Continuous-State SARSA for the Mountain Car Problem

M.S.C. Winokan

A working implementation of the SARSA algorithm with discrete states to solve the mountain car benchmark was adapted to use radial basis functions (RBFs). The two algorithms were investigated to determine the optimal parameters choices, and their performance was quantified. The continuous state implementation was found to provide improved performance over the discrete implementation even with a low number of basis functions.

1 Introduction

The mountain car problem describes a car trapped at the bottom of a one-dimensional sinusoidal valley, with a target at the top of the right hill.[1] Specifically, the car is underpowered. An underpowered car must learn to employ its own momentum to overcome gravity, unlike an overpowered car can simply drive up the hill.[2]

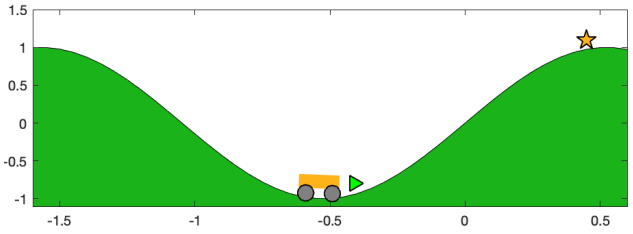


Figure 1: The mountain car problem. Adapted from [3].

The mountain car problem is a classic example of a continuous control task where the system must take seemingly counter-productive steps in order to reach the goal.[2] The control algorithm must be developed in such a way that the delayed gratification of reaching the target justifies the many steps away from the target position, otherwise human aid will be necessary to solve the problem.[1, 2]

2 Methods and Definitions

2.1 Defining the State and Action Space

To define the mountain car problem a two-dimensional state space, describing the position and velocity, is needed. In Martin's given implementation (see [3]) the state space is discretised.

$$S = \begin{bmatrix} (x_{min}, -v_{max}) & \dots & (x_{min}, +v_{max}) \\ \dots & \dots & \dots \\ (x_{max}, -v_{max}) & \dots & (x_{max}, +v_{max}) \end{bmatrix} \quad (1)$$

Ten discrete positions ($x_{min} = -1.5, x_{max} = 0.5, dx = 0.205$), and six discrete velocities ($v_{max} = 0.07, dv = 0.028$) form the discrete position-velocity state space. This is illustrated in Figure 2 where a discretised state lies on every vertex in the grid.

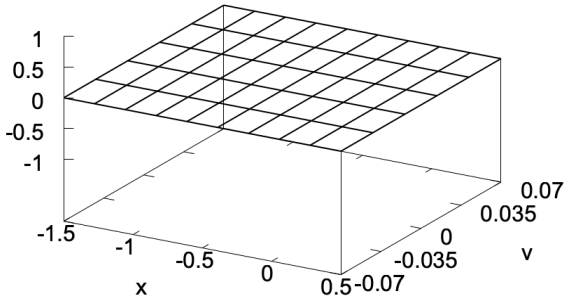


Figure 2: The discretised state space in Martin's given implementation (see [3]).

The actions are discretised such that they resemble a simple motor on the car which is either applying a force forward, backwards, or is switches off.[3] The action space is thus:

$$A = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \quad (2)$$

The goal of this work was to modify the given MATLAB implementation to use continuous-valued radial basis functions (RBFs) to describe the state space. Every point in the state space is represented by a feature vector $\phi(s)$:[4] Each state $\phi(s)$ is a linear superposition of a set of basis functions/features.[2]

$$\boldsymbol{\phi}(\boldsymbol{s}) = \begin{pmatrix} \phi_1(\boldsymbol{s}) \\ \phi_2(\boldsymbol{s}) \\ \dots \\ \phi_n(\boldsymbol{s}) \end{pmatrix} \quad (3)$$

Each element of the feature vector depends on the proximity to a defined feature via:[2]

$$\phi_i(\boldsymbol{s}) = \exp\left(-\frac{||\boldsymbol{s} - \boldsymbol{c}_i||}{2\sigma_i^2}\right) \quad (4)$$

Where $\boldsymbol{c}_i$ is a vector describing the centre of the RBF in state-space, and σ_i is the standard deviation of the Gaussian distribution. In this adap-

tation the generalisation is asymmetric, meaning that σ_i is different along position coordinate compared to the velocity coordinate. Each RBF is now described via:

$$\phi_i(x, v) = \exp\left(-\frac{|x - c_{x,i}|}{2\sigma_{x,i}^2} - \frac{|v - c_{v,i}|}{2\sigma_{v,i}^2}\right) \quad (5)$$

The value of the function in 5 is shown in Figure 3.

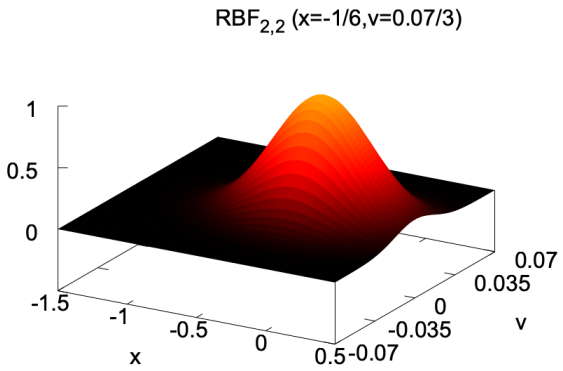


Figure 3: A single Radial Basis Function in the position-velocity state space.

2.2 Reinforcement Learning and SARSA

State-action-reward-state-action (SARSA) is an algorithm in reinforcement learning whose name reflects the parameters considered by the Q-value function:[2, 1]

$$Q = Q(\boldsymbol{s}_t, a_t, r_t, \boldsymbol{s}_{t+1}, a_{t+1}) \quad (6)$$

The value function Q considers the reward r_t for taking action a_t when in state $\boldsymbol{s}_t$, and reaching state $\boldsymbol{s}_{t+1}$ from which action a_{t+1} is taken. Function approximation estimates the value function and we implement a scheme such that after many iterations the approximation mimics the true policy.[4]

$$Q_\theta(\boldsymbol{s}, a) \approx Q^\pi(\boldsymbol{s}, a) \quad (7)$$

A SARSA agent interacts with the environment, and updates the policy Q based on the actions taken and their reward.[2, 1] For a discretised state-action-space the lookup table $Q(\boldsymbol{s}, a)$ is updated via:

$$\Delta Q(\boldsymbol{s}, a) = \alpha[r + \gamma Q(\boldsymbol{s}', a') - Q(\boldsymbol{s}, a)] \quad (8)$$

Where α is a scalar learning rate, and γ is a scalar discount factor. A discount factor of 0 makes the agent "opportunistic" by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward.[2]

For a continuous state definition, the value function is now evaluated with respect to a parameter vector $\boldsymbol{\theta}_a$ corresponding to the value of each basis feature for a given (discrete) action.[4]

$$Q_\theta(\boldsymbol{s}, a) = \boldsymbol{\theta}_a.\boldsymbol{\phi}(\boldsymbol{s}) \quad (9)$$

The equivalent of Equation 8 for a SARSA update of $\boldsymbol{\theta}_a$ is:

$$\Delta\boldsymbol{\theta}_a = \alpha[r + \gamma Q(\boldsymbol{s}', a') - Q(\boldsymbol{s}, a)]\boldsymbol{\phi}(\boldsymbol{s}) \quad (10)$$

Substituting in Equation 9 yields:

$$\Delta\boldsymbol{\theta}_a = \alpha[r + \gamma\boldsymbol{\theta}_{a'}.\boldsymbol{\phi}(\boldsymbol{s}') - \boldsymbol{\theta}_a.\boldsymbol{\phi}(\boldsymbol{s})]\boldsymbol{\phi}(\boldsymbol{s}) \quad (11)$$

We can see that Equations 11 and 8 are equivalent if we substitute one of the pure basis functions $\boldsymbol{\phi}_i(\boldsymbol{s})$.

2.2.1 Action Selection: epsilon-greedy

In the SARSA algorithm description above, it has not been discussed how an action is chosen given that the actor is in state $\boldsymbol{s}$. The action should be decided based on the optimal policy (or the current estimate) but allow for some random exploration. One such strategy is epsilon-greedy (ϵ-greedy).[2, 3] Where a parameter ϵ determines the probability of random exploration. The algorithm chooses a random action from the action-list with a rate of ϵ. Otherwise the best action is chosen by evaluating the value Q_a and choosing the action corresponding to the maximal Q-value.[2]

3 Initial Investigations

The SARSA algorithm with a continuous state space built from Radial Basis Functions was applied to the mountain car problem by modifying the discrete space implementation made by Martin (see [3]). The MATLAB code can be found at `https://github.com/mwinokan/Delft_MountainCar`.

Following experimentation with various parameters the algorithm was found to produce successful policies within 200 episodes, which is

comparable to the original discrete implementation. The following basis and parameters yielded relatively successful convergence:

$$
\begin{aligned}
&c_{x,i} = [-1.5 : 2/3 : 0.5] \\
&c_{v,i} = [-0.07 : 0.035 : 0.07] \\
&\sigma_x = 1/3, \sigma_v = 0.0175 \\
&\text{max steps} = 1500 \\
&\epsilon_0 = 0.005, \epsilon_{t+1} = 0.999\epsilon_t \\
&\alpha = 0.004 \\
&\gamma = 0.99
\end{aligned}
\tag{12}
$$

Figure 4 shows the number of steps taken until the car reaches the target as the algorithm iterates over 200 episodes. It can be seen that as the policy is updated the variance in the number of steps taken decreases. In the initial episode domain, the car often does not reach the target. At the point where it reliably does, a convergence is reached.

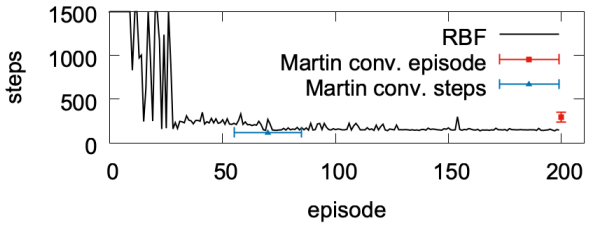


Figure 4: A single Radial Basis Function in the position-velocity state space.

In a trial of fifteen executions the implementation reaches rough conversion after an average of 70 ± 10 episodes. Rough convergence is achieved when the agent will always reach the target within the maximum number of steps, in Figure 4 this is reached around episode 30. The average number of steps taken by the final policy to reach the target was 180 ± 10. In Figure 4 the average performance over 20 trials of the provided discrete solution is plotted for comparison. The initial parameters given in Equation 12 produce improved behaviour over the given discrete implementation. The two approaches will further be compared in the next section.

Figures 5 to 7 show the "cost-to-goal" function at episodes 20, 100, and 200. The cost-to-goal function takes the parameter vector corresponding to the action with the optimal Q-value and expresses this in terms of all points in the state space. As the episode number increases, the algorithm iterates on the parameter vectors and the 'true' cost-to-goal function is approximated.

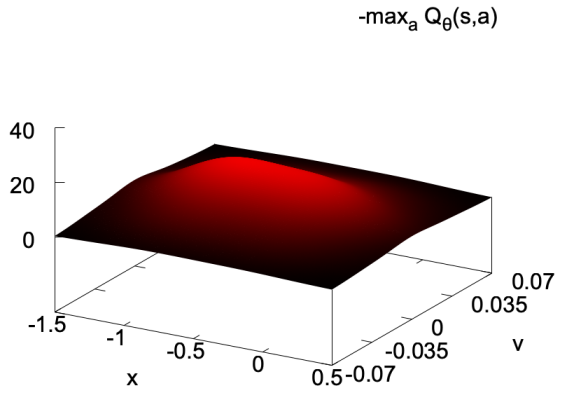


Figure 5: Cost-to-goal function for the position-velocity state space. At episode 20.

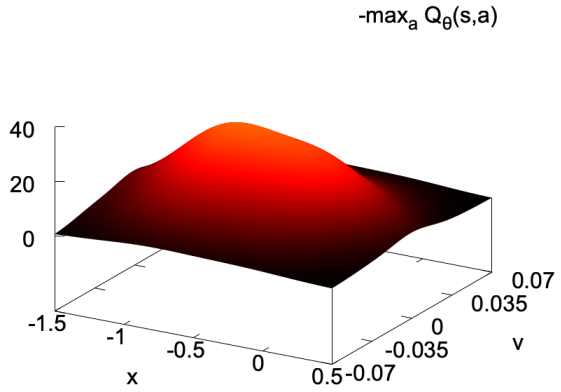


Figure 6: Cost-to-goal function for the position-velocity state space. At episode 100.

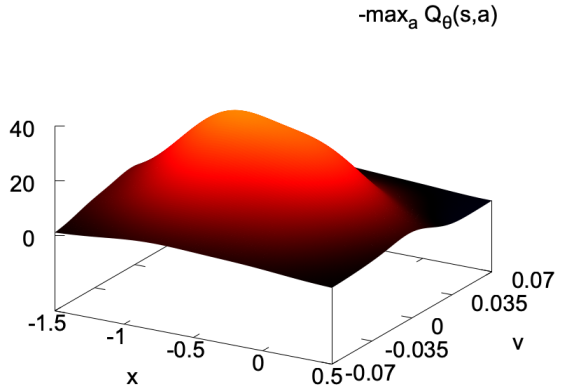


Figure 7: Cost-to-goal function for the position-velocity state space. At episode 200.

4 Quantifying the performance of the algorithms

To quantify the performance of the two implementations discussed in this report, two metrics have been devised.

The first is simply the number of steps taken to reach the target after a set number of episodes have been completed. In this case we chose to study 200 episodes, as this results in reliable convergence from the discrete implementation, which we wish to improve upon. This metric will allow for comparison in the quality of the solutions provided by the algorithm.

The second metric applies to the rate at which the algorithm converges on reliable solutions. It is defined as the last episode at which the goal was not reached, or the "episode of last non-solution". This metric will allow for comparison in the rate of convergence.

In the following studies 20 trials are run for each set of parameters and the aforementioned metrics are averaged with a mean and standard error value.

4.1 Determining optimal parameters for the continuous implementation

In this section the performance of various parameter sets, such as those given in Equation 12, are investigated.

Figure 8 shows that the value chosen for ϵ (the rate of random action) of 0.005 produces a good quality policy with small uncertainty.

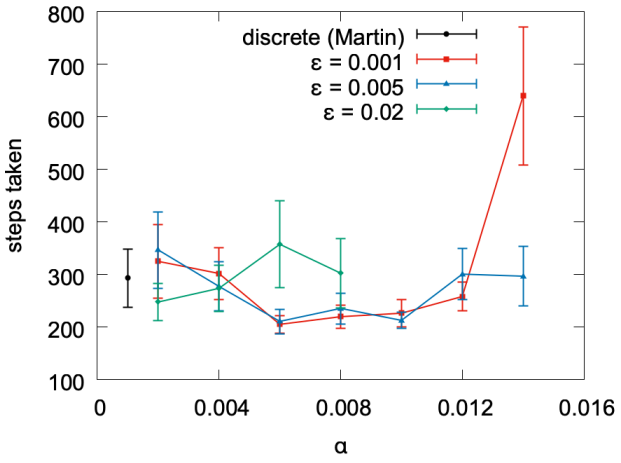


Figure 8: The quality of the policy after 200 episodes for different α and ϵ parameters in the continuous state implementation.

The above is only if a suitable value for the learning rate α is selected also. From Figure 8 we can conclude that a more suitable choice for the learning rate would be 0.005.

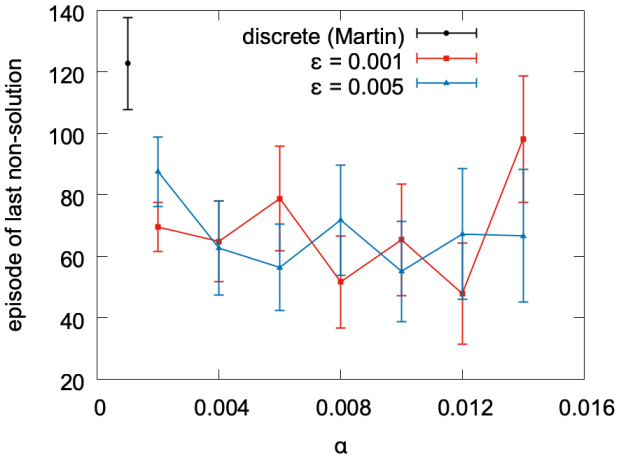


Figure 9: The rate of convergence for different α and ϵ parameters in the continuous state implementation.

From Figure 9, it can be concluded that varying the learning rate and rate of random action does not have a large effect on the rate of convergence in the range of parameters considered. The following parameter set was selected for investigations into the basis selection:

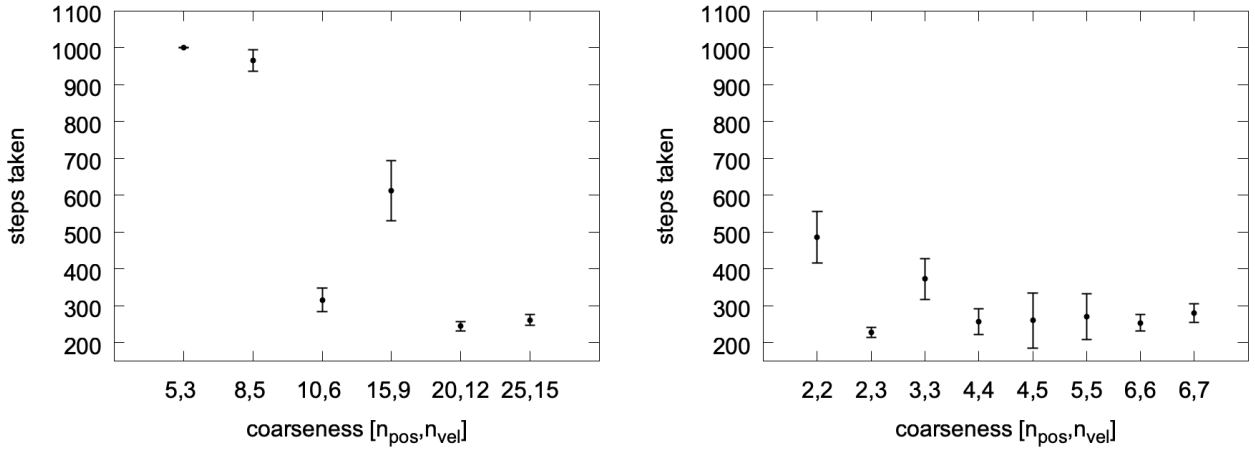
$$\begin{aligned} &c_{x,i} = [-1.5 : 2/3 : 0.5] \\ &c_{v,i} = [-0.07 : 0.035 : 0.07] \\ &\sigma_x = 1/3, \sigma_v = 0.0175 \\ &\text{max steps} = 1500 \\ &\epsilon_0 = 0.005, \epsilon_{t+1} = 0.999\epsilon_t \\ &\alpha = 0.005, \gamma = 0.99 \end{aligned} \tag{13}$$

4.1.1 Comparison to discrete performance

In Figures 8 and 9, a data point for the provided discrete implementation is also given. That algorithm was run with the following parameters:

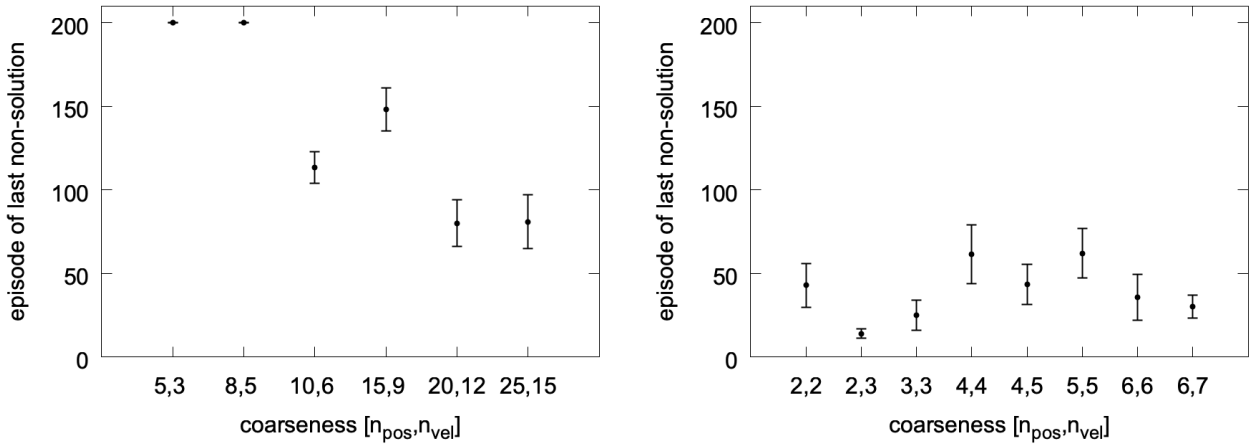
$$\begin{aligned} &[x] = [-1.5 : 0.205 : 0.345] \\ &[v] = [-0.07 : 0.0028 : 0.07] \\ &\text{max steps} = 1000 \\ &\epsilon_0 = 0.01, \epsilon_{t+1} = 0.99\epsilon_t \\ &\alpha = 0.5, \gamma = 1.0 \end{aligned} \tag{14}$$

It can be seen that the continuous implementation outperforms the provided algorithm in both of the studied metrics, if optimal parameters are chosen for the former.



(a) (b)

Figure 10: Effect of coarseness on number of steps taken after 200 episodes. Each data point reflects the statistics of 20 trials.



(a) (b)

Figure 11: Effect of coarseness on episode of last non-solution. Each data point reflects the statistics of 20 trials.

5 Investigating the effect of coding coarseness

In this section the performance of the two implementations was investigated as the coarseness of the state space was varied. In the discrete implementation the coarseness is the number of discrete positions and velocities in the tile coding. In the continuous state implementation the coarseness is the number of radial basis functions (RBFs) in each of the state dimensions.

Figures 10 and 11 show the quality of the policy converged upon and the rate of convergence for the two implementations.

From Figure 10 the several conclusions can be drawn. The general trend that a higher discretisations and number of features increases the policy quality is not as strongly reflected in the continuous state implementation. The continuous state approximations produces good quality policies even with a small number of RBFs, notably for six RBFs with two positions and three velocity values. The reason that this state description improves so drastically upon [2,2] and even [3,3] is likely due to the odd number of discrete velocities. This results in a direct weight-

ing for points in state space with zero velocity - rather than the superposition of two features in the basis.

In Figure 10, the rate of convergence for both implementations is shown. It can be seen that for the tile coding implementation, an increase in the number of states in the statelist results in faster convergence. The same cannot be said for the continuous state implementation, where a satisfactory rate of convergence is achieved even with a very small number of basis functions.

From all the investigations performed in this work the following parameter set for the continuous state implementation is recommended as it produces very favourable results with low uncertainty and low computational expense:

$$\begin{aligned} &c_{x,i} = [-1.5, 0.5] \\ &c_{v,i} = [-0.07, 0.00, 0.07] \\ &\sigma_x = 1.0, \sigma_v = 0.035 \\ &\text{max steps} = 1500 \\ &\epsilon_0 = 0.005, \epsilon_{t+1} = 0.999\epsilon_t \\ &\alpha = 0.005, \gamma = 0.99 \end{aligned} \tag{15}$$

6 Conclusion

In this work, a working implementation of the SARSA algorithm with discrete state definition to solve the mountain car benchmark (see [3]), was adapted to use a continuous state space described by radial basis functions (RBFs). The behaviour of the two algorithms was investigated to determine the optimal parameters choices for successful convergence. The quality of the two algorithms for various parameter sets was determined by considering the quality of produced policy as well as the rate of convergence. The effect of the number of states in the state space of the discrete state implementation and the number of RBFs for the continuous state implementation was investigated.

The SARSA algorithm with continuous states defined by a feature vector defined by a superposition of Gaussian basis features was found to provide improved performance over the discrete implementation even with a low number of basis functions.

While this simple benchmark does not guarantee such a scheme will work with a more complicated system, with more parameters and a noisy environment, it does showcase the value of describing the space in terms of features.[2, 5, 6] This implementation performs comparably to much more complicated schemes such as in [7].

References

[1] Andrew William Moore. “Efficient memory-based learning for robot control”. In: (1990).

[2] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998.

[3] Jose Antonio Martin H. *Matlab SARSA implementation of the Mountain Car Problem*. (Available at: `https://jamh-web.appspot.com/download.htm`).

[4] David Silver. *Value Function Approximation*. University College London. Reinforcement Learning Course COMPM050. 2015.

[5] Verena Heidrich-Meisner and Christian Igel. “Variable Metric Reinforcement Learning Methods Applied to the Noisy Mountain Car Problem”. In: *Recent Advances in Reinforcement Learning*. Ed. by Sertan Girgin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 136–150.

[6] Shahab Jabbari Arfaee. “Comparing Different Function Approximations Using Mountain-Car Problem”. University of Alberta.

[7] Alexey A Melnikov, Adi Makmal, and Hans J Briegel. “Projective simulation applied to the grid-world and the mountaincar problem”. In: *arXiv preprint arXiv:1405.5459* (2014).