plouy_martin_tp1

November 12, 2018

In [2]:
```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import time
```

1 1) Dynamic Programming

1.1 Q1. MDP Model

For a MDP Model we need to define a Tuple M = (S, A, p, r) with - S the state space. Here S = s0, s1, s2 - A the action space. Here A = a0, a1, a2 - p the transition probability. We'll define a 3d matrix such that P[s',s,a] = p(s' | s,a) - r the reward of transition (s, a, s'). We'll define a 2d Matrix R[s, a] = r(s, a)

The optimal policy is obvious and is

$$\pi^*$$

such that

$$\pi^*(s0) = a1$$

$$\pi^*(s1) = a1$$

$$\pi^*(s2) = a2$$

In [3]:
```
lenState = 3
lenAction = 3

# Probability of going to x from s0 after taking action a0
Px_s0_a0 = [0.55, 0.45, 0]
# Probability of going to x from s0 after taking action a0
Px_s0_a1 = [0.3, 0.7, 0]
# same pattern ...
Px_s0_a2 = [1, 0, 0]
# same pattern ...
Px_s1_a0 = [1, 0, 0]
# same pattern ...
Px_s1_a1 = [0, 0.4, 0.6]
```

```
# same pattern ...
Px_s1_a2 = [0, 1, 0]
# same pattern ...
Px_s2_a0 = [0, 1, 0]
# same pattern ...
Px_s2_a1 = [0, 0.6, 0.4]
# same pattern ...
Px_s2_a2 = [0, 0, 1]

# we create and fill the matrix P
P = np.zeros((lenState, lenState, lenAction))
P[:,0,0] = Px_s0_a0
P[:,0,1] = Px_s0_a1
P[:,0,2] = Px_s0_a2
P[:,1,0] = Px_s1_a0
P[:,1,1] = Px_s1_a1
P[:,1,2] = Px_s1_a2
P[:,2,0] = Px_s2_a0
P[:,2,1] = Px_s2_a1
P[:,2,2] = Px_s2_a2

# We create and fill the matrix R, whose value R[i,j] is the reward of taking action j a
R = np.zeros((lenState, lenAction))
# reward of taking action a0 at state s0
R[0,0] = 0
# reward of taking action a1 at state s0
R[0,1] = 0
# same pattern ...
R[0,2] = 0.05
# same pattern ...
R[1,0] = 0
# same pattern ...
R[1,1] = 0
# same pattern ...
R[1,2] = 0
# same pattern ...
R[2,0] = 0
# same pattern ...
R[2,1] = 1
# same pattern ...
R[2,2] = 0.9
```

Since we have π^* we can compute V^* in order to plot the difference between V^k and V^*
We need to solve the following system:

$$V^*(0) = r(s0, a1) + 0.95 * (0.3 * V^*(0) + 0.7 * V^*(1))$$
$$V^*(1) = r(s1, a1) + 0.95 * (0.4 * V^*(1) + 0.6 * V^*(2))$$

$$V^*(2) = r(s2, a2) + 0.95 * (1 * V^*(2))$$

which leads to

$$V^* = \begin{pmatrix} 15.39 \\ 16.55 \\ 18 \end{pmatrix}$$

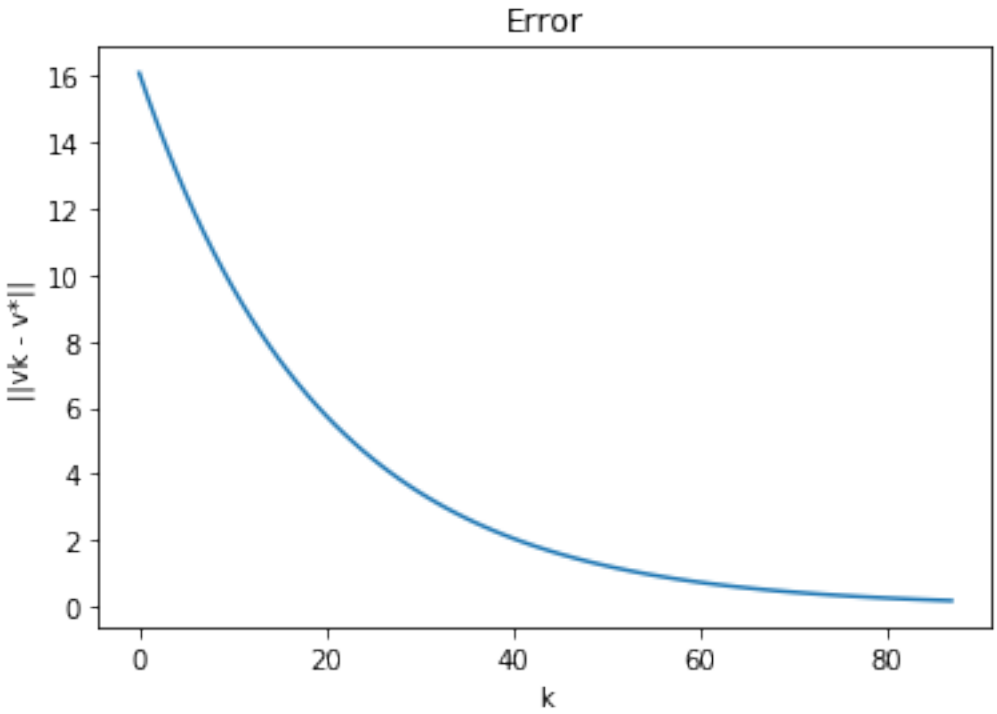
1.2 Q2. Implementation of Value Iteration

We are going to implement the algorithm described in the slides of the lecture 2.
Here N = lenState = 3

```
In [4]: def ValueIteration():
            time_start = time.clock()
            #run your code
            V_ast = np.copy([15.39, 16.55, 18])
            V = [1,1,1]
            gamma = 0.95
            maxItera = 10000
            precision = 0.01
            errors = []

            for k in range(maxItera):
                oldV = np.copy(V)
                # compute Vk term by term
                tmpV = [0,0,0]
                for i in range(len(V)):
                    for j in range(lenAction):
                        tmpV[j] = R[i, j] + gamma * (P[0,i,j]*V[0]+P[1,i,j]*V[1]+P[2,i,j]*V[
                    V[i]=max(tmpV)

                error = np.max(np.abs(V-V_ast))
                errors.append(error)
                if np.max(np.abs(V-oldV)) < precision:
                    break


            time_elapsed = (time.clock() - time_start)

            return V, errors, time_elapsed


In [5]: V, errors, time_elapsed = ValueIteration()
        print("{} iterations were needed ".format(len(errors)))
        print("Optimal value iteration found is : {}".format(V))
        print("Function took {:.2f} ms to run".format(time_elapsed * 1000))
        plt.plot(range(len(errors)),errors)
        plt.xlabel("k")
        plt.ylabel("||vk - v*||")
```

```
        plt.title('Error')
        plt.show()
```

```
88 iterations were needed
Optimal value iteration found is : [15.205719035396244, 16.36294890230353, 17.814561805529337]
Function took 4.95 ms to run
```



1.3 Q3. Implementation of Policy Iteration

Similarly as 2), we are going to implement the algorithm descrived in the slides of the lecture 2.

```
In [6]: def PolicyIteration():
                time_start = time.clock()
                V_ast = np.copy([15.39, 16.55, 18])
                V = [1,1,1]
                Pi = [0,0,0]
                gamma = 0.95
                maxItera = 10
                precision = 0.01
                errors = []

                for k in range(maxItera):
                    oldV = np.copy(V)
```

```
            # we use the formula slide 82 of the lecture 2
            r_pi = [R[i,Pi[i]] for i in range(lenState)]
            P_pi = np.array([P[i, :,Pi[i]] for i in range(lenState)])
            V = np.dot(np.linalg.inv(np.eye(3)-gamma*P_pi), r_pi)

            for i in range(len(Pi)):
                a = [0,0,0]
                for j in range(len(a)):
                    a[j]= R[i, j] + gamma * (P[0,i,j]*V[0]+P[1,i,j]*V[1]+P[2,i,j]*V[2])
                Pi[i] = np.argmax(a)

            if np.max(np.abs(V-oldV)) == 0:
                break

        time_elapsed = (time.clock() - time_start)
        return Pi, k, time_elapsed
```

```
In [7]: Pi, numIter, time_elapsed = PolicyIteration()
        print("{} iterations were needed ".format(numIter))
        print("Optimal Policy found is : {}".format(Pi))
        print("Function took {:.2f} ms to run".format(time_elapsed * 1000))
```

```
3 iterations were needed
Optimal Policy found is : [1, 1, 2]
Function took 1.02 ms to run
```

The time complexity of the Value iteration is higher than the time complexity of the Policy iteration (see lectures). In the example we see that it is indeed the case as the Value iteration took 5ms to run whereas the policy iteration took less than 1 ms to run. We also note that the number of iteration of loops is much lower for the Policy iteration (3) than for the Value Iteration(88). However the computation at each step is cheaper for the value iteration than for the Policy iteration

2 2 Reinforcement Learning

```
In [8]: from gridworld import GridWorld1
        import gridrender as gui
        import numpy as np
        import time

        env = GridWorld1

        ##############################################################################
        # investigate the structure of the environment
        # - env.n_states: the number of states
        # - env.state2coord: converts state number to coordinates (row, col)
```

```
# - env.coord2state: converts coordinates (row, col) into state number
# - env.action_names: converts action number [0,3] into a named action
# - env.state_actions: for each state stores the action availables
#   For example
#       print(env.state_actions[4]) -> [1,3]
#       print(env.action_names[env.state_actions[4]]) -> ['down' 'up']
# - env.gamma: discount factor
################################################################################
print(env.state2coord)
print(env.coord2state)
print(env.state_actions)
for i, el in enumerate(env.state_actions):
        print("s{}: {}".format(i, env.action_names[el]))

################################################################################
# Policy definition
# If you want to represent deterministic action you can just use the number of
# the action. Recall that in the terminal states only action 0 (right) is
# defined.
# In this case, you can use gui.renderpol to visualize the policy
################################################################################
# pol = [1, 2, 0, 0, 1, 1, 0, 0, 0, 0, 3]
# gui.render_policy(env, pol)

################################################################################
# Try to simulate a trajectory
# you can use env.step(s,a, render=True) to visualize the transition
################################################################################

# env.render = True
# state = 0
# fps = 1
# for i in range(5):
#         action = np.random.choice(env.state_actions[state])
#         print(state, action)
#         nexts, reward, term = env.step(state,action)
#         state = nexts
#         time.sleep(1./fps)


# ################################################################################
# # You can also visualize the q-function using render_q
# ################################################################################
# # first get the maximum number of actions available
# max_act = max(map(len, env.state_actions))
# q = np.random.rand(env.n_states, max_act)
# gui.render_q(env, q)
```

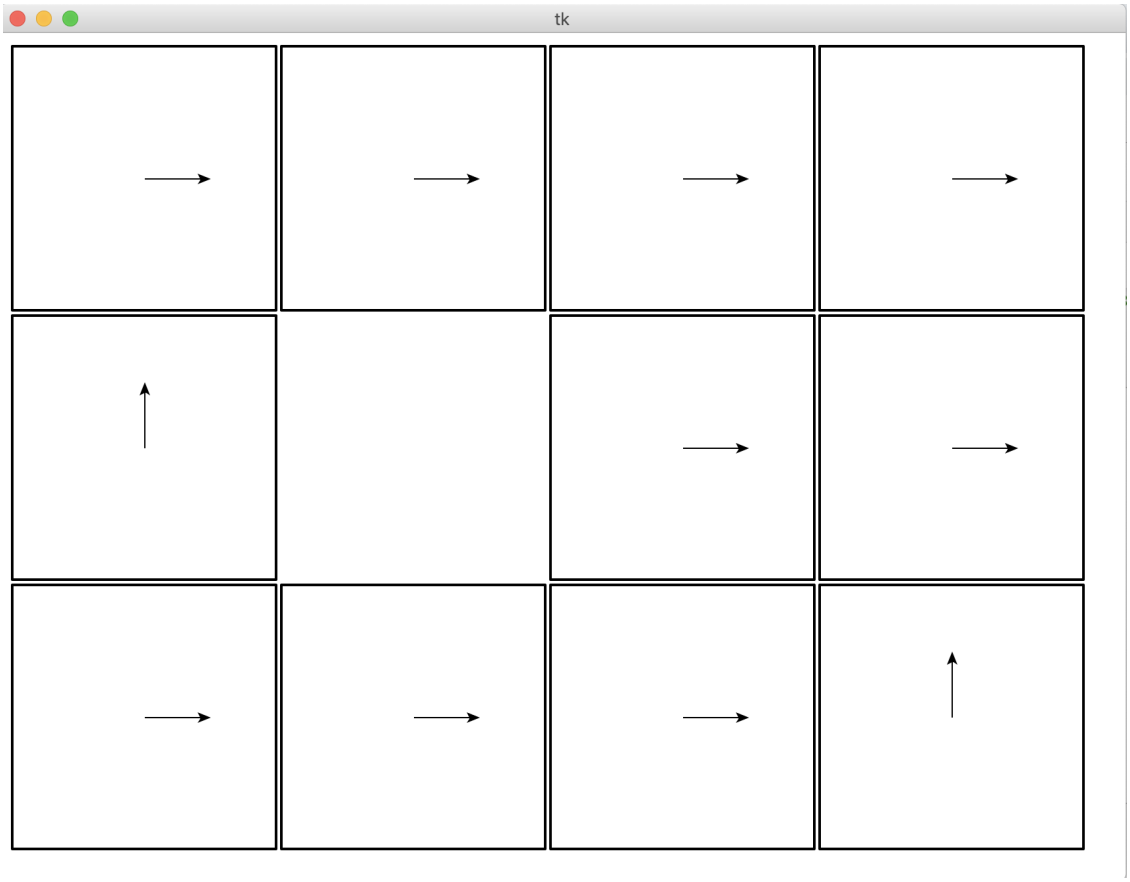
```
        # ##############################################################################
        # # Work to do: Q4
        # ##############################################################################
        # here the v-function and q-function to be used for question 4
        v_q4 = [0.87691855, 0.92820033, 0.98817903, 0.00000000, 0.67106071, -0.99447514, 0.00000
                -0.93358351, -0.99447514]


        # ##############################################################################
        # # Work to do: Q5
        # ##############################################################################
        v_opt = [0.87691855, 0.92820033, 0.98817903, 0.00000000, 0.82369294, 0.92820033, 0.00000
                 0.87691855, 0.82847001]
```

```
[[0, 0], [0, 1], [0, 2], [0, 3], [1, 0], [1, 2], [1, 3], [2, 0], [2, 1], [2, 2], [2, 3]]
[[ 0  1  2  3]
 [ 4 -1  5  6]
 [ 7  8  9 10]]
[[0, 1], [0, 2], [0, 1, 2], [0], [1, 3], [0, 1, 3], [0], [0, 3], [0, 2], [0, 2, 3], [2, 3]]
s0: ['right' 'down']
s1: ['right' 'left']
s2: ['right' 'down' 'left']
s3: ['right']
s4: ['down' 'up']
s5: ['right' 'down' 'up']
s6: ['right']
s7: ['right' 'up']
s8: ['right' 'left']
s9: ['right' 'left' 'up']
s10: ['left' 'up']
```

In [9]:
```
# justification of policy :

        for i, el in enumerate(env.state_actions):
                print("s{}: {}".format(i, env.action_names[el]))

        # if we want the policy right when available otherwise up then for all states except 4 a
        # and for state 4 and 10 select the action up

        selectedPol = [0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 3]
        # let's verify that we chose the correct policy
        gui.render_policy(env, selectedPol)
```

```
s0: ['right' 'down']
s1: ['right' 'left']
s2: ['right' 'down' 'left']
s3: ['right']
```

```
s4: ['down' 'up']
s5: ['right' 'down' 'up']
s6: ['right']
s7: ['right' 'up']
s8: ['right' 'left']
s9: ['right' 'left' 'up']
s10: ['left' 'up']
```



2.1 Q4.

We compute μ_0

```
In [10]: def getMuFrequencies():
             mu = np.zeros(11)
             numIter = 20000
             for i in range(numIter):
                 x = env.reset()
                 mu[x] +=1
             mu = mu/numIter
             return mu
```

```
In [11]: getMuFrequencies()
```

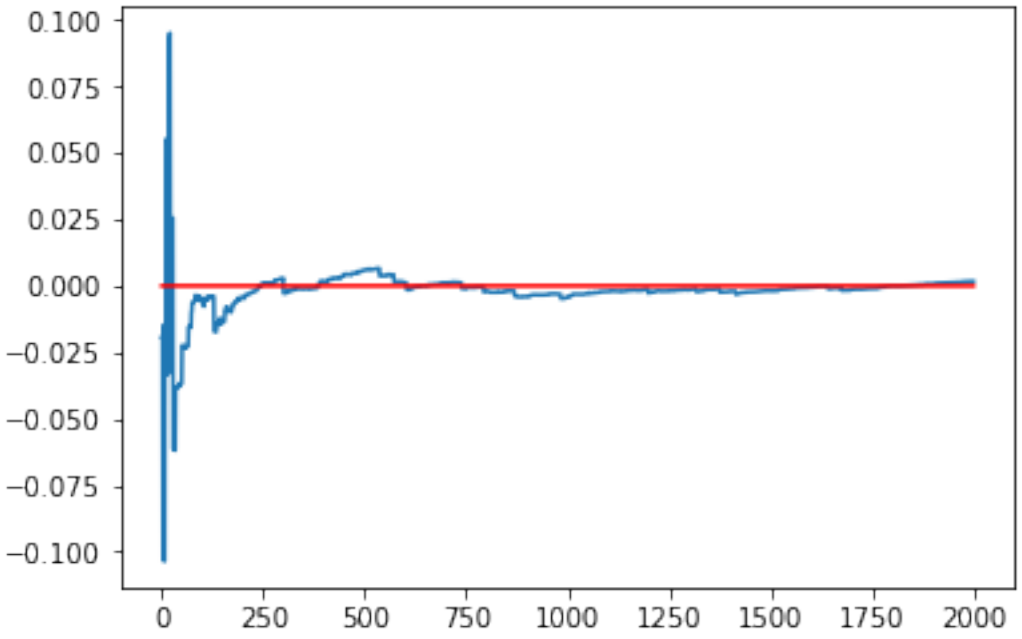
```
Out[11]: array([0.14225, 0.09005, 0.0884 , 0.0425 , 0.0909 , 0.08915, 0.09135,
                0.08825, 0.0936 , 0.09385, 0.0897 ])
```

```
In [12]: def normalizeV(V):
             normalized_V = np.zeros(11)
             for i in V:
                 discountedRewards = V[i]
                 normalized_V[i] = np.sum(discountedRewards)/len(discountedRewards)
             return normalized_V

         mu_0 = getMuFrequencies()
         def computeJ(V):
             j = 0
             for i in range(len(mu_0)):
                 j += mu_0[i] * V[i]
             return j
```

```
In [13]: gamma = 0.95
         # here Rmax is bounded by 1, if we choose delta = 0.2 then
         T_max = -np.log(0.2)/(1-gamma)

         def fillVAndComputeErrors(V, nIter, errors):
             V_star = [0.87691855, 0.92820033, 0.98817903, 0.00000000, 0.67106071, -0.99447514,
                 -0.93358351, -0.99447514]
             J_star = computeJ(V_star)
             for i in range(nIter):
                 term = False
                 x_0 = env.reset()
                 state = x_0
                 t = 0
                 discountedReward = 0
                 while not term and t < T_max:
                     action = selectedPol[state]
                     state, reward, term = env.step(state,action)
                     discountedReward += (gamma**t)*reward
                     t += 1

                 if x_0 not in V:
                     V[x_0] = []
                 V[x_0].append(discountedReward)
                 V_normalized = normalizeV(V)
                 J_n = computeJ(V_normalized)
                 errors.append(J_n - J_star)

             return V, errors
```

```
In [14]: E = 2000
```

```
V = dict()
errors = []
V, errors = fillVAndComputeErrors(V, E, errors)
while len(V) < 11:
    V = fillVAndComputeErrors(V, E, errors)

plt.plot(range(0,len(errors)), errors)
plt.plot(range(0,len(errors)), [0]*len(errors), color ="red")
plt.show()
```



2.2 Q6.

What was said in class is that the optimal policy of an MDP isn't affected by the change of the initial distribution as long as there isn't a state $ s_i $ such that

$$\mu_0(s_i) = 0$$