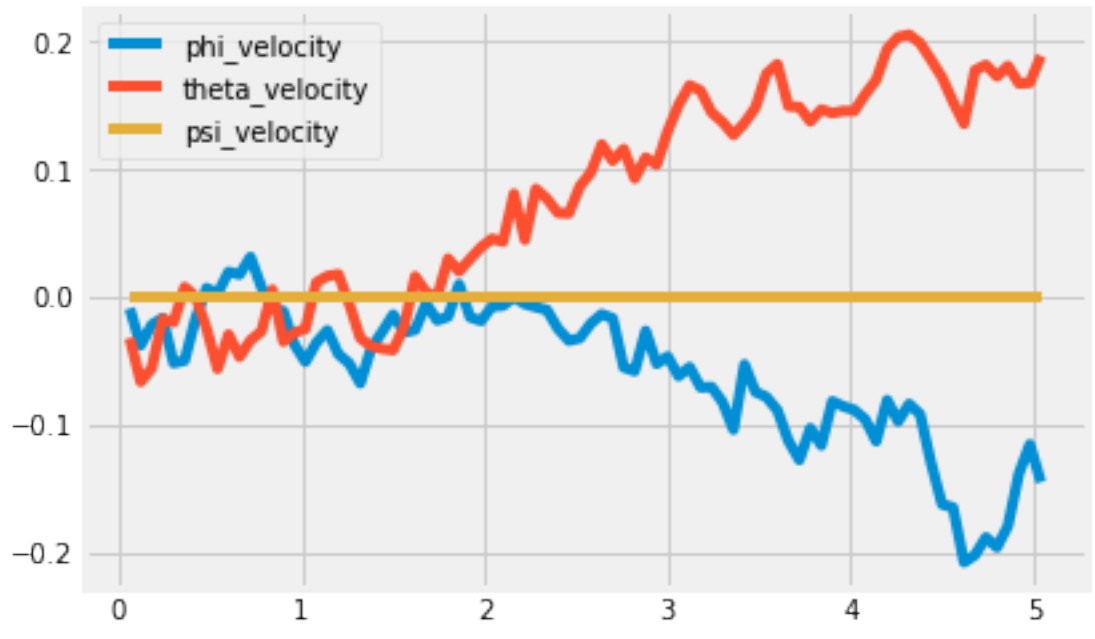
```
plt.plot(results['time'], results['phi'], label='phi')
plt.plot(results['time'], results['theta'], label='theta')
plt.plot(results['time'], results['psi'], label='psi')
plt.legend()
_ = plt.ylim()
```



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

In [7]:

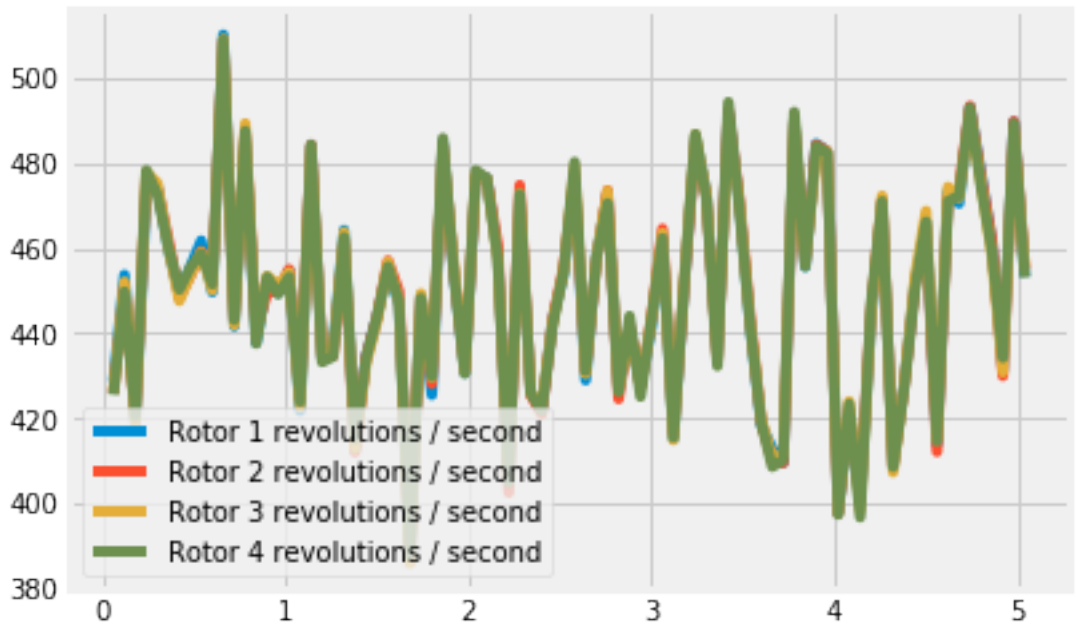
```
plt.plot(results['time'], results['phi_velocity'], label='phi_velocity')
plt.plot(results['time'], results['theta_velocity'], label='theta_velocity')
plt.plot(results['time'], results['psi_velocity'], label='psi_velocity')
plt.legend()
_ = plt.ylim()
```



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

In [8]:

```
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 [9]:

```
# 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)
```

```
[ -1.75160048  -5.05596021  30.93817143   6.00001416   0.39099011
0.        ]
[-4.1267544  -3.97126265  5.52003926]
[-0.144821  0.188892  0.      ]
```

In the sample task in `sample_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.

The Sample Task

A sample task has been provided for you in `sample_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 (https://arxiv.org/abs/1509.02971).
- 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.

The Sample Agent `PolicySearch_Agent` performing the Sample Hovering Task

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 [15]:

```
num_episodes = 1000
target_pos = np.array([0., 0., 10.])
task = Sample_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_sc
ale = {}".format(
                i_episode, agent.score, agent.best_score, agent.noise_scale), en
d="")  # [debug]
            sys.stdout.flush()
            break
```

```
Episode = 1000, score =  -1.056 (best =  -0.104), noise_scale = 3.26
25
```

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

My Work Begins Here:

Plot the rewards and flight behavior when running sample `PolicySearch_Agent` on the Sample Hovering Task:

Before creating my own agent, I wanted to plot the actual rewards earned when the `PolicySearch_Agent` (included inside `agents/policy_search.py`) performed the sample hovering task. This will give me a baseline for comparison when I create my own custom agent. I intend to eventually run my own custom agent on the `Sample_Task` in the file `sample_task.py`, and observe how its earned rewards and flight behavior compared to those of the `PolicySearch_Agent`.

To review: in the `Sample_Task` class in the file `sample_task.py`, the agent needs to maintain the quadcopter at a height of 10 meters above the ground, at x-y coordinates of (0,0). The target x-y-z coordinates are (0,0,10). The copter's initial x-y-z position is also (0,0,10). This is the default starting position as specified in physics_sim.py, and the design of the `Sample_Task` class does not override this default.

First, let's re-run the `PolicySearch_Agent` on the `Sample_Task` and this time keep track of rewards earned, as well as save the `PolicySearch_Agent`'s flight behavior to a .csv file.

In [27]:

```
# Setup
num_episodes = 1000
target_pos = np.array([0., 0., 10.])          # copter's target position in Samp
le_Task task
task = Sample_Task(target_pos=target_pos)
agent = PolicySearch_Agent(task)
rewards_list = []                             # store the total rewards earned f
or each episode
best_reward = -np.inf                         # keep track of the best reward ac
ross episodes
best_episode = 0                              # episode in which best reward is
earned
episode_steps = 0                             # keep track of number of steps in
each episode

# In order to save the simulation's results to a CSV file.
file_output = 'policy_search_agent_sample_task_data.txt'       # file name for s
aved results
labels = ['episode', 'time', 'reward', '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']

# Run the simulation, and save the results.
with open(file_output, 'w') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(labels)
    # Begin the simulation by starting a new episode.
    state = agent.reset_episode()
    # Run the simulation for each episode.
    for i_episode in range(1, num_episodes+1):
        total_reward = 0
        while True:
            action = agent.act(state)
```

```
            next_state, reward, done = task.step(action)
            total_reward += reward
            agent.step(reward, done)

            # Save quadcopter's behavior during each timestep of each episode
            # of the simulation to the CSV file.
            to_write = [i_episode] + [task.sim.time] + [reward] + list(task.sim.
pose) + list(task.sim.v) + list(task.sim.angular_v) + list(action)
            writer.writerow(to_write)
            # Increase episode step count by one.
            episode_steps += 1

            if done:
                rewards_list.append((i_episode, total_reward))
                if total_reward > best_reward:
                    best_reward = total_reward
                    best_episode = i_episode
                print("\rEpisode = {:4d} (duration of {} steps); Reward = {:7.3f
} (best reward = {:7.3f}, in episode {})   ".format(i_episode, episode_steps, to
tal_reward, best_reward, best_episode), end="")  # [debug]
                sys.stdout.flush()
                state = agent.reset_episode() # start a new episode
                episode_steps = 0 # Reset for the next episode
                break
            else:
                state = next_state
```

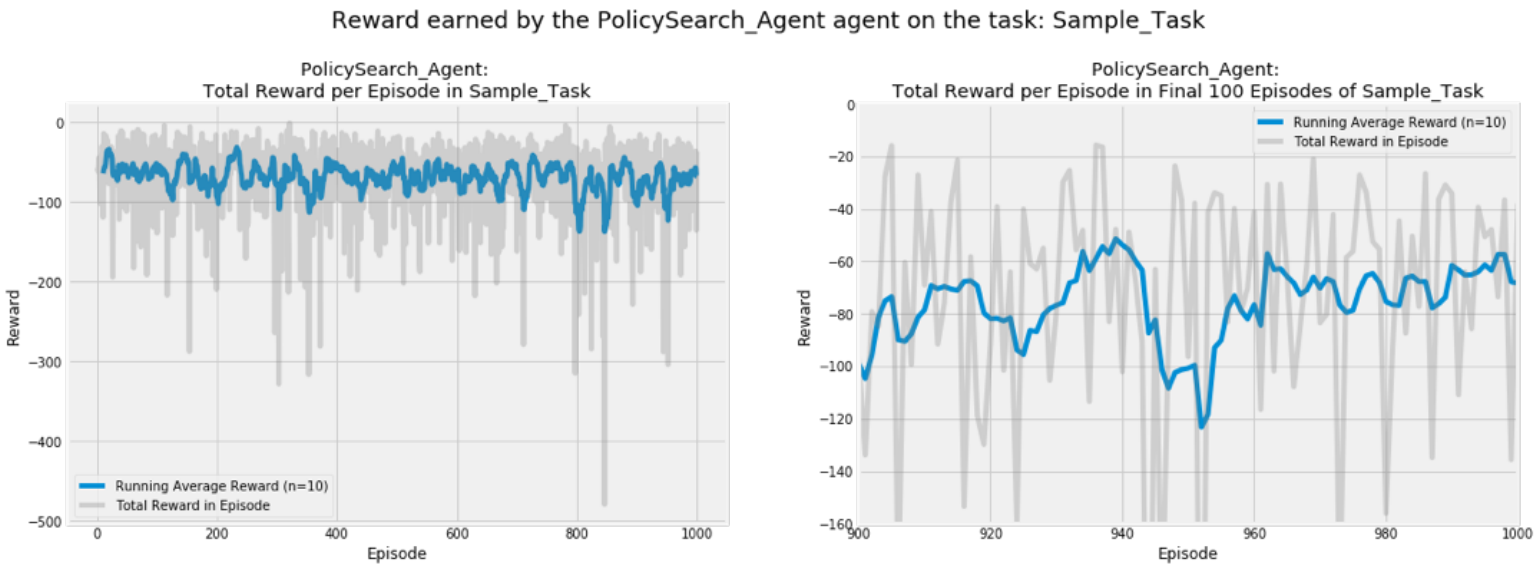
```
Episode = 1000 (duration of 23 steps); Reward = -37.953 (best reward
=  -1.414, in episode 321)
```

First let's plot the reward. Below on the left I plot the total rewards the `PolicySearch_Agent` earned in each episode of the `Sample_Task`, along with the running average reward of the previous ten episodes.

On the right, I zoom in to observe the agent's reward during the last 100 episodes of the simulation. In this period, a decent agent would conceivably have adequately learned the sample task.

In [3]:

```
# Load simulation results from the .csv file
results = pd.read_csv('policy_search_agent_sample_task_data.txt')

# Plot the reward
vs.plot_rewards(results=results, zoomed_x_range=(900,1000), zoomed_y_range=(-160
,0), agent_name= "PolicySearch_Agent", task_name="Sample_Task", n=10);
```



As we can see in the plot above, during 1,000 episodes of the `Sample_Task` the `PolicySearch_Agent` earns a reward that varies within a stable range of roughly between -160 and -20. After I design my custom agent, I will run it on the `Sample_Task` and hopefully it's performance will at least be superior to what I see here from the `PolicySearch_Agent`.

Before moving on, let's first observe how the quadcopter physically behaved during these 1,000 episodes. The graphs below will indicate how well the `PolicySearch_Agent` was able to learn the goal of the sample task, which was to keep the copter continuously hovering at a height of 10 meters off the ground, above the x-y coordinates of (0,0).

To summarize: the copter begins each episode at (0,0,10) and it is supposed to remain in this position indefinitely. Let's see how successful the `PolicySearch_Agent` was in inducing this behavior:

- The first graph in the upper left hand corner is quadcopter's average position in, x, y, and z values, across all episodes at the first n timesteps, where n is the average episode duration in timesteps. This gives the best indication of the copter's behavior, on the whole, over all episodes of the simulation.

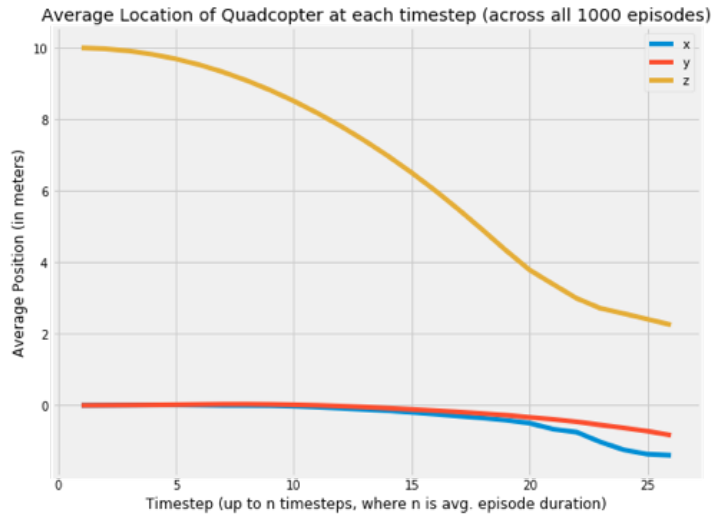
- The second graph in the upper right hand corner shows how the copter's location (in terms of its x, y, and z values) varied at each timestep during the episode when the copter earned its highest total reward. This shows the behavior that earned the highest reward for the copter in an episode.

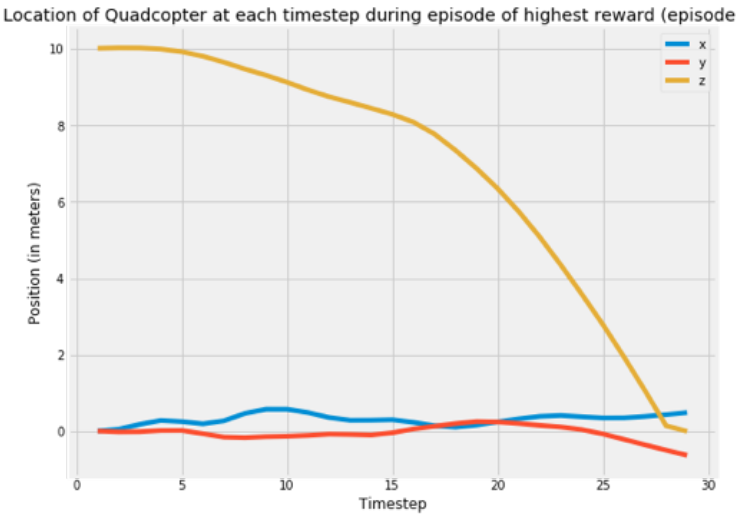
- The final four graphs plot x, y, z values of the copter at each timestep, for 4 episodes chosen at random from the final 100 episodes of the simulation. This is the period when a decent agent would conceivably have adequately learned the task.

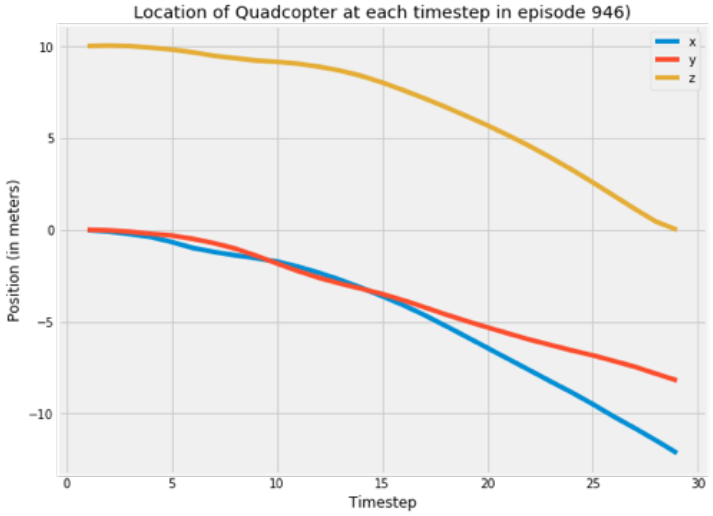
In [29]:

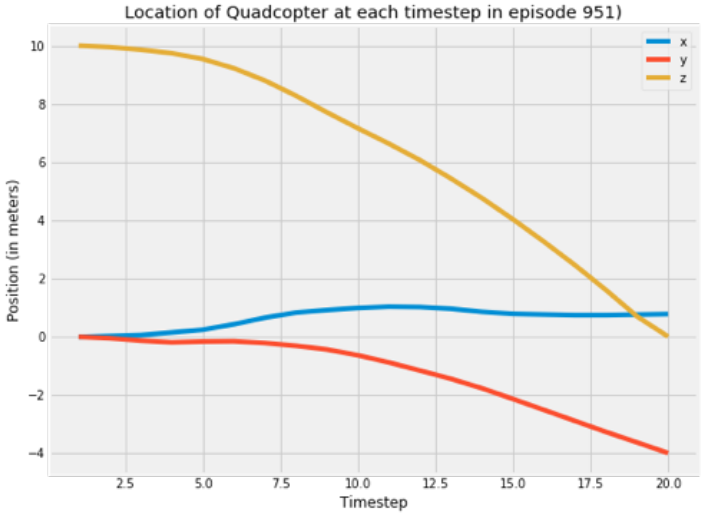
```
# Plot the flight behavior
vs.plot_behavior(results=results, agent_name="PolicySearch_Agent", task_name="Sa
mple_Task")
```

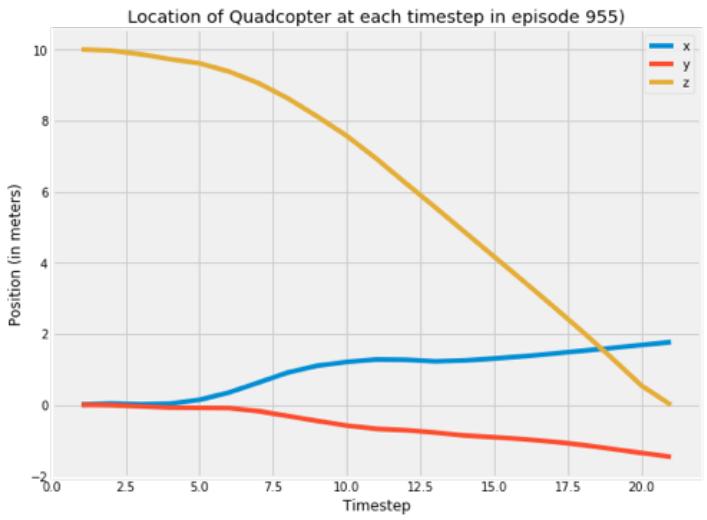
Flight behavior of the PolicySearch_Agent agent on the Sample_Task task.

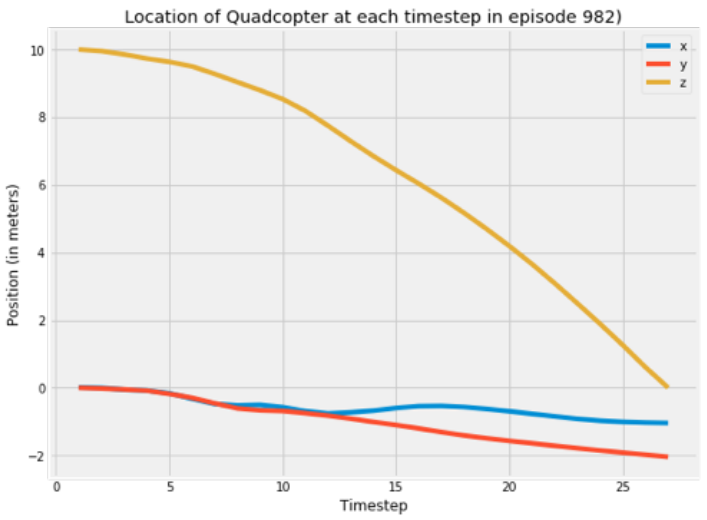












As can be seen above, the agent begins each episode at location (0,0,10), and as timesteps progress, the agent drifts downward to a height of 0, while also experiencing some drift away from the center (0,0) of the x-y plane. A good agent would keep the copter at a height of 10, and centered over the center (0,0) of the x-y plane.

In the upper right hand graph above, we can see that in episode 321 this ideal behavior occured for a slightly longer number of timesteps than was typical, and it should come as no surprise that the behavior depicted in this graph was the behavior that earned the agent its highest reward in an individual episode.

Overall, as indicated by the graphs above, the `PolicySearch_Agent` clearly isn't a very successful agent. And I hope that my custom agent will have better performance on this `Sample_Task`. However, I also beginning to have some doubts about the rewards structure in the `Sample_Task`. More specifically, I am wondering whether the rewards structure of the task is sufficient to induce any agent, no matter how good, to learn maintain a position at (0,0,10) across an entire episode:

```
reward = 1.-.3*(abs(self.sim.pose[:3] - self.target_pos)).sum()
```

Therefore, as I create and refine my own custom agent, I will not use the `Sample_Task` as a source of feedback. Instead, I will benchmark my custom agent's performance on OpenAi Gym's `MountainCarContinuous-v0` task (from https://github.com/openai/gym/wiki/MountainCarContinuous-v0 (https://github.com/openai/gym/wiki/MountainCarContinuous-v0)).

Although `MountainCarContinuous-v0`'s action space and environment physics are different from those of our quadcopter, I feel far more confident using the well-known and widely used `MountainCarContinuous-v0` continuous task, with its validated rewards structure, as a yardstick for measuring my custom agent's proficiency.

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.

My custom agent: a DDPG Agent

I created a DDPG (Deep Deterministic Policy Gradient) agent inspired by the algorithm outlined by Lillicrap, Timothy P., et al., in their 2015 paper Continuous Control with Deep Reinforcement Learning https://arxiv.org/pdf/1509.02971.pdf (https://arxiv.org/pdf/1509.02971.pdf).

My agent's class itself is titled `DDPG_Agent_Mountain_Car_Continuous`, and this is stored in the file `agents/ddpg_agent_mountain_car_continuous.py`.

I began by using as a template the sample code included in the Quadcopter Project starter code for the DDPG_Agent, ReplayBuffer, Actor, Critic, OUNoise (Ornstein-Uhlenbeck noise process) classes that are part of my agent's model. These classes are stored in the file `model_ddpg_agent_mountain_car_continuous.py`.

I first modified the init method in the agent class so that it could parse the smaller state size and action space of the mountain car task (as opposed to the quadcopter environment, which has higher dimensional state and action spaces). I then began the process of tweaking my agent's deep net structure and other model parameters through a process of trial and error.

The majority of my modifications were inspired by the deep net structure and parameters used by Lillicrap, Timothy P., et al. in their paper https://arxiv.org/pdf/1509.02971.pdf (https://arxiv.org/pdf/1509.02971.pdf). I made some final tweaks to my agent that were inspired by the approach of user 'lirnli,' whose used DDPG in their solution (https://github.com/lirnli/OpenAI-gym-solutions/blob/master/Continuous_Deep_Deterministic_Policy_Gradient_Net/DDPG%20Class%20ver2.ipynb (https://github.com/lirnli/OpenAI-gym-solutions/blob/master/Continuous_Deep_Deterministic_Policy_Gradient_Net/DDPG%20Class%20ver2.ipynb)) to the `MountainCarContinuous-v0` task. Their solution is currently in 3rd place on OpenAi Gym's leaderboard (https://github.com/openai/gym/wiki/Leaderboard#mountaincarcontinuous-v0 (https://github.com/openai/gym/wiki/Leaderboard#mountaincarcontinuous-v0)).

Here is a list of my modifications:

1. Use τ of 0.001, which is recommended by Lillicrap, Timothy P., et al. (The Udacity base code had a τ of 0.01, and this didn't work at all for me.)

2. Use memory batch size of 256 and a buffer size of 10,000. This is what 'lirnli' used, and I found it improved my agent's average performance by roughly 20%. (different from both Udacity base code and recommendations of Lillicrap, Timothy P., et al. where both recommended a batch size of 64 and buffer of 100,000)

3. Use an initial exploration policy to warm up for a duration that's 3 times longer than typical (3 times the batch size). (I adopted this approach from user 'lirnli' as well.)

4. Use learning rate of 0.0001 in actor's neural net, in line with the recommendations of Lillicrap, Timothy P., et al. (different from 0.001 that Udacity recommended in their base code)

5. OU noise perturbs only a fraction of the value of the predicted actions, as opposed to the entire value. This fraction decreases as number of episodes increases. And after episode 100, the "exploration phase" is terminated and no OU noise is added to the actions.

For the purposes of solving this task, it was more convenient to write a simple method to implement OU noise, instead of using the OUNoise class originally included in the Udacity starter code. (I adopted this approach from user 'lirnli' as well.)

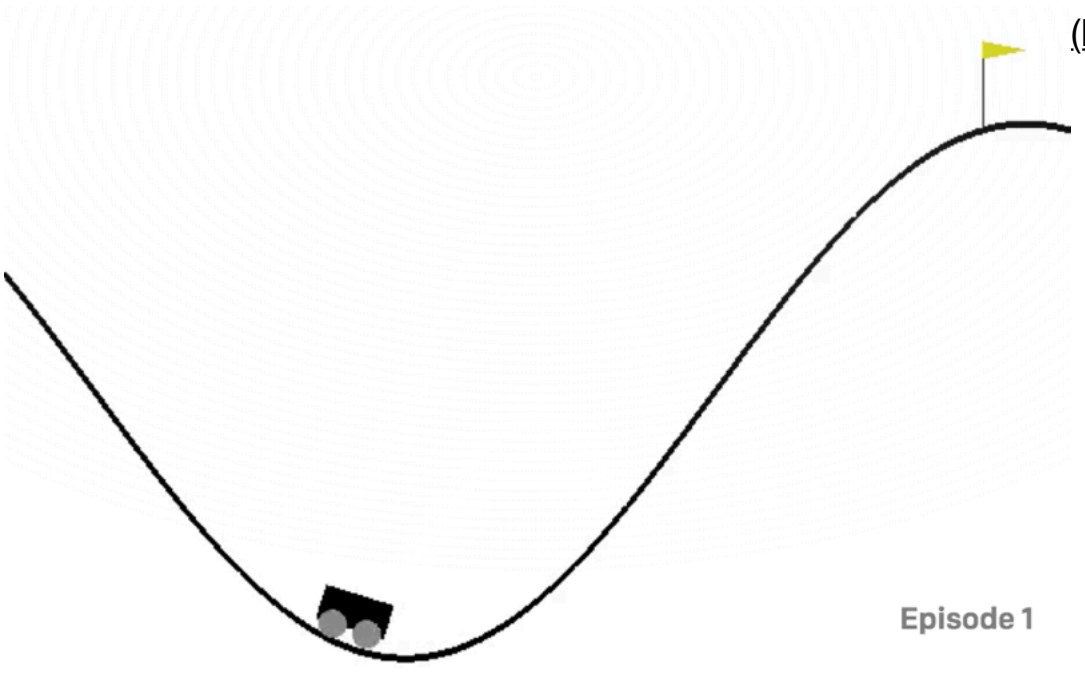
6. For actor and critic nets: 1st hidden layer has 400 units, and 2nd hidden layer has 300 units. This greatly improved performance/consistency. Also initialized all layers in actor and critic with normal distributions of $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$, where f is the fan-in of the layer. This also came from Lillicrap, Timothy P., et al., except I used normal distributions instead of uniform, which was what they used.

7. **This final modification was the most meaningful**: use elu (Exponential Linear Units) activation function in neural net hidden layers (instead of relu activations as recommended by Lillicrap, Timothy P., et al. and the Udacity starter code). This was also inspired by lirnli's approach. Indeed, it wasn't until I changed my activation function from relu to elu that my agent was able to demonstrate that it learned to solve the `MountainCarContinuous-v0` task.

   Here is more on elu activation functions, as published in this paper: https://arxiv.org/abs/1511.07289 (https://arxiv.org/abs/1511.07289) by Clevert, Unterthiner, and Hochreiter.

Interesting Note: Of the customizations that I made to my agent in order to maximize its reward on the `MountainCarContinuous-v0` task, I would expect those inspired by Lillicrap, Timothy P., et al. to still be beneficial to my agent when I run it on a quadcopter task. This is because Lillicrap, Timothy P., et al. designed their deep net and model to generalize to learning tasks run in a variety of physical environments. It will be interesting to see which, if any, of my modifications that were inspired by the DDPG implementation of user 'lirnli' are also still helpful during quadcopter tasks. I am somewhat more skeptical that these modifications will generalize as well. This is because lirnli's implementation ostensibly was optimized solely toward maximizing reward on the `MountainCarContinuous-v0` task, and on that task alone.

OpenAI Gym `MountainCarContinuous-v0` cart task



(MountainCarContinuous-v0)

Here is a summary of OpenAI Gym's `MountainCarContinuous-v0` task (from https://github.com/openai/gym/wiki/MountainCarContinuous-v0 (https://github.com/openai/gym/wiki/MountainCarContinuous-v0)):

"An underpowered car must climb a one-dimensional hill to reach a target. Unlike MountainCar v0, the action (engine force applied) is allowed to be a continuous value.

The target is on top of a hill on the right-hand side of the car. If the car reaches it or goes beyond, the episode terminates.

On the left-hand side, there is another hill. Climbing this hill can be used to gain potential energy and accelerate towards the target. On top of this second hill, the car cannot go further than a position equal to -1, as if there was a wall. Hitting this limit does not generate a penalty (it might in a more challenging version)."

OpenAI Gym considers this task solved when the agent earns an average reward of 90.0 over 100 consecutive trials.

In [2]:

```
#OU Noise method for this task
def OUNoise():
    theta = 0.15
    sigma = 0.2
    state = 0
    while True:
        yield state
        state += -theta*state+sigma*np.random.randn()

# Setup
env = gym.make('MountainCarContinuous-v0')      # make the environment
agent = DDPG_Agent_Mountain_Car_Continuous(env)
action_repeat = 3                               # my DDPG implementation uses ac
tion_repeat
num_episodes = 200
rewards_list = []                               # store the total rewards earned
for each episode
best_reward = -np.inf                           # keep track of the best reward
across episodes
max_explore_eps = 100                           # duration of exploration phase
using OU noise
episode_steps = 0
noise = OUNoise()

# In order to save the simulation's reward results to a CSV file.
file_output = 'ddpg_agent_mountain_car_continuous_data.txt'       # file name fo
r saved results
labels = ['episode', 'timestep', 'reward']

# Run the simulation, and save the results.
with open(file_output, 'w') as csvfile:
```

```
    writer = csv.writer(csvfile)
    writer.writerow(labels)
    # Begin the simulation by starting a new episode
    state = agent.reset_episode()
    # Run the simulation for each episode.
    for i_episode in range(1, num_episodes+1):
        total_reward = 0
        while True:
            action = agent.act(state)

            # exploration policy
            if i_episode < max_explore_eps:
                p = i_episode/max_explore_eps
                action = action*p + (1-p)*next(noise) # Only a fraction of the a
ction's value gets perturbed

            next_state, reward, done, _ = env.step(action)
            # Ensure that size of next_state as returned from the
            # 'MountainCarContinuous-v0' environment is increased in
            # size according to the action_repeat parameter's value.
            next_state = np.concatenate([next_state] * action_repeat)
            total_reward += reward
            agent.step(action, reward, next_state, done)

            # Save agent's rewards earned during each timestep of each episode
            # of the simulation to the CSV file.
            to_write = [i_episode] + [episode_steps] + [reward]
            writer.writerow(to_write)
            # Increase episode timestep count by one.
            episode_steps += 1

            if done:
                rewards_list.append((i_episode, total_reward))
                if total_reward > best_reward:
                    best_reward = total_reward
                    best_episode = i_episode
                print("\rEpisode = {:4d} (duration of {} steps); Reward = {:7.3f
} (best reward = {:7.3f}, in episode {})   ".format(
                    i_episode, episode_steps, total_reward, best_reward, best_ep
isode), end="")  # [debug]
                sys.stdout.flush()
                state = agent.reset_episode() # start a new episode
                episode_steps = 0 # Reset for the new episode
                break
            else:
                state = next_state
```

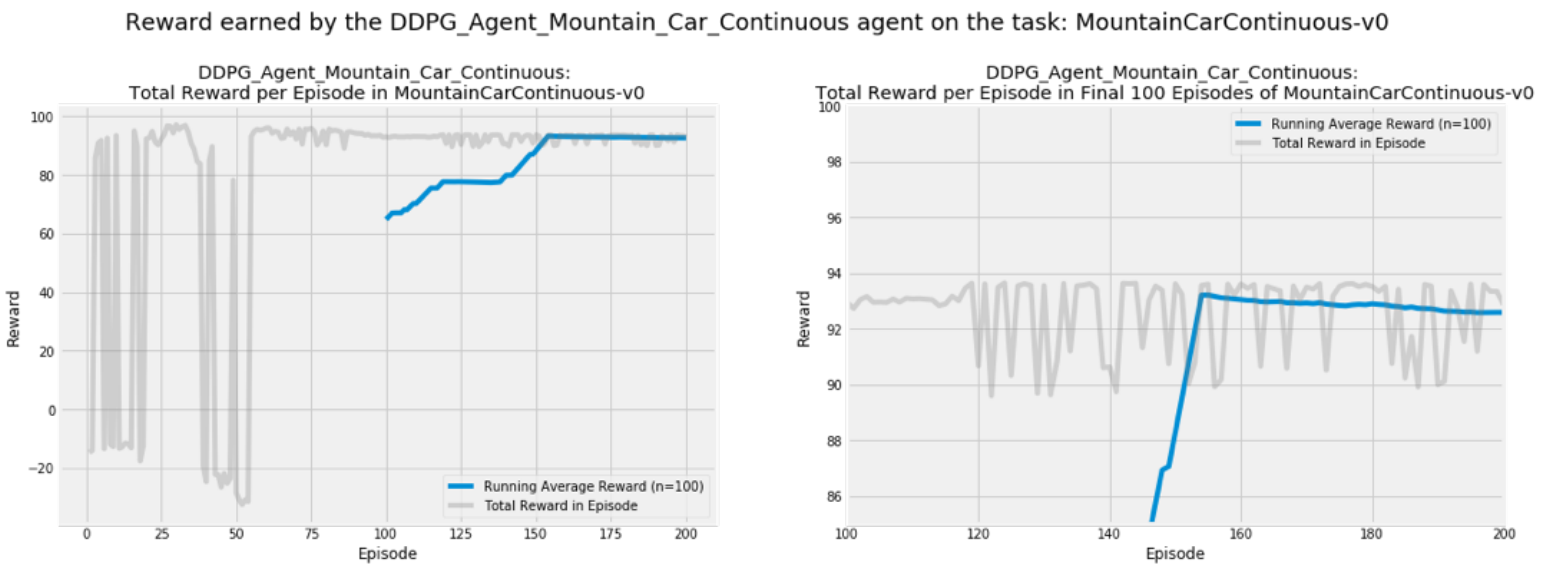
```
WARN: gym.spaces.Box autodetected dtype as <class 'numpy.float32'>.
Please provide explicit dtype.
WARN: gym.spaces.Box autodetected dtype as <class 'numpy.float32'>.
Please provide explicit dtype.
Episode =  200 (duration of 74 steps); Reward =  92.862 (best reward
=  97.069, in episode 30)
```

Plot the Rewards from running my agent `DDPG_Agent_Mountain_Car_Continuous`

on the `MountainCarContinuous-v0` cart task:

In [4]:

```
# Load simulation results from the .csv file
results = pd.read_csv('ddpg_agent_mountain_car_continuous_data.txt')

# Plot the reward
vs.plot_rewards(results=results, zoomed_x_range=(100,200), zoomed_y_range=(85,10
0), agent_name= "DDPG_Agent_Mountain_Car_Continuous", task_name="MountainCarCont
inuous-v0", n=100);
```



My `DDPG_Agent_Mountain_Car_Continuous` agent has solved OpenAI Gym's `MountainCarContinuous-v0` task. OpenAI considers this task solved when an agent earns an average reward of 90.0 over 100 consecutive trials. As can be seen above in the graph on the right, begininng just after episode 150, my agent's running average reward (over the previous 100 consecutive episodes) rocketed past 92 and remained safely in between 92 and 94 points for the final 50 episodes.

I am pleased that my agent did a masterful job of solving the `MountainCarContinuous-v0` task. I now have enough confidence in this agent's potential to learn quadcopter tasks. However, before I run this agent on my custom quadcopter task, I will first run it on the sample hovering task,`Sample_Task`, and observe how its performance compares to that of the sample `PolicySearch_Agent`.

Running my custom DDPG agent on the Sample Hovering Task

I took my custom `DDPG_Agent_Mountain_Car_Continuous` agent that I used to solve OpenAI Gym's `MountainCarContinuous-v0` task and modified it so that it would be compatible with both the state space and action space of quadcopter tasks. Everything else in this agent and its model was left unchanged. This modified agent is called `DDPG_Agent_Sample_Task` and it is located in `agents/ddpg_agent_sample_task.py`. Its model file is `model_ddpg_agent_sample_task.py`.

In [3]:

```
#OU Noise method for this task
def OUNoise():
    theta = 0.15
    sigma = 0.2
    state = 0
    while True:
        yield state
        state += -theta*state+sigma*np.random.randn()

# Setup
num_episodes = 1000
target_pos = np.array([0., 0., 10.])          # copter's target position in Samp
le_Task task
task = Sample_Task(target_pos=target_pos)
agent = DDPG_Agent_Sample_Task(task)
rewards_list = []                             # store the total rewards earned f
or each episode
best_reward = -np.inf                         # keep track of the best reward ac
ross episodes
best_episode = 0                              # episode in which best reward is
earned
episode_steps = 0                             # keep track of number of steps in
each episode
max_explore_eps = 500                         # duration of exploration phase us
ing OU noise
noise = OUNoise()

# In order to save the simulation's results to a CSV file.
file_output = 'ddpg_agent_sample_task_data.txt'       # file name for saved resu
lts
labels = ['episode', 'time', 'reward', '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']

# Run the simulation, and save the results.
with open(file_output, 'w') as csvfile:
```

```
    writer = csv.writer(csvfile)

    writer.writerow(labels)
    # Begin the simulation by starting a new episode.
    state = agent.reset_episode()
    # Run the simulation for each episode.
    for i_episode in range(1, num_episodes+1):
        total_reward = 0
        while True:
            action = agent.act(state)

            # exploration policy: explore for half of the episodes
            # (max_explore_eps = 500) in MountainCarContinuous task,
            # this agent also explored for half of the episodes
            if i_episode < max_explore_eps:
                p = i_episode/max_explore_eps
                action = action*p + (1-p)*next(noise) # Only a fraction of the a
ction's value gets perturbed

            next_state, reward, done = task.step(action)
            total_reward += reward
            agent.step(action, reward, next_state, done)

            # Save quadcopter's behavior during each timestep of each episode
            # of the simulation to the CSV file.
            to_write = [i_episode] + [task.sim.time] + [reward] + list(task.sim.
pose) + list(task.sim.v) + list(task.sim.angular_v) + list(action)
            writer.writerow(to_write)
            # Increase episode step count by one.
            episode_steps += 1

            if done:
                rewards_list.append((i_episode, total_reward))
                if total_reward > best_reward:
                    best_reward = total_reward
                    best_episode = i_episode
                print("\rEpisode = {:4d} (duration of {} steps); Reward = {:7.3f
} (best reward = {:7.3f}, in episode {})   ".format(i_episode, episode_steps, to
tal_reward, best_reward, best_episode), end="")  # [debug]
                sys.stdout.flush()
                state = agent.reset_episode() # start a new episode
                episode_steps = 0 # Reset for the next episode
                break
            else:
                state = next_state
```

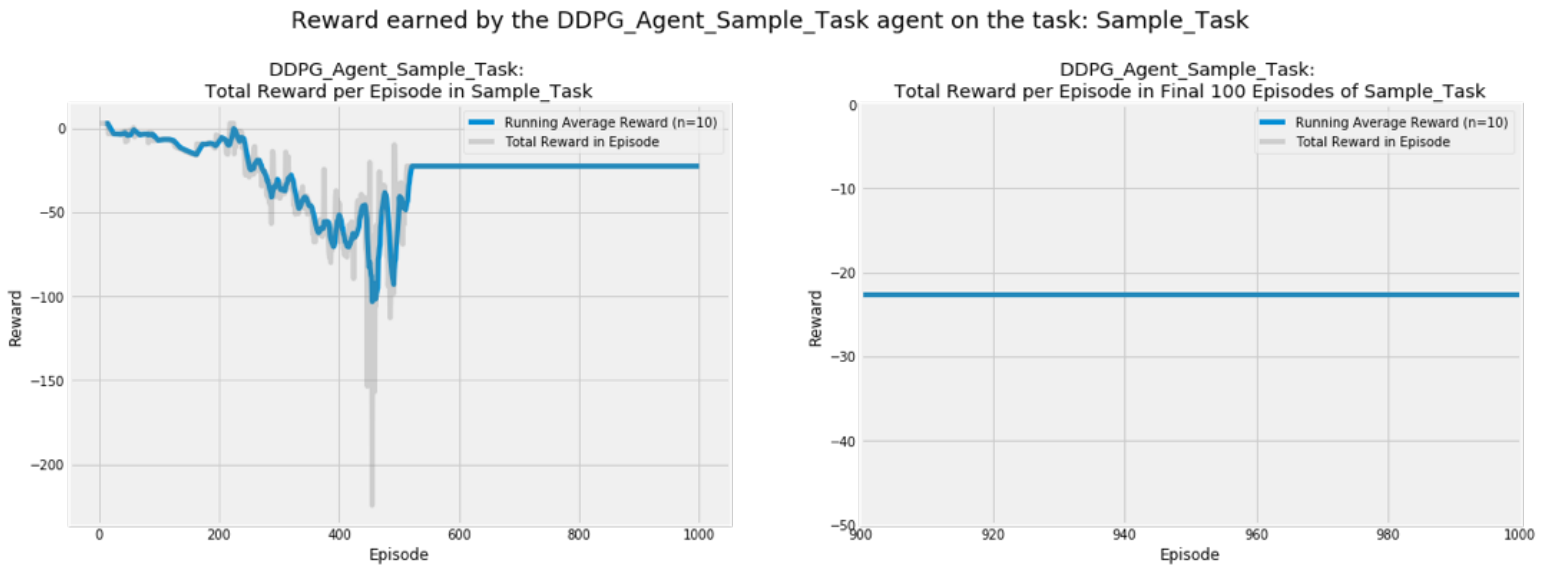
```
Episode = 1000 (duration of 26 steps); Reward = -22.696 (best reward
=   2.709, in episode 218)
```

Plot the rewards and flight behavior from running my custom agent

`DDPG_Agent_Sample_Task` on the Sample Hovering Task:

In [2]:

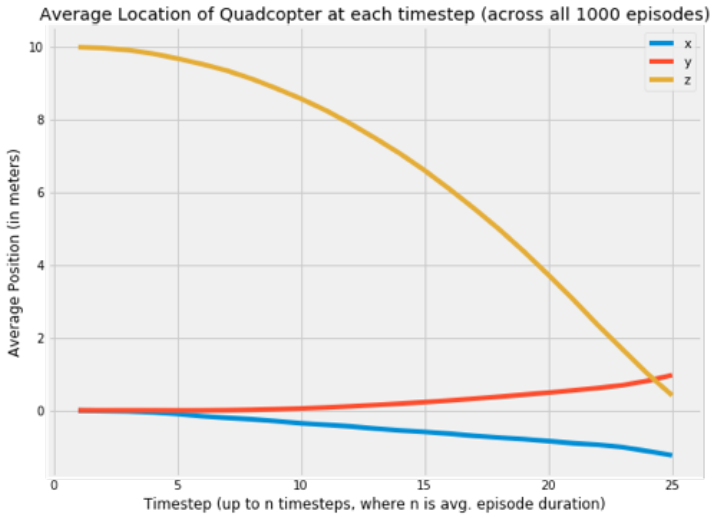
```
# Load simulation results from the .csv file
results = pd.read_csv('ddpg_agent_sample_task_data.txt')

# Plot the reward
vs.plot_rewards(results=results, zoomed_x_range=(900,1000), zoomed_y_range=(-50,
0), agent_name="DDPG_Agent_Sample_Task", task_name="Sample_Task", n=10);
```

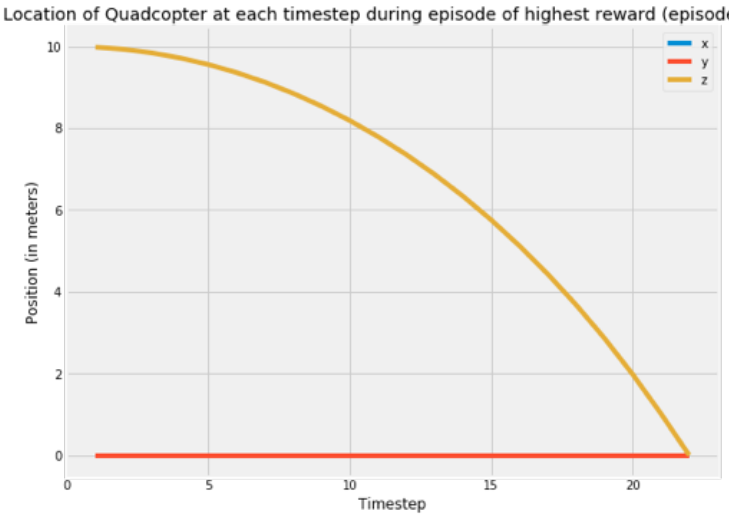


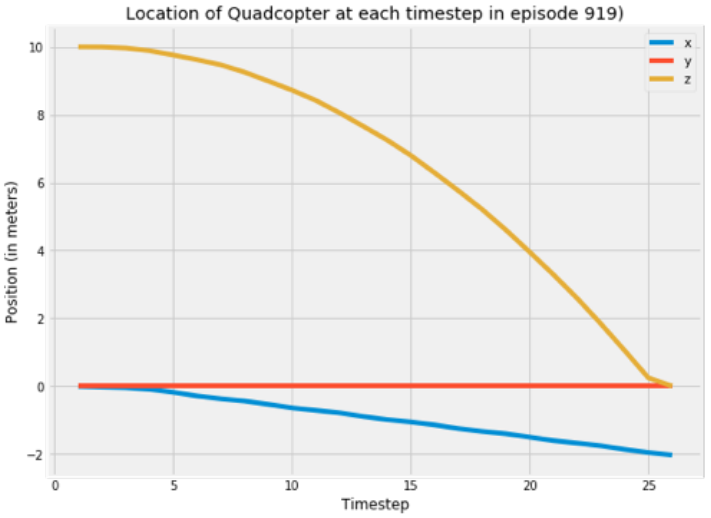
In [3]:

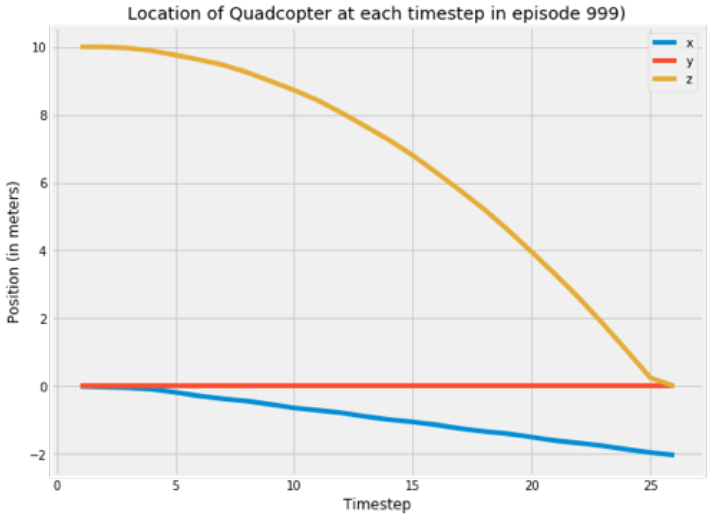
```
# Plot the quadcopter's flight behavior
vs.plot_behavior(results=results, agent_name="DDPG_Agent_Sample_Task", task_name
="Sample_Task");
```

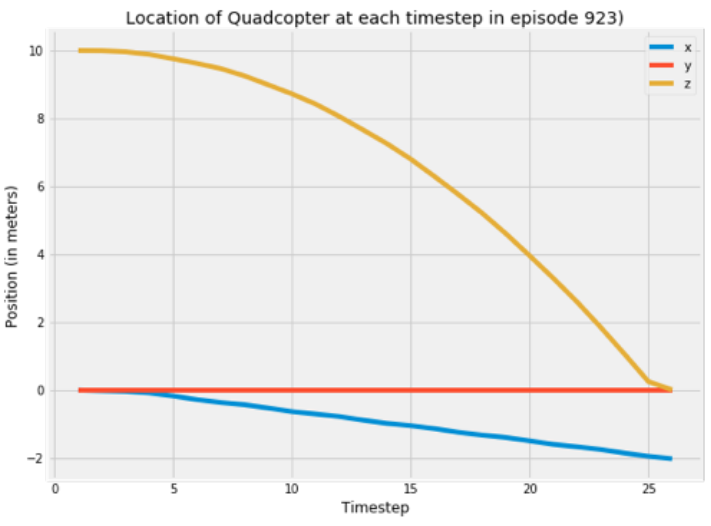
Flight behavior of the DDPG_Agent_Sample_Task agent on the Sample_Task task.

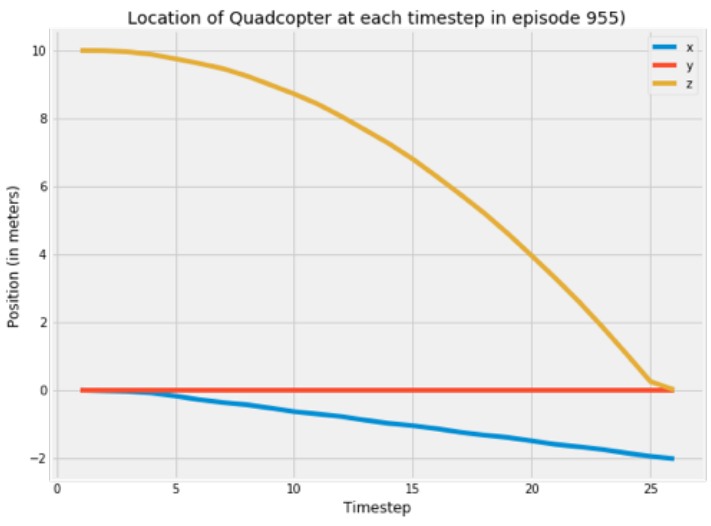












Looking at rewards graphs above, I can clearly see that my custom DDPG agent performs much better on the `Sample_Task` than the sample `PolicySearch_Agent` did -- my agent earns an average reward just under -20 in its final 100 episodes, compared to the `PolicySearch_Agent`, which earned average rewards around -80 in its final episodes.

Also, looking at the above graphs of the copter's flight behavior, it appears that my DDPG agent did a better job of keeping the quadcopter at a height of 10 meters and more closely centered above (0,0) in the x-y plane for a longer period of time than the `PolicySearch_Agent` was able to do. On its episode of best performance, my custom DDPG agent kept the copter centered exactly above the center of the x-y plane for the complete duration of the episode.

Nonetheless, even using my superior DDPG agent, in most episodes the copter eventually drops down to a height of 0. My agent seems to do a better job of delaying this, but it does not prevent this from happening. Given that this same agent did such a good job of learning the `MountainCarContinuous-v0` task, I am more suspicious than ever that the rewards structure of the `Sample_Task` could be substantially improved in order to adequately incentivize the agent to learn to keep the copter 10 meters above the ground.

Improve my custom DDPG_Agent to perform better on the Sample Hovering Task

However, before proceeding to the custom task, I decided to spend a little extra time to see if I could apply some further tweaks to my custom DDPG agent (`DDPG_Agent_Sample_Task`), in order to see if I could improve its performance on the `Sample_Task`.

This agent is called `DDPG_Agent_Improved_Sample_Task` and it is located in `agents/ddpg_agent_improved_sample_task.py`. Its model file is `model_ddpg_agent_improved_sample_task.py`.

Over the course of several simulations on the `Sample_Task`, I found that my custom DDPG agent that had performed so reliable well on OpenAI Gym's `MountainCarContinuous-v0` task did not consistently and reliably earn decent rewards on the `Sample_Task`. In some simulations, the agent would converge to a respectfully decent reward. Indeed, this was the case in the simulation that I ran and graphed just above. However, more often than not, the agent would fail learn how to earn a somewhat decent reward from the task, and the reward amounts earned during the final 100 episodes could be just as volatile (+/- 1,000 or more from one episode to the next), as they were during the first 100 episodes of the simulation.

What I found worked best was undoing the tinkering and tweaks that had made my DDPG agent so successful at the `MountainCarContinuous-v0` task, and implementing an agent that was as identical as possible, both in terms of parameters and deep neural net structure, to the agent which Lillicrap, Timothy P., et al. (https://arxiv.org/pdf/1509.02971.pdf (https://arxiv.org/pdf/1509.02971.pdf)) had used when testing their DDPG actor/critic model. I will elaborate more on this agent's structure and parameter values below in my response to **Question 2** in the Reflections section.

In [2]:

```
# Setup
num_episodes = 1000
target_pos = np.array([0., 0., 10.])          # copter's target position in Samp
le_Task task
task = Sample_Task(target_pos=target_pos)
agent = DDPG_Agent_Improved_Sample_Task(task)
rewards_list = []                             # store the total rewards earned f
or each episode
best_reward = -np.inf                         # keep track of the best reward ac
ross episodes
best_episode = 0                              # episode in which best reward is
earned
episode_steps = 0                             # keep track of number of steps in
each episode

# In order to save the simulation's results to a CSV file.
file_output = 'ddpg_agent_improved_sample_task_data.txt'       # file name for s
aved results
labels = ['episode', 'time', 'reward', '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']

# Run the simulation, and save the results.
```

```
with open(file_output, 'w') as csvfile:

    writer = csv.writer(csvfile)
    writer.writerow(labels)
    # Begin the simulation by starting a new episode.
    state = agent.reset_episode()
    # Run the simulation for each episode.
    for i_episode in range(1, num_episodes+1):
        total_reward = 0
        while True:
            action = agent.act(state)
            next_state, reward, done = task.step(action)
            total_reward += reward
            agent.step(action, reward, next_state, done)

            # Save quadcopter's behavior during each timestep of each episode
            # of the simulation to the CSV file.
            to_write = [i_episode] + [task.sim.time] + [reward] + list(task.sim.
pose) + list(task.sim.v) + list(task.sim.angular_v) + list(action)
            writer.writerow(to_write)
            # Increase episode step count by one.
            episode_steps += 1

            if done:
                rewards_list.append((i_episode, total_reward))
                if total_reward > best_reward:
                    best_reward = total_reward
                    best_episode = i_episode
                print("\rEpisode = {:4d} (duration of {} steps); Reward = {:7.3f
} (best reward = {:7.3f}, in episode {})   ".format(
                    i_episode, episode_steps, total_reward, best_reward, best_ep
isode), end="")  # [debug]
                sys.stdout.flush()
                state = agent.reset_episode() # start a new episode
                episode_steps = 0 # Reset for the next episode
                break
            else:
                state = next_state
```

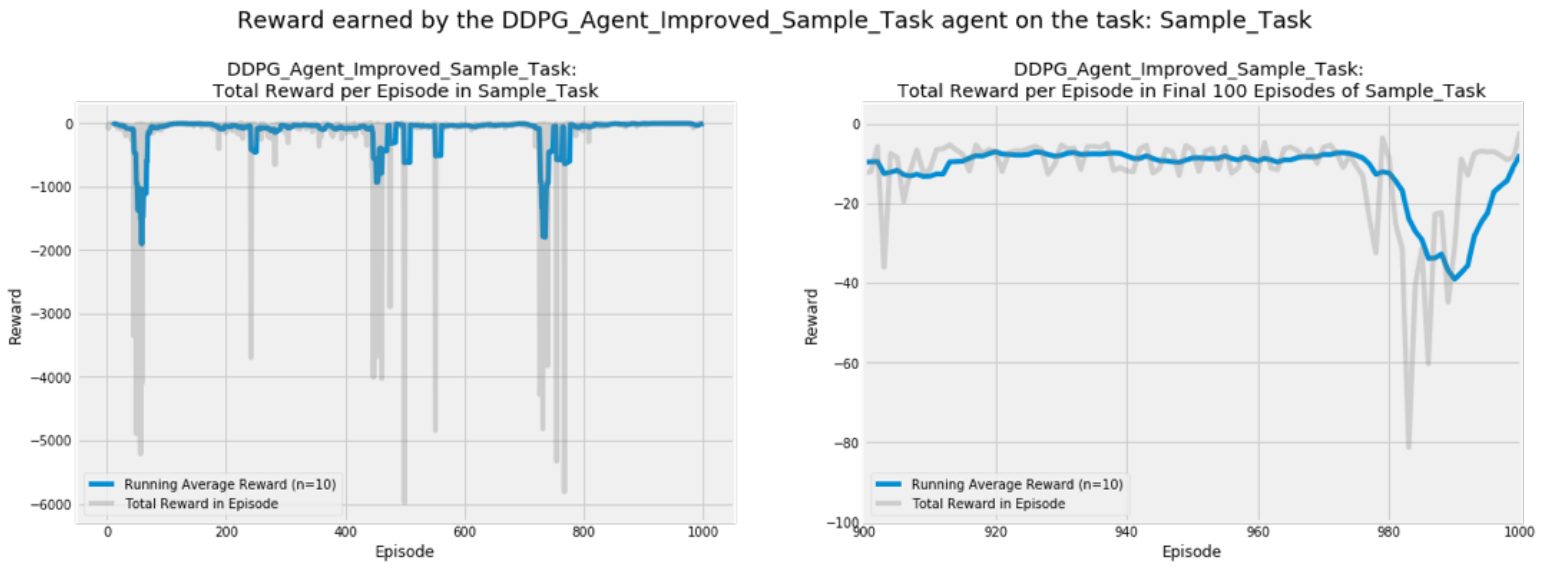
```
Episode = 1000 (duration of 22 steps); Reward =  -2.006 (best reward
=   3.042, in episode 225)
```

Plot the rewards and flight behavior from running my improved custom agent

`DDPG_Agent_Improved_Sample_Task` on the Sample Hovering Task:

In [3]:

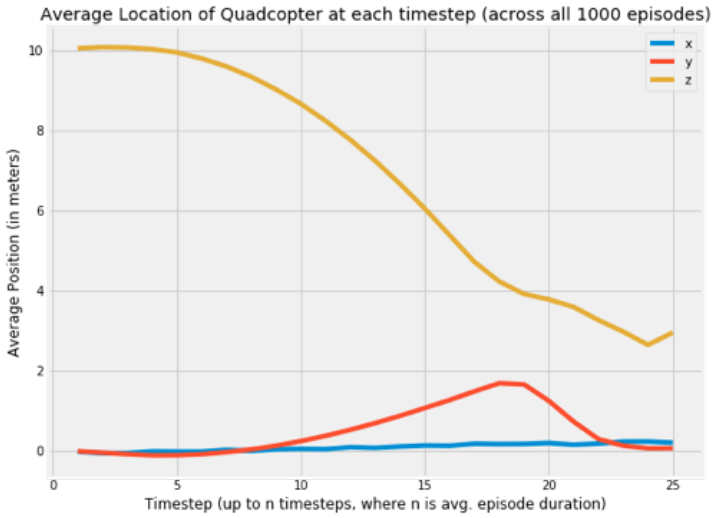
```
# Load simulation results from the .csv file
results = pd.read_csv('ddpg_agent_improved_sample_task_data.txt')

# Plot the reward
vs.plot_rewards(results=results, zoomed_x_range=(900,1000), zoomed_y_range=(-100
,5), agent_name="DDPG_Agent_Improved_Sample_Task", task_name="Sample_Task", n=10
);
```

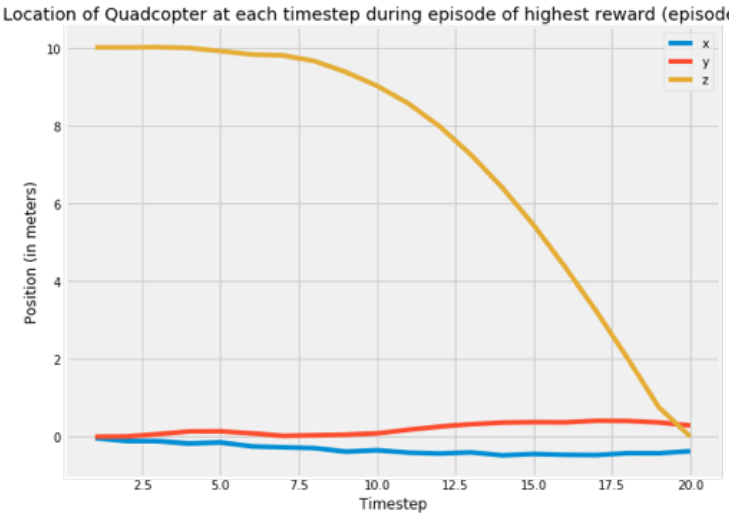


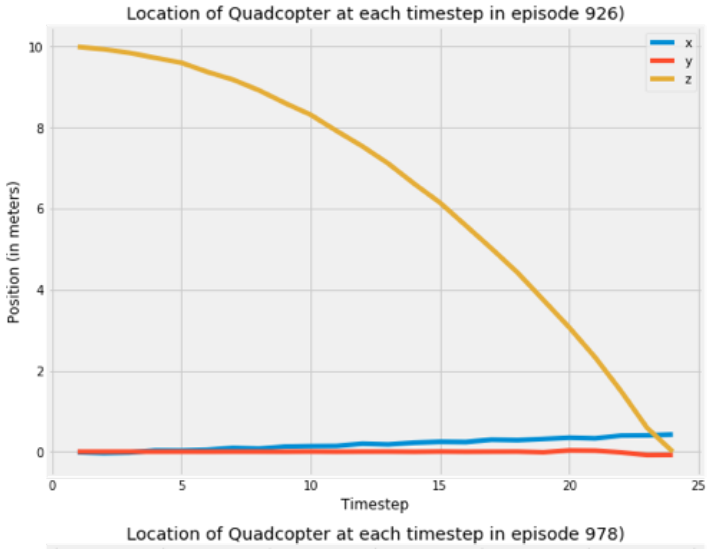
In [4]:

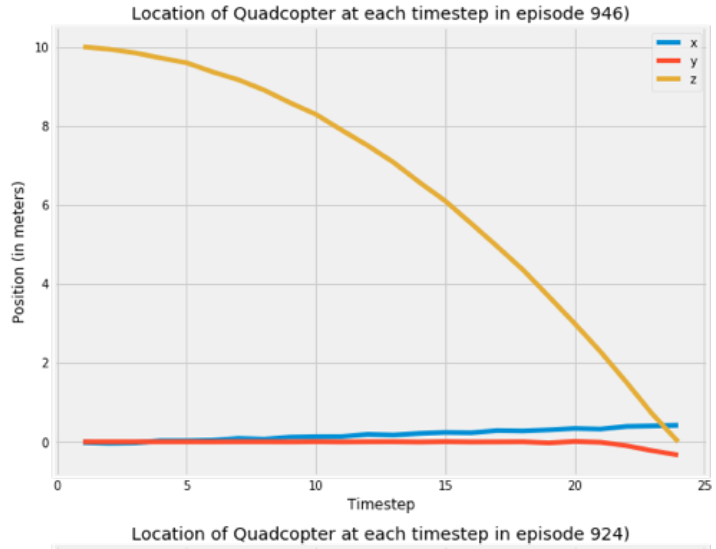
```
# Plot the quadcopter's flight behavior
vs.plot_behavior(results=results, agent_name="DDPG_Agent_Improved_Sample_Task",
task_name="Sample_Task");
```

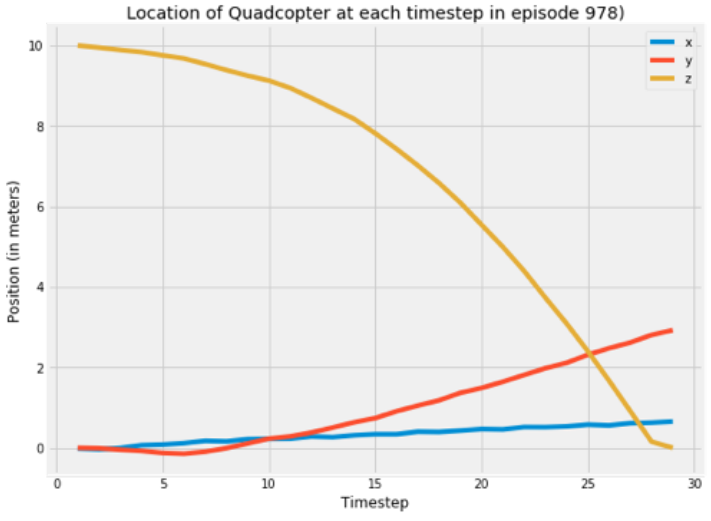
Flight behavior of the DDPG_Agent_Improved_Sample_Task agent on the Sample_Task task.

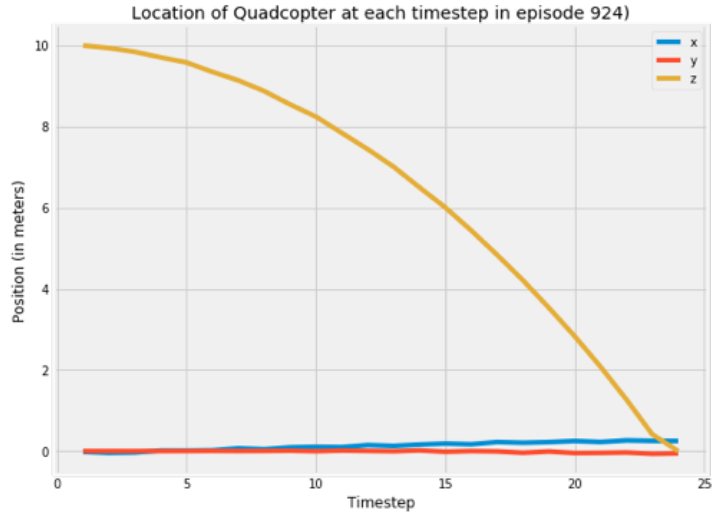












Based on the rewards graphs above, I found that my improved DDPG agent was able to continuously earn an average reward that was better (between 0 and -20) than that which the previous version of my custom agent had been able to earn (which was betwen -20 and -25).

Granted, there is a period toward the end of the final 100 episodes of this simlation where the improved agent's average reward dips down to around -40, however this "dip" is a tiny blip when comparing it to bigger drops in performance that had occurred around 200, 400, 600, and 800 episodes prior in the same simulation. This indicates to me the agent is indeed learning the task, and that I could expect average reward to continue to converge to around -10 over subsequent episodes.

Looking at graphs of the quadcopter's behavior during this simulation, the first thing I notice is that the improved agent more consistetly keeps the copter centered above (0,0) in the x-y plane. Compared to the previous version of my custom agent, this improved agent also does a better job of keeping the copter aloft at a height of 10 meters for a greater length of time.

Now that I've finalized my agent, I have copied it over to the file `agents/agent.py`. The title I've given this agent's class is `DDPG_Agent_Custom_Task`. This agent's model is stored in the file `model.py`.

Running my final DDPG custom agent `DDPG_Agent_Custom_Task` on my custom quadcopter task `Custom_Task`:

In [2]:

```
## TODO: Train your agent here.
# Setup
num_episodes = 200
task = Custom_Task()
agent = DDPG_Agent_Custom_Task(task)
rewards_list = []                                   # store the total rewards earned f
or each episode
best_reward = -np.inf                               # keep track of the best reward ac
ross episodes
best_episode = 0                                    # episode in which best reward is
earned
episode_steps = 0                                   # keep track of number of steps in
each episode

# In order to save the simulation's results to a CSV file.
file_output = 'ddpg_agent_custom_task_data.txt'       # file name for saved resu
lts
labels = ['episode', 'time', 'reward', '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']
```

```
# Run the simulation, and save the results.
with open(file_output, 'w') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(labels)
    # Begin the simulation by starting a new episode.
    state = agent.reset_episode()
    # Run the simulation for each episode.
    for i_episode in range(1, num_episodes+1):
        total_reward = 0
        while True:
            action = agent.act(state)
            next_state, reward, done = task.step(action)
            total_reward += reward
            agent.step(action, reward, next_state, done)

            # Save quadcopter's behavior during each timestep of each episode
            # of the simulation to the CSV file.
            to_write = [i_episode] + [task.sim.time] + [reward] + list(task.sim.
pose) + list(task.sim.v) + list(task.sim.angular_v) + list(action)
            writer.writerow(to_write)
            # Increase episode step count by one.
            episode_steps += 1

            if done:
                rewards_list.append((i_episode, total_reward))
                if total_reward > best_reward:
                    best_reward = total_reward
                    best_episode = i_episode
                print("\rEpisode = {:4d} (duration of {} steps); Reward = {:7.3f
} (best reward = {:7.3f}, in episode {})   ".format(
                    i_episode, episode_steps, total_reward, best_reward, best_ep
isode), end="")  # [debug]
                sys.stdout.flush()
                state = agent.reset_episode() # start a new episode
                episode_steps = 0 # Reset for the next episode
                break
            else:
                state = next_state
```

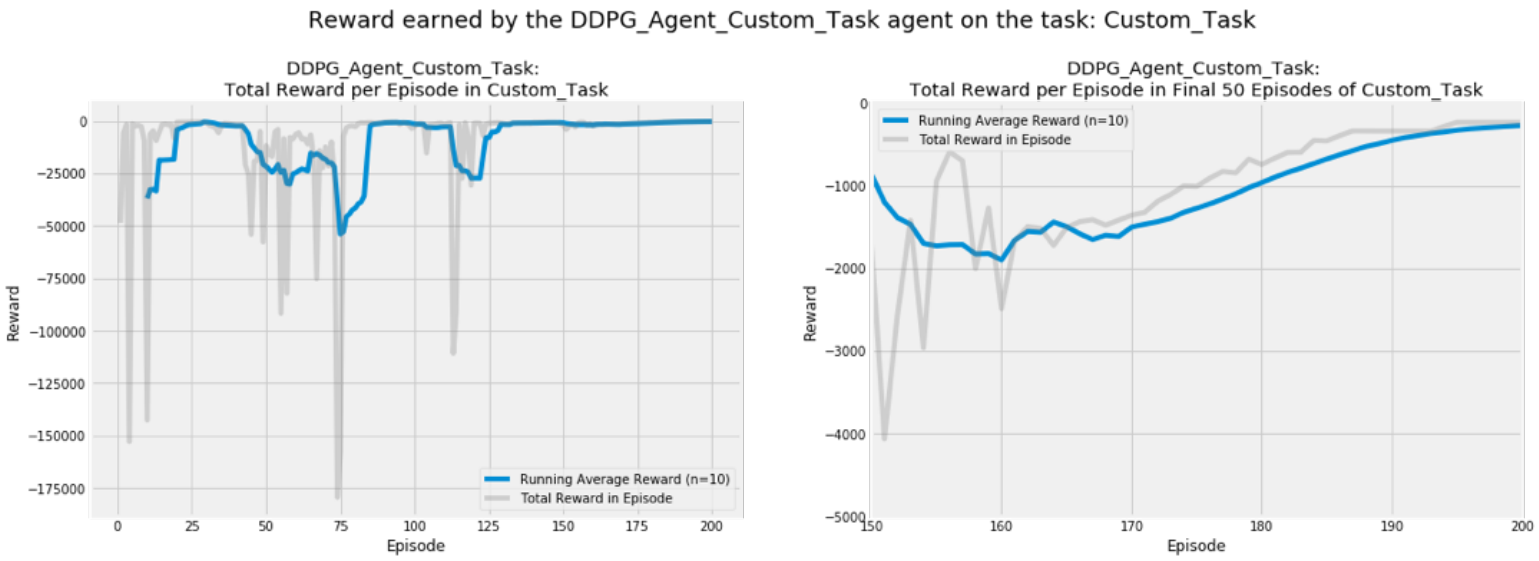
```
Episode =  200 (duration of 7 steps); Reward = -233.919 (best reward
= -232.777, in episode 197)
```

Plot the rewards and flight behavior from running my finalized custom agent

`DDPG_Agent_Custom_Task` on my custom quadcopter task `Custom_Task`:

In [15]:

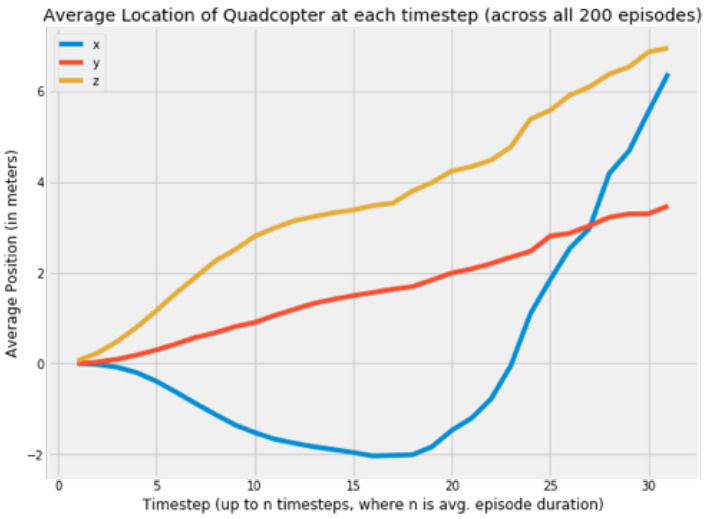
```
# Load simulation results from the .csv file
results = pd.read_csv('ddpg_agent_custom_task_data.txt')

# Plot the reward
vs.plot_rewards(results=results, zoomed_x_range=(150,200), zoomed_y_range=(-5000
, 5), agent_name="DDPG_Agent_Custom_Task", task_name="Custom_Task", n=10);
```

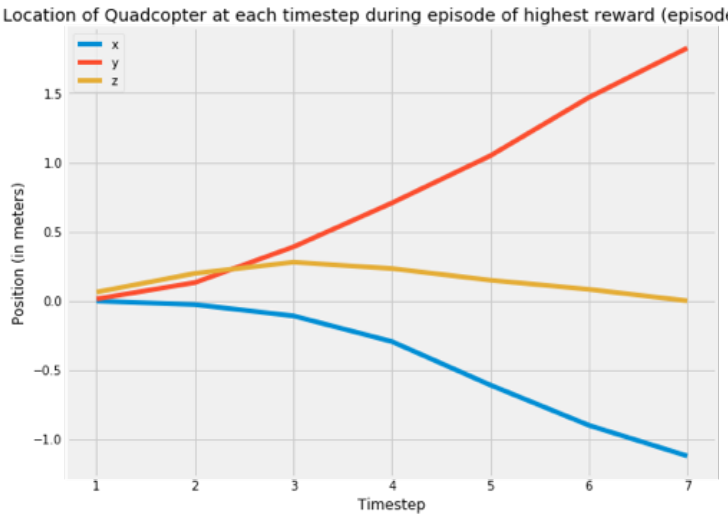


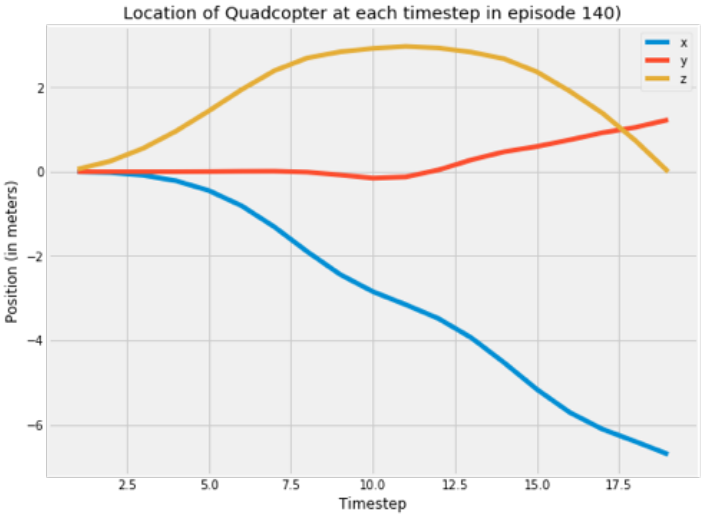
In [4]:

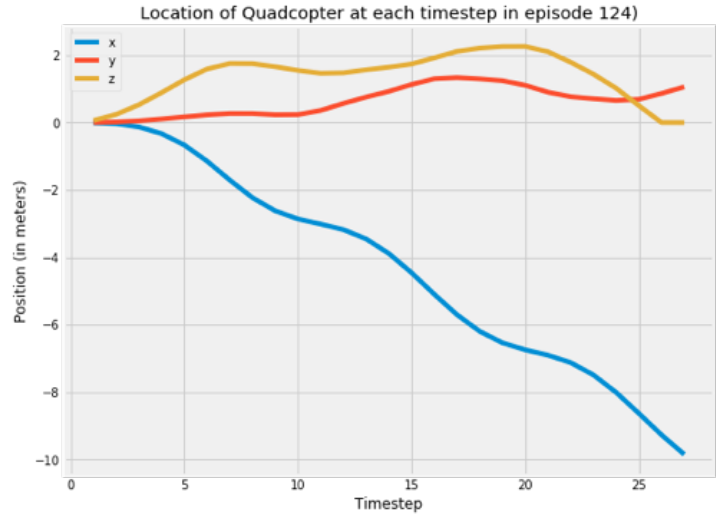
```
# Plot the quadcopter's flight behavior
vs.plot_behavior(results=results, agent_name="DDPG_Agent_Custom_Task", task_name
="Custom_Task");
```

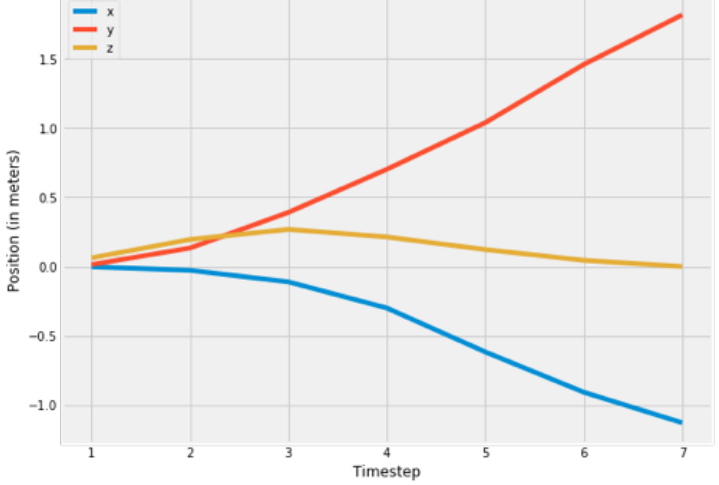
Flight behavior of the DDPG_Agent_Custom_Task agent on the Custom_Task task.

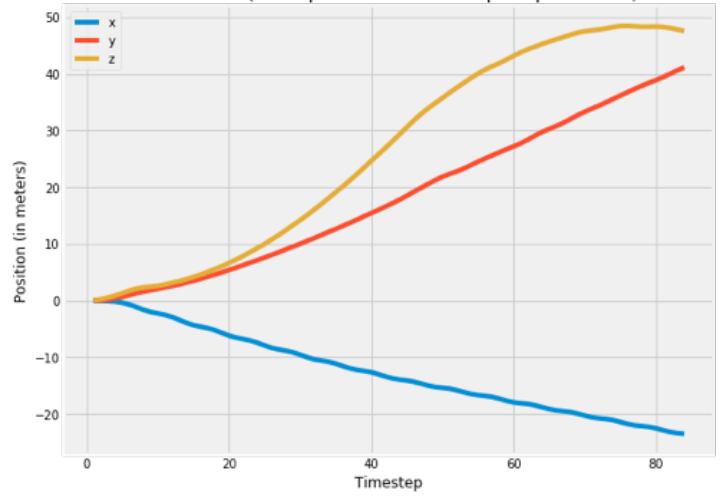












Reflections

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

Answer:

I designed a simple take-off and hover task: the quadcopter begins the task at rest on the ground at the center of the map -- at x-y-z coordinates of (0,0,0). Because the copter begins at rest, its initial velocities and its initial angular velocities are both 0.

Once the task commences, the quadcopter must takeoff and elevate to an altitude of 10 meters above the ground as rapidly as possible. It should do this while maintaining its location above the center of the x-y plane.

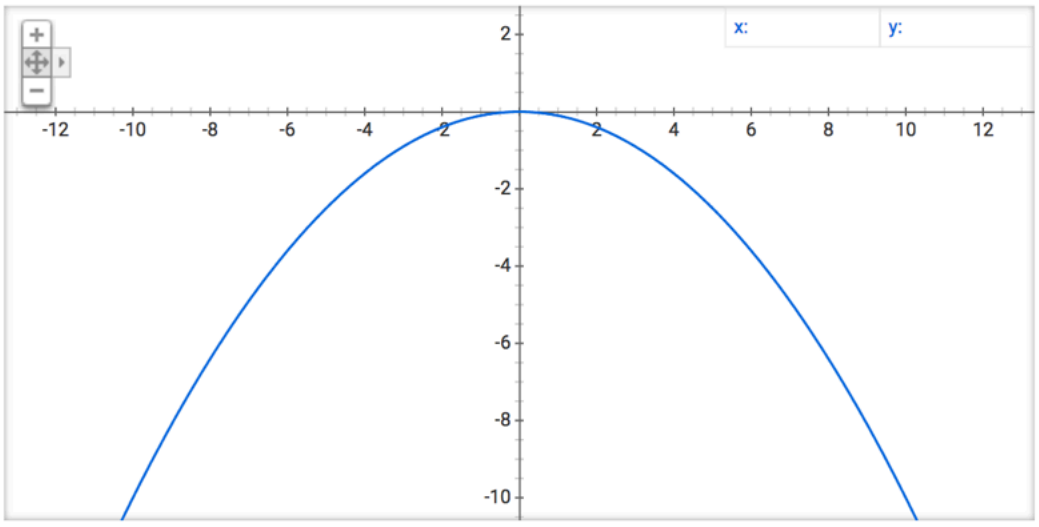
When the quadcopter reaches a height of 10 meters, it should continue hovering at this height, and at a position directly above the center of the map at x-y coordinates of (0,0).

I used a reward function that penalizes the copter when it is far away from its target position. The closer the copter gets to its target, the smaller this penalty becomes:

```
reward = 1 - .1*((abs(current_position - target_position)).sum())**2
```

An astute observer will notice that the rate of decrease of the penalty actually decreases as the copter gets closer to its goal location. Indeed, I based the penalty term's calculation off the equation: $f(x) = -\frac{1}{10}x^2$, where $f(x)$ is the amount of the penalty, and x is the absolute distance between the copter's current position and its goal position, in terms of x-y-z coordinates.

Graph for -(0.1*x^2) (-x2)

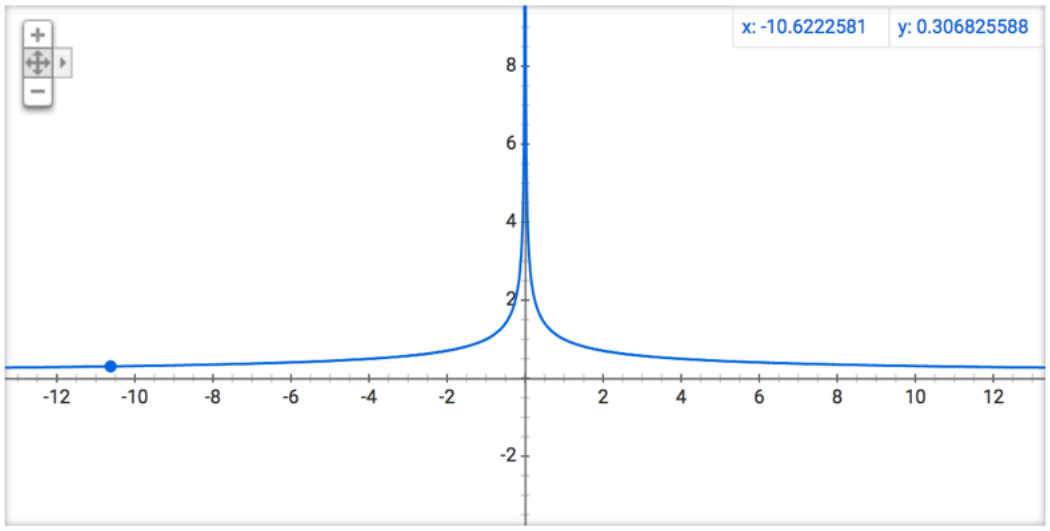


My original intuition was that I ought to use reward functions whose rate of increase actually increased as the copter got closer to its target goal. Two reward functions I tried were:

$$f(x) = \frac{1}{\sqrt{|x|}}$$

Graph for 1/sqrt(abs(x))

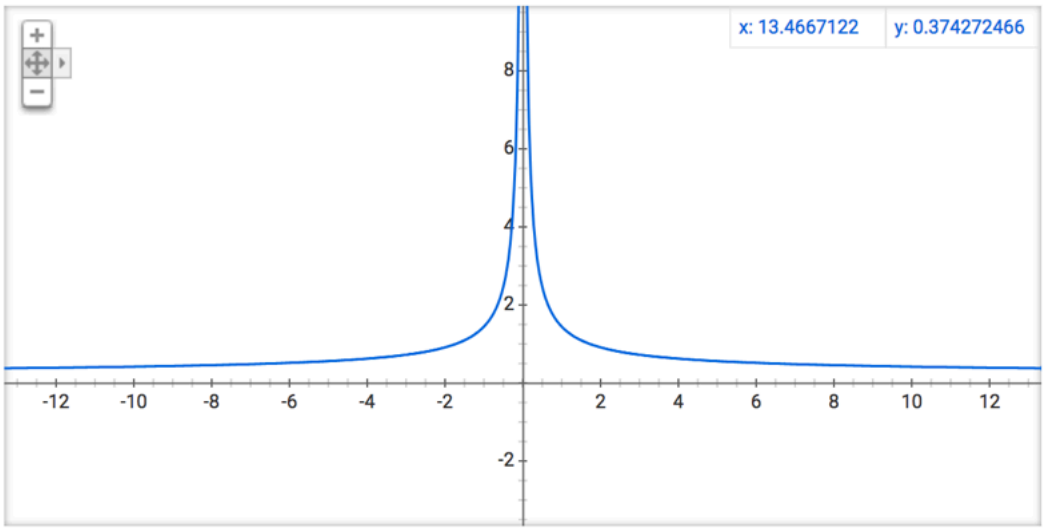
(-x2)



$$f(x) = \frac{1}{\ln(|x|+1)}$$

Graph for 1/ln(abs(x)+1)

(-x2)



I reasoned that an increasing rate of increase would do a better job of "pulling" the copter up toward its goal position as the copter made greater progress toward its goal. In other words, I imagined that as the copter got closer to its goal, the likelihood of it actually reaching that goal would increase exponentially. Note: in the above two functions, if the copter actually reaches its goal location, the solution to the function would be undefined (it would be positive infinity). I worked around this by applying these functions as reward functions only when the copter hadn't reached its goal. If the copter did reach its goal, I gave it a discrete outsized reward as a special bonus.

Unfortunately my intuition was not borne out by what actually happened. It turned out that using the above two functions exacerbated my agent's biggest flaw that it exhibited when simulating this task: after 150 or so episodes, my agent had a tendency to converge to a very small local maximum reward. It did this by learning to crash the copter on the ground in as few timesteps as possible.

While my agent was technically doing its job of maximizing reward, it did this by inducing the copter to behave in a manner quite opposite of what I had hope for -- I wanted the copter to hover indefinitely, not crash as early as possible. After prolonged trial and error, I found that using $f(x) = -\frac{1}{10}x^2$ as my penalty function helped stave off this tendency, somewhat.

Limiting the state size to a size of 3 (just the x-y-z coordinate values) also helped. With an action_repeat value of 3, my complete state size was 9. This is in contrast to the state size of the `Sample_Task` which was 6 (and which became 18 once multiplied by an action_repeat of 3).

Finally, limiting the range of the action space (the possible rotor speeds) to a very high, very narrow range helped ensure that my agent would learn to make the quadcopter actually take off. I used a minimum rotor speed of 800, and a maximum rotor speed of 900. Using too low a value for the minimum rotor speed led to the agent making the copter crash far too early in many episodes.

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:

I. Algorithm Choice

I chose to use the deep deterministic policy gradient, or DDPG, algorithm invented by Lillicrap, Timothy P., et al. and presented in their paper at: https://arxiv.org/pdf/1509.02971.pdf (https://arxiv.org/pdf/1509.02971.pdf). DDPG is a model-free, off-policy actor-critic algorithm. DDPG is based, in part, on two breakthroughs that made Deep Q Learning successful:

1. The deep neural network is trained off-policy with samples from a memory replay buffer. This minimizes correlations between samples.
2. The network is trained with a target Q network. This ensures that targets are kept stable during temporal difference backups.

Because our Quadcopter task has a continuous action space, the Deep Q Learning algorithm on its own would not be sufficient. Deep Q learning can only handle discrete, low-dimensional action spaces. This is because it needs to find an action that, if it were chosen, would maximize the action-value function. In continuous action spaces, there are an infinite number of possible actions, and thus, it is impossible for one to ever know for sure that they have found the unique action value that does, in fact, maximize the action-value function. One workaround could be to discretize the continuous action space, however in this case, computation is still unfeasible, mostly due to the curse of dimensionality -- as degrees of freedom increase, the number of possible actions increase exponentially. This means that a continuous action space, even after it had been discretized, would still be very difficult to explore efficiently.

DDPG sidesteps the limititations of Deep Q learning on continuous spaces by using an actor-critic model. However, this alone is not sufficient. Since the network being updated would also be the same network that's being used to calculate the target value, it is likely that value estimates of state/action pairs would be likely to diverge.

DDPG's core breakthrough is it combines the actor-critic approach, along with target network idea that made Deep Q learning successful: DDPG makes copies of both the actor and critic networks. One set of actor/critic networks is referred to as the "target" networks, while the other set is called the "local" networks. The breakthrough in this approach is that learning is done on the "local" actor/critic networks, and the weights of the "target" networks are then slowly updated based on what is learned in the "local" networks. An update parameter τ, which will have a value between 0 and 1, is used to constrain the rate at which target values change. This is called "soft" target updates. This approach enables the DDPG algorithm to learn stable solutions to tasks with continuous action spaces.

The one remaining challenge is how to explore the continuous action space. Since DDPG is an off-policy algorithm, exploration can be handled independently of learning. DDPG does this by adding a small amount of "noise" to target actions (the actor policy). These slightly perturbed actions are then taken by the algorithm at the beginning of the next timestep, and the learning process continues. Lillicrap, Timothy P., et al. recommended using an Ornstein-Uhlenbeck process, which models the velocity of a Brownian particle with friction. The OU process produces temporally correlated noise that is centered around 0. It has been demonstrated to be useful for exploring physical environments that have momentum.

II. Hyperparameters and architecture:

I have already discussed at length earlier in this notebook my journey and trial and error of finding the right network architecture and hyperparameters. I won't rehash too much of this here. To summarize, I began by using the sample DDPG agent and model classes graciously provided in the Udacity starter code. However, I found that the parameters were woefully inadequate for solving OpenAI Gym's `MountainCarContinuous-v0` task:

- Actor and critic deep networks each had two layers of 64 and 32 units (too few units).
- Soft target update parameter, τ was 0.01 (too large).
- Actor deep network learning rate, α was default rate of 0.001 (too large).

I then revised this agent and model to include some of recommendations of Lillicrap, Timothy P., et al.:

- Actor deep network learning rate, α of 0.0001. (Left critic deep network learning rate unchanged from default at α = 0.001.

- Soft target update parameter, τ was 0.001.
- Actor and critic deep networks each have two layers of 400 and 300 units
- All layers in actor and critic networks are initialized from normal distributsions of $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$, where f is the fan-in of the layer (Lillicrap, Timothy P., et al. recommended initializing from uniform distributions but for `MountainCarContinuous-v0` I found that normal worked better for `MountainCarContinuous-v0`).

I then added in some inspirations that I gleaned from the 3rd place solution to `MountainCarContinuous-v0` by user "lirnli," (https://github.com/lirnli/OpenAI-gym-solutions/blob/master/Continuous_Deep_Deterministic_Policy_Gradient_Net/DDPG%20Class%20ver2.ipynb (https://github.com/lirnli/OpenAI-gym-solutions/blob/master/Continuous_Deep_Deterministic_Policy_Gradient_Net/DDPG%20Class%20ver2.ipynb)) which also was a DDPG implementation:

- Use memory batch size of 256 and a buffer size of 10,000.
- Use an initial exploration policy to warm up for a duration that's 3 times longer than typical (3 times the batch size).
- OU noise perturbs only a fraction of the value of the predicted actions, as opposed to the entire value. This fraction decreases as number of episodes increases. And after episode 100, the "exploration phase" is terminated and no OU noise is added to the actions.
- Use an 'elu' (Exponential Linear Units) activation function in neural net hidden layers. Indeed, it wasn't until I changed my activation function from relu to elu that my agent was able to demonstrate that it learned to solve the MountainCarContinuous-v0 task.

The above modifications resulted in an agent and model that was extremely proficient at solving `MountainCarContinuous-v0`. However, I found that this agent and model could have very unstable performance when trying to solve a task in a quadcopter environment. I found that I couldn't get my agent to have stable performance on quadcopter tasks until I undid most of the specialized tweaks I had included for `MountainCarContinuous-v0` and implemented an architecture virtually identical to that which Lillicrap, Timothy P., et al. used when testing out DDPG on various physical tasks. Here are the final changes:

- Use memory batch size of 64 and a buffer size of 100,000.
- Use an initial exploration policy to warm up for a duration that's same as batch size of 64. (I previously was warming up for a duration of 3 times the batch size.)
- All layers in actor and critic networks are initialized from **uniform** distributions of $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$, where f is the fan-in of the layer.
- Use a 'relu' activation function in all hidden layers of both actor and critic network.
- Initialize final layer weights of both actor and critic networks from a uniform distribution of [-3 x 10^{-3}, 3 x 10^{-3}]. (Lillicrap, Timothy P., et al. also recommended initializing biases on these final layers from the same uniform distribution. I did not bother with biases in these final layers, however.)
- Output layer of the actor network used a 'tanh' activation function. I then mapped this output to the action space, where the middle of the 'tanh' output (0) corresponded to the middle value of the quadcopter environment action range.
- Add OU noise in every timestep of every episode, with θ = 0.15 and σ = 0.2.
- Use batch normalization on the state input and on all layers of the actor network, as well as on the

state input and all layers of the critic network before the action subnetwork is merged with the state subnetwork.
- Initialize final output layer of critic network with kernel L2 loss regularizer of L2 = 10^{-2}

In all iterations of my agent and model I used a discount factor γ = 0.99, and in my critic network I merged the state and action subnetworks in the second hidden layer. I also used an action repeat of 3. This means that for each timestep of the agent, the agent's action and its results in the simulation is repeated 3 times. This basically means that when the agent takes a "step" in the task, that the size of the "next_state" is 3 times the actual state size. Having an action repeat helps our agent to do a better job of inferring velocities.

This final agent is called `DDPG_Agent_Custom_Task` and it is located in `agents/agent.py`. The model for this agent is in `model.py`.

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:

- I found it very difficult to get my agent to learn how to make the quadcopter take off from a complete rest on the ground. If we refer to the four graphs above of my copter's flight behavior when it was controlled by my agent, we can see that at its best, the copter does learn to take off and hover for a period of time. However, after 200 episodes of training, it still very unpredictable as to whether the copter would rise and hover at a low height (say, 2 meters), or a much higher height of 60 meters (the goal was always 10 meters). Efforts to induce the copter to learn to stay centered above (0,0) in the x-y plane were even less successful.

  At its worst, my agent displayed a tendency to seek a local maximum reward by learning to crash the copter in as few timesteps as possible (on the 1st or 2nd timestep of an episode). In my answer to **Question 1** above I outlined a few strategies I employed to counteract this tendency. Along the way, I also found that if I started my simulation with the agent already above the ground (say, at a height of 10 meters), that this drastically reduced the probability that the agent would crash the copter to the ground in 1-2 timesteps. However, seeing as how my goal was to teach the copter to actually perform a real takeoff, I was unwilling to cut this corner.

  It is possible that with more training, the agent would learn to perform better. However, my intuition was that, owing to the complexity of the environment (we're talking about flying, afterall) and assuming there was nothing wrong with my rewards function, that this would require several hours, if not days, of training time -- an amount of both time and money that's beyond my budget.

- Looking at the two rewards graphs above, we can see that learning was a gradual process, with spurts of noise that decreased in frequency and magnitude as more episodes were experienced.

  Furthermore, based on the upper left hand graph of flight behavior, we see that on during an epside on average, as more timesteps pass, the agent flies higher. This is good and this is what we want to see, as it indicates that the agent is taking off.

Unfortunately, taking off is only half of the task -- we also want to see that the agent can hover at a constant height. The graphs depicting flight behavior do not indicate that the agent ever really learned to hover at a constant height for an indefinite period of time. Sooner or later, the copter would inevitable crash to the ground. In episodes where highest reward was earned, this crash-down would happen after a greater number of timesteps had passed.

- In the final 10 episodes, the agent consistently earned its highest mean reward (averged over 10 previous episodes) of the entire simulation. This indicates that the agent was making progress. However I have a hunch that were I to train for say, one million episodes, that I would still see recurring periods of extended sub-optimal performance several thousand episodes into the simulation. My experience of training my agent on this task leads me to believe that it would take a very long time for this agent to learn to avoid the temptation to minimize penalty and earn a local maximum reward by crashing the copter into the ground after only 1-2 timesteps.

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:

- The hardest part of this project frankly was not knowing beyond a shadow of doubt that the physics environment was up to snuff. If I had designed this project I would have used a proven and widely-used physics engine, say the torcs simulator (http://torcs.sourceforge.net/ (http://torcs.sourceforge.net/)).

  Designing either an agent or a rewards equation are each, on their own, difficult enough. Doing both together further compounds the uncertainty of knowing what to debug (e.g. Do I try to fix the agent, the rewards, or both?) when something goes wrong. This is all fine and well, but having to do this troubleshooting on top of a physics engine that doesn't appear to be widely validated is, in my opinion, pushing things a bit too far. Who wrote this physics engine? Where did it come from? Who else uses it? We don't know, and nowhere in either the source code or lesson materials does anyone from Udacity tell us. Indeed, the physics engine .py file itself is a big block of code with no comments whatsoever.

- The most notable aspect of my agent's behavior was, as I stated above in my answers to **Question 1** and **Question 3**, its tendency for limited periods of episodes to learn to minimize penalty by crashing the copter as early as possible in an episode.

  I assumed that this was due either to a flaw in my agent or my rewards equation. However, as I stated just above, I could never shake the sneaking suspicion that this might, at least in part, owe to a quirk endemic to the physics engine itself. There's no way I can know for sure -- but that's sort of my point: it would have been nice to only have to deal with two unknowns (the agent and rewards function) at most. Adding in that third unknown of an unproven and unvalidated environment is, in my opinion, injecting a bit too much unnecessary frustration into what, at the end of the day, is supposed to be a learning experience.