1. My CID is 01060785, hence my personalised p and γ are $p = 0.90$ and $\gamma = 0.45$. We have verified `PersonalisedGridWorld.p` by comparing its outputs (sets of states and actions and matrices T/R) against that obtained by our custom function `MDPGridWorld.m`. The output were identical, which increases our belief that `PersonalisedGridWorld.p` is correct.

2. We assume the MDP is operating under an unbiased policy π^u define for each state s by the vector $\pi^u_s = [0.25\ 0.25\ 0.25\ 0.25]$, $s = 1 : S$, where $S = 14$ is the total number of states in the MDP. This vector offers equiprobable action selections and satisfies the boundary condition that its elements sum to one. We then apply an iterative policy evaluation algorithm to the MDP, using the relevant outputs of `PersonalisedGridWorld.p` (matrices T/R and list of absorbing states) as inputs to a function `PolicyIteration.m`, where we systematically apply Bellman's equation (convergence is guaranteed), until the maximum error between the elements of the value vector computed on two successive algorithm iterations is smaller than a parameter `tolerance`. At each iteration, for each state s we operate a full backup through all successor states by sweeping through every action from s and weighting each term according to the policy. A term is the product of the transition probability to a successor state given an action by the sum of the expected immediate reward and discounted future reward.

State	s_1	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}
Value	-1.2161	-4.5530	-1.7111	-1.5915	-4.1776	-2.4763	-1.8023	-1.9158	-1.8158	-1.8181	-1.8202	-1.8326

Table 1: **Value function for the Grid World MDP using Iterative Policy Evaluation**

3a. From the Markov property, the future is independent of the past given the present, hence the state occupied at step $i+1$ only depends on the state occupied on step i. Therefore, each transition occurring between steps i and $i+1$ is independent from the previous transition. As a result, the sequence of states generated under a given policy π can be decomposed into a sequence of independent transitions, for which the probability is given by the sum over all actions of the product of the probability $p_\pi(a|s(i))$ of choosing an action a, given by the policy π, by the probability $p(s(i+1)|a(i), s(i))$ of ending up in a specific state given the action taken and the currently occupied state, given by the transition matrix T. From the independence property, the likelihood of obtaining a given sequence, given the policy, is the product of all the transition probabilities of jumping from one state in the sequence to the next one, multiplied by the probability of starting in the corrects state, which is 0.25.

Sequence	s_{14}, s_{10}, s_8, s_4, s_3	s_{11}, s_9, s_5, s_6, s_6, s_2	s_{12}, s_{11}, s_{11}, s_9, s_5, s_9, s_5, s_1, s_2
Likelihood	$3.906\,25 \times 10^{-3}$	$9.765\,625 \times 10^{-4}$	$3.051\,757\,812\,5 \times 10^{-5}$

Table 2: **Likelihood of observing Sequences under an Unbiased policy**

3b. To define a policy π^M which has higher likelihood than π^u to have generated the sequences observed we consider, for each state appearing in any sequence, the successor states observed from that state, and determine the action most likely to have conducted to this set of successor state. If we consider the state `S5`, it has transitioned **once** to successor state `S6` (sequence 2), **once** to `S9` (sequence 3) and **once** to `S1` (sequence 3). The most likely (in fact, the only possible) action for these transitions is a step to the East, since this results in a transition to `S6` with probability p, or a transition to `S1` or `S9` with a probability $\frac{1-p}{2}$ each.

State	s_1	s_4	s_5	s_6	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{14}
Action	2	4	2	4	1	1	1	4	4	1

Table 3: **Deterministic Policy offering higher-than-unbiased likelihood for Observed Sequences**

Assuming independent sequence generation, the probability of obtaining all 3 sequences is the product of the probabilities of observing each sequence separately. For π^u, $p_{\pi^u} \approx 1.82 \times 10^{-12}$ and for the biased policy, $p_{\pi^b} \approx 8.09 \times 10^{-11}$, making it 44 times more likely to observe the sequences under the biased policy with respect to the unbiased one.

4a. We have assumed a policy π^u and have generated 10 traces of the MDP, shown below. We have selected an initial starting state in a random unbiased way between possible starting states `S11`, `S12`, `S13` and `S14`. For each iteration, the agent randomly selects an action, where the probability of selecting each action is $p_\pi(a|s(i))$ in accordance with the policy π^u. Given the action, the agent transitions randomly to a state according to probabilities $p(s(i+1)|a(i), s(i))$, given by T. For each transition, the action and reward taken were recorded.

1. S13,S,-1,S13,W,-1,S12,N,-1,S12,W,-1,S11,N,-1,S9,N,-1,S5,S,-1,S9,E,-1,S5,E,-1,S1,W,-1,S1,W,-1,S1,W,-1,S1,E,0

2. S13,S,-1,S13,N,-1,S13,E,-1,S14,S,-1,S14,W,-1,S13,S,-1,S13,N,-1,S13,N,-1,S13,N,-1,S13,S,-1,S13,E,-1,S14,S,-1,S14,E,-1,S14,W,-1,S13,S,-1,S12,S,-1,S12,N,-1,S13,N,-1,S13,S,-1,S13,E,-1,S14,S,-1,S14,W,-1,S13,S,-1,S13,W,-1,S12,W,-1,S12,S,-1,S12,E,-1,S13,S,-1,S13,S,-1,S13,E,-1,S14,W,-1,S13,E,-1,S14,N,-1,S10,S,-1,S14,W,-1,S13,S,-1,S13,S,-1,S13,N,-1,S13,W,-1,S12,W,-1,S11,S,-1,S11,E,-1,S12,E,-1,S13,N,-1,S13,E,-1,S14,E,-1,S14,S,-1,S14,S,-1,S14,N,-1,S10,N,-1,S8,N,-1,S4,N,-1,S4,E,-1,S4,E,-1,S4,S,-1,S8,N,-1,S4,W,-10

3. S14,E,-1,S14,S,-1,S14,S,-1,S13,N,-1,S13,E,-1,S13,S,-1,S13,E,-1,S14,S,-1,S14,N,-1,S10,E,-1,S10,W,-1,S10,W,-1,S8,S,-1,S10,S,-1,S10,S,-1,S14,S,-1,S14,E,-1,S14,N,-1,S10,N,-1,S8,S,-1,S10,S,-1,S14,S,-1,S14,N,-1,S10,S,-1,S10,W,-1,S8,W,-1,S7,S,-1,S7,S,-1,S7,W,-1,S6,E,-1,S7,W,-1,S6,N,0

4. S12,E,-1,S12,N,-1,S12,N,-1,S12,N,-1,S12,S,-1,S12,E,-1,S13,E,-1,S13,W,-1,S12,W,-1,S11,E,-1,S12,N,-1,S13,S,-1,S13,S,-1,S13,E,-1,S14,E,-1,S14,W,-1,S13,S,-1,S13,W,-1,S12,S,-1,S11,W,-1,S11,E,-1,S12,N,-1,S12,N,-1,S12,S,-1,S12,W,-1,S11,S,-1,S11,W,-1,S11,S,-1,S11,N,-1,S9,E,-1,S9,N,-1,S9,N,-1,S5,W,-1,S5,N,-1,S6,S,-1,S6,S,-1,S6,N,0

5. S14,W,-1,S13,S,-1,S13,N,-1,S13,W,-1,S12,N,-1,S12,N,-1,S12,N,-1,S12,N,-1,S12,S,-1,S12,S,-1,S12,E,-1,S13,W,-1,S13,W,-1,S12,N,-1,S12,W,-1,S11,E,-1,S12,W,-1,S12,N,-1,S12,W,-1,S12,W,-1,S11,S,-1,S11,S,-1,S11,E,-1,S12,N,-1,S12,N,-1,S12,W,-1,S12,N,-1,S12,W,-1,S11,W,-1,S11,N,-1,S11,W,-1,S11,W,-1,S11,N,-1,S9,N,-1,S5,N,-1,S1,S,-1,S5,W,-1,S1,W,-1,S1,S,-1,S5,N,-1,S1,W,-1,S1,S,-1,S5,S,-1,S9,E,-1,S9,N,-1,S9,W,-1,S9,E,-1,S9,W,-1,S9,W,-1,S9,N,-1,S5,N,-1,S5,S,-1,S9,S,-1,S11,W,-1,S11,W,-1,S11,W,-1,S11,E,-1,S12,N,-1,S12,W,-1,S11,N,-1,S9,N,-1,S9,E,-1,S9,E,-1,S9,E,-1,S9,W,-1,S9,N,-1,S5,S,-1,S9,W,-1,S9,W,-1,S9,W,-1,S9,S,-1,S11,N,-1,S9,E,-1,S9,E,-1,S9,W,-1,S9,N,-1,S5,W,-1,S5,E,-1,S6,E,-1,S7,W,-1,S6,W,-1,S5,S,-1,S6,E,-1,S7,S,-1,S7,S,-1,S7,N,-10

6. S12,S,-1,S12,S,-1,S12,N,-1,S12,E,-1,S13,E,-1,S14,W,-1,S13,W,-1,S12,E,-1,S13,N,-1,S13,E,-1,S14,W,-1,S13,E,-1,S14,W,-1,S13,W,-1,S12,N,-1,S11,N,-1,S9,S,-1,S9,S,-1,S9,S,-1,S11,E,-1,S12,W,-1,S11,S,-1,S11,W,-1,S11,S,-1,S11,W,-1,S11,N,-1,S9,W,-1,S9,S,-1,S11,S,-1,S11,W,-1,S11,E,-1,S12,N,-1,S12,W,-1,S11,E,-1,S12,E,-1,S13,W,-1,S12,E,-1,S12,W,-1,S12,S,-1,S12,E,-1,S13,S,-1,S13,E,-1,S13,N,-1,S13,S,-1,S13,N,-1,S13,N,-1,S13,W,-1,S12,N,-1,S12,W,-1,S11,N,-1,S9,S,-1,S11,E,-1,S12,W,-1,S11,E,-1,S12,S,-1,S12,E,-1,S13,E,-1,S14,W,-1,S10,W,-1,S10,E,-1,S10,W,-1,S10,S,-1,S14,S,-1,S14,S,-1,S14,W,-1,S13,N,-1,S13,W,-1,S12,N,-1,S12,W,-1,S11,N,-1,S9,S,-1,S11,W,-1,S11,W,-1,S11,S,-1,S11,S,-1,S11,N,-1,S9,E,-1,S9,N,-1,S5,W,-1,S5,E,-1,S6,N,0

7. S13,N,-1,S13,S,-1,S13,S,-1,S13,E,-1,S13,E,-1,S14,W,-1,S13,N,-1,S13,N,-1,S13,W,-1,S12,W,-1,S11,W,-1,S11,N,-1,S9,E,-1,S9,E,-1,S9,S,-1,S11,E,-1,S11,E,-1,S12,S,-1,S12,S,-1,S12,N,-1,S12,E,-1,S13,W,-1,S12,N,-1,S12,S,-1,S12,S,-1,S12,W,-1,S11,W,-1,S11,E,-1,S12,W,-1,S11,N,-1,S9,E,-1,S9,S,-1,S11,N,-1,S9,E,-1,S9,E,-1,S9,W,-1,S11,W,-1,S11,E,-1,S12,E,-1,S13,W,-1,S13,N,-1,S13,E,-1,S14,W,-1,S13,E,-1,S14,E,-1,S14,S,-1,S14,S,-1,S14,W,-1,S13,W,-1,S13,W,-1,S13,W,-1,S12,E,-1,S13,W,-1,S12,N,-1,S12,S,-1,S12,E,-1,S13,W,-1,S13,S,-1,S13,S,-1,S13,N,-1,S13,W,-1,S12,W,-1,S12,S,-1,S12,W,-1,S11,E,-1,S12,W,-1,S11,W,-1,S11,E,-1,S12,N,-1,S12,W,-1,S11,W,-1,S11,W,-1,S11,N,-1,S9,W,-1,S9,W,-1,S9,E,-1,S9,E,-1,S9,E,-1,S9,N,-1,S5,E,-1,S6,W,-1,S5,W,-1,S5,E,-1,S6,W,-1,S5,E,-1,S6,S,-1,S6,S,-1,S6,S,-1,S6,W,-1,S5,S,-1,S9,E,-1,S9,W,-1,S9,E,-1,S9,W,-1,S9,W,-1,S9,N,-1,S5,N,-1,S1,S,-1,S5,N,-1,S1,S,-1,S5,N,-1,S1,E,0

8. S13,W,-1,S12,W,-1,S11,S,-1,S11,S,-1,S11,N,-1,S11,W,-1,S9,S,-1,S11,W,-1,S11,S,-1,S11,W,-1,S11,N,-1,S9,W,-1,S9,E,-1,S9,W,-1,S9,E,-1,S9,S,-1,S11,S,-1,S11,S,-1,S11,S,-1,S11,N,-1,S9,E,-1,S9,W,-1,S5,W,-1,S5,N,-1,S1,N,-1,S1,N,-1,S1,W,-1,S1,W,-1,S1,E,0

9. S11,W,-1,S11,S,-1,S11,W,-1,S11,S,-1,S11,W,-1,S11,W,-1,S11,E,-1,S12,N,-1,S12,S,-1,S12,S,-1,S12,S,-1,S12,S,-1,S12,S,-1,S12,N,-1,S12,N,-1,S12,W,-1,S11,S,-1,S11,N,-1,S9,S,-1,S9,E,-1,S9,W,-1,S9,W,-1,S9,E,-1,S9,S,-1,S11,N,-1,S9,N,-1,S5,E,-1,S6,E,-1,S6,S,-1,S6,W,-1,S5,S,-1,S9,W,-1,S9,E,-1,S9,S,-1,S11,S,-1,S11,S,-1,S11,N,-1,S9,E,-1,S9,S,-1,S11,E,-1,S12,W,-1,S11,S,-1,S11,W,-1,S11,N,-1,S9,N,-1,S5,W,-1,S5,E,-1,S6,E,-1,S7,S,-1,S7,N,-10

10. S11,E,-1,S12,N,-1,S12,N,-1,S12,N,-1,S12,N,-1,S12,W,-1,S11,E,-1,S11,N,-1,S9,W,-1,S9,S,-1,S11,W,-1,S11,N,-1,S9,E,-1,S11,E,-1,S12,S,-1,S12,N,-1,S11,W,-1,S11,N,-1,S9,N,-1,S5,N,-1,S1,E,0

4b. We apply First-Visit Batch Monte-Carlo Policy Evaluation (FVMCPE) using the 10 traces shown above to obtain an estimate of the value function. We accumulate, for each trace and for each state, the sum of discounted future rewards obtained from the first visit to that state until termination at an absorbing state. For each state, we then average the return obtained over all sequences generated using two methods: either we consider the return obtained for a non-visited state in a sequence as having a value of zero (method M1), or we ignore traces where the state was not visited (method M2).

State	s_1	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}
Value (M1)	-0.6958	-0.1969	-1.3254	-0.8385	-0.9171	-0.3704	-1.4532	-0.5455	-1.6361	-1.6363	-1.4545	-1.0909
Value (M2)	-1.7395	-1.9692	-1.6568	-1.6770	-3.0569	-1.8522	-1.8165	-1.8182	-1.8178	-1.8181	-1.8182	-1.8182

Table 4: **Value function for the Grid World MDP using FVMCPE, under two different methods**

4c. We use the cosine similarity (normalized dot product) to compare value function vectors obtained using FVMCPE (labeled MC) and Iterative Policy Evaluation (labeled I). This measure is defined as $err_C = \frac{V_{MC} \cdot V_I}{||V_{MC}|| \times ||V_I||}$, and gives information about the alignment of the two vectors in spatial domain, irrespective of their magnitude, and converges to 1 for perfectly aligned vectors. Since this measure ignores scaling, it is particularly relevant to value functions comparison since the value of each state is interpreted relative to the value of other states.

Indeed, if we scaled our set of rewards, the resulting value function would consequently also be scaled, however the intrinsic dynamics of the MDP would remain unchanged. As an example, the optimal policy for that system would be the same, since the optimal state/action function which maximizes return (relative to others) would be left unchanged. Although mean absolute percentage error (MAPE), defined as $err_E = 100 \times \sum_{i=1}^{S} |\frac{V_I(i) - V_{MC}(i)}{V_I(i)}|$, could also have been used for the same semi-scale-free property, literature suggests that it is biased since it penalizes negative errors more heavily than positive errors.

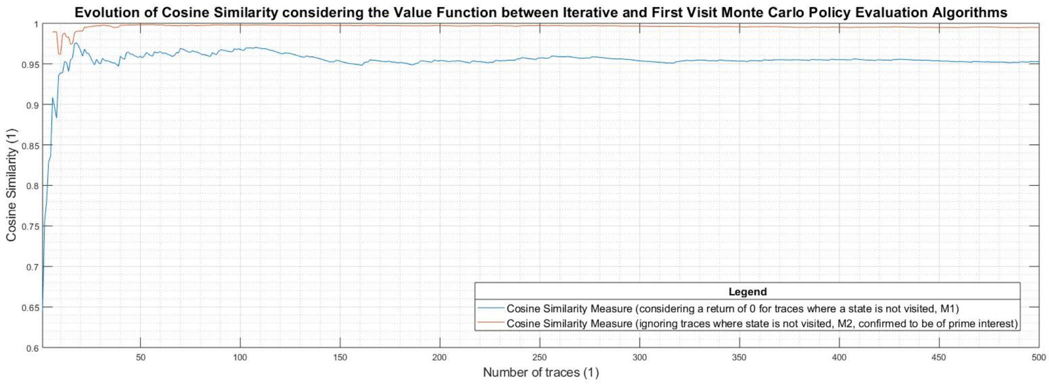

Figure 1: **Evolution of cosine similarity between model-based and model-free value vector as a function of number of traces considered**

Increasing the number of traces causes the value function vectors to align as the cosine similarity measure converges towards 0.995 over the scale of $10-50$ traces. This observation highlights the power of FVMCPE, as a few collected traces suffice for a model-free system to reliably estimate the value function. Results indicate that convergence varies significantly when computing V with method M1 or M2 (it has been confirmed that M2 should be used, it indeed offers stronger similarity when converged). We hypothesize that an every-visit algorithm should converge over less traces, since each trace would offer several samples for the return associated with each state, offering a larger data-set for mean computation.

5. We have evaluated ϵ-greedy first-visit Monte-Carlo learning and control, using total return and trace length per episode against episodes as learning curves (including mean and standard deviation). 100 learning experiments were simulated from an initialization of the policy to π^u. The evolution of the policy and Q were taken as measures of control. The algorithm starts by generating a trace using an initial arbitrary policy and an initial arbitrary Q, and records the return collected after the first visit to each state/action pair, appended to a list of returns collected over previous traces. Q is the average of the returns for each action/state pair over previous traces (empirical mean return). An optimal action is selected which maximizes Q for a given state, and is assigned a probability of $1 - \epsilon + \frac{epsilon}{|A|}$ and the others a probability of $\frac{epsilon}{|A|}$. A new trace is then generated, yielding new returns, a new estimate for Q, and a new policy, until convergence to the optimal policy (guaranteed). The evolution of Q reveals that state/action pairs which tend to lead to the goal state increase in value, and conversely for state/action pairs which tend to lead to the penalty state. The policy evolves accordingly - as expected, the converged policy is almost deterministic for $\epsilon = 0.1$ while the distribution of probabilities is more uniform for $\epsilon = 0.75$, reflecting that a high value of ϵ favours exploration. Consequently, for $\epsilon = 0.75$ the learning curves converge to have a **lower performance** (longer traces, smaller returns) than for $\epsilon = 0.1$, since the latter ultimately values performance over exploration (**asymptotic behaviour**).

Trace length In early episodes, for $\epsilon = 0.1$, the effect of ϵ-greedy learning seems counter-intuitive: trace lengths per episode start by increasing. We propose here a possible explanation. In early episodes, the estimation of Q relies upon a small data-set of returns and is hence less

accurate, which in turn may cause the agent to select a sub-optimal policy. This policy may lead the agent to become trapped in a **loop of states**. This issue would be enforced by two factors: (1) for a small ϵ, the probability of randomly breaking free from this policy-induced loop is low; and (2) if the agent eventually breaks free from the loop, in the context of a first-visit algorithm and for a small γ, the agent would give a small weight to the reward obtained after breaking free from the loop. Indeed, after n runs through a loop of m states, the reward obtained from the first state reached after escape from the loop would be weighted by γ^{mn}. Due to those loops, the initial standard deviation for $\epsilon = 0.1$ is several orders of magnitude greater than for $\epsilon = 0.75$, reflecting the occurrence of abnormally long traces in the former case. In addition, while trace lengths for $\epsilon = 0.75$ converge around 30 over 100 episodes, for $\epsilon = 0.1$ it takes approximately 200 episodes for trace lengths to converge to 20, revealing a **trade-off** between the short-term benefit of a high ϵ (faster convergence, smaller initial variance) against the long-term benefits of a low ϵ (shorter traces and smaller variance on the long term).
The issue of state loops could potentially be solved by implementing every-visit control: the return associated with one loop state would be the average of the return obtained from the first passage to this state (artificially low, since throughout the loop the agent accumulates the -1 move cost) **and** the return obtained from subsequent visits to that loop state. During visits to the loop state occurring shortly before escape from the loop, the return associated with that state/action pair would become more representative of the value of this state. Indeed, after escaping the loop the agent would be given a chance to reach a terminal state, therefore the trace length after escape would be shorter when compared to the abnormally long trace length occurring within the loop, which in turn would result in a higher return and eventually make the state/action pair of escape more attractive, breaking the policy loop for the next episode. In addition, implementing a running mean for Q with a forgetting rate α would accelerate the process of diluting the effect of the loop over time.

Return Since we have a low $\gamma = 0.45$, return depends mostly on the first rewards collected (heavy discount of future rewards). Since for a high ϵ, the exploration-prone agent is unlikely to follow a direct trajectory from start to the goal state `S2` (even when the optimal policy has been learned), return for $\epsilon = 0.75$ remains constant around -1.82 throughout learning. In comparison, the more focused $\epsilon = 0.1$ agent becomes more likely to follow a direct trace to the goal state as it learns an optimal policy (less exploration, more likely to follow optimal policy leading to `S2`), hence returns converge to a greater value of -1.79 over the scale of 50 episodes.

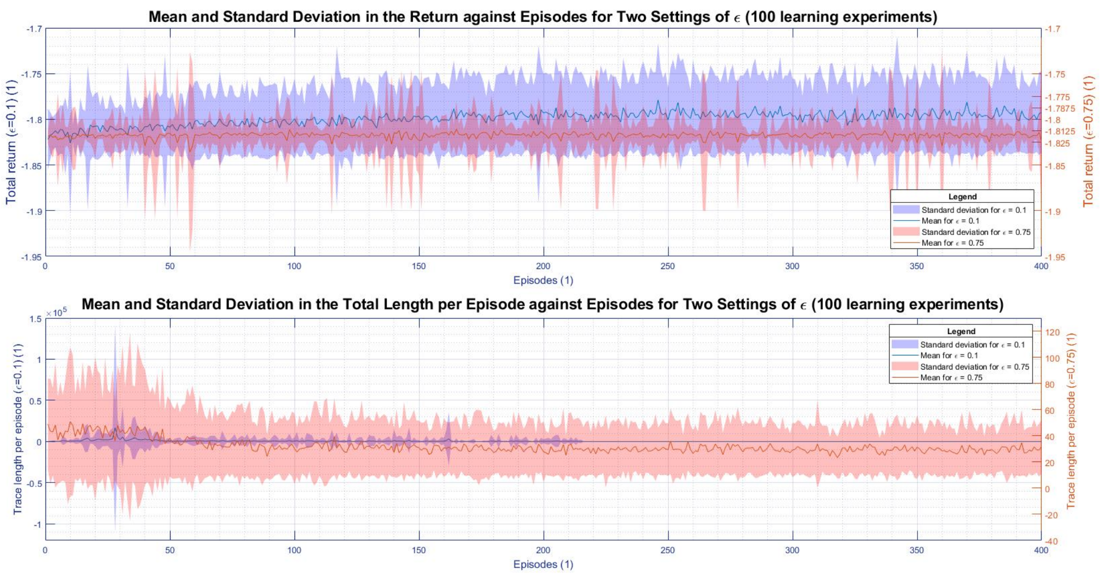

Figure 2: **Learning Curves computed for** $\epsilon = 0.1$ **and** $\epsilon = 0.75$ **over 50 Learning Experiments for the Grid World MDP, showing collected reward instead of return**

1 Appendix

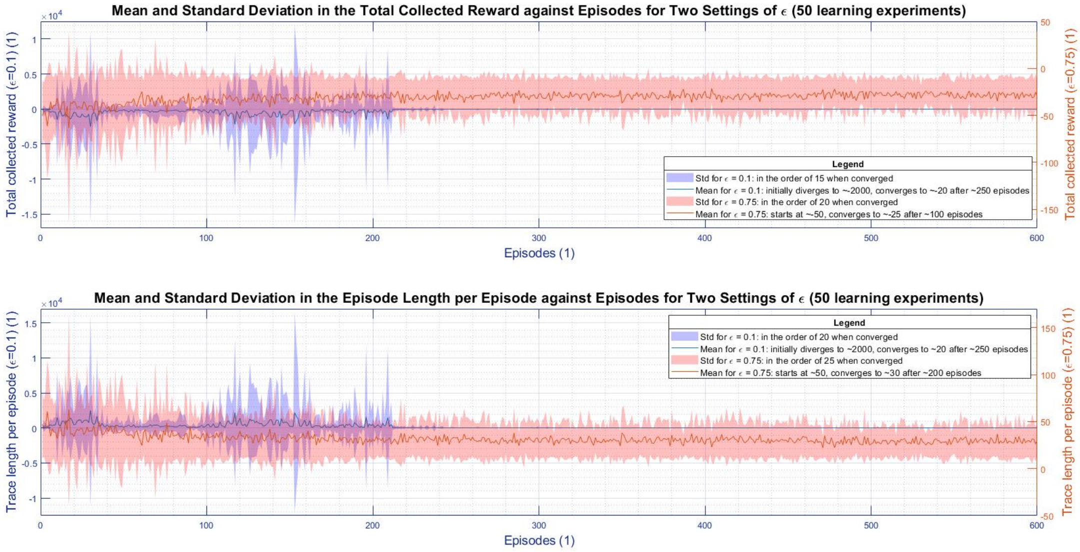

Figure 3: **Learning Curves computed for $\epsilon = 0.1$ and $\epsilon = 0.75$ over 50 Learning Experiments for the Grid World MDP, showing collected reward instead of return**

Please find the fully commented Matlab code on the next pages.

```
%%% Machine Learning and Neural Computation > Coursework 1: MDP grid World
%%% Bastien Caba, CID: 01060785 > Department of Bioengineering, Imperial College↙
London

function [answers] = RunCoursework()
%% Q1: Definition of parameters p and gamma
%Clearing the workspace, command windows, and closing all figures
clear all; close all; clc;

%Definition of parameters p and gamma
fprintf('QUESTION 1\n');
p = 0.9; gamma = 0.45;
fprintf('We define the parameters p = %.2f, gamma = %.2f.\n\n', p, gamma);

%%%%%%%%%% Verification of PersonalisedGridWorld.p %%%%%%%%%%

%Declare the Grid World MDP using our custom function
[S_num_veri, S_names_veri, A_num_veri, A_names_veri, S_absorbing_veri, T_veri, R_veri]↙
= MDPGridWorld(p);

%Declare the Gird World MDP using PersonalisedGridWorld.p
[S_num, A_num, T, R, S_names, A_names, S_absorbing] = PersonalisedGridWorld(p);

%Comparison of the outputs of PersonalisedGridWorld.p with those of our custom↙
function
%Verification is a boolean indicating if the outputs are identical
verification = MDP_Verification(S_num, S_num_veri, A_num, A_num_veri, T, T_veri, R,↙
R_veri, S_absorbing, S_absorbing_veri);
msg = 'Warning: verification of PersonalisedGridWorld.p has failed.';

if(verification == 0)    %If the outputs are different, then:
    error(msg);          %Display error message
else                     %Else, if verification succeeds:
    var_veri = {'S_num_veri', 'S_names_veri', 'A_num_veri', 'A_names_veri',↙
'S_absorbing_veri', 'T_veri', 'R_veri'};
    clear(var_veri{:}); %Clear workspace of verification variables
end

%Object containing all Grid World Parameters
GridWorld = struct('S', S_num, 'A', A_num, 'T', T, 'R', R, 'S_names', S_names,↙
'A_names', A_names, 'S_absorbing', S_absorbing, 'p', p, 'gamma', gamma);

%Storing useful outputs from QUESTION 1
Q1 = struct('p', p, 'gamma', gamma, 'Verification', verification);

%% Q2: Value function evaluation
fprintf('QUESTION 2\n');

%Defining the unbiased policy and tolerance
Pi_u = 1./(GridWorld.A) * ~GridWorld.S_absorbing'*ones(1,(GridWorld.A));
tol = 0.0001;

%Perform Iterative Policy Evaluation
[V, num_iteration, V_evolution] = PolicyEvaluation(Pi_u, GridWorld.T, GridWorld.R,↙
GridWorld.S_absorbing, GridWorld.gamma, tol);
```

```
%Display V obtained, and number of iterations until convergence
temp_printV = sprintf('%.2f ', V);
fprintf('The value function is V = %s (converged after %d iterations).\n\n',
temp_printV, num_iteration);

%Storing useful outputs from QUESTION 2
Q2 = struct('V', V, 'Number_iterations', num_iteration, 'Evolution_V', V_evolution);

%% Question 3: Likelihood of obtaining specific sequences of states under specific
policies
fprintf('QUESTION 3\n');

%Defining the observed sequences of visited states
Sequence1 = [14 10 8 4 3];
Sequence2 = [11 9 5 6 6 2];
Sequence3 = [12 11 11 9 5 9 5 1 2];

%Computing the likelihood of observing each sequence under unbiased policy
p1 = likelihoodSequence(Sequence1, Pi_u, GridWorld.T);
p2 = likelihoodSequence(Sequence2, Pi_u, GridWorld.T);
p3 = likelihoodSequence(Sequence3, Pi_u, GridWorld.T);

%Defining the biased deterministic policy (1 is North, 2 East, 3 South, 4 West)
Pi_biased = zeros(GridWorld.S,GridWorld.A);      %Initialize the policy
destination = [2 0 0 4 2 4 0 1 1 1 4 4 0 1];    %List of action for each state (0 when
irrelevant)

for i = 1:S_num                                  %Iterate over states
    if(destination(i)~= 0)                       %If action from this state is not 0
        Pi_biased(i, destination(i)) = 1;        %Set a probability of 1
(deterministic) in the policy for that action given that state
    end
end

%Computing the new likelihood under biased policy
p1biased = likelihoodSequence(Sequence1, Pi_biased, T);
p2biased = likelihoodSequence(Sequence2, Pi_biased, T);
p3biased = likelihoodSequence(Sequence3, Pi_biased, T);

%Probability of observing all three sequences under an unbiased/biased policy
p_u = p1*p2*p3;                         %Under unbiased policy
p_b = p1biased*p2biased*p3biased;       %Under biased policy

%Display probabilities obtained
fprintf('The likelihood of sequence 1 to be generated by an unbiased policy is %.3g,
whereas under the biased policy it is %.3g.\n', p1, p1biased);
fprintf('The likelihood of sequence 1 to be generated by an unbiased policy is %.3g,
whereas under the biased policy it is %.3g.\n', p2, p2biased);
fprintf('The likelihood of sequence 1 to be generated by an unbiased policy is %.3g,
whereas under the biased policy it is %.3g.\n', p3, p3biased);
fprintf('Overall, the likelihood of observing all three sequences altogether under an
unbiased policy is %.3g, while under our biased policy it is %.3g.\n', p_u, p_b);
fprintf('In other terms, the biased policy makes it approximately %.2f times more
likely to observe all three sequences.\n\n', p_b/p_u);
```

```
fprintf('The policy used is shown below:\n');
DisplayPolicy(Pi_biased, GridWorld.S_names, GridWorld.A_names);
fprintf('\n\n');

%Storing useful outputs from QUESTION 3
Q3 = struct('P_unbiased1', p1, 'P_unbiased2', p2, 'P_unbiased3', p3, 'P_biased1',
p1biased, 'P_biased2', p2biased, 'P_biased3', p3biased);

%% Question 4: First-Visit Batch Monte-Carlo Policy Evaluation
fprintf('QUESTION 4\n');
fprintf('Below is a list of 10 traces generated from the MDP:\n');

%Generate 10 sequences of the MDP
numSeq = 10;          %Number of sequences to generate
for i = 1:numSeq      %Reapeat numSeq times
    %First, generate an episode: record sequence of states, actions and rewards
    [Sequence, Actions, Rewards] = Generatetrace(GridWorld.T, GridWorld.R, Pi_u,
GridWorld.S_absorbing);
    %Then, append episode to list of sequence, rewards and actions generated
    All_seq{i} = Sequence; All_act{i} = Actions; All_rew{i} = Rewards;
    %Last, display the formatted episode
    displayTrace(Sequence, GridWorld.S_names, Actions, GridWorld.A_names, Rewards);
end

%Perform Monte Carlo policy evaluation to obtain an estimate for V from 10 sequences
[V_MC, V_MC_non0] = MC_estimation(All_seq, All_rew, S_num, gamma);

%Display the estimated V using the two methods (M1 and M2)
temp_printV_MC = sprintf('%.2f ', V_MC);              %V-estimate using M1
fprintf('\nThe value function obtained using these 10 traces (method M1) is V = %s\n',
temp_printV_MC);
temp_printV_MC_non0 = sprintf('%.2f ', V_MC_non0);  %V-estimate using M2
fprintf('The value function obtained using these 10 traces (method M2) is V = %s\n\n',
temp_printV_MC_non0);

%Number of sequences used for the plot
numberof_sequences = 500;

%Plot convergence of error between V under MC and iterative policy evaluation for
increasing number of traces considered
fprintf('For question 4c, please refer to the appropriate figure for the convergence
plot of the value function using FVMCPE.\n\n\n');
[V_ev, V_ev_non0] = error_v_numtraces(numberof_sequences, gamma, S_num, T, R, Pi_u,
S_absorbing, V);

%Computation of raw ratio between MC-estimated V and IPE-estimated V
for row = 1:numberof_sequences
    V_ev(row,:) = V_ev(row,:)./V;
    V_ev_non0(row,:) = V_ev_non0(row,:)./V;
end
%The above was used to show that there was no particular trend in the final
%ratio of model-based V to model-free V estimate when converged.

%Storing useful outputs from QUESTION 4
Q4 = struct('Sequence', Sequence, 'Actions', Actions, 'Rewards', Rewards, 'V_M1',
```

```
V_MC, 'V_M2', V_MC_non0, 'EvolutionRatioErrV_M1', V_ev, 'EvolutionRatioErrV_M2',↙
V_ev_non0);

%% Question 5: Epsilon-greedy first-visit MC control
exp_num = 100;       %Number of learning experiments
numiter = 500;       %Number of iterations for each experiment

%Initialization
%Matrices of return across traces (1st dim) for different learning experiments (2nd↙
dim)
EVret_E75 = zeros(exp_num, numiter);     %For epsilon = 0.75
EVret_E10 = zeros(exp_num, numiter);     %For epsilon = 0.1

%Matrices of total rewards across traces (1st dim) for different learning experiments↙
(2nd dim)
EVrew_E75 = zeros(exp_num, numiter);    %For epsilon = 0.75
EVrew_E10 = zeros(exp_num, numiter);    %For epsilon = 0.1

%Matrices of trace length across traces (1st dim) for different learning experiments↙
(2nd dim)
EVtraces_E75 = zeros(exp_num, numiter);  %For epsilon = 0.75
EVtraces_E10 = zeros(exp_num, numiter);  %For epsilon = 0.1

%Repeated Learning Experiments
for exp = 1:exp_num
    epsilon = 0.1;      %For epsilon = 0.1
    %Get total reward and trace length across episodes for low epsilon
    [EVQ_E10, EVPi_E10, Return_E10, RewardE10, TraceLen_E10] = e_greedy(epsilon,↙
GridWorld.gamma, GridWorld.S, GridWorld.A, GridWorld.T, GridWorld.R, GridWorld.↙
S_absorbing, numiter);
    EVret_E10(exp,:) = Return_E10; EVrew_E10(exp,:) = RewardE10; EVtraces_E10(exp,:) =↙
TraceLen_E10;

    epsilon = 0.75;     %For epsilon = 0.75
    %Get total reward and trace length across episodes for high epsilon
    [EVQ_E75, EVPi_E75, Return_E75, RewardE75, TraceLen_E75] = e_greedy(epsilon,↙
GridWorld.gamma, GridWorld.S, GridWorld.A, GridWorld.T, GridWorld.R, GridWorld.↙
S_absorbing, numiter);
    EVret_E75(exp,:) = Return_E75; EVrew_E75(exp,:) = RewardE75; EVtraces_E75(exp,:) =↙
TraceLen_E75;
end

%Initialize all means and standard deviations
Mret_E10 = zeros(1,numiter);
STDret_E10 = zeros(1,numiter);
Mret_E75 = zeros(1,numiter);
STDret_E75 = zeros(1,numiter);

Mtrace_E10 = zeros(1,numiter);
STDtrace_E10 = zeros(1,numiter);
Mtrace_E75 = zeros(1,numiter);
STDtrace_E75 = zeros(1,numiter);

%Compute mean and standard deviation across learning experiments
for i = 1:numiter   %Iterate over all episodes
```

```
    %Compute return means and standard deviations
    Mret_E10(i) = mean(EVret_E10(:,i));
    STDret_E10(i) = std(EVret_E10(:,i));
    Mret_E75(i) = mean(EVret_E75(:,i));
    STDret_E75(i) = std(EVret_E75(:,i));

    %Compute trace length means and standard deviations
    Mtrace_E10(i) = mean(EVtraces_E10(:,i));
    STDtrace_E10(i) = std(EVtraces_E10(:,i));
    Mtrace_E75(i) = mean(EVtraces_E75(:,i));
    STDtrace_E75(i) = std(EVtraces_E75(:,i));
end

%Plot return against episodes for both values of epsilon
figure; subplot(2,1,1);
dispReward(Mret_E10, STDret_E10, Mret_E75, STDret_E75, numiter);

%Plot trace length against episodes for both values of epsilon
subplot(2,1,2);
dispTrace(Mtrace_E10, STDtrace_E10, Mtrace_E75, STDtrace_E75, numiter);

fprintf('For question 5, please refer to the appropriate figure for the learning↙
curves, described in the PDF file attached.\n\n');

%Defining a struct containg useful outputs from QUESTION 5
Q5 = struct('Evolution_Q_LowEpsilon', EVQ_E10, 'Evolution_Pi_LowEpsilon', EVPi_E10,↙
'Evolution_Q_HighEpsilon', EVQ_E75, 'Evolution_Pi_HighEpsilon', EVPi_E75);

%Final code output, containing all important code outputs in an organised fashion
answers = struct('Answers_to_Question1', Q1, 'Answers_to_Question2', Q2,↙
'Answers_to_Question3', Q3, 'Answers_to_Question4', Q4, 'Answers_to_Question5', Q5);
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%↙
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% LIST OF FUNCTIONS USED↙
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%↙
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%% Function declaring our custom MDP for verification of PersonalisedGridWorld.p
function [numStates, StateNames, numActions, ActionNames, AbsorbingVector, T, R] =↙
MDPGridWorld(p)
%Defining the set of states (including absorbing)
numStates = 14;                                                %Numer of states
StateNames = ['S1'; 'S2'; 'S3';'S4';'S5';'S5';'S6';'S7'];     %Name of states
AbsorbingStates = [2 3];                                       %Set of absorbing states
AbsorbingVector = zeros(1,numStates);                          %Boolean absorbing state↙
vector (1 if absorbing, 0 otherwise)
AbsorbingVector(AbsorbingStates) = 1;

%Defining the set of actions
```

```
numActions = 4;                          %Number of actions
ActionNames = ['N';'E'; 'S'; 'W'];  %Name of actions

%Defining the transition matrix T
% Transition matrix given action selected is N (North)

TN = zeros(numStates, numStates);        %Initialization
for priorState = 1:numStates             %Iterating over all prior states
    %Function returns neighbours of state (if neighbour is blocked, returns current↙
state)
    [postN, postE, postS, postW] = neighbours(priorState);

    if(AbsorbingVector(priorState))                         %If state is absorbing
        TN(priorState,priorState) = 1;                      %There will be no↙
transition away from this state
    else                                                    %If state is not↙
absorbing
        TN(postN, priorState) = TN(postN, priorState) + p;       %Probability p to↙
transition in intended direction
        TN(postE, priorState) = TN(postE, priorState) + (1-p)/2; %Transition in first↙
adjacent direction
        TN(postW, priorState) = TN(postW, priorState) + (1-p)/2; %transition in second↙
adjacent direction
    end
end

%Transition matrix given action is E (East), same as above
TE = zeros(numStates, numStates);
for priorState = 1:numStates
    [postN, postE, postS, postW] = neighbours(priorState);
    if(AbsorbingVector(priorState))
        TE(priorState,priorState) = 1;
    else
        TE(postE, priorState) = TE(postE, priorState) + p;
        TE(postN, priorState) = TE(postN, priorState) + (1-p)/2;
        TE(postS, priorState) = TE(postS, priorState) + (1-p)/2;
    end
end

%Transition matrix given action is S (South), same as above
TS = zeros(numStates, numStates);
for priorState = 1:numStates
    [postN, postE, postS, postW] = neighbours(priorState);
    if(AbsorbingVector(priorState))
        TS(priorState,priorState) = 1;
    else
        TS(postS, priorState) = TS(postS, priorState) + p;
        TS(postE, priorState) = TS(postE, priorState) + (1-p)/2;
        TS(postW, priorState) = TS(postW, priorState) + (1-p)/2;
    end
end

%Transition matrix given action is W (West), same as above
TW = zeros(numStates, numStates);
for priorState = 1:numStates
```

```
    [postN, postE, postS, postW] = neighbours(priorState);
    if(AbsorbingVector(priorState))
        TW(priorState,priorState) = 1;
    else
        TW(postW, priorState) = TW(postW, priorState) + p;
        TW(postN, priorState) = TW(postN, priorState) + (1-p)/2;
        TW(postS, priorState) = TW(postS, priorState) + (1-p)/2;
    end
end

%Complete transition matrix from concatenating all action matrices
T = cat(3,TN,TE,TS,TW);

%Defining reward matrix R
R = zeros(numStates, numStates, numActions);            %Initialize
for priorState = 1:numStates                            %Iterate over prior states
    for action = 1:numActions                           %Iterate over actions
        for postState = 1:numStates                     %Iterate over post states
            if (((priorState == 1) && (postState == 2))||((priorState == 6) &&↙
(postState == 2)))
                R(postState, priorState, action) = 0;   %If we jump into reward state↙
2, then 0 reward
            elseif (((priorState == 4) && (postState == 3))||((priorState == 7) &&↙
(postState == 3)))
                R(postState, priorState, action) = -10; %If we jump into penality↙
state 3, then -10 reward
            else
                R(postState, priorState, action) = -1;  %Any other move, -1 reward↙
(move cost)
            end
        end
    end
end
end



%% Function that returns neighbours of state in all directions, and returns current↙
state if target state is blocked
function [nextN, nextE, nextS, nextW] = neighbours(state)
%Creating a spatial grid world representation (0 for blocked states) and locating↙
state on grid
grid = [0  0  0  0  0  0;
        0  1  2  3  4  0;
        0  5  6  7  8  0;
        0  9  0  0 10  0;
        0 11 12 13 14  0;
        0  0  0  0  0  0;];

[row, col] = find(grid == state);   %Find coordinates of current state on grid

%% Find neighbour one step north from current state (N)
targetN = grid(row-1, col);     %Find content of cell one step north of current state
if(targetN == 0)                %If cell blocked, then step in that direction results↙
in current state
```

```
    nextN = state;
else                            %If cell is free, then step in that direction will↙
result in move to free cell
    nextN = targetN;
end

%% Find neighbour one step south from current state (S), same as above
targetS = grid(row+1, col);
if(targetS == 0)
    nextS = state;
else
    nextS = targetS;
end

%% Find neighbour one step east from current state (E), same as above
targetE = grid(row, col+1);
if(targetE == 0)
    nextE = state;
else
    nextE = targetE;
end

%% Find neighbour one step west from current state (W), same as above
targetW = grid(row, col-1);
if(targetW == 0)
    nextW = state;
else
    nextW = targetW;
end
end



%% Function that compares two MDPs (returns 1 if identical, 0 if different)
function veri = MDP_Verification(S1, S2, A1, A2, T1, T2, R1, R2, Absorbing1,↙
Absorbing2)
veri = 1;           %Start with verification true

%Test for number of states
if((S1-S2) ~= 0)    %If different, verification is false (0)
    veri = 0;
end

%Test for number of actions, same
if((A1-A2) ~= 0)
    veri = 0;
end

%Test for T matrices, same
if(~isequal((T1-T2), zeros(S1,S1,A1)))
    veri = 0;
end

%Test for R matrices, same
if(~isequal((R1-R2), zeros(S1,S1,A1)))
```

```
    veri = 0;
end

%Test for absorbing vectors, same
if(~isequal((Absorbing1-Absorbing2), zeros(1,S1)))
    veri = 0;
end
end



%% Iterative policy evaluation (gives V and its evolution across iterations, plus
number of iterations)
function [V, iter, Vevolution] = PolicyEvaluation(Policy, T, R, AbsorbingStates,
gamma, tolerance)
%Defining number of actions and states from policy
numActions = length(Policy(1,:));
numStates = length(Policy);

%Initialization
iter = 0;                          %Initialize iteration counter to 0
delta = tolerance + 1;             %Initialize error between two successive value
vectors (initially > tolerance)

V = zeros(1, numStates);           %Initiliaze value vector to 0
newV = V;                          %Set V at next step
Vevolution = zeros(1,numStates);   %Initialize empty matrix of V at each step i (memory
of evolution)

%Iterative Policy Evaluation
while(delta > tolerance)                %Iterate until error is within tolerance level
    for priorState = 1:numStates        %Iterate over prior states
        if AbsorbingStates(priorState)  %If the prior state is absorbing, ignore and:
            continue;                   %Jump out of the loop to next prior state
        end
        tempV = 0;                      %Temporary value function V for priorState
        for action = 1:numActions       %Iterate over actions from priorState
            tempQ = 0;                  %Temporary state-action value function Q
(priorState, action)
            for postState = 1:numStates %Iterate over post states, accumulate
immediate reward and discounted future reward
                tempQ = tempQ + T(postState, priorState, action)*(R(postState,
priorState,action) + gamma*V(postState));
            end
            tempV = tempV + Policy(priorState, action)*tempQ;    %Value for priorState
is sum of Q for one state over all actions
        end
        newV(priorState) = tempV;                                %Update the value of
this state in V
    end
    delta = max(abs(newV - V));         %Calculate max error between value of state at
two consecutive steps
    Vevolution = [Vevolution; newV];    %Append new V (at iter+1) to matrix of V
vectors
    iter = iter + 1;                    %Iteration counter increment
```

```
    V = newV;                           %The new V is now the old V for next iteration
end
end



%% This function returns the likelihood of obtaining a given sequence under a give↙
policy
function [p] = likelihoodSequence(seq, policy, T)
%Define number of actions from policy
numActions = length(policy(1,:));

%Initialize to 0.25, as there is 0.25 probability to start in each of the starting↙
states
p = 0.25;

for i = 1:(length(seq)-1)       %Iterate over every state of the sequence
    state = seq(i);             %Temporary variable containing the current state
    postState = seq(i+1);       %The post state is the next state in sequence
    temp_p = 0;                 %Temporary variable for the probability of transition↙
from current state to next
    for action = 1:numActions   %Iterate over actions available from current state
        %Accumulate probability of transitionning from current state to next across↙
all possible actions
        temp_p = temp_p + policy(state, action)*T(postState, state, action);
    end
    p = p * temp_p;             %Successively multiply the transition probabilites↙
(independance property)
end
end



%% This function displays a policy
function [] = DisplayPolicy(Policy, StateNames, ActionNames)
for s = 1:length(Policy)            %Iterate over states
    for a = 1:length(Policy(1,:))   %Iterate over actions
        if(Policy(s,a) == 1)        %If we have a probability of 1 to select this↙
action given this state
            disp(strcat('Policy(',StateNames(s,:),')=', ActionNames(a,:)));
        end                         %Policy for that state is displayed
    end
end
end



%% This a function to generate a trace from the GridWorld MDP
function [Sequence, Actions, Rewards] = Generatetrace(T,R, Policy, Absorbing)
%% Determination of the initial state
%Draw a random number between 0 and 1 from a standard uniform distribution
rand_start = rand(1); i = 1;  %Initialize step index to 1

%We create 'windows' of sizes equal to the probability of selecting each initial↙
state;
```

```
%If the random number generated falls inside of that probability window;
%We select the initial state corresponding to that window;
%This logic is repeated for action selection and next state selection.

%Initial state selection
if(rand_start < 0.25)
    Sequence(i) = 11;
elseif((rand_start >= 0.25) && (rand_start < 0.5))
    Sequence(i) = 12;
elseif((rand_start >= 0.5) && (rand_start < 0.75))
    Sequence(i) = 13;
else
    Sequence(i) = 14;
end

%% Generate next state
while(not(Absorbing(Sequence(i))))  %Repeat until absorbing state is reached
    %Find possible target states from current state
    [nextN, nextE, nextS, nextW] = neighbours(Sequence(i));
    TargetVec = [nextN nextE nextS nextW];  %Vector containing the target states from↙
current state for each direction

    %Random action selection under constraints
    rand_action = rand(1);  %Generate random number between 0 and 1 from a standard↙
uniform distribution
    if(rand_action < Policy(Sequence(i), 1))
        action = 1;         %Action is 1 with probability Policy(Sequence(i-1), 1)
    elseif((rand_action >= Policy(Sequence(i), 1)) && (rand_action < (Policy(Sequence↙
(i), 1) + Policy(Sequence(i), 2))))
        action = 2;         %Action is 2 with probability Policy(Sequence(i-1), 2)
    elseif((rand_action >= (Policy(Sequence(i), 1) + Policy(Sequence(i), 2))) &&↙
(rand_action < (Policy(Sequence(i), 1) + Policy(Sequence(i), 2) + Policy(Sequence(i),↙
3))))
        action = 3;         %Action is 1 with probability Policy(Sequence(i-1), 3)
    else
        action = 4;         %Action is 1 with probability Policy(Sequence(i-1), 4)
    end

    Actions(i) = action;    %Store chosen action

    %Compute transition probabilities given action
    T_from_current = zeros(1, 4); %Transition probability of moving to neighbours of↙
state, given action
    for j = 1:length(TargetVec)   %Iterate over possible target states (neighbours)
        postState = TargetVec(j); %Set postState to a target state
        %Record probability of transition from current state to target, given action
        T_from_current(j) = T(postState, Sequence(i), action);
    end

    %Normalize the probabilities
    T_from_current = T_from_current ./ sum(T_from_current);

    %Compute next state (random), same as for action selection
    rand_next = rand(1);
    if(rand_next < T_from_current(1))
```

```
        Sequence(i+1) = nextN;
    elseif((rand_next >= T_from_current(1)) && (rand_next < (T_from_current(1) +
T_from_current(2))))
        Sequence(i+1) = nextE;
    elseif((rand_next >= (T_from_current(1) + T_from_current(2))) && (rand_next <
(T_from_current(1) + T_from_current(2) + T_from_current(3))))
        Sequence(i+1) = nextS;
    else
        Sequence(i+1) = nextW;
    end

    Rewards(i) = R(Sequence(i+1), Sequence(i), action);     %Store acquired reward
    i = i + 1;  %Next step
end
end



%% This is a function that displays a formatted trace of the MDP
function [] = displayTrace(Sequence, S_names, Actions, A_names, Rewards)
for i = 1:length(Rewards) %Iterate over elements of episode
    %Display under required formatting
    fprintf('S%d,%c,%d', Sequence(i), A_names(Actions(i)), Rewards(i));
    %Separate elements by a comma, except last
    if(i~=length(Sequence)-1)
        fprintf(',');
    end
end
fprintf('\n');
end



%% This is a function that estimates value function from n traces of MDP
function [V, V_non0] = MC_estimation(S_sequences, S_rewards, S_num, gamma)
%Initialize Vs
V = zeros(1, S_num);
V_non0 = zeros(1, S_num);

%Matrix of total return from a given state (2nd dim) for a given sequence (1st dim)
S_return = zeros(numel(S_sequences), S_num);

%Matrix of first visit time for a given state (2nd dim) for a given sequence (1st dim)
first_visit = zeros(numel(S_sequences), S_num);

%Compute first visit time for each state
for i = 1:numel(S_sequences)              %Iterate over each trace
    seq = S_sequences{i};                 %Assign trace to a temporary variable seq
    for j = 1:length(seq)                 %Iterate over states in sequence seq
        S = seq(j);                       %Assign sequence element (state) to a
temporary variable
        if(first_visit(i, S) == 0)        %If this is the first time this state is
visited
            first_visit(i, S) = j;        %Record the first visit time for state S in
sequence i
```

```
        end
    end
end

%Compute total return for state, by accumulating reward obtained after first visit to↙
that state
for seq_num = 1:numel(S_sequences)                           %Iterate over each↙
sequence
    rew = S_rewards{seq_num};                                %Assign rewards to a↙
temporary variable
    S_return(seq_num,:) = zeros(1,S_num);                    %Reset the return of all↙
states to 0 (security)
    for k = 1:S_num                                          %Iterate over states
        if(first_visit(seq_num,k) ~= 0)                      %If state was visited in↙
this sequence
            for l = first_visit(seq_num,k):length(rew)       %Iterate over all time↙
steps after first visit
                %Accumulate discounted reward from first visit time until trace↙
termination
                S_return(seq_num, k) = S_return(seq_num, k) + gamma^(l-first_visit↙
(seq_num, k))*rew(l);
            end
        end
    end
end

%For each state, compute mean return obtained across episodes
for state = 1:S_num     %Iterate over states
    %M1 (if state not visited in one trace, return of 0)
    V(state)= mean(S_return(:, state));
    %M2 (if state not visited in one trace, trace ignored)
    V_non0(state) = mean(nonzeros(S_return(:, state)));
end

%Correct for NaN values at terminating states 2 and 3
V_non0(2) = 0;
V_non0(3) = 0;
end



%% This is a function that returns three error measures between value functions
function [err_cosine, err_dist, err_rel] = error_measure(Vtest, Vref)
Vt = [Vtest(1) Vtest(4:length(Vtest))];         %Ignore absorbing states
Vr = [Vref(1) Vref(4:length(Vref))];            %Ignore absorbing states
err_cosine = dot(Vt,Vr)/(norm(Vt)*norm(Vr));    %Cosine similarity measure (normalized↙
dot product)
err_dist = norm(Vr-Vt)/length(Vr);              %Normalised Euclidian distance measure
err_rel = 100*mean(abs((Vr-Vt)./Vr));           %Mean Absolute Percentage Error
end



%% This is a function that plots similarity measure between model-free and model-based↙
Vs against number of traces used
```

```
function [V_MC_ev, V_MC_ev2] = error_v_numtraces(num_seq, gamma, S_num, T, R, Pi,↙
S_absorbing, V_true)
%Closes all figures, creates a new figure
close all;
figure; hold on;

%Initialization
e_cos = zeros(1,num_seq);                  %Cosine similarity for each number of episodes↙
(M1)
e_dist = zeros(1,num_seq);                 %Normalised euclidian distance for each↙
defined number of episodes (M1)
e_rel = zeros(1,num_seq);                  %MAPE for each number of episodes (M1)
e_cos_2 = zeros(1,num_seq);             %Cosine similarity for each number of episodes↙
(M2)
e_dist_2 = zeros(1,num_seq);            %Normalised euclidian distance for each number of↙
episodes (M2)
e_rel_2 = zeros(1,num_seq);             %MAPE for each number of episodes (M2)
V_MC_ev = zeros(num_seq,S_num);            %Matrix containing evolution of estimated V↙
using MC (M1)
V_MC_ev2 = zeros(num_seq,S_num);       %Matrix containing evolution of estimated V using↙
MC (M2)
X = 1:num_seq;                             %X-axis for plots

%Compute similarity/error measures for increasing number of traces considered
for i = 1:num_seq                                                           %Iterate↙
num_seq times
    [seq, actions, rewards] = Generatetrace(T, R, Pi, S_absorbing);         %Generate↙
a trace
    All_seq{i} = seq; All_act{i} = actions; All_rew{i} = rewards;           %Record↙
sequences of states, actions, rewards
    [V, V_non0] = MC_estimation(All_seq, All_rew, S_num, gamma);            %Estimate↙
value function from all traces already computed
    [e_cos(i), e_dist(i), e_rel(i)] = error_measure(V, V_true);             %Compute↙
similarity/error measure for new V estimate (M1)
    [e_cos_2(i), e_dist_2(i), e_rel_2(i)] = error_measure(V_non0, V_true);  %Compute↙
similarity/error measure for new V estimate (M2)
    V_MC_ev(i,:) = V;
    V_MC_ev2(i,:) = V_non0;
end

%Plot cosine similarity measure
subplot(2,1,1); plot(X, e_cos); hold on; plot(X, e_cos_2);
title('Evolution of Cosine Similarity considering the Value Function between Iterative↙
Policy Evaluation and Monte Carlo First Visit');
ylabel('Cosine Similarity'); xlabel('Number of traces'); xlim([1 num_seq]);
legend({'Cosine Simlarity (M1)','Cosine Similarity (M2)'}, 'Location', 'southeast');
grid on; grid minor;

%Plot normalised euclidian distance measure
% subplot(3,1,2); plot(Xaxis, e_dist); hold on; plot(Xaxis, e_dist_2);
% title('Evolution of Normalised Euclidean Distance considering the Value Function↙
between Iterative Policy Evaluation and Monte Carlo First Visit');
% ylabel('Normalised Euclidian Distance'); xlabel('Number of traces'); xlim([1↙
num_seq]);
% legend({'Normalised Euclidian Distance Error','Normalised Euclidian Distance Non-↙
```

```
0'}, 'Location', 'northeast');
% grid on; grid minor;

%Plot MAPE similarity measure results
subplot(2,1,2); plot(X, e_rel); hold on; plot(X, e_rel_2);
title('Evolution of MAPE considering the Value Function between Iterative Policy↙
Evaluation and Monte Carlo First Visit');
ylabel('MAPE (%)'); xlabel('Number of traces'); xlim([1 num_seq]);
legend({'MAPE (M1)','MAPE (M2)'}, 'Location', 'northeast');
grid on; grid minor;

%Plot separate figure with focus on cosine similarity (prime interest)
figure; plot(X, e_cos); hold on; plot(X, e_cos_2);
title('Evolution of Cosine Similarity considering the Value Function between Iterative↙
and First Visit Monte Carlo Policy Evaluation Algorithms');
ylabel('Cosine Similarity (1)'); xlabel('Number of traces (1)'); xlim([1 num_seq]);
lgd = legend({'Cosine Similarity Measure (considering a return of 0 for traces where a↙
state is not visited, M1)','Cosine Similarity Measure (ignoring traces where state is↙
not visited, M2, confirmed to be of prime interest)'}, 'Location', 'southeast');
title(lgd, 'Legend');
grid on; grid minor;

end



%% This is a function that performs epsilon-greedy learning and control (with M2)
function [Q_ev, Pi_e_ev, TotRet, Tot_Reward, trace_length] = e_greedy(epsilon, gamma,↙
S_num, A_num, T, R, S_absorbing, numiter)
%Initialization
Q_ev = zeros(S_num, A_num, numiter);           %Evolution of state-action Q value↙
function along episodes
Pi_e_ev = zeros(S_num, A_num, numiter);        %Evolution of policy along episodes
returns = zeros(S_num, A_num, numiter);        %Evolution of returns along episodes

TotRet = zeros(1, numiter);                    %Total (discounted) return obtained↙
for each episode
Tot_Reward = zeros(1, numiter);                %Total reward obtained for each↙
episode
trace_length = zeros(1, numiter);              %Total trace length for each episode

Q = zeros(S_num, A_num); Q_ev(:,:,1) = Q;      %Initial Q set to 0 (promote↙
exploration at first)

%Initialize arbitrary policy (unbiased in this case)
Pi_e = 0.25*ones(S_num, A_num); Pi_e_ev(:,:,1) = Pi_e;

%% Create numiter traces of the MDP and apply MC control
for iter = 1:numiter                                           %Repeat numiter times

    [seq, act, rew] = Generatetrace(T, R, Pi_e, S_absorbing);  %Generate a trace
    TotRet(iter) = f_return(rew, gamma);                       %Calculate return for↙
that trace
    Tot_Reward(iter) = sum(rew);                               %Calculate reward for↙
that trace
```

```
    trace_length(iter) = length(seq);                          %Calculate length of↙
that trace

    SA_visited = zeros(S_num, A_num);    %Re-initialize empty array of visisted↙
couples of states/actions
    first_visit = zeros(S_num, A_num);   %Re-initialize empty array of times of first↙
visit for couples

    %Obtain first passage times for state/action couples for this trace
    for temp_act = 1:length(act)                               %Iterate over actions↙
of trace
        temp_state = seq(temp_act);                            %Assign state to a↙
temporary variable
        temp_action = act(temp_act);                           %Assign action to a↙
temporary variable
        if(SA_visited(temp_state, temp_action) == 0)           %If this is the first↙
time state/action couple is visited
            SA_visited(temp_state, temp_action) = 1;           %Record that this↙
couple was visited
            first_visit(temp_state, temp_action) = temp_act;   %Record the first↙
visit time for couple
        end
    end

    %Compute return for each state/action couple for this trace
    for k = 1:S_num                                     %Iterate over states
        for l = 1:A_num                                 %Iterate over actions
            temp_return = 0;                            %Temporary variable for the↙
return from state/action
            if(SA_visited(k, l) == 1)                   %If couple was visited in↙
this sequence
                for m = first_visit(k,l):length(rew)    %Iterate over all time steps↙
after first visit
                    %Accumulate discounted reward for state-action pair
                    temp_return = temp_return + gamma^(m-first_visit(k, l))*rew(m);
                end
            end
            returns(k,l,iter) = temp_return;   %Return obtained for a state/action↙
pair on episode number iter
        end
    end

    %Obtain updated Q
    for stateQ = 1:S_num
        for actionQ = 1:A_num
            %Average return for action/state pair across all previous traces,
            %Ignoring traces where action/state pair was not visited (M2)
            Q(stateQ, actionQ) = mean(nonzeros(returns(stateQ,actionQ,:)));
        end
    end

    Q_ev(:,:,iter+1) = Q;   %Store newly obtained Q function

    %Obtain action arg_max which maximizes Q
    for i = 1:S_num                                            %Iterate over states
```

```
        if(ismember(i,seq))                                     %If state was visited↙
in this sequence
            [max_val, arg_max] = max(Q(i,:));                   %Compute action↙
maximizing Q for this state
            for a = 1:A_num                                     %Iterate over actions
                if(a == arg_max)                                %For optimal action;
                    Pi_e(i,a) = 1 - epsilon + epsilon/A_num;    %Update policy↙
according to epsilon-gredy formula
                else
                    Pi_e(i,a) = epsilon/A_num;                  %Exploring probability↙
epsilon/|A|
                end
            end
        end
    end

    Pi_e_ev(:,:,iter+1) = Pi_e;     %Record newly obtained epsilon-greedy policy
end
end



%% This is a function that performs epsilon-greedy learning and control (with M1,↙
unused)
function [Q_ev, Pi_e_ev, TotRet, trace_length] = e_greedy_visited0(epsilon, gamma,↙
S_num, A_num, T, R, S_absorbing, numiter)
%% Initialization
Q_ev = zeros(S_num, A_num, numiter);        %Evolution of state-action Q value↙
function along episode
Pi_e_ev = zeros(S_num, A_num, numiter);     %Evolution of policy along episode
returns = zeros(S_num, A_num);              %List of matrices containing return for↙
each action/state pair for each episode

TotRet = zeros(1, numiter);                  %Total rewards obtained for each episode
trace_length = zeros(1, numiter);               %Total trace length obtained for each↙
episode
Q = -10*ones(S_num, A_num); Q_ev(:,:,1) = Q;    %Initial Q set to 0

%% Initialize arbitrary soft policy (unbiased in this case)
Pi_e = 0.25*ones(S_num, A_num);

%Place this policy in first row of policy evolution matrix
Pi_e_ev(:,:,1) = Pi_e;

%% Create numiter traces of the MDP and apply MC control
for iter = 1:numiter                                            %Repeat numiter times
    [seq, act, rew] = Generatetrace(T, R, Pi_e, S_absorbing);   %Generate a trace

    %Calculate total reward for that trace (excluding discount factor)
    TotRet(iter) = sum(rew);

    %Calculate length of that trace
    trace_length(iter) = length(seq);

    SA_visited = zeros(S_num, A_num);                    %Re-initialize empty array of↙
```

```
visisted couples of states and actions
    first_visit = zeros(S_num, A_num);                        %Re-initialize empty array of↙
times of first visit for couples

    %% Obtain first passage times for state/action couples for one trace
    for u = 1:length(act)                                     %Iterate over actions of↙
trace
        temp_state = seq(u);                                  %Assign state to a temporary↙
variable
        temp_action = act(u);                                 %Assign action to a temporary↙
variable
        if(SA_visited(temp_state, temp_action) == 0)          %If this is the first time↙
state/action couple is visited
            SA_visited(temp_state, temp_action) = 1;          %Record that this couple was↙
visited
            first_visit(temp_state, temp_action) = u;         %Record the first visit time↙
for couple
        end
    end

    %% Calculate accumulated return for each state/action couple for one trace
    %Create an empty matrix of returns collected for each state/action pair
    returns_mat = zeros(S_num, A_num);
    for k = 1:S_num                                           %Iterate over states
        for l = 1:A_num                                       %Iterate over actions
            temp_return = 0;                                  %Temporary variable for the↙
total return from state given action
            if(SA_visited(k, l) == 1)                         %If couple was visited in↙
this sequence
                for m = first_visit(k,l):length(rew)          %Iterate over all time steps↙
after first visit
                    %Accumulate discounted reward for state-action pair
                    temp_return = temp_return + gamma^(m-first_visit(k, l))*rew(m);
                end
            end
            returns_mat(k,l) = temp_return;                  %Return obtained for a↙
state/action pair on trace iter
        end
    end
    returns = cat(3, returns, returns_mat);                  %Append returns to list of↙
returns across iterations

    %% Obtain updated Q
    for stateQ = 1:S_num
        for actionQ = 1:A_num
            %Average return for action/state pair across all previous
            %traces, including traces where action/state pair was not visited
            Q(stateQ, actionQ) = mean((returns(stateQ,actionQ,2:length(returns↙
(1,1,:)))));
        end
    end

    %Store newly obtained Q function
    Q_ev(:,:,iter+1) = Q;
```

```
    %% Obtain action arg_max which maximizes Q
    for i = 1:S_num                                          %Iterate over all↙
states visited in the sequence
        if(ismember(i,seq))
            [max_val, arg_max] = max(Q(i,:));                %Obtain action which↙
maximizes state-action value function from this state
            for a = 1:A_num                                  %Iterate over all↙
possible actions
                if(a == arg_max)                             %For optimal action
                    Pi_e(i,a) = 1 - epsilon + epsilon/A_num; %Update policy↙
according to epsilon-gredy formulas
                else
                    Pi_e(i,a) = epsilon/A_num;               %Exploring probability
                end
            end
        end
    end

    %Record newly obtained epsilon-greedy policy
    Pi_e_ev(:,:,iter+1) = Pi_e;
end
end



%% This is a function that computes discounted reward (unused)
function tol_return = f_return(reward_seq, gamma)
temp_return = 0;                  %Initialize return to 0
for i = 1:length(reward_seq)    %Iterate over reward sequence
    %Accumulate discounted reward over episode
    temp_return = temp_return + gamma^(i-1)*reward_seq(i);
end
tol_return = temp_return;
end



%% This function displays the mean and std of reward collected against episodes
function [] = dispReward(mean_low, std_low, mean_high, std_high, numiter)
%Define x-axis
x = 1:numiter;
X = [x, fliplr(x)];

%Define top and bottom limits using standard deviation
top_low = mean_low + std_low;
bottom_low = mean_low - std_low;
top_high = mean_high + std_high;
bottom_high = mean_high - std_high;

%Plot return/reward for low epsilon
Y_low = [bottom_low, fliplr(top_low)];
fill(X,Y_low,'b','LineStyle','none'); hold on;
plot(x, mean_low, '-');
alpha(0.25);
ylabel('Total return (\epsilon=0.1) (1)');
```

```
% set(gca, 'YScale', 'log');

%Plot return/reward for high epsilon
yyaxis right;
Y_high = [bottom_high, fliplr(top_high)];
fill(X,Y_high,'r','LineStyle','none'); hold on;
plot(x, mean_high, '-');
alpha(0.25);
ylabel('Total return (\epsilon=0.75) (1)');

%Title, legend and grid
title('Mean and Standard Deviation in the Return against Episodes for Two Settings of↙
\epsilon (100 learning experiments)');
xlabel('Episodes (1)');
lgd = legend({'Std for \epsilon = 0.1','Mean for \epsilon = 0.1', 'Std for \epsilon =↙
0.75','Mean for \epsilon = 0.75'},'Location','southeast');
title(lgd,'Legend'); grid on; grid minor;
end



%% This function displays mean and std of trace length per episode against episodes
function [] = dispTrace(mean_low, std_low, mean_high, std_high, numiter)
%Define x-axis
x = 1:numiter;
X = [x, fliplr(x)];

%Define top and bottom limits
top_low = mean_low + std_low;
bottom_low = mean_low - std_low;
top_high = mean_high + std_high;
bottom_high = mean_high - std_high;

%Plot return/reward for low epsilon
Y_low = [bottom_low, fliplr(top_low)];
fill(X,Y_low,'b','LineStyle','none'); hold on;
plot(x, mean_low,'-');
alpha(0.25);
ylabel('Trace length per episode (\epsilon=0.1) (1)');
% set(gca, 'YScale', 'log');

%Plot return/reward for high epsilon
yyaxis right;
Y_high = [bottom_high, fliplr(top_high)];
fill(X,Y_high,'r','LineStyle','none'); hold on;
plot(x, mean_high,'-');
alpha(0.25);
ylabel('Trace length per episode (\epsilon=0.75) (1)');

%Title, legend and grid
title('Mean and Standard Deviation in the Total Sequence Length per Episode against↙
Episodes for Two Settings of \epsilon (100 learning experiments)');
xlabel('Episodes (1)');
lgd = legend({'Std for \epsilon = 0.1','Mean for \epsilon = 0.1', 'Std for \epsilon =↙
0.75','Mean for \epsilon = 0.75'},'Location','northeast');
```

```
title(lgd,'Legend'); grid on; grid minor;
end
```