question 1

We have modified the following code :

```
walls = [7, 8, 9,10,21,27,30,31,32,33,45,46,47]
height = 6
width = 9
```

question 2

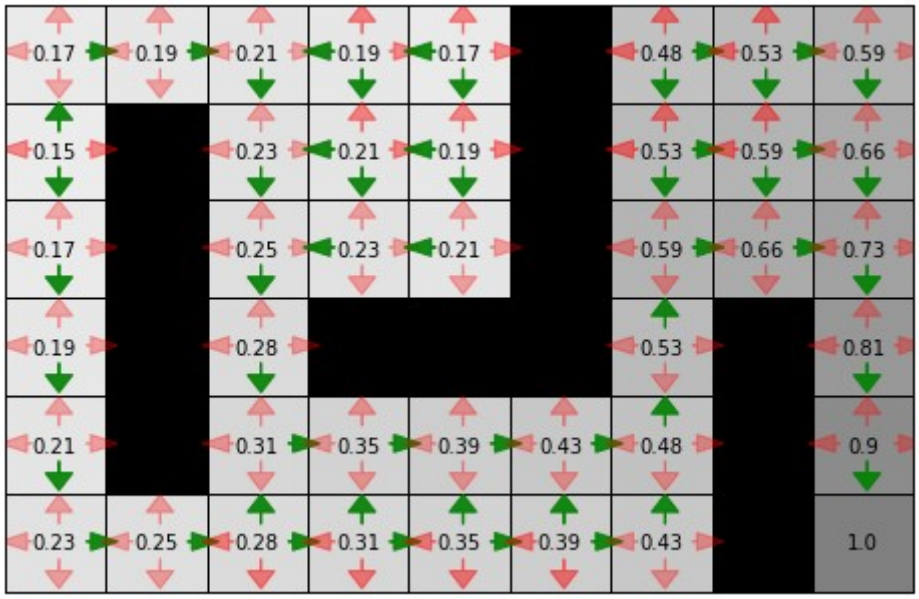
We take the list of the possible actions, we remove the one which is chosen and then, if the random number is below the success rate then we don’t modify anything. In the other case we take a random action,

```
action = [0,1,2,3].remove(u)
r = random_sample((1))[0]
if r < self.proba_success :
    pass
else :
    u = np.random.shuffle(np.array(action))[0]
```

question 3

Here is the missing part of the code :

```
for y in range(mdp.nb_states):
    summ += mdp.P[x, u, y] * np.max(qold[y, :])
q[x, u] = mdp.r[x, u]+mdp.gamma * summ
```



question 4

```
def get_policy_from_q(q):
    return np.argmax(q,axis=1)
```

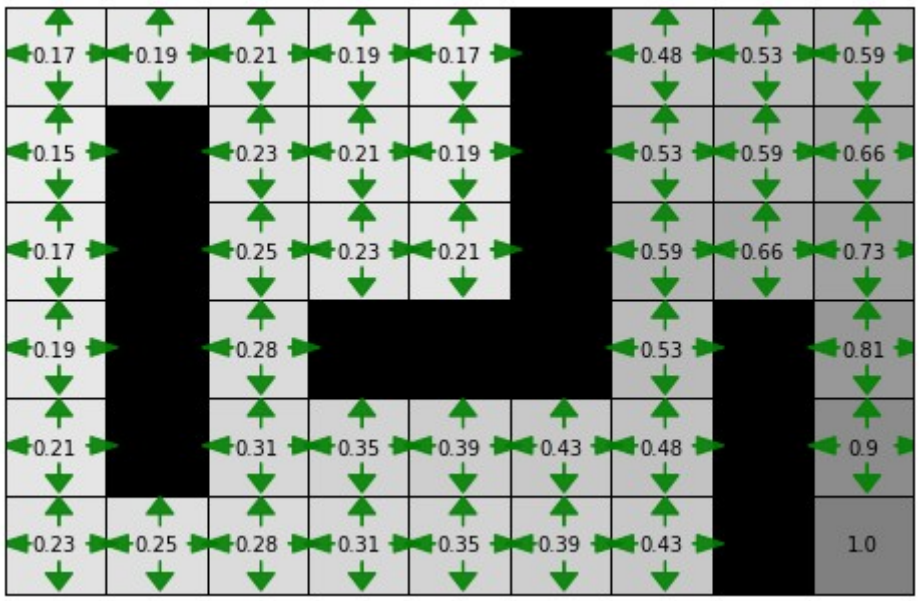
question 5

```
def improve_policy_from_v(mdp, v, policy):
    return np.argmax(mdp.r[x,u]+mdp.gamma*np.sum(mdp.P[x,u,:]*v[:]))
```

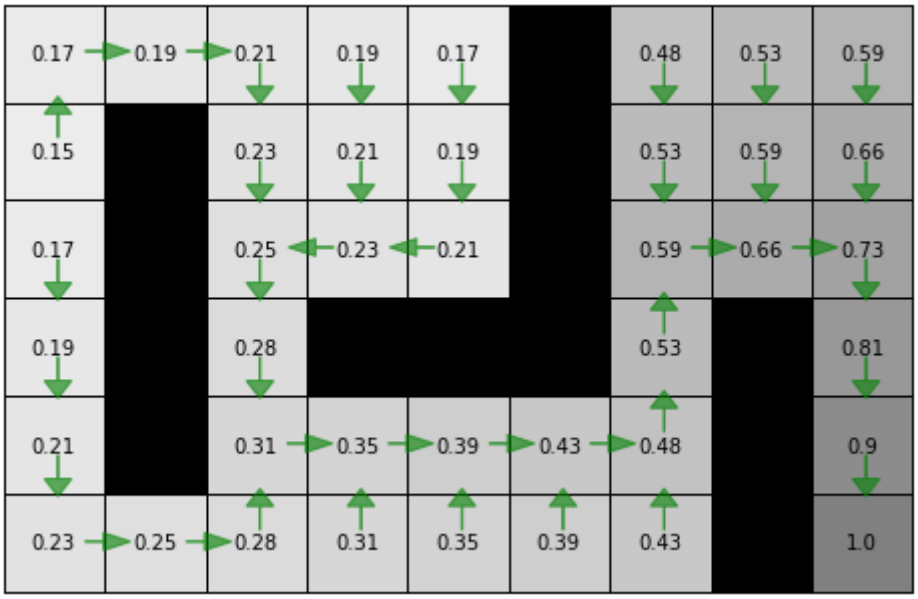
question 6

```
def evaluate_one_step_q(mdp, q, policy):
    # Outputs the state value function after one step of policy evaluation
    q_new = np.zeros((mdp.nb_states, mdp.action_space.size))  # initial action values
    for x in range(mdp.nb_states):  # for each state x
        # Compute the value of the state x for each action u of the MDP action space
        q_temp = []
        if x not in mdp.terminal_states:
            # Process sum of the values of the neighbouring states
            summ = 0
            for y in range(mdp.nb_states):
                summ = summ + mdp.P[x, :, y] * q[y,policy[y]]
            q_temp.append(mdp.r[x, :] + mdp.gamma * summ)
        else:  # if the state is final, then we only take the reward into account
            q_temp.append(mdp.r[x, :])

        # Select the highest state value among those computed
        q_new[x] = np.max(q_temp)
    return q_new


def evaluate_q(mdp, policy):
    # Outputs the state value function of a policy
    q = np.zeros((mdp.nb_states, mdp.action_space.size))  # initial action values are
    stop = False
    while not stop:
        qold = q.copy()
        q = evaluate_one_step_q(mdp, qold, policy)

        # Test if convergence has been reached
        if (np.linalg.norm(q - qold)) < 0.01:
            stop = True
    return q
```

question 7



question 8



question 9

We can see that the policy_iteration_q method is more efficient because it takes only 2 iterations and 0,72sec. For the other one, it takes 16 iterations and 2,75sec.

```
policy iteration Q
2
0.7204508781433105
policy iteration V
16
2.7518985271453857
```

Question 11 :

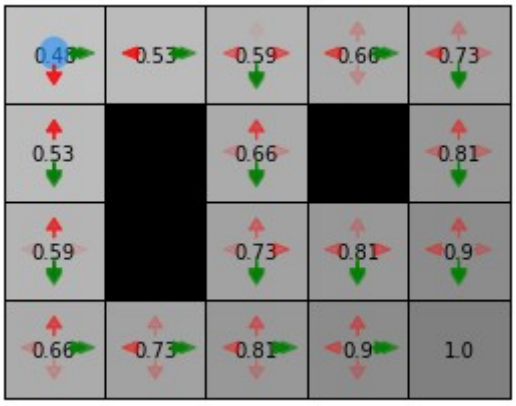
Here is the code for the temporal_difference and the result we obtain for 200 iterations :

```
# Update the state value of x
if x in mdp.terminal_states:
    v[x] = r
else:
    delta = r + mdp.gamma*v[y]-v[x]
    v[x] = v[x]+alpha*delta
```

Question 12 :

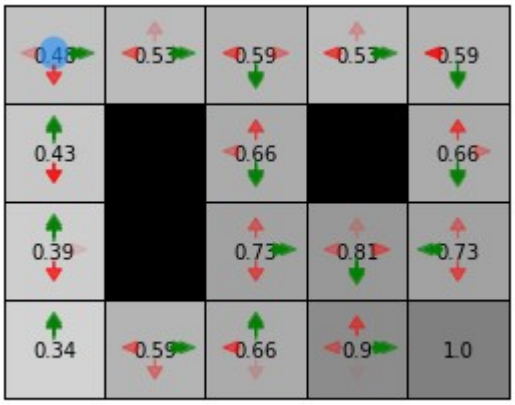
Here is the code for the q_learning function and the result we obtain for 200 iterations :

```
# Update the state-action value function with q-Learning
if x in mdp.terminal_states:
    q[x, u] = r
else:
    delta = r + mdp.gamma * np.max(q[y, :]) - q[x, u]
    q[x, u] = q[x, u] + alpha*delta
```



Question 13 :

Here is the result of the Q-learning-greedy for 200 iterations :

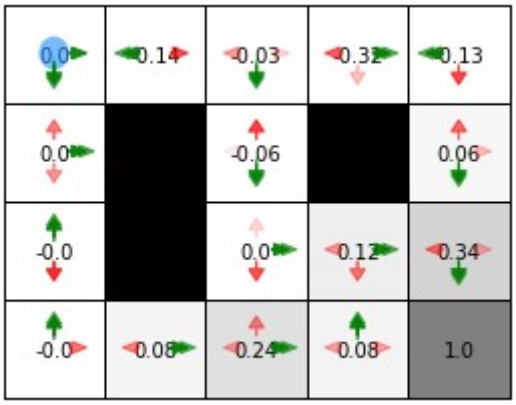


we can see that greedy is faster than softmax to converge,

Question 14 :

Here is the code for the sarsa function and the result we obtain for 200 iterations :

```
# Update the state-action value function with q-Learning
if x in mdp.terminal_states:
    q[x, u] = r
else:
    u1 = np.random.choice(np.where(q[x,:]==q[x,:].max())[0])
    delta =r + mdp.gamma*q[y, u1]-q[x, u]
    q[x,u] = q[x, u] + alpha*delta
```


0.0
0.14
0.03
0.32
0.13
0.0
0.06
0.06
0.0
0.0
0.12
0.34
0.0
0.08
0.24
0.08
1.0