PLOUY_MARTIN_TP3

December 17, 2018

```
In [1]: import numpy as np
        import scipy
        import lqg1d
        import matplotlib.pyplot as plt
        import utils
```

0.1 1.4 Experiments

0.1.1 Q1.

Here we have $\mu_\theta(s) = \theta s$ hence $\nabla_\theta log\pi(a|s) = \frac{(a-\theta s)}{\sigma_\omega^2(s)} s$

Note that we use small n_itr, N and T because otherwise it take too much time to run

```
In [2]: class ConstantStep(object):
            def __init__(self, learning_rate):
                self.learning_rate = learning_rate

            def update(self, gt):
                return self.learning_rate * gt

        class Policy(object):
            def __init__(self, theta, sigma):
                self.theta = theta
                self.sigma = sigma

            def draw_action(self, state):
                return scipy.stats.norm(state * self.theta, self.sigma).rvs()

            def update(self, theta):
                self.theta = theta
```

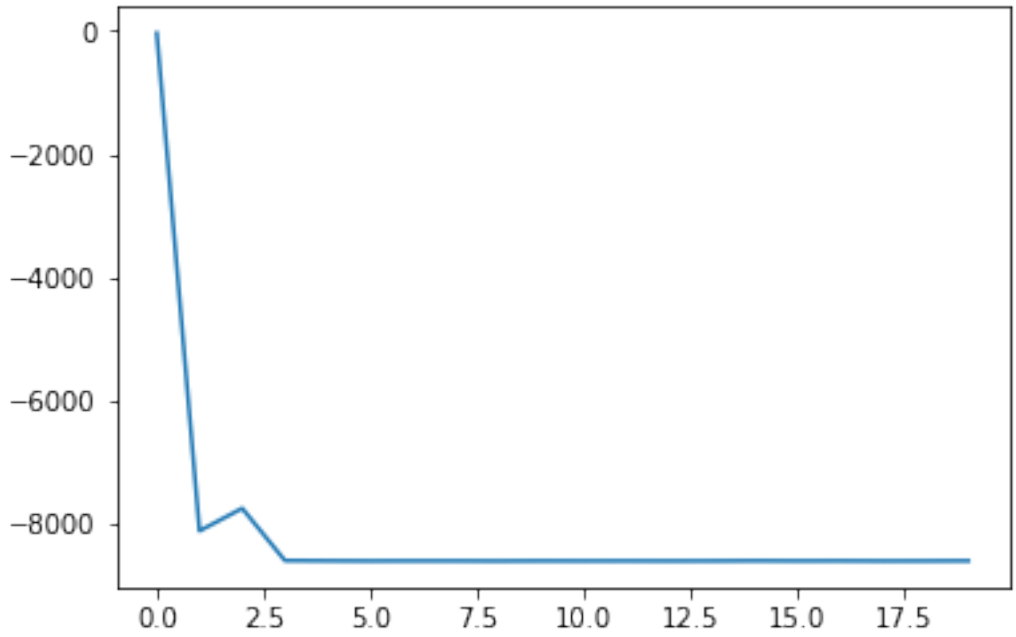
```
In [3]: ######################################################
        # Define the environment and the policy
        ######################################################
        env = lqg1d.LQG1D(initial_state_type='random')

        optimal_mean_parameter = -0.6
```

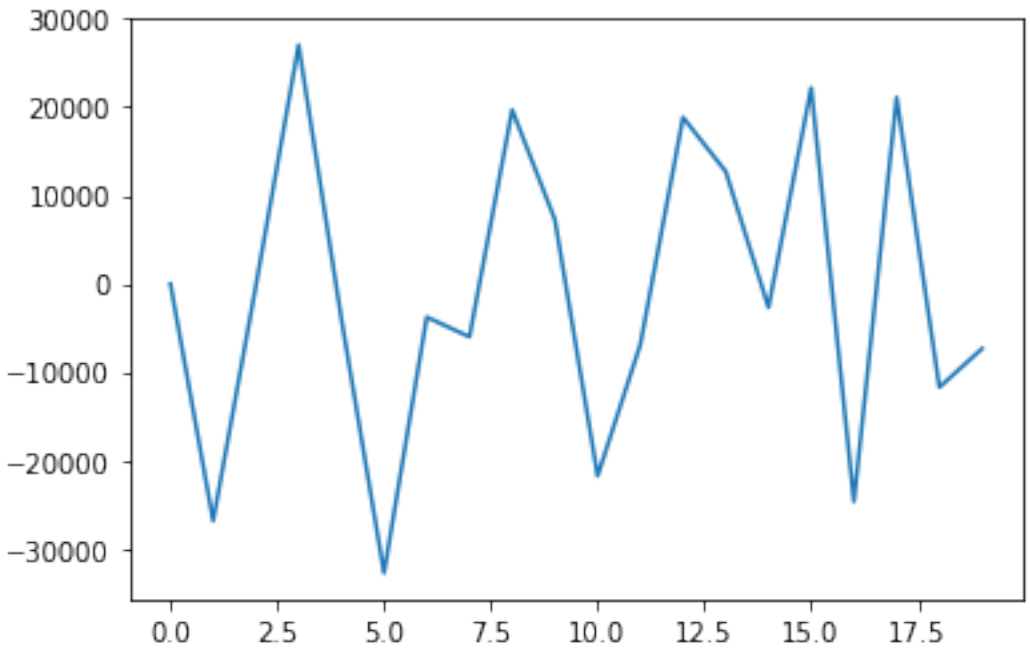
In [4]:
```
def computeTheta(policy, stepper, n_itr, N, T, theta, sigma):
    mean_parameters = []
    avg_return = []

    for i in range(n_itr):
        paths = utils.collect_episodes(env, policy=policy, horizon=T, n_episodes=N)
        avg_return.append(utils.estimate_performance_paths(paths, discount))
        discounts = [discount ** k  for k in range(T)]
        grad_J = 0
        for n in range(N):
            path = paths[n]
            rewards = path["rewards"]
            states = path["states"].reshape([T])
            actions = path["actions"].reshape([T])
            R = np.sum(rewards[:] * discounts[:])
            delta = np.dot(actions - theta * states, states) / (sigma * sigma)
            grad_J += R * delta

        grad_J /= N
        theta = stepper.update(grad_J)
        policy.update(theta)

        mean_parameters.append(theta)
    return avg_return, mean_parameters
```

In [5]:
```
avg_return = []
mean_parameters = []
num_exp = 10
for i in range(num_exp):
    theta_init = -0.2
    sigma = 0.4
    policy = Policy(theta = theta_init, sigma = sigma)

    ######################################################
    # Experiments parameters
    ######################################################
    # We will collect N trajectories per iteration
    N = 20
    # Each trajectory will have at most T time steps
    T = 20
    # Number of policy parameters updates
    n_itr = 20
    # Set the discount factor for the problem
    discount = 0.9
    # Learning rate for the gradient update
    learning_rate = 0.1
    ######################################################
    # define the update rule (stepper)
```

```
        stepper =  ConstantStep(learning_rate = learning_rate)
        tmp_avg_return, tmp_mean_parameters = computeTheta(policy, stepper, n_itr, N, T, the
        if i == 0:
            avg_return =  tmp_avg_return
            mean_parameters = tmp_mean_parameters
        else:
            avg_return = [x + y for x, y in zip(avg_return, tmp_avg_return)]
            mean_parameters = [x + y for x, y in zip(mean_parameters, tmp_mean_parameters)]

    averaged_avg_return = [ a / num_exp for a in avg_return]
    averaged_mean_parameters = [ m / num_exp for m in mean_parameters]

    # plot the average return obtained by simulating the policy
    # at each iteration of the algorithm (this is a rough estimate
    # of the performance)
    plt.figure()
    plt.plot(averaged_avg_return)

    # plot the distance mean parameter
    # of iteration k
    plt.figure()
    distance = [mean_parameter - optimal_mean_parameter for mean_parameter in averaged_mean_
    plt.plot(distance)
```

Out[5]: [<matplotlib.lines.Line2D at 0x1a1550ef98>]





We notice that the results are terrible, theta is way too instable. It is probably due to the learning rate which is too big. We should try to change it to see if we can improve the model.

Note that I tried with bigger n_itr, N and T and it improve the results

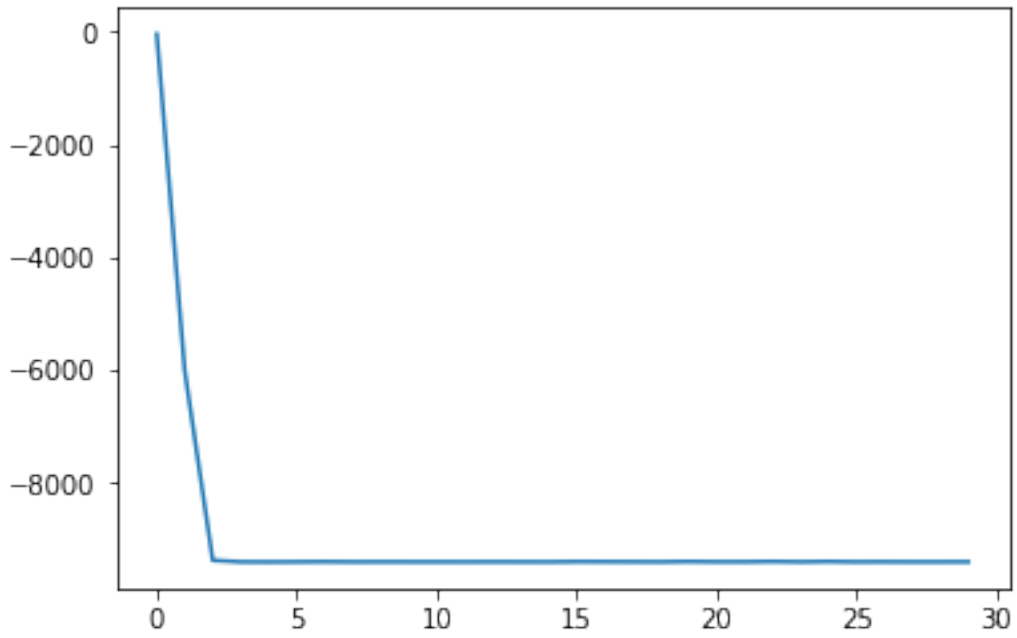
Learning rate of 0.01

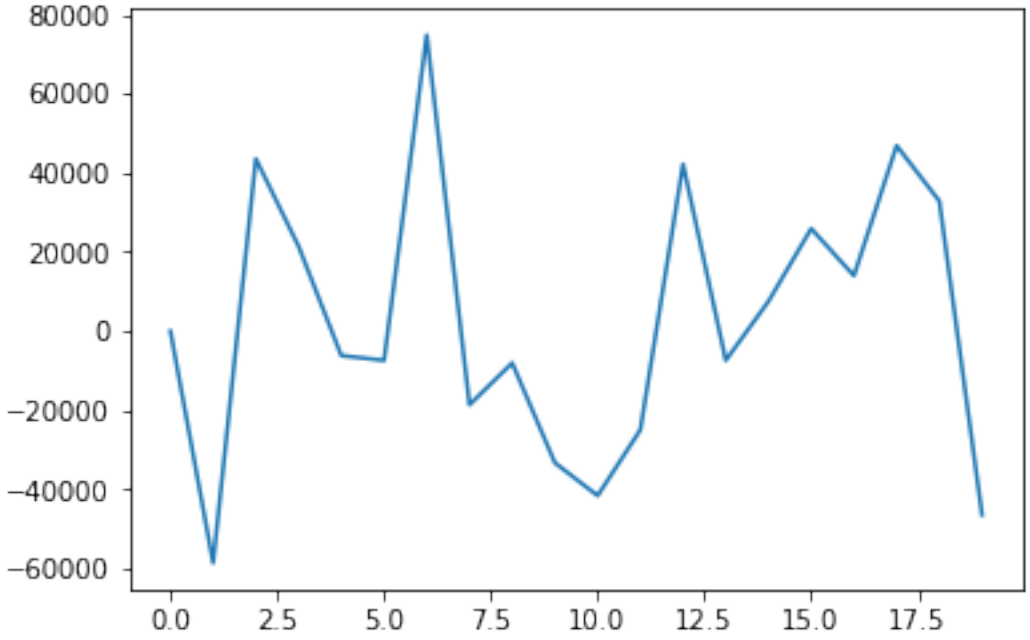
```
In [6]: avg_return = []
        mean_parameters = []
        num_exp = 10
        for i in range(num_exp):
            theta_init = -0.2
            sigma = 0.4
            policy = Policy(theta = theta_init, sigma = sigma)

            ######################################################
            # Experiments parameters
            ######################################################
            # We will collect N trajectories per iteration
            N = 30
            # Each trajectory will have at most T time steps
            T = 30
            # Number of policy parameters updates
            n_itr = 30
            # Set the discount factor for the problem
            discount = 0.9
            # Learning rate for the gradient update
```

```
            learning_rate = 0.01
            ######################################################
            # define the update rule (stepper)
            stepper =  ConstantStep(learning_rate = learning_rate)
            tmp_avg_return, mp_tmean_parameters = computeTheta(policy, stepper, n_itr, N, T, the
            if i == 0:
                avg_return =  tmp_avg_return
                mean_parameters = tmp_mean_parameters
            else:
                avg_return = [x + y for x, y in zip(avg_return, tmp_avg_return)]
                mean_parameters = [x + y for x, y in zip(mean_parameters, tmp_mean_parameters)]

        averaged_avg_return = [ a / num_exp for a in avg_return]
        averaged_mean_parameters = [ m / num_exp for m in mean_parameters]

        # plot the average return obtained by simulating the policy
        # at each iteration of the algorithm (this is a rough estimate
        # of the performance)
        plt.figure()
        plt.plot(averaged_avg_return)

        # plot the distance mean parameter
        # of iteration k
        print(optimal_mean_parameter)
        print(mean_parameters)
        plt.figure()
        distance = [mean_parameter - optimal_mean_parameter for mean_parameter in averaged_mean_
        plt.plot(distance)
```

```
-0.6
[-481.9374814441716, -586532.8113183227, 435032.01767823944, 213800.50793756777, -62385.92279107
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x1a15661e10>]
```



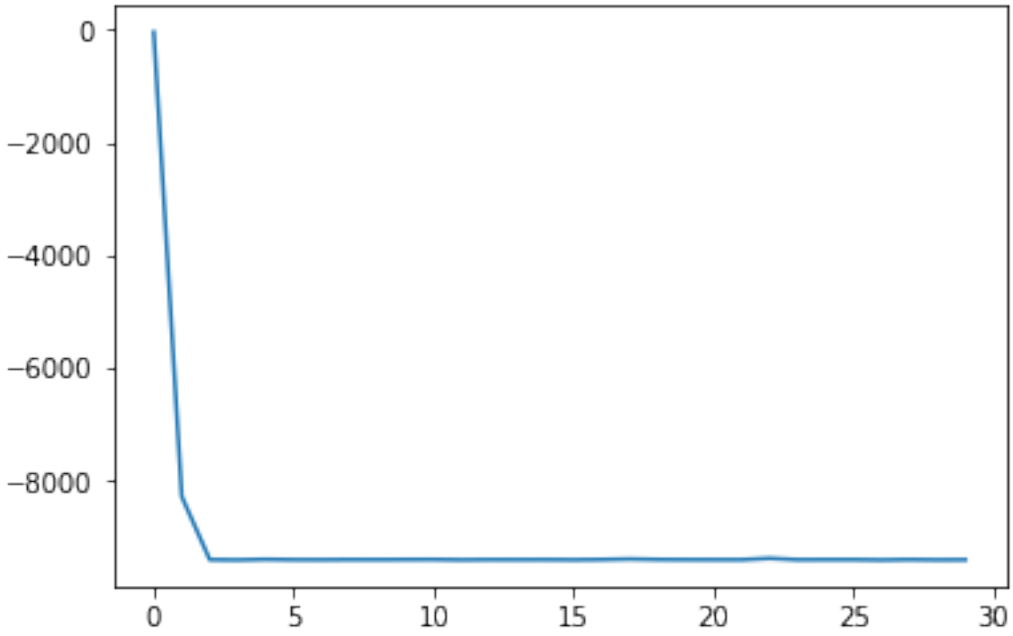


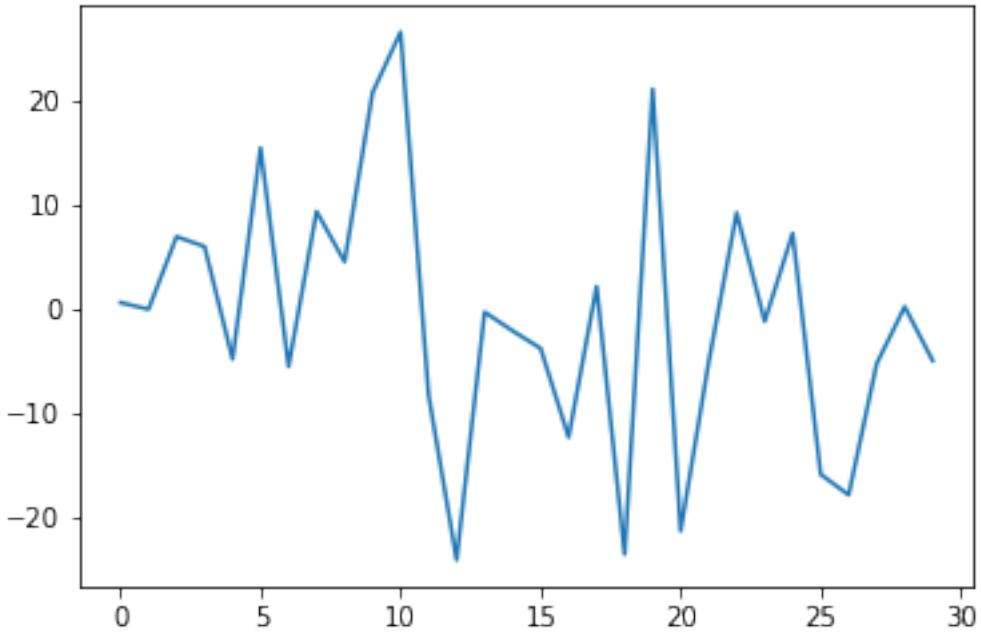
learning rate of 0.0001

```
In [7]: avg_return = []
        mean_parameters = []
```

```
num_exp = 10
for i in range(num_exp):
    theta_init = -0.2
    sigma = 0.4
    policy = Policy(theta = theta_init, sigma = sigma)

    ######################################################
    # Experiments parameters
    ######################################################
    # We will collect N trajectories per iteration
    N = 30
    # Each trajectory will have at most T time steps
    T = 30
    # Number of policy parameters updates
    n_itr = 30
    # Set the discount factor for the problem
    discount = 0.9
    # Learning rate for the gradient update
    learning_rate = 0.0001
    ######################################################
    # define the update rule (stepper)
    stepper =  ConstantStep(learning_rate = learning_rate)
    tmPavg_return, tmp_mean_parameters = computeTheta(policy, stepper, n_itr, N, T, thet
    if i == 0:
        avg_return =  tmp_avg_return
        mean_parameters = tmp_mean_parameters
    else:
        avg_return = [x + y for x, y in zip(avg_return, tmp_avg_return)]
        mean_parameters = [x + y for x, y in zip(mean_parameters, tmp_mean_parameters)]

averaged_avg_return = [ a / num_exp for a in avg_return]
averaged_mean_parameters = [ m / num_exp for m in mean_parameters]


# plot the average return obtained by simulating the policy
# at each iteration of the algorithm (this is a rough estimate
# of the performance)
plt.figure()
plt.plot(averaged_avg_return)

# plot the distance mean parameter
# of iteration k
plt.figure()
distance = [mean_parameter - optimal_mean_parameter for mean_parameter in averaged_mean_
plt.plot(distance)
```

Out[7]: [<matplotlib.lines.Line2D at 0x1a156cadd8>]





We note that when we decrease the learning rate the results get less unstable. But it still doesn't seem to converge.

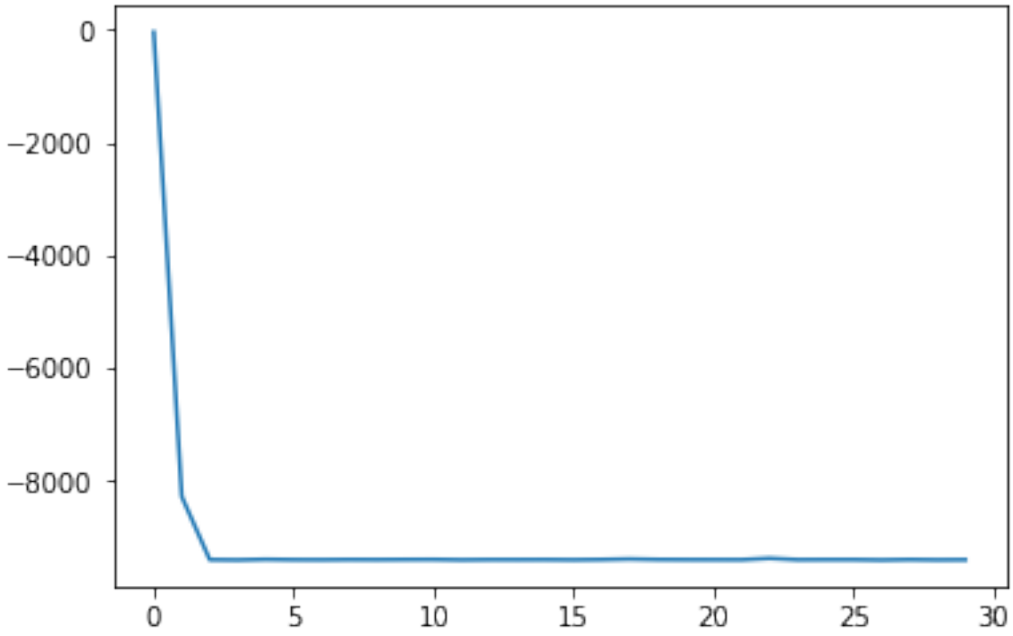
Let's try with an initial theta further from the optimal value to see if we have the same results.

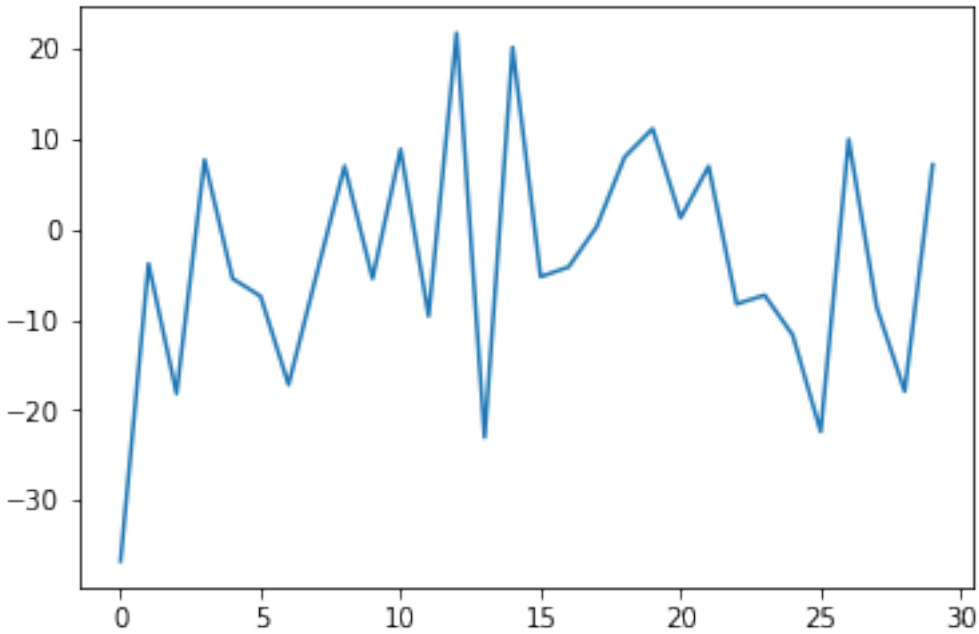
Learning rate of 0.001 but initial theta further from optimal value

```
In [9]: avg_return = []
        mean_parameters = []
        num_exp = 10
        for i in range(num_exp):
            theta_init = -100
            sigma = 0.4
            policy = Policy(theta = theta_init, sigma = sigma)

            ######################################################
            # Experiments parameters
            ######################################################
            # We will collect N trajectories per iteration
            N = 30
            # Each trajectory will have at most T time steps
            T = 30
            # Number of policy parameters updates
            n_itr = 30
            # Set the discount factor for the problem
            discount = 0.9
            # Learning rate for the gradient update
            learning_rate = 0.0001
            ######################################################
            # define the update rule (stepper)
            stepper =  ConstantStep(learning_rate = learning_rate)
            tmPavg_return, tmp_mean_parameters = computeTheta(policy, stepper, n_itr, N, T, thet
            if i == 0:
                avg_return =  tmp_avg_return
                mean_parameters = tmp_mean_parameters
            else:
                avg_return = [x + y for x, y in zip(avg_return, tmp_avg_return)]
                mean_parameters = [x + y for x, y in zip(mean_parameters, tmp_mean_parameters)]

        averaged_avg_return = [ a / num_exp for a in avg_return]
        averaged_mean_parameters = [ m / num_exp for m in mean_parameters]


        # plot the average return obtained by simulating the policy
        # at each iteration of the algorithm (this is a rough estimate
        # of the performance)
        plt.figure()
        plt.plot(averaged_avg_return)

        # plot the distance mean parameter
        # of iteration k
        plt.figure()
        distance = [mean_parameter - optimal_mean_parameter for mean_parameter in averaged_mean_
```

```
plt.plot(distance)
```

Out[9]: [<matplotlib.lines.Line2D at 0x18150277b8>]





We have the same kind of results. It is much betther but still it doesn't seem to converge.

What we could do to improve the model is to use a more fine-grained update rule like ADAM or ADAGRAD which have better results

The learning rate, here α_t is the size of the step toward the gradient that we are taking at each iteration of the algorithm. If the step is too big, then we might leave the region of optimality. If the step is too small, then we almost don't update the parameter and the model doesn't improve. Also, if the step is too small we might never get out of a local optimal region

0.1.2 Q2.

For this

In [ ]: