Exercises

a) Implement the value iteration algorithm. Turn your code for the value iteration and plot the value function and policy for the model created by the `gridworld` script.
Value iteration algorithm is model-based method in Reinforcement learning that iteratively updates the state-value function V_π via:

$$V_{k+1}(s) = max_a \sum_{s'} P(s' \mid s, a)(r_{a,s} + \gamma V_k(s')) \tag{1}$$

where we maximize the current Q-value function $Q_k(s, a)$ over all possible actions. Convergence is ensured as long as the Bellman operator is a contraction map. Stopping occurs when the difference between updates in the L_1 norm stays below a threshold (`= 0.01`). Then, a compute greedy policy $\pi(s) = argmax_a Q(s, a)$ using the last update before stopping.

```
function [v, pi] = valueIteration(model, maxit)
v = zeros(model.stateCount, 1);    % initialize the value function
for i = 1:maxit,
   %pi = ones(model.stateCount, 1); no need to Initialize the policy
   v_ = zeros(model.stateCount, 1);    % initialize new value function
   Q_val = 0;             % Compute Q-value
   for s_ = 1:model.stateCount,  % s_ next state visited
   term = reshape(model.P(:,s_,:),model.stateCount,4).*(model.R(:,:) + model.gamma*v(s_));
   Q_val = Q_val + term;
   end
   v_ = max(Q_val,[],2);    % compute new value-function (1)
   stop_criterion = norm(v-v_,1);  % Stop-stopping criterion in L1-norm
   v = v_;           % Update value-function
   if stop_criterion < 0.01,  % exit early
      display('exit early');
      display(i);
      break;
   end
end
[v, pi] = max(Q_val,[],2);   % Compute greedy policy from last Q-value update
end
```

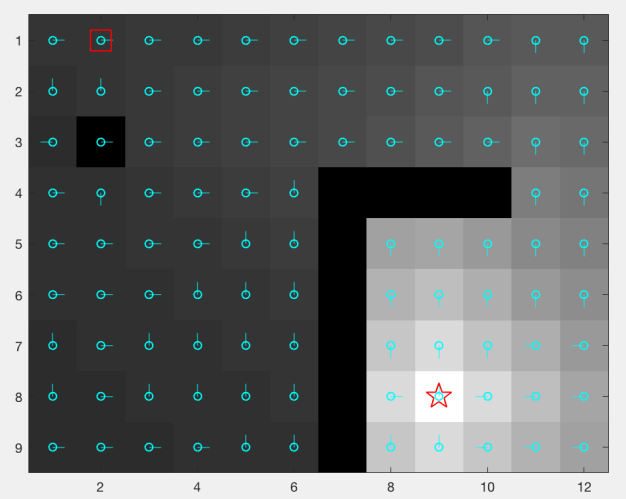


Figure 1: `gridworld plotVP(v, pi, paramSet)` exiting `valueIteration` after 47 iterations

b) Implement the policy iteration algorithm. Turn your code for this algorithm and a plot for the optimal value function and policy for the `gridworld` model.
Policy Iteration algorithm is a model-based method where **policy evaluation** and **policy improvement** are performed in each step. First, in policy evaluation, the state value function $V_\pi(s)$ of the current policy π is iteratively approximated using:

$$V_{k+1}(s) = \sum_a \pi_k(a,s) \sum_{s'} P(s' \mid s,a)(r_{a,s} + \gamma V_k(s')) \tag{2}$$

Since in `gridworld` we are dealing with deterministic policies, this reduces to:

$$V_{k+1}(s) = \sum_a \pi_k(a,s) Q_k(s,a) \tag{3}$$

$$= Q_k(s, \pi_k(s)) \tag{4}$$

Stopping again occurs when the difference between state-value function updates in the L_1 norm stays below a threshold (`= 0.01`) or the maximum number of iterations `maxit` is reached.

```
function v = policyEvaluation(model, pi, maxit)
v = zeros(model.stateCount, 1);
   for i = 1:maxit, % Evaluate policy pi iterative updating V_pi
   v_ = zeros(model.stateCount, 1); % initialize new value function
   Q_val = bellman_update(model, v);
      for s = 1:model.stateCount,
         v_(s) = Q_val(s,pi(s)); % VALUE FUNCTION UPDATES(4)
      end
   stop_criterion = norm(v-v_,1);
   v = v_;
      if stop_criterion < 0.01, %exit early
         break;
      end
   end
end
```

where, for convenience, we introduced a function that computes the Q-value $Q_\pi(s,a)$ from a given value function $V_\pi(s)$ and discount factor γ using Bellman's equation:

$$Q_\pi(s,a) = \mathbb{E}(R_{t+1} | s_t = s, a_t = a) = \sum_{s',r} P(s', r \mid s, a)(r + \gamma V_\pi(s')) \tag{5}$$

```
function value = bellman_update(model, val_func)
value = 0;
   for s_ = 1:model.stateCount,
      term = reshape(model.P(:,s_,:),model.stateCount,4).*(model.R(:,:)
            + model.gamma*val_func(s_));
      value = value + term;
   end
end
```

Policy Improvement Theorem 1 *Let π' and π be two deterministic policies such that for all states s, $Q_\pi(s, \pi'(s)) \geq V_\pi(s)$. Then, policy π' must be as good as, or better than, π. That is, for all states s, $V_{\pi'}(s) \geq V_\pi(s)$.*

Hence, in policy improvement, the current value function $V_\pi(s)$ is used to (greedily) update the current policy:

$$\pi'(s) = argmax_a \sum_{s'} P(s' \mid s, a)(r_{a,s} + \gamma V_\pi(s')) \tag{6}$$

$$= argmax_a Q_\pi(s, a) \tag{7}$$

We implement into the full Policy Iteration algorithm via:

```
function [v, pi] = policyIteration(model, maxit)
pi = ones(model.stateCount, 1); % initialize policy pi
%(value-function v initialised inside policyEvaluation)
policy_stable = false;
   while policy_stable == false,  % Repeat until stable policy reached
   v = policyEvaluation(model, pi, maxit);
   Q_val_ = bellman_update(model, v); % POLICY (GREEDY) IMPROVEMENT (7)
   [temp, pi_] = max(Q_val_,[],2);
      if pi == pi_,       % exit while loop when no changes in policy observed
         policy_stable = true;
      end
   pi = pi_;
   end
v = policyEvaluation(model, pi, maxit); % Compute Value-function for last policy update
end
```

exiting the while loop when no changes in policy are observed.

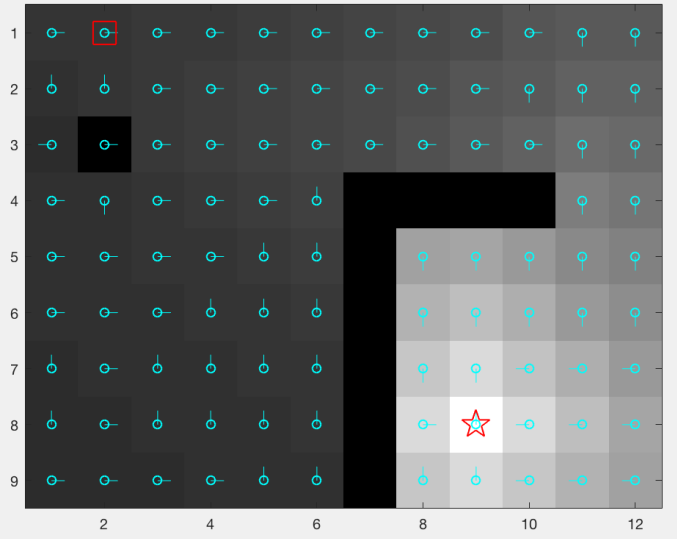


Figure 2: `gridworld` after exiting `policyIteration` after 6 policy updates

Although `policyIteration` reaches convergence to optimal policy after only 6 policy-updates, policy evaluation is performed for each update, which can take up to 70 (approx.) iterations converge.

c) Implement the SARSA algorithm as a function `[v, pi] = sarsa (model, maxit, maxeps)` which takes the model, a maximum number of iterations per episode, and maximum number of iterations per episode. Turn your code for this algorithm and a plot for the value function `smallworld` model.

```
function [v, pi] = sarsa(model, maxit, maxeps)
Q = zeros(model.stateCount, 4); % initialize the Q-value function
%rewards = []; LATER
alpha = 0.3;
for i = 1:maxeps, % FOR EACH EPISODE
s = model.startState;  % Start at startState and tot_reward = 0;
[~,pi] = max(Q,[],2);  % Choose action a from Q (eps-greedy)
a = greedy(pi(s),0.4);
   for j = 1:maxit, % FOR EACH STEP
   % compute acummulated reward: tot_reward = tot_reward + model.R(s,a);
   % Sample s_ from p(s_|s,a)
   p = 0; r = rand;
      for s_ = 1:model.stateCount,
      p = p + model.P(s, s_, a);
         if r <= p,
            break;
         end
      end
   [~,pi_] = max(Q,[],2);  % Take action a_ from s_ using policy pi_ derived Q (eps-greedy)
   a_ = greedy(pi_(s_),0.4);
   % Update Q-value for current state and action
   Q(s,a) = Q(s,a) + alpha*(model.R(s,a) + model.gamma*Q(s_, a_) - Q(s, a));
   % Update new state and action
   s = s_; a = a_;
         if s == model.goalState,
            Q(s,:) = 0;
            % tot_reward = tot_reward + model.R(s,a);
            break
         end
   end
% rewards = [rewards, tot_reward];
end
[v, pi] = max(Q,[],2); % Compute value function and greedy policy from Q
end
```

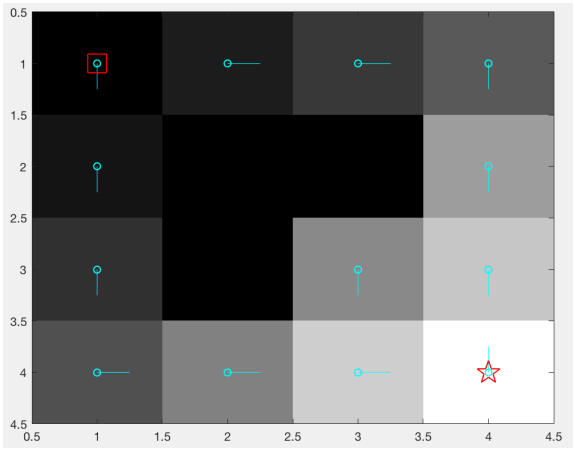


Figure 3: `smallworld` using `sarsa(model,15,1500)` with `alpha = 0.1`, `eps = 0.4`

d) Implement the Q-learning algorithm as a function `[v, pi] = Qlearning (model, maxit, maxeps)` which takes the model, a maximum number of iterations per episode, and maximum number of iterations per episode. Turn your code for this algorithm and a plot for the value function `smallworld` model.

```
function [v, pi] = qLearning(model, maxit, maxeps)
Q = zeros(model.stateCount, 4); % initialize the Q-value function
%rewards = []; LATER
alpha = 0.3;
   for i = 1:maxeps, % FOR EACH EPISODE
   s = model.startState; % tot_reward = 0; start epsiode at startState
      for j = 1:maxit, % FOR EACH STEP
      [~,pi] = max(Q,[],2);  % Choose action a from s using policy derived from Q (eps-greedy)
      a = greedy(pi(s),0.4);
      % compute acummulated reward: tot_reward = tot_reward + model.R(s,a);
      % Sample s_ from p(s_|s,a)
      p = 0; r = rand;
         for s_ = 1:model.stateCount,
         p = p + model.P(s, s_, a);
            if r <= p,
               break;
            end
         end
      % Update Q-value for current state and action using Q-learning off-policy update
      Q_max = max(Q,[],2);
      Q(s,a) = Q(s,a) + alpha*(model.R(s,a) + model.gamma*Q_max(s_) - Q(s, a));
      s = s_; % Update new state
         if s == model.goalState, %Stop episode when goal state reached
            Q(s,:) = 0;  %Set value of Goal-state to be zero
            %Add Goal state reward to accumulated reward: tot_reward = tot_reward + model.R(s,1);
            break
         end
      end
   %rewards = [rewards, tot_reward];
   end
[v, pi] = max(Q,[],2); % Evaluate value function and policy from Q
end
```

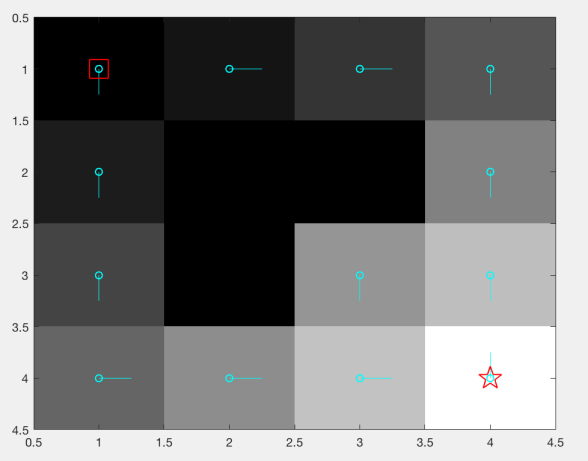


Figure 4: `smallworld` using `qLearning(model,15,1000)` with `alpha = 0.1`, `eps = 0.4`

For `sarsa` and `qLearning` scripts above, we introduce `greedy(pi,eps)` to implement ϵ-greedy policy selection:

```
function a_ = greedy(pi,eps)
r = rand;
   if r > eps,
      a_ = pi;
   else
      a_ = randi([1,4]);
   end
end
```

Where we choose policy $\pi(s) = argmax_a Q(a, s)$ with probability $1-\epsilon$ and sample from a uniform distribution $U(up, down, right, left)$ with probability ϵ.
Both SARSA and Q-learning below to the family of Temporal Difference (TD) learning methods, where estimates of Q-value function are learnt from experience based on previous estimates through interaction with environment (without competition of episodial task being required for these updates).
SARSA is an on-policy method, meaning it uses the current estimate of the optimal policy to generate behavior while learning.
Q-learning is an off-policy method, where we distinguish between the target policy and the policy generating the behavior. This directly converges to the optimal policy.

e) Modify SARSA and and Q-learning algorithms to save the accumulated reward per episode. Run these algorithms using the `cliffworld` model and plot the rewards obtained throughout the learning process.

In `cliffworld` model, rewards are -1 on all states (indep. of action) except for those in the "cliff" (`paramSet.badSet`) with reward -6 and goal state with reward 10. Everytime agent steps into a cliff state transition tensor `model.P(s,s_,a)` sends it to start state with probability 1. For this exercise, we introduced variable `tot_reward` to keep track of accumulated rewards in an episode and history array `rewards` of total rewards after each episode commented out in scripts for exercises (c) and (d).

In order to observe the on-line performance of SARSA versus Q-learning in `cliffworld`, since the accumulated reward signal is quite noisy, we apply a filter that averages over a window of 50 iterations.

```
window_size = 50;
coef = ones(1, window_size)/window_size;
avg_rewards = filter(coef, 1, rewards);
```

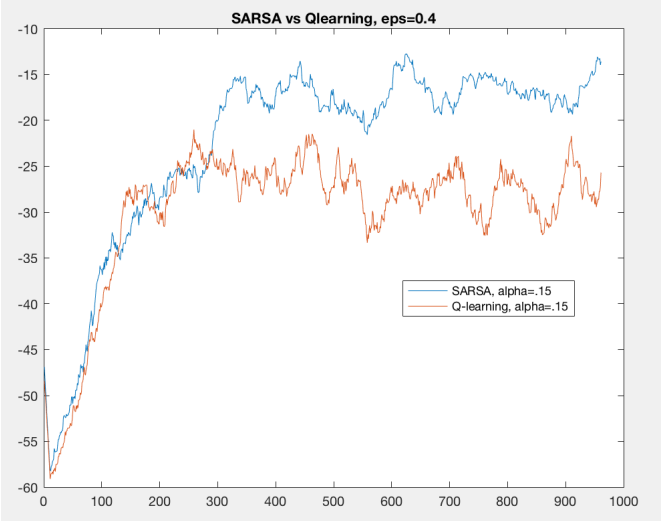
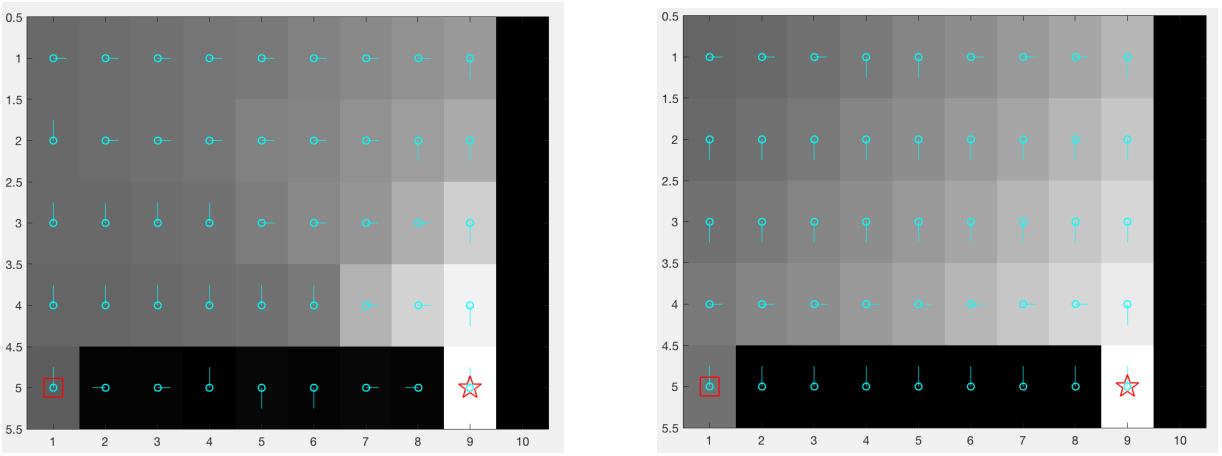


Figure 5: Accumulated reward/ep `sarsa` vs `qLearning` in `cliffworld` with `maxit=50`, `alpha=.15` and `eps=.4`

Figure above highlights the difference between SARSA (on-policy) and Q-learning (off-policy) methods in `cliffworld`. While Q-learning directly approximates the optimal action/state value function Q^*, resulting in learning the values of the optimal policy, it has lower on-line performance than SARSA, which learns the roundabout policy. For figure 6, we set `eps = 0.6` , `maxep = 10000` and `maxit= 100` to ensure more exploration of the state space while increasing the uncertainty in action selection. This exaggerates the effect observed in figure 5. We see that while Q-learning learns optimal policy choosing paths that go along the cliff causing it to occasionally fall into the cliff, SARSA for which the behavior and target policies are the same chooses safer (and longer) paths through the upper-side of the grid.



(a) SARSA (b) Q-Learning

Figure 6: State-value function and policy for `cliffworld` of `sarsa` and `qLearning` in 10000 ep.

For the graphs in figure 5, `eps = 0.4` to allow the agent to explore state space through ϵ-greedy policies. However, when `eps` is reduced to small values `0.01` almost no difference can be observed between both methods:

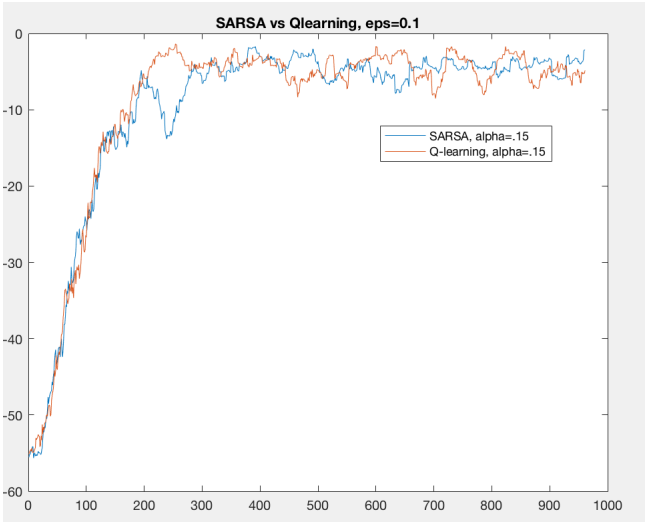


Figure 7: Accumulated reward/ep `sarsa` vs `qLearning` in `cliffworld` with `eps=.1`