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]:
```
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.

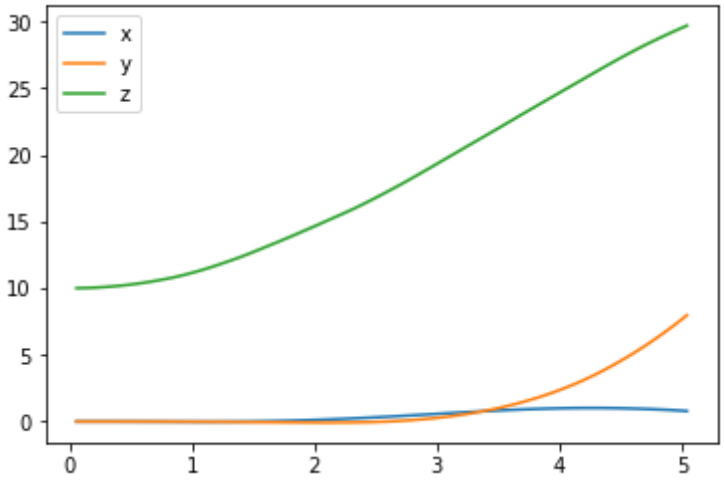
The `labels` list below annotates statistics that are saved while running the simulation. All of this information is saved in a text file `data.txt` and stored in the dictionary `results`.

In [2]:
```
%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 po
sition.
runtime = 5.                                     # time limit of the epi
sode
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 velocit
ies
file_output = 'data.txt'                         # file name for saved r
esults

# 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.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
```

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

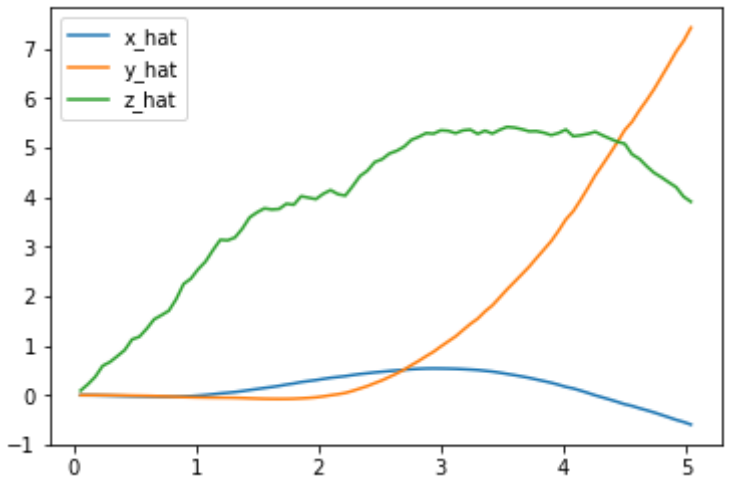
Run the code cell below to visualize how the position of the quadcopter evolved during the simulation.

In [3]:
```
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()
```



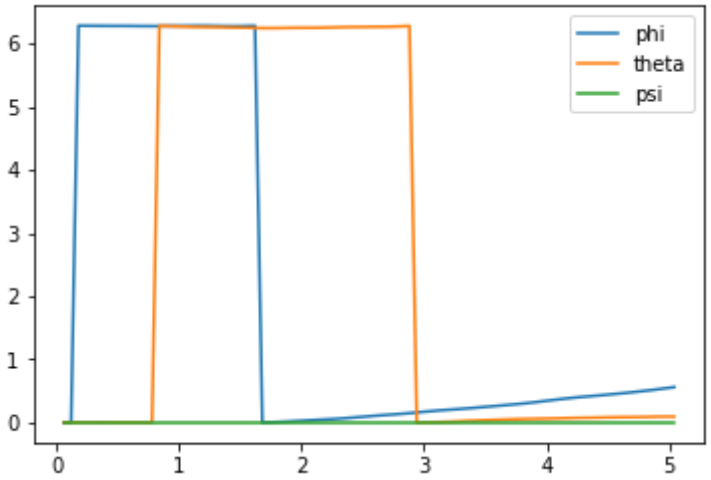
The next code cell visualizes the velocity of the quadcopter.

In [4]:
```
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()
```



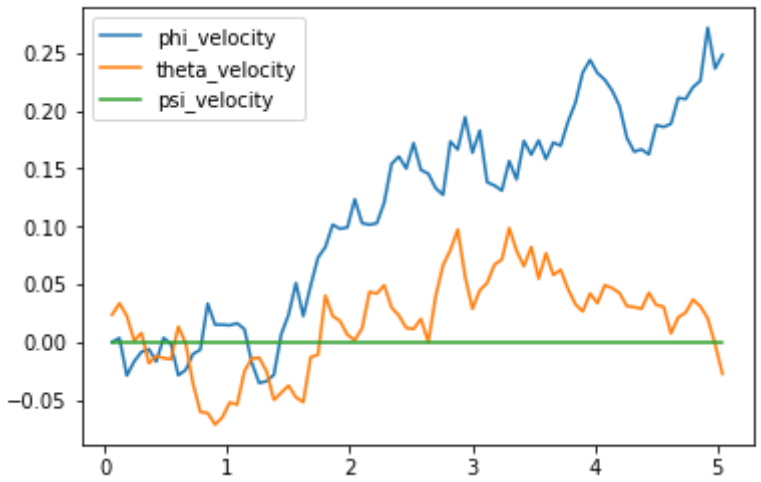
Next, you can plot the Euler angles (the rotation of the quadcopter over the x-, y-, and z-axes),

In [5]:
```
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()
```



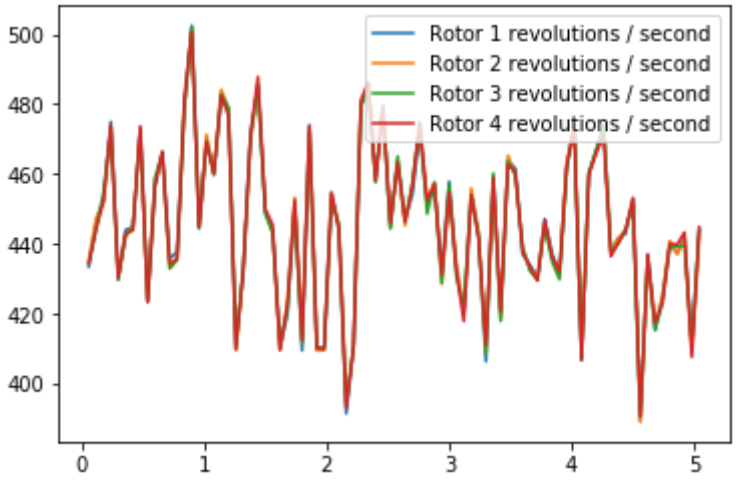
before plotting the velocities (in radians per second) corresponding to each of the Euler angles.

In [6]:
```
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 [7]:
```
plt.plot(results['time'], results['rotor_speed1'], label='Rotor 1 revolu
tions / second')
plt.plot(results['time'], results['rotor_speed2'], label='Rotor 2 revolu
tions / second')
plt.plot(results['time'], results['rotor_speed3'], label='Rotor 3 revolu
tions / second')
plt.plot(results['time'], results['rotor_speed4'], label='Rotor 4 revolu
tions / 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 [8]:
```
# 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)
```

```
[ 0.78828974  7.97439091 29.69447853  0.56009105  0.0933042   0.
]
[-0.59411629  7.42890715  3.90983039]
[ 0.24841487 -0.02722739  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.

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 (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 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 [17]:
```
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 = {}".format(
                i_episode, agent.score, agent.best_score, agent.noise_sc
ale), end="")  # [debug]
            break
    sys.stdout.flush()
```

```
Episode = 1000, score =  -0.934 (best =  -0.196), noise_scale = 3.2
```

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

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.

Defining the task

flying to a specific position and standing there.

Training the agent

In [18]:
```
import sys
import os
import glob
import csv
import pandas as pd
import numpy as np
from test import test, plot_tests
from plot_results import *

labels = [
    'time',
    'done',
    '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',
    'position_reward',
    'euler_reward',
    'velocity_reward',
    'angular_velocity_reward',
    'linear_accel_reward',
    'angular_accel_reward',
    'time_reward',
    'total_reward'
]

def remove_file_with_prefix(prefix, suffix):
   for filename in glob.glob(f'data/{prefix}*{suffix}*'):
        os.remove(filename)

results = []
def write_to_file(prefix, suffix, labels, file_output, data):
    remove_file_with_prefix(prefix, suffix)

    with open(file_output, 'w+') as csvfile:
        writer = csv.writer(csvfile)
        if sum(1 for row in csvfile) == 0:
            results = {x : [] for x in labels}
            writer.writerow(labels)
        for row in data:
            for ii in range(len(labels)):
                results[labels[ii]].append(row[ii])
            writer.writerow(row)
```

```
def write_rewards_to_file(file_output, data):
    rewards_labels = ["episode", "rewards"]
    with open(file_output, 'a+') as csvfile:
        writer = csv.writer(csvfile)
        if sum(1 for line in open(file_output)) == 0:
            writer.writerow(rewards_labels)
        for row in data:
            writer.writerow(row)


def train(episodes, task, agent, prefix):
    remove_file_with_prefix(prefix, "rewards")
    for i_episode in range(1, episodes+1):
        state = agent.reset_episode() # start a new episode
        data_to_write = []
        while True:
            action = agent.act(state)
            next_state, all_rewards, done = task.step(action)
            net_reward = all_rewards[-1]
            agent.step(action, net_reward, next_state, done, i_episode)
            state = next_state
            data_to_write.append([task.sim.time, done] + list(task.sim.p
ose) + list(task.sim.v) + list(task.sim.angular_v) + list(action) + list
(all_rewards))
            if done:
                write_rewards_to_file(f'data/{prefix}_rewards.txt', [[in
t(task.num_episode), round(task.score, 2)]])
                if task.num_episode % 25 == 0:
                    test(task.num_episode, task, agent, prefix)
                if task.score > task.best_score:
                    remove_file_with_prefix(prefix, "best")
                    write_to_file(prefix, "best", labels, f'data/{prefi
x}_best.txt', data_to_write)
                if i_episode == episodes:
                    remove_file_with_prefix(prefix, "last")
                    write_to_file(prefix, "last", labels, f'data/{prefi
x}_last.txt', data_to_write)
                print("\rEpisode = {:4d}, score = {:7.3f} (best episode
 ({:4d}) = {:7.3f}), noise_scale = {}".format(
                    task.num_episode, task.score, task.best_score_episod
e, task.best_score, agent.noise.state), end="")  # [debug]
                break
        sys.stdout.flush()
```

Train model (actor critic 1 - lr=0.001 for actor and critic)

In [19]:
```
from task2 import Task
from ddpg.agent import DDPG

def get_agent(lr, task, sigma=0.2):
    return DDPG(
        task,
        # noise
        mu=0.,
        theta=.15,
        sigma=sigma,
        # replay buffer
        batch_size=64,
        # an episode lasts 50 experiences max (runtime 3. / 0.06), we wa
nt to keep only the last ~200 episodes in memory
        buffer_size=1000000,
        # algorithm parameters
        gamma=0.99,
        tau=0.01,
        # learning
        learning_per_n=1,
        lr=lr
    )
```

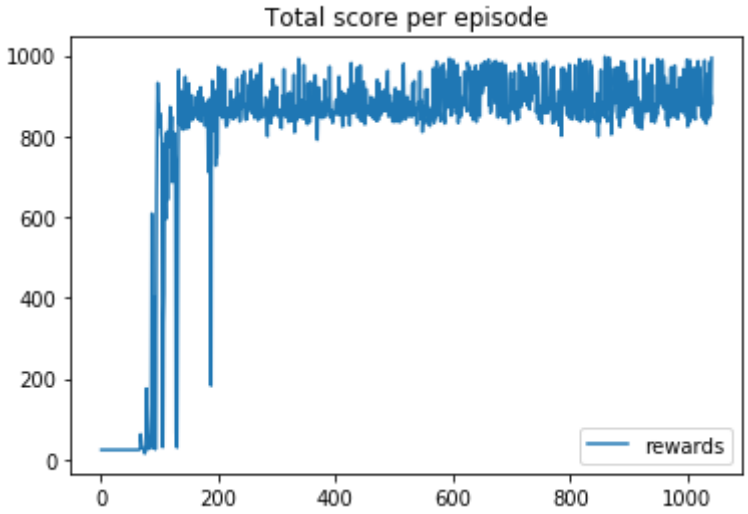
In [20]:
```
%matplotlib inline

test_name = "actor_critic1"
task = Task()
agent = get_agent([0.001, 0.001], task)
```

In [21]:
```
!rm -rf data/actor_critic1*
```

In [22]:
```
train(1000, task, agent, test_name)
```

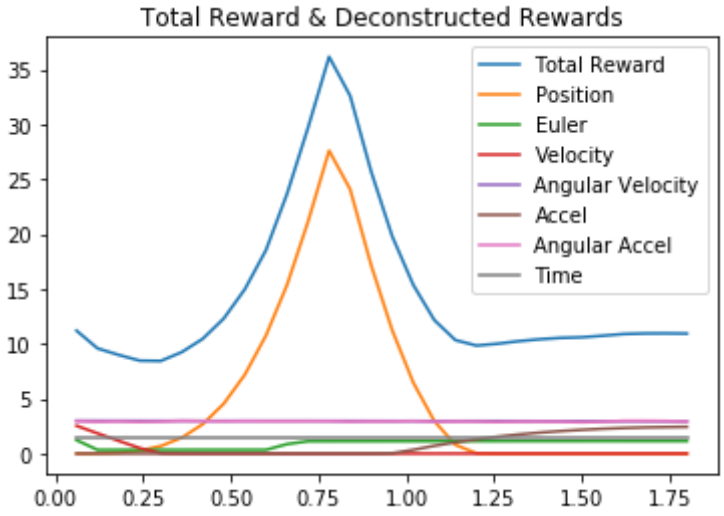
```
Episode = 1041, score = 993.845 (best episode ( 751) = 1002.717), noise
_scale = [ 0.29069121 -0.43444868 -0.18000837  0.51194494]
```

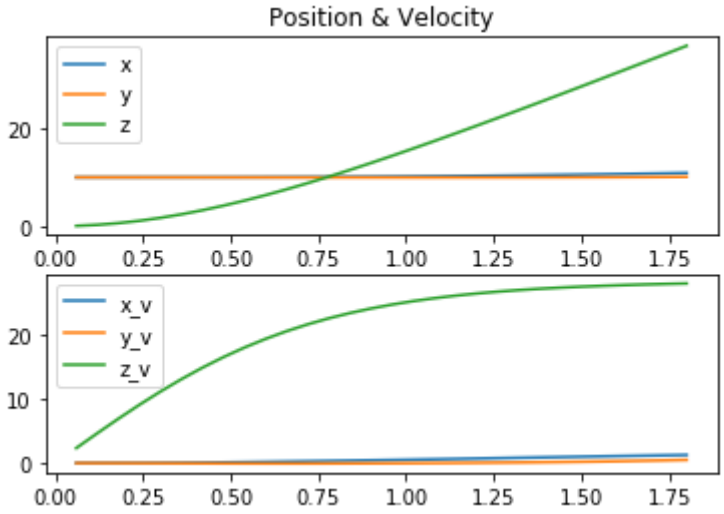
In [23]:
```
results_rewards = pd.read_csv(f'data/{test_name}_rewards.txt')
plot_reward_over_episodes(results_rewards)
```

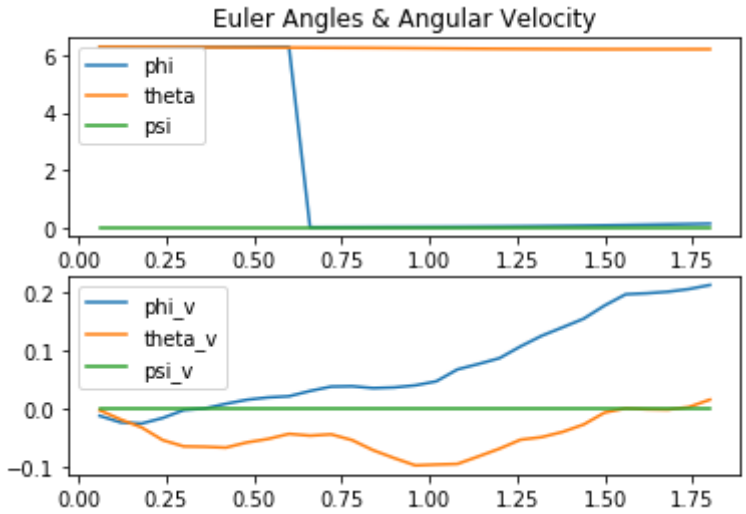


In [24]:
```
plot_tests(test_name)
```

In [25]:
```
results_best = pd.read_csv(f'data/{test_name}_best.txt')
plot_all(results_best)
```







Training with a new (simpler) task definition

See `task3.py`.

In [26]:
```
import sys
import os
import glob
import csv
import pandas as pd
import numpy as np
from test import test, plot_tests
from plot_results import *

labels = [
    'time',
    'done',
    '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',
    'position_reward',
    'euler_reward',
    'time_reward',
    'total_reward'
]

def remove_file_with_prefix(prefix, suffix):
   for filename in glob.glob(f'data/{prefix}*{suffix}*'):
        os.remove(filename)

results = []
def write_to_file(prefix, suffix, labels, file_output, data):
    remove_file_with_prefix(prefix, suffix)

    with open(file_output, 'w+') as csvfile:
        writer = csv.writer(csvfile)
        if sum(1 for row in csvfile) == 0:
            results = {x : [] for x in labels}
            writer.writerow(labels)
        for row in data:
            for ii in range(len(labels)):
                results[labels[ii]].append(row[ii])
            writer.writerow(row)

def write_rewards_to_file(file_output, data):
    rewards_labels = ["episode", "rewards"]
    with open(file_output, 'a+') as csvfile:
        writer = csv.writer(csvfile)
```

```
        if sum(1 for line in open(file_output)) == 0:
            writer.writerow(rewards_labels)
        for row in data:
            writer.writerow(row)


def train(episodes, task, agent, prefix):
    remove_file_with_prefix(prefix, "rewards")
    for i_episode in range(1, episodes+1):
        state = agent.reset_episode() # start a new episode
        data_to_write = []
        while True:
            action = agent.act(state)
            next_state, all_rewards, done = task.step(action)
            net_reward = all_rewards[-1]
            agent.step(action, net_reward, next_state, done, i_episode)
            state = next_state
            data_to_write.append([task.sim.time, done] + list(task.sim.p
ose) + list(task.sim.v) + list(task.sim.angular_v) + list(action) + list
(all_rewards))
            if done:
                write_rewards_to_file(f'data/{prefix}_rewards.txt', [[in
t(task.num_episode), round(task.score, 2)]])
                if task.num_episode % 25 == 0:
                    test(task.num_episode, task, agent, prefix)
                if task.score > task.best_score:
                    remove_file_with_prefix(prefix, "best")
                    write_to_file(prefix, "best", labels, f'data/{prefi
x}_best.txt', data_to_write)
                if i_episode == episodes:
                    remove_file_with_prefix(prefix, "last")
                    write_to_file(prefix, "last", labels, f'data/{prefi
x}_last.txt', data_to_write)
                print("\rEpisode = {:4d}, score = {:7.3f} (best episode
 ({:4d}) = {:7.3f}), noise_scale = {}".format(
                    task.num_episode, task.score, task.best_score_episod
e, task.best_score, agent.noise.state), end="")  # [debug]
                break
        sys.stdout.flush()
```

Train model (actor critic 2 - lr=0.001 for actor and critic)

In [27]:
```
from task3 import Task
from ddpg.agent import DDPG
```

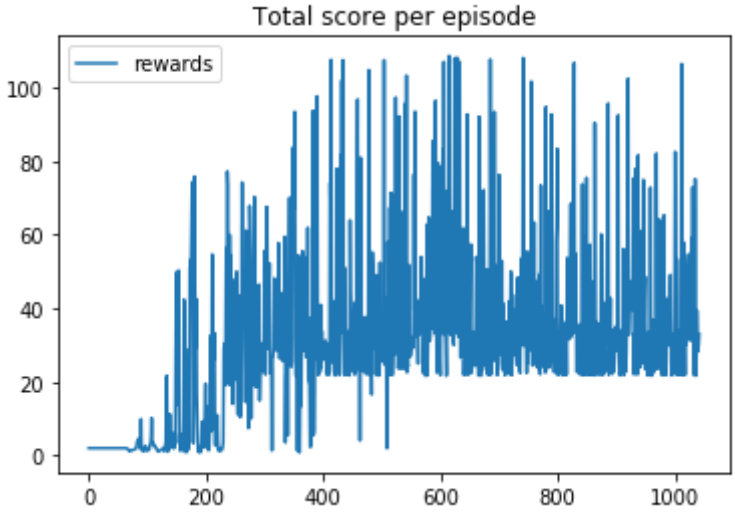
In [28]:
```
%matplotlib inline

test_name = "actor_critic2"
task = Task()
agent = get_agent([0.001, 0.001], task)
```

In [29]:
```
!rm -rf data/actor_critic2*
```

In [30]:
```
train(1000, task, agent, test_name)
```

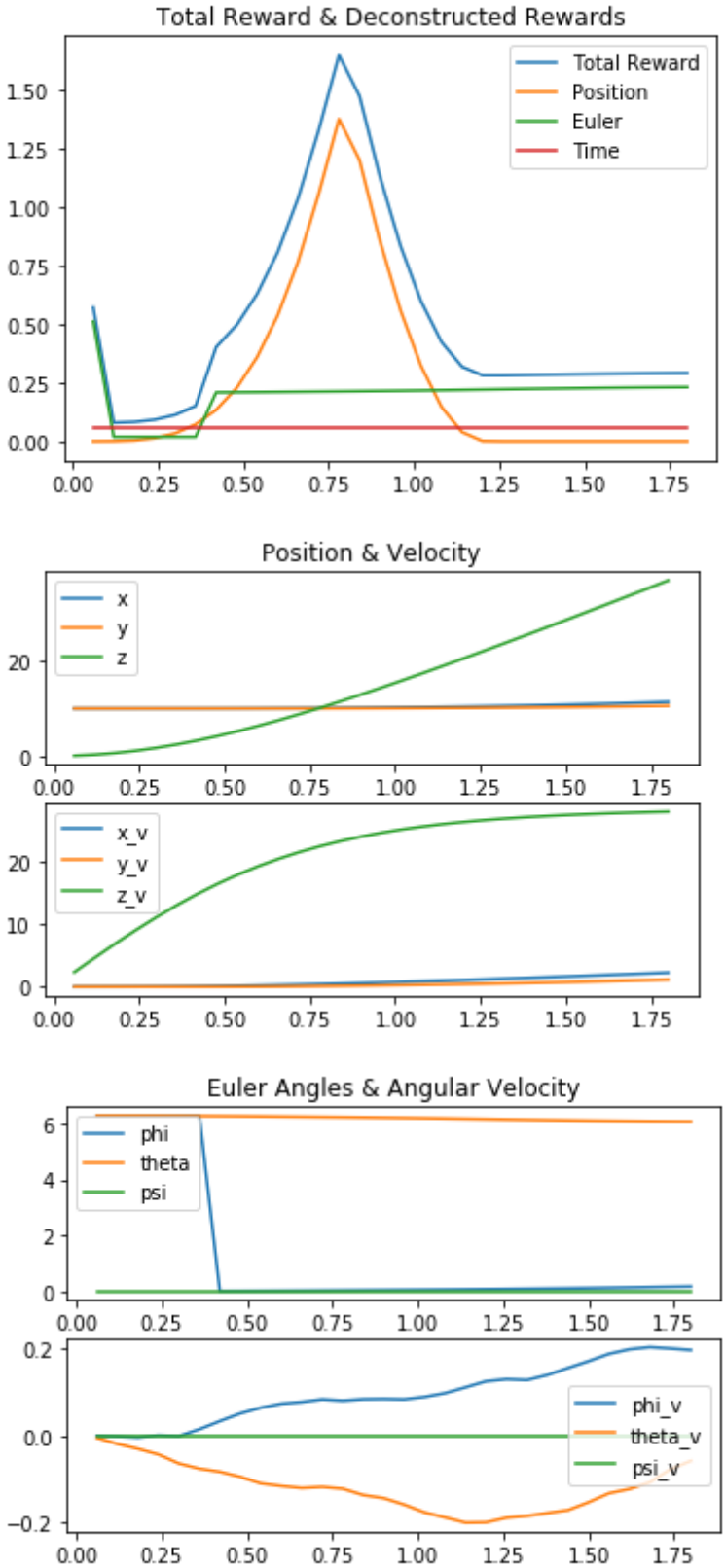
```
Episode = 1041, score =  33.020 (best episode ( 501) = 109.034), noise_
scale = [-0.85246472 -0.85325513  0.42324931  0.30088596]
```

In [31]:
```
results_rewards = pd.read_csv(f'data/{test_name}_rewards.txt')
plot_reward_over_episodes(results_rewards)
```

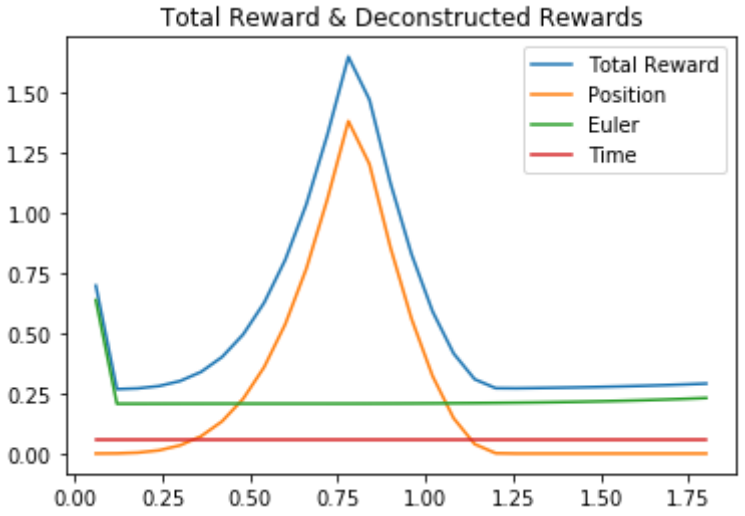


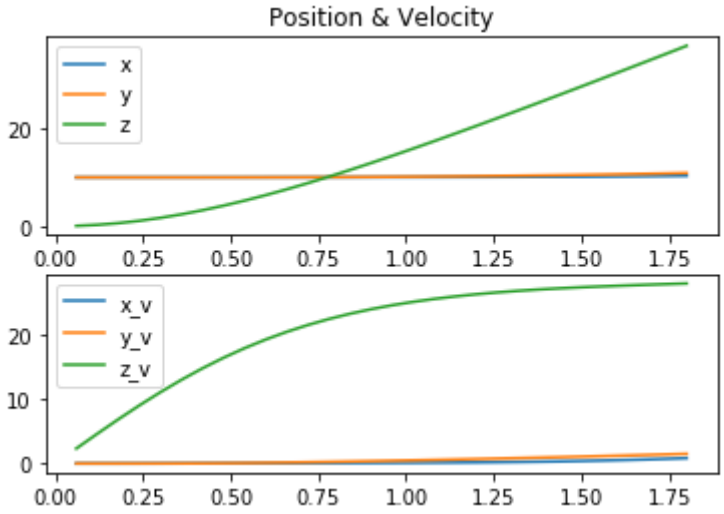
In [32]:
```
plot_tests(test_name)
```

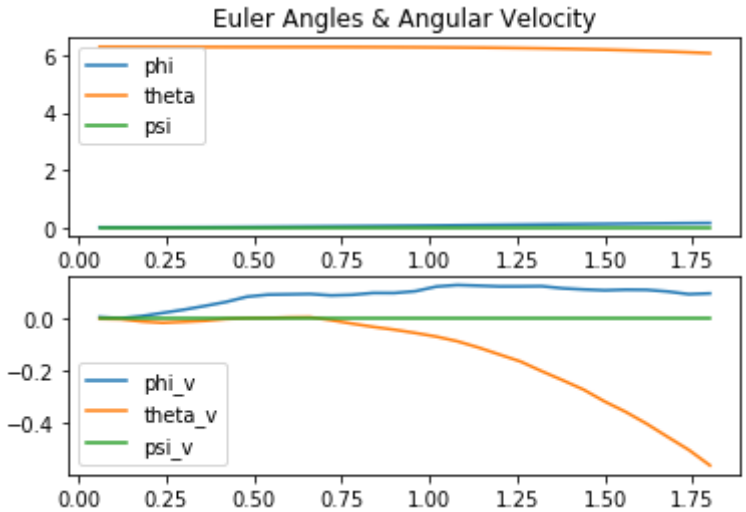
In [33]:
```
results_best = pd.read_csv(f'data/{test_name}_best.txt')
plot_all3(results_best)
```



In [34]:
```
results_last = pd.read_csv(f'data/{test_name}_last.txt')
plot_all3(results_last)
```





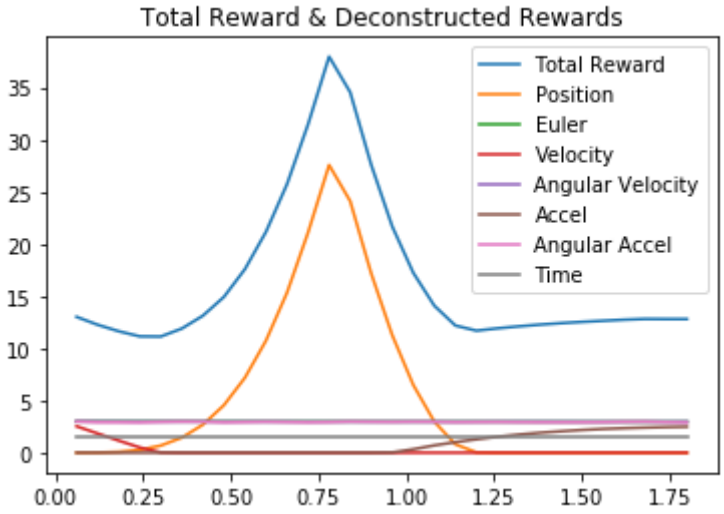


Plot the Rewards

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

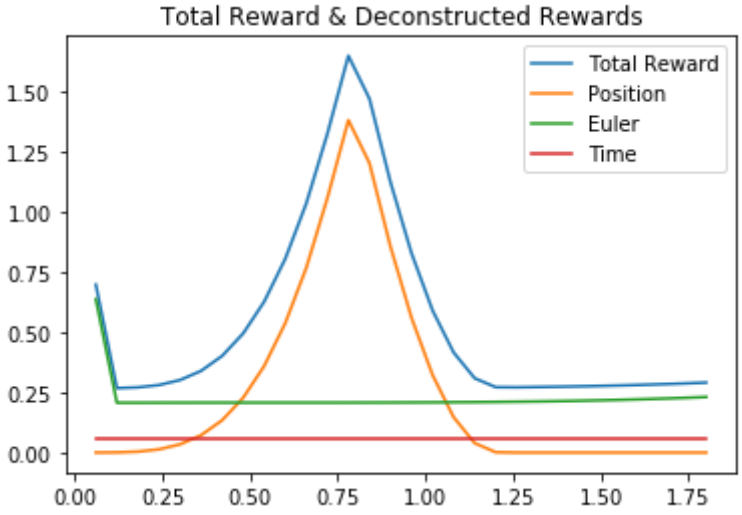
First task design

In [38]:
```
results_last = pd.read_csv(f'data/actor_critic1_last.txt')
plot_reward_over_time(results_last)
```



Second task design

In [41]:
```
results_last = pd.read_csv(f'data/actor_critic2_last.txt')
plot_reward_over_time3(results_last)
```



Reflections

Question 1: Describe your task. How did you design the reward function?

Answer:
Note: For both tasks design, the reward function can be seen inside the `get_rewards` method.

The task was basically to make a quadcopter take off vertically from a specific position where z=0 to a position where z=10, with x and y remaining the same as in the original position. Once the target position reached, the quadcopter should also learn to stay at this specific position indefinitely.

Task designs

We designed the reward function in multiple components:

For both task designs, we used the euler distance to determine how close is the quadcopter to the target on a specific component.

Task (design 1 - task2.py)

The main component was the more the quadcopter was moving closer to the right x-y-z position, the more it received points.

Furthermore, we gave additional time points as a constant to indicate to the quadcopter that remaining in the environment for as long as it can is good.

Lastly, we gave additional points to the quadcopter the closer it got to the final target velocity, angular velocity, euler angles, linear accel, and angular accel (which should all converge to [0, 0, 0].

Task (design 2 - task3.py)

There was two main components.

The first one was the more the quadcopter was moving closer to the right x-y-z position, the more it received points.

The second one was the more the quadcopter was moving closer to euler angles (0,0,0), the more it received points.

Lastly, we gave additional time points as a constant to indicate to the quadcopter that remaining in the environment for as long as it can is good.

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

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

Answer:

1. We use a variant of the DDPG code provided, which we adjusted based on the original DDPG article (https://arxiv.org/abs/1509.02971). Since our task was kind of a "reversed landing" problem, other suited algorithms may have been a better fit (e.g. see https://arxiv.org/abs/1709.03339 (https://arxiv.org/abs/1709.03339)).
2. Several unfruitful variations of hyperparameters were tried until we reverted to values fairly similar to the original DDPG implementation. Here they are in a nutshell:

```
# noise
mu=0.,
theta=.15,
sigma=0.2,
# replay buffer
batch_size=64,
buffer_size=1000000,
# algorithm parameters
gamma=0.99,
tau=0.01,
# learning rate
lr_actor=0.001,
lr_critic=0.001
```

1. We mirrored the DDPG implementation with the addition of two extra hidden layers with 300 units for the actor and critic, which helped to improve the performance of the agent.

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

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

Answer:

1. It was a hard reinforcement learning task and research is still trying to learn how to use RL for accomplishing this kind of task https://arxiv.org/abs/1709.03339 (https://arxiv.org/abs/1709.03339). Unfortunately, we did not achieve a satisfactory result.
2. There seems to be a aha moment, but converging toward the wrong solution. The agent seems to learn to go faster once reaching the right z position, but should remain at this position instead.
3. The final performance was poor (see answer to point 2 right above for details), since it is converging toward the wrong "pattern".

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

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

Answer:

1. The hardest part of the project is to find the right combination of hyperparameters + model architecture + reward function design. It may be simpler for more simple tasks such as just flying to a specific position without wanting the drone to stop and keep flying still at this position.
2. I find that **you get what you incentivize, not what you intend** (source (https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0)). The agent won't behave like you wanted because you **think** it will. Conclusion: the reward function seems very important in the reinforcement learning realm.