University of Münster
School of Business & Economics

Sebastian Deisel

Implementation of the Policy Iteration Algorithm for an Energy Storage Problem

Seminar Thesis

in the context of the seminar “Dynamic Decision Making in Energy Systems and Transportation”

at the Research Group Quantitative Methods for Logistics
(University of Münster)

Principal Supervisor:	Prof. Dr. Stephan Meisel
Associate Supervisor:	Tanja Merfeld, M.Sc.
Presented by:	Sebastian Deisel [394 951] Bentelerstr. 39 48149 Münster sebastian.deisel@uni-muenster.de
Submission:	3rd July 2020

Contents

Figures

Listings

1 Introduction

The victory of AlphaGo over Lee Sedol in the board game Go in 2016 indicated a milestone and reinforcement learning became a hot topic not only among scientists. Already in the 1950s Richard Bellmann defined a functional equation, known as the Bellman equation, that can be solved with dynamic programming by using the concept of a dynamical system's state and a value function. Both approaches have in common that they minimize some kind of measure in a dynamic system and it's goal is to solve the optimal control problem (Sutton and Barto 1998).

Richard Bellman further formulated this problem by introducing a discrete stochastic version of it and Ronald Howard worked out the policy iteration technique to solve this problem iteratively (Foster and Howard 1962). All of these elements are important for the theory of modern reinforcement learning and therefore still relevant topics these days (Sutton and Barto 1998).

The aim of the following thesis in the context of the seminar "Dynamic Decision Making in Energy Systems and Transportation" deals with how a dynamic program can be implemented to solve a real world problem of today. In particular Howard's policy iteration algorithm was implemented to solve a simplified energy storage problem where energy can be stored in a battery and sold and bought at a market. Challenges may occur due to the uncertain development of the market price or other exogenous information. The implemented algorithm generates a model that decides how much energy should be transferred in or out of a battery at a point in time to maximize a certain value. Furthermore, the solution is extensible such that other information can be added to the model when the problem appears to be more complex. The model is used to elaborate on computing the marginal value of one storage unit that might help a decision maker for finding the optimal capacity of her/his battery.

The thesis begins by defining the fundamental elements of Markov Decision Processes, the basis for any dynamic program. First, an overview of the elements of dynamic programming is given in section 2.1. Next in section 2.2 the components of Markov decision processes were defined formally. Section 2.3 presents the value functions, which is the functional equation Bellman introduced. At the end of this chapter in section 2.4 a modelling framework is introduced that helps to model any problem instance. Furthermore a quick summary of the introduced concepts is presented.

The thesis continues by presenting the policy iteration in chapter 3. For that the algorithm is first described in general in 3.1. Afterwards the substep policy evaluation is introduced in 3.2 as well as the substep policy improvement in 3.3. This chapter further presents the implemented policy iteration algorithm in section 3.4.

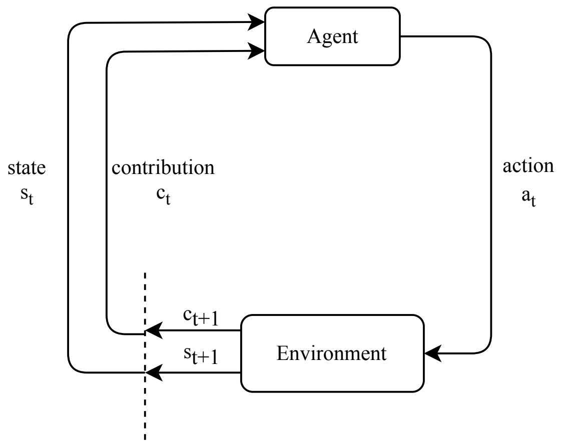
Chapter 4 is meant to apply the policy iteration algorithm developed in the previous chapter onto the energy storage case. First this chapter gives an overview about the problem and formalizes the problem formally in section 4.1. The following section 4.2 explores the energy storage problem by analyzing the marginal value of a storage unit. This examination might help decision makers for choosing an appropriate battery capacity. Presented in section 4.3 a second experiment was conducted to examine how long it will take to find the optimal policy dependent upon the number of states given in the system.

The work concludes in chapter 5 with a recap of the presented thesis and an outlook about further investigations.

2 Markov Decision Process

2.1 Overview

The basis for a dynamic program is the Markov Decision Process which is a formalization of sequential decision making. In a Markov Decision Process a decision maker subsequentially called an agent interacts with an environment. Since this paper focuses on discrete timesteps the agent receives information about the state $s_t \in S$ of the environment at each point in time $t = 0, 1, 2, 3, ..., T$. Based on that knowledge the agent decides on an action $a_t \in A(s)$, with A representing the set of feasible decision variables for s_t. After his decision the agent receives a numerical contribution c_{t+1} as a consequence of it's action. Moreover the agent finds itself in a new state s_{t+1} resulting from the action made in t and the contribuftion it received (Sutton and Barto 1998). To illustrate this basic concept figure 1 is provided to give an overview about the idea of Markov Decision Processes.



Own representation based on Sutton and Barto (1998)

Figure 1 Agent-Environment Interaction in a Markov Decision Process

The state variable is defined as the minimally dimensioned function of history that is necessary and sufficient to calculate the cost function, the decision function and the transition function. Therefore they contain the information needed to model the system from time t onwards (Powell 2011). While the components of this definition should be explained later in this seminar thesis the state variable itself can be distinguished into three different types (Powell and Meisel 2016; Powell 2011).

(1) The physical state - Snapshot of the status of the physical resources of the model. This might include the level of energy in a battery, the level of water in a reservoir or the location of a sensor on a network.

(2) The informational state - Other information that is relevant to make a decision. This might include the current price of electricity on a grid.

(3) The belief/knowledge state - Describing unobservable parameters with a probability distribution. This might include the remaining lifetime of a battery.

State variables and decision variables are both sets which might either be discrete, continuous or categorical (Powell 2011). For continuous variables that means that the space should be approximated by discretization which usually results in a trade-off decision. On the one hand a very fine space with many points gives a much better approximation of the continuous space on the other hand it leads in a costly computation in terms of time (Judd 1998; Powell 2011).

2.2 Components

The simplified model in figure 1 provides a first intuition about how the system evolves over time. It can be written as a sequence of state-action-pairs, each leading to another state and contribution. (Powell and Meisel 2015; Sutton and Barto 1998).

$$s_0, a_0, c_1, s_1, a_1, c_2, s_2, a_2, ..., s_T$$

In a totally deterministic environment the outcome of choosing an action a_t in s_t would a priori lead to exactly one successor state. However, in a real world scenario a deterministic environment is barely the case and therefore at each point in time the outcome of selecting an action a_t in s_t is uncertain. Thus, when choosing a_t in s_t the outcome is a set of states with each state having a certain probability of occurence. The uncertainty can be modeled by introducing the exogenous information w_{t+1} which becomes known in $t + 1$ (Powell and Meisel 2016; Powell 2011). Another concept is the post-decision state s_t^a which represents the state of the system immediatly after the agent has decided on an action a_t (Powell and Meisel 2016, 2015).

Hence, the sequence of state-action-pairs can be updated with the post-decision state s_t^a as well as the exogenous information w_t.

$$s_0, a_0, s_0^a, w_1, c_1, s_1, a_1, s_1^a, w_2, c_2, s_2, a_2, s_2^a, ..., s_T$$

While state variables and decision variables are deterministic the finite set of states can be easily determined. For exogenous information a way of formulating randomness is by introducing a delta parameter that represents the difference of a state variable in t and $t + 1$. For instance considering a price p in t, the price p in $t + 1$ can be written as

$$p_{t+1} = p_t + \hat{p}_{t+1}$$

while $\hat{p}$ can be for instance a normal distributed random variable with $\mu = 0$ and $\sigma^2 = 1$. For the price, however, this means that in $t + 1$ the price could possibly lead into an invalid state due to the discretization done beforehand. A standard way of dealing with this problem is the assumption of a state-dependent information process. It allows a probability distribution of new information to be a function of the state of the system itself (Powell 2011). Applying this on the price example that means that the price in $t + 1$ depends on the price in t. In concrete terms that means that a negative $\hat{p}$ is more probable when the price is high and a positive $\hat{p}$ is more probable when we price is low.

To further formalize how the system evolves over time the concept of a transition function $S^M(\cdot)$ is being introduced. The state s_{t+1} is determined as a function of the model M. It is described by the state parameter s_t, the selected action parameter a_t along with the exogenous information parameter w becoming known in $t + 1$ (Powell and Meisel 2015).

$$s_{t+1} = S^M(s_t, a_t, w_{t+1})$$

It is common to express exogenous information formally as a probability of transitioning to a state s' given that the agent is in state s_{t-1} and take action a_{t-1}. Here, the state-transition probabilities is defined with the following equation (Sutton and Barto 1998).

$$p(s'|s, a) = Pr(s_t = s'|s_{t-1} = s, a_{t-1} = a) = \sum_{c \in C} p(s', c|s, a)$$

When modelling the exogenous information the transition function is usually considered as given. Hence, referring to the price example the concrete characteristic of p_{t+1} is given as a discrete probability distribution. One way of declaring that is by using a transition matrix with rows representing s and columns representing s' (Powell 2011).

In each timestep t the agent receives a contribution such that the agent receives a sequence of contributions over time: $c_1, c_2, c_3, ..., c_T$. The agent's goal in a Markov Decision Process is to maximize the contribution function C_t over all timesteps until the agent reaches a terminal state in T (Sutton and Barto 1998).

$$C_t = c_1 + c_2 + c_3 + ... + c_T$$

When the agent finds itself in a terminal state the episode ends and the agent starts a new episode independent of the previous one. These kind of tasks are called episodic tasks. A second kind of tasks are continuing tasks. Here, the problem is stated as an infinite time horizon and the final step would be in $T = \infty$ (Sutton and Barto 1998). Is this the case the previous described sum would also yields to infinity. To overcome this issue a fixed discount ratio γ is introduced with $0 < \gamma < 1$ to discount future contributions. (Bellman 1957).

$$C_t = c_t + \gamma c_{t+1} + \gamma^2 c_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k c_{t+k+1}$$

A low γ results in a shortsighted agent while a higher γ results in a more farsighted agent (Sutton and Barto 1998). If $\gamma = 0$ the agent would consider only the contribution in t but not contributions occurring in the future. If $\gamma = 1$ the agent would give all future contributions the same importance as the contribution in t.

2.3 State-Value Function and Action-Value Function

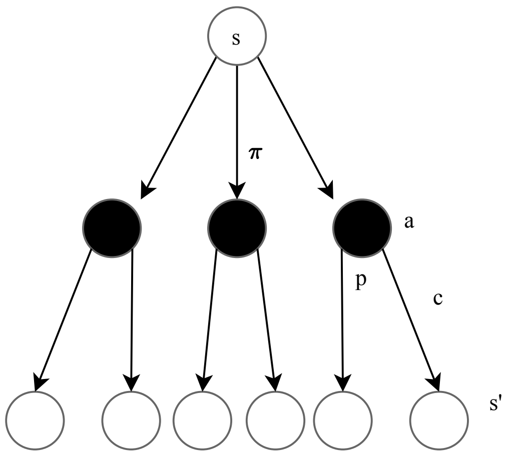
As a next step the value function of a Markov Decision Process will be defined. A value function is an estimate about how good it is for the agent to be in a certain state with respect to the contribution C_t. It is therefore a mapping from state to the expected contribution (Littman 2001; Sutton and Barto 1998). As already depicted the contribution in the future is determined by the action the agent selects at each point in time. Therefore, the value function is directly related to how the agent acts in a certain state. This scheme of acting are called policies and describes the behaviour of the agent (Littman 2001). Denoted as $\pi(a|s)$ the policy describes how likely it is for an agent to pick an action $a \in A(s), \forall s \in S$ (Sutton and Barto 1998). To clarify this even further the agent changes its policy to maximize its return C.

Since the return C is uncertain due to the exogenous information w and hence directly dependent on the state-transition probability distribution $p(s'|s, a)$, the state-value-function $v_\pi(s)$ states the expected return in s while following a specific policy π (Sutton and Barto 1998).

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[C_t | S_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k c_{t+k+1} | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_c p(s', c|s, a)[c + \gamma v_\pi(s')] \end{aligned}$$

It is noticeable that v_π is of recursive nature and thus expresses a relation of the value of one state to its successor states. The last equation is the Bellman equation for v_π and its averages over all possibilities and gives them a weight of its occurring probability (Sutton and Barto 1998). It follows Bellman's principle of optimality which describes that often a complex problem of a time series can be broken into smaller subproblems in a recursive manner. The Bellman equation expresses this principle by defining a relation between the value of the complex problem and the values of its subproblem (Bellman 1957; K.Vinotha 2014; Adda and Cooper 2003).

To illustrate this equation the diagram in figure 2 is given. Here, each empty circle represents a state $s \in S$ while each filled circle represents a state-action pair. While in s the agent can decide on any action $a \in A(s)$ with respect to it's policy π. When the agent has decided on a particular action the outcome of its action is uncertain due to the randomness of the environment and is determined by $p(s'|s, a)$. In any case the agent finds itself in a successor state s' and receives a contribution c.



Own representation based on Sutton and Barto (1998)

Figure 2 Diagram for v_π

Similar to the state-value function $v_\pi(s)$ the action-value function $q_\pi(s, a)$ is defined as the expected return starting from s, taking the action a and after that following the policy π.

$$q_\pi(s, a) = \sum_{s'} \sum_{c} p(s', c|s, a)[c + \gamma q_\pi(s', a')]$$

Having a decent function to optimize for, the optimization problem for a Markov Decision Process is to find a policy π which maximizes the value of that function this is to say the optimal policy π^* is the one which maximizes the value function (Sutton and Barto 1998).

$$q_*(s, a) = \sum_{s'} \sum_{c} p(s', c|s, a)[c + \gamma \max_{a'} q_*(s', a')]$$

2.4 Modelling Framework

The relevant components to model a dynamic program is now fully described. The components should now be formulated in an appropriate framework which should help to structure a given problem instance. Therefore the components should be taken into a condensed overview. A modeler can take this overview as a framework and guidance that might help him in the modeling process. Each dynamic program consists of five elements: (Powell 2011)

(1) State variables - Describing what it needs to be known at a point in time t.

(2) Decision variables - Variables under control. These variables represent the actions the agent can decide on.

(3) Exogenous information processes - Variables that describes information that arrives exogenously and representing randomness of the environment.

(4) Transition function - Function that describes how the system evolves over time from one point in time to another.

(5) Contribution function - The agent either tries to maximize a contribution function i.e. maximizing expected contribution or to minimize a cost function. It describes how well the agent is performing at a given point in time.

3 Dynamic Programming

3.1 Policy Iteration

Bearing in mind the previous points, one must now consider the question on how to find the optimal policy to maximize the value function obtained in the previous chapter. For solving dynamic programs the two methods value iteration and policy iteration are well known. Both are model-based methods to find the optimal policy π^* by recursively updating the values using the Bellman optimality equation (Krueger et al. 2017). The following section should outline how policy iteration works and how it can be applied for a dynamic program.

Policy iteration can be understood as a repeated execution of evaluating the current policy (policy evaluation) and improving the policy afterwards (policy improvement). The algorithm begins by initializing an arbitrary policy $\pi_{i=0}$. Afterwards a policy evaluation step is performed to compute the value V_π for the initialized policy π, followed by generating an improved policy $\pi_{k=1}$. Each policy is a strict improvement over the previous one. The policy iteration terminates if the updated policy is the same as the previous one: $\pi_k = \pi_{k-1}$ and hence the derived policy converges to the optimal policy $\pi*$ (Rust 1996; Sutton and Barto 1998). This can be visualized with the following excerpt, while E represents the evaluation step and I represents the improvement step.

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi 1} \xrightarrow{I} ... \xrightarrow{I} \pi* \xrightarrow{E} v*$$

Furthermore, listing 1 is given to provide a condensed overview in pseudo code. $\forall s \in S$ an initial policy is generated by randomly selecting from the set of possible actions A. Next, the policy evaluation and the policy improvement step is repeatedly executed one after the other until the policy appears to be stable (Sutton and Barto 1998). This can be computed by element-wise comparison of the current policy π and the improved policy π'. If the array appears to be identical, the policy is considered to be stable and hence the optimal policy $\pi*$ is determined. In any case the improved policy is assigned to the current policy and eventually being utilized in the next iteration until the mentioned termination criteria is met.

```
function policy_iteration(A, S)
    in: all possible actions with $a \in A$
    in: all possible states with $s \in S$
    out: optimal policy

    init array $\pi(s)$ = random_choice(A), $\forall s \in$ S

    Repeat
        $V$ = policy_evaluation($\pi, S$)
        $\pi'$ = policy_improvement($V, A, S$)
        policy_stable = is_identical($\pi$, $\pi'$)
        $\pi = \pi'$
    until policy_stable

    return $\pi$
```

Listing 1 Policy Iteration

An alternative to the policy iteration algorithm is the value iteration. This algorithm is considered to be weaker in terms of performance compared to policy iteration. However, researcher showed that the later one is not always superior and depends on the parameters of the problem and the algorithm (Pashenkova et al. 1996; Puterman 1994). The value iteration is an iterative procedure that calculates the expected return of a state using the utilies of the neighboring states until the utilities are satisfactory enough. It is a direct implementation of the Bellman equation by turning the equation into an update rule (Pashenkova et al. 1996; Sutton and Barto 1998). Note that a detailed explanation of value iteration was omitted due to the limited scope of this seminar paper. However, interested readers are referred to Puterman (1994); Sutton and Barto (1998); Pashenkova et al. (1996); Russell (2002).

3.2 Policy Evaluation

As stated, the first step of each iteration in policy iteration is to evaluate the current policy. It therefore asks how the state-value function $v_\pi(s)$ respectively the action-value function $q_\pi(s, a)$ under a policy π can be computed. As mentioned the Bellman equation for v_π is of recursive manner and therefore the value in $s \in S$ depends on the value of $s' \in S$. To solve this problem the iterative policy evaluation is given.

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_c p(s', c|s, a)[c + \gamma v_k(s')]$$

It iteratively calculates the values of each state under the policy π. It therefore generates a sequence of values v_k with each value being an approximation of the "real" state-value function under the policy π. With $k \longrightarrow \infty$ the approximation v_k converges to v_π. This convergence of the state-value function is guaranteed as long as the policy π ends up in a terminal state or the discount ratio γ fulfills it's constraint $\gamma < 1$ (Sutton and Barto 1998).

Since v_k is only an approximation of v_π and converges in $k \longrightarrow \infty$ it is common to introduce a parameter ϵ which is a small number and represents the termination criterion. This parameter helps to decrease the computing time. The iterative policy evaluation stops when $|v_k - v_{k-1}| < \epsilon$ (Adda and Cooper 2003).

The policy evaluation algorithm is given in listing 2. The function accepts the policy π which is being evaluated as well as all possible states S. It initializes an array of the same size as S which represents the values of $s \in S$. It further iterates over all $s \in S$ and calculates its state-value function $v_\pi(s)$ under the given policy π. Now, $\forall s \in S$ the difference of the values of the current iteration and the values of the previous iteration is being calculated. The result is summed up to calculate δ and the termination criteria is met when $\delta < \epsilon$. After termination the values for π are sufficiently approximated with respect to ϵ and the policy evaluation function returns the values (Sutton and Barto 1998; Adda and Cooper 2003). Note that the state-value function can easily be exchanged by the action-value function $q(s, a)$, since optimal policies also share the same optimal action-value function (Sutton and Barto 1998).

```
init $\epsilon$ with a small number

function policy_evaluation($\pi$, S)
    in: policy being evaluated
    in: all states for whose policy the values are being computed with $s \in S$
    out: values under the policy $v_\pi$

    init array $V(s)$ = 0, $\forall s \in$ S

    Repeat
        $\delta = 0$
        prev_values = V

        For each $s \in$ S:
            $V(s) = \sum_a \pi(a|s) \sum_{s'} \sum_c p(s', c|s, a)[c + \gamma V_k(s')]$

            $\delta$ += V(s) - prev_values(s)
    until $\delta < \epsilon$

    return $V$
```

Listing 2 Policy Evaluation

3.3 Policy Improvement

The policy improvement step is the step in policy iteration. Here, a new and improved policy π_{i+1} is generated with respect to the current values for π_i. $\forall s \in S$ the policy improvement step determines if the current policy for this state should be changed or not. To do so the action-value function $q_\pi(s,a), \forall a \in A(s)$ is computed. The action which maximizes q is being selected afterwards and hence forming an improved greedy policy $\pi'(s)$. (Sutton and Barto 1998).

$$\begin{aligned}\pi'(s) &= \arg\max_a q_\pi(s,a)\\ &= \arg\max_a \mathbb{E}[c_{t+1}\gamma v_\pi(s')|S_t = s, A_t = a]\\ &= \sum_{s'}\sum_{c} p(s',c|s,a)[c+\gamma v_\pi(s')]\end{aligned}$$

The policy improvement algorithm is given in listing 3. The function accepts the values of the current policy being calculated in the policy evaluation step. Further, the function accepts all possible actions, all possible states and the policy itself. The function initializes a boolean variable that indicates whether the function has found an improved policy. It further initializes an array of the size of the possible state whose elements is null. Further the function iterates over all $s \in S$, calculating $q_\pi(s,a)$ and selecting the action which maximizes q as the best action for s. The policy improvement step is done by returning the respective policy π'.

```
function policy_improvement(V, A, S)
    in: values of current policy
    in: all possible actions with $a \in A$
    in: all possible states with $s \in S$
    in: policy of current iteration
    out: new policy

    init array $\pi'(s)$ = Null, $\forall s \in$ S

    For each $s \in$ S:
        $\pi'(s) = \arg\max_a \sum_{s'} \sum_c p(s',c|s,a)[c+\gamma V(s')]$

    return $\pi'$
```

Listing 3 Policy Improvement

3.4 Implementation

To demostrate how the policy iteration algorithm can be implemented the following section outlines the sample framework which was developed during this seminar thesis. Importance was given to implement a generalizable framework instead of code which can not be reused for other problems. This is particularly important if the modeler wants to further complicate the problem by introducing other state variables or decision variables.

Furthermore, it was taken care to ensure that the concepts and algorithms presented earlier are clearly and unambiguously reflected in the code. That includes naming variables and functions appropriately as well as separating the concepts introduced. Note that the presented framework lacks of some optimization techniques. For the implementation the python programming language 3.7 (Python Software Foundation, https://www.python.org/) was chosen.

For the sake of completeness the following paragraphs outline the implemented code which might not be reflected in any listing. However, for reference the code is available at [1].

For seperation concerns the three classes *Agent, State* and *Environment* were created. As conducted the *Agent* class performs the actual policy iteration and holds an instance of the environment it is in. It further holds instances $\forall s \in S$ by using the *State* class which itself is responsible for holding the elements of the state variables and performing a transition. Lastly, the *Environment* class is responsible for injecting randomness and providing the contribution for the agent.

[1] https://github.com/sebdei/Policy-Iteration-for-Simplified-Energy-Storage-Problem

The *Agent* class starts by creating a set of actions A as well as the set of states S. This is done by discretizing the decision space respectively the state space. According to listing 1 the agent initializes a policy that satisfies the post decision constraints $\forall s \in S$. This is done by randomly selecting a feasible action from the set of possible actions for each state. After the initialization the policy is repeatedly evaluated by a policy evaluation function and improved by a policy improvement function. Next, a function determines whether the policy of the previous iteration is identical with the improved policy. It simultaneously iterates over these two lists and do a element-wise comparison of the contained actions both policies suggests. Note that the element are arrays themselves, representing the action and therefore another element-wise comparison is mandatory. In the next step the old policy is being overwritten by the improved policy. Now, if the policy of the previous run appears to be the same as the improved policy the loop breaks and the optimal policy is being determined and being provided as a return value. The implemented python code for the policy iteration can be found in listing 4.

```python
def policy_iteration(self, possible_actions, possible_states):
    policy = [self.pick_feasible_action_for_state(state, possible_actions) for
        state in possible_states]
    policy_stable = False

    while not policy_stable:
        values = self.policy_evaluation(policy, possible_states)
        new_policy = self.policy_improvement(values, possible_actions,
            possible_states)

        policy_stable = self.is_identical(policy, new_policy)

        if policy_stable:
            break

        policy = new_policy

    return policy
```

Listing 4 Policy Iteration Function in Python

To outline how the policy evaluation step can be computed listing 5 is given. The function accepts the policy which should be evaluated as well as a list of possible states. The function starts by initializing an array with zeroes of the same size as the list of possible states. After that a while loop determines the values of the policy by copying the values of the previous iteration for later comparison with the termination criterion. Further, the function iterates over the policy and all possible states simultaneously and calculates the action-value function for each state under the given policy. If the sum of the element-wise difference of the values of the previous iteration and the evaluated values of the current iteration is small enough w.r.t to ϵ the values of the given policy were approximated and returned by the function.

```python
EPSILON = 1e-2

def policy_evaluation(self, policy, possible_states):
    values = np.zeros(len(possible_states))

    while True:
        prev_values = np.copy(values)

        for i, (action, state) in enumerate(zip(policy, possible_states)):
            values[i] = self.calculate_action_value(state, action, values,
                possible_states)

        if (np.sum(np.fabs(prev_values - values)) < EPSILON):
            break

    return values
```

Listing 5 Policy Evaluation Function in Python

Listing 6 shows how the policy which was being evaluated beforehand can be improved. The function accepts the evaluated values, all possible actions and all possible states. It initializes an empty array which is being filled throughout the function with the same size as the possible states. It further iterates over all possible states and computes the action-value function for all possible actions of that state. The best action w.r.t. to the action-values is being selected which is determined by numPy's argmax function and chosen as the best action for this particular state. Lastly, the function returns the improved policy π'.

```python
def policy_improvement(self, values, possible_actions, possible_states):
    new_policy = [None] * (len(possible_states))

    for s, state in enumerate(possible_states):
        action_values = [self.compute_action_value(state, action, values,
            possible_states) for action in possible_actions]

        best_action = possible_actions[np.argmax(action_values)]
        new_policy[s] = best_action

    return new_policy
```

Listing 6 Policy Improvement Function in Python

Listing 7 shows how the action-value function presented in listing 5 and listing 6 can be represented. The function accepts the state and the action for which the value should be calculated as well as the action-values that were calculated until now and all possible actions. It first checks whether the given state-action satisfies all given constraints and returns a big negative number if this is not the case to force the algorithm to not consider these actions. Otherwise, for the given state a transition is executed to observe all expected successor states w.r.t to a probability distribution provided by the environment with the given action. The transition function further provides the contribution to which the transition leads. Now, having a possible successor state, the related value must be found in the set of values. This is simply done by simultaneously iterating over all values and their related states and comparing the states with the successor state. Before that it is being assured that the lookup state is in the set of possible states by simply approximating to a valid state. After finding the value to a success state the action-value is calculated according to its definition. These steps are repeated for all possible successor states with its probability. The outcome is summed up and returned.

```python
    def calculate_action_value(self, state, action, values, possible_states):
        if not self.are_constraints_satisfied(state, action):
            result = -100
        else:
            result = 0

            for prob, successor_state, contribution in zip(*state.transition(
                action, self.env)):
                value_of_successor_state = self.find_value_for_state(
                    successor_state, values, possible_states)
                result += prob * (contribution + DISCOUNT_RATIO *
                    value_of_successor_state)

        return result
```

Listing 7 Action-Value Calculation in Python

4 Experimental Setup

4.1 Problem Modelling

The underlying theory of policy iteration as well as the concrete implementation has been laid out in the previous chapter. This chapter in turn deals with applying the presented principle on a practical example. An energy storage problem should serve as a problem instance. To mitigate the curse of dimensionality (Bellman 1961) the problem is modelled as a simplified version.

Energy can be put in a battery R to store it temporarily (a_t^{GR}) or put out of the battery to sell it on a grid G (a_t^{RG}). The battery has a maximum capacity denoted as R^c and the battery level for t is denoted as R_t. Whenever the battery is charged or discharged a small portion of the energy gets lost and is described as the transfer loss with a function η. Moreover in one step in time the battery can be charged/discharged with a maximum transfer rate δ_{in}/δ_{out} s.t. at each point in time the agent can decide to buy energy with $0 \leq a_t^{RG} \leq \delta_{in}$ and to sell energy with $0 \leq a_t^{GR} \leq \delta_{out}$. Energy can be bought/sold at the grid G for a variable market price MP at time t that reflects the uncertainty of the environment. This whole problem instance for this simplified energy storage case is visualized in figure 3.

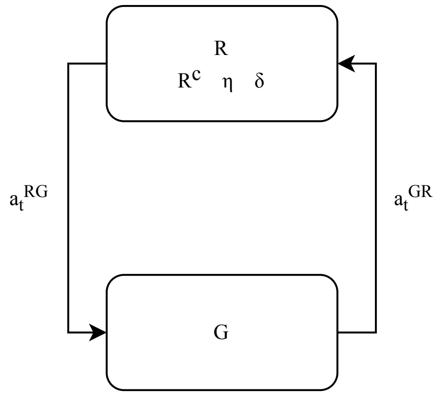


Figure 3 Energy Storage Problem

To model the energy storage problem the model formulation framework of 2.4 is conducted to structure the problem appropriately. The problem is considered to be a continous task with an infinite time horizon to make the problem slightly more realistic, since selling and buying energy can be done 24/7.

The state variables for this example represents the level of the battery in t alongside the market price MP in t. According to section 2.1 the states are further distinguished between physical states, informational states and belief state. To map this categories onto the energy problem the level of the battery is considered to be a physical state variable and the market price p is considered to be an informational state variable. Moreover, in this simplified storage example the market price is considered to be exogenous and both state variables R_t and MP_t are assumed to be independent from each other.

$$s_t = (R_t, MP_t)$$

In each point in time t the agent can simultaneously sell energy at the market and buy energy at the market. The decision variables are therefore defined as follows.

$$a_t = (a_t^{RG}, a_t^{GR})$$

While the decision variables having the following constraints.

$$0 \le a_t^{RG} \le \delta_{in}$$
$$0 \le a_t^{GR} \le \delta_{out}$$

An action a_t is only feasible if the constraints are satisfied. That is the amount of energy putting into the battery in time t must be smaller or equal to the difference of the maximal capacity R^c and the current level of energy R_t with respect to the transfer loss function η. Moreover, the amount of discharging in one time step must be smaller than the current level of energy in the battery. The following constraints must be satisfied.

$$\eta(a_t^{GR}) \le R^c - R_t$$
$$a_t^{RG} \le R_t$$

As depicted, the exogenous information for the simplified energy storage example is the uncertain market price. To model the exogenous information a state-dependent information process is the underlying assumption. Therefore MP_{t+1} depends on the market price in time t. To model the exogenous information a probability transition matrix is considered as given.

The transition function for the battery R from t to $t+1$ can be derived from the current level of energy in t plus the amount of energy the agent puts into the battery w.r.t. the transfer loss function η minus the amount of energy the agent pulls out of the battery w.r.t the transfer loss function η.

$$R_{t+1} = R_t + \eta(a_t^{GR}) - \eta(a_t^{RG})$$

The contribution of a decision in time t is the difference of the amount of energy the agents sells in t time times the market price MP_t w.r.t the transfer loss function η and the amount of energy the agent buys at the grid times the price MP_t.

$$C(s_t, a_t) = \eta(MP_t a_t^{RG}) - MP_t a_t^{GR}$$

4.2 Experiment 1 - Marginal Value

A typical objective for discussion of the introduced storage problem is the analysis of the marginal value for a storage unit. In other words, this objective deals with the question of how a dependent variable changes when the independent variable is increased for one unit. For this goal several experiments with different price probability matrices were conducted to present how the marginal value behaves when the storage unit is incremented.

For the objective analysis of the marginal value the probability transition matrix was fixed for each experiment. Only the capacity of the resource R^c was slightly incremented by a step size of two and starting with $R^c = 4$. In total 32 optimal policies were calculated with different capacities. To summarize how the remaining parameters were chosen table 1 is provided to give a condensed overview about the problem being modeled.

Remark that in one timestep the agent can decide on two different actions. Therefore, the discretization of the action space with $|a_t^{RG}| = |a_t^{GR}| = 21$ different discrete steps per action results in 441 different actions. The discretization of the state space in turn were conducted with 81 different steps per resource, where $0 \le R_t \le R^c$. The market price in turn can take on the values $MP = \{1, 2, 3, 4, 5\}$. In total this results in 405 different states.

Furthermore, the probability transition matrix in table 2 were chosen to describe how the market price evolves from one time step to another with a certain probability.

For this transition matrix it is obvious that the market prices are "symmetric" in the sense that it fluctuates around it's mean $\mu = 3$ i.e. the probability of transition from $MP = 3$ to $MP = 2$ is

Parameter	**Value**
η	0.8
γ	0.9
δ_{in}	2.5
δ_{out}	2.5
ϵ	0.01

Table 1

	1.0	2.0	3.0	4.0	5.0
1.0	0.40	0.30	0.20	0.10	0.00
2.0	0.20	0.40	0.25	0.10	0.05
3.0	0.10	0.20	0.40	0.20	0.10
4.0	0.05	0.10	0.25	0.40	0.20
5.0	0.00	0.10	0.20	0.30	0.40

Table 2

the same as the probability of transitioning from $MP = 3$ to $MP = 4$. The same holds true for $MP = 5$ to $MP = 4$ and $MP = 1$ to $MP = 2$ etc.. Further, the elements of the main diagonal, which indicate how likely it is to keep the market price in $t + 1$, is always equal to 0.4.

To see how the marginal value behaves when the underlying probability transition matrix differs a second matrix in table 3 was used for the experiment. It again describes how the market price evolves from one time step to another with a certain probability.

	1.0	2.0	3.0	4.0	5.0
1.0	0.40	0.40	0.20	0.00	0.00
2.0	0.20	0.40	0.30	0.10	0.00
3.0	0.00	0.20	0.40	0.30	0.10
4.0	0.00	0.00	0.20	0.40	0.40
5.0	0.30	0.00	0.10	0.20	0.40

Table 3

Instead of having a "symmetric" shape as given in the previous matrix the desired behaviour is to have a market price probability distribution that tends to be more increasing. If $MP_{max} = 5$ is reached it means there is a relatively high probability that the market price crashes and ends up in $MP_{min} = 1$. For all other market prices the probability of ending up with an increased price in $t + 1$ is strictly higher than ending up with a decreased price in $t + 1$. The probability of ending up with the same price in $t + 1$ is also always equal to 0.4.

Figure 4 provides a condensed overview about the experiment results. It is shown the battery capacity on the x-axis as well as the summed action-values $\forall s \in S$ of the determined optimal policy on the y-axis. While the black points represent the "symmetric" probability transition matrix, the red points represent the price probability which tends to be more increasing.

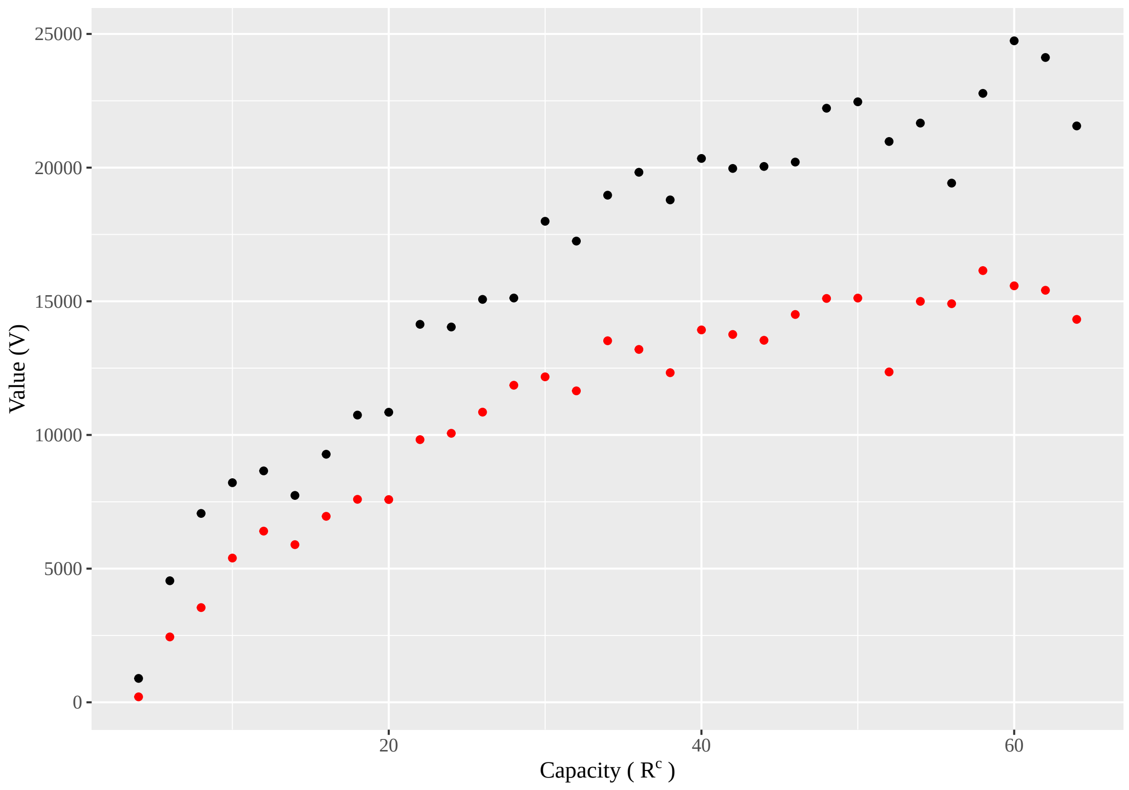


Figure 4 Experiment 1 - Results

The summed values are increasing while the capacity also increases. With lower capacities the value increase is even stricly monotonic. With higher capacities the values do not increase strictly anymore but an upward trend is still observable for both probability matrices. Moreover, it can be observed that the marginal value of a storage unit is higher when having a low capacity compared to the marginal value which is observable when the capacity is high. The plot assumes that the values of the "symmetric" matrix are overall higher than the values of the price probability which tends to be more increasing. This especially becomes apparent with higher capacities. Here the difference of the values are significant.

Furthermore, it seems like the higher the capacity, the more the value looses stability. It can be assumed that this occurs due to the fact that the state space was discretized beforehand. With increasing capacity the gaps between the state space are also increasing and therefore a value lookup of a valid successor state is only possible by rough approximation.

By further analyzing the given plot graphically it is assumed that both values increase logarithmic when the battery capacity is increased. Therefore, the marginal value of a storage unit decreases with increasing capacity. A logarithmic regression model was applied for both datasets to observe underlying functions that most likely fit the given data at best. The result is plotted in figure 5 and is described for the "symmetric" price probability matrix (black) with the following function.

$$v(R^c)_s = 7768.8 \ln(R^c) - 9566.5$$

For the price probability matrix which tends to be more increasing (red) is described with the following function.

$$v(R^c)_i = 5813.719 \ln(R^c) - 8319.052$$

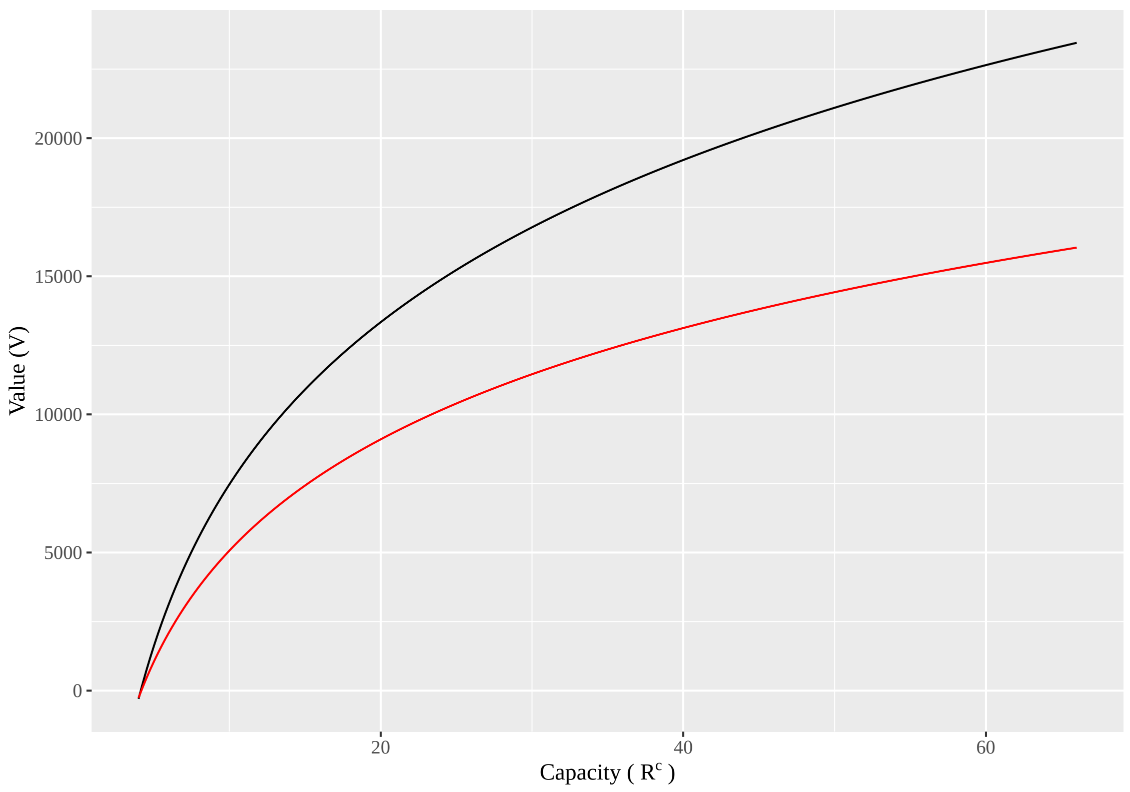


Figure 5 Experiment 1 - Regression Model

Apart from visualizing the fitted regression model in figure 5 the function can be used to calculate the marginal value for a specific storage unit. For instance it might be compute the value of one additional storage capacity when the capacity is $R^c = 15$. Note that an example calculation for the price probability matrix which tends to be more increasing was omitted.

$$\begin{aligned}\Delta v(R^c)_s &= |v(R^c) - v(R^c + 1)| \\ \Delta v(15)_s &= |v(15) - v(16)| \\ \Delta v(15)_s &= 501.3856\end{aligned}$$

This in turn can also be visualized with a function. Figure 6 shows the marginal values for the storage capacities from 4 to 65. Note that the y-axis represents the values for R^c when the battery is increased by exactly 1. The x-axis represents R^c in turn again.

As expected and discussed before, the marginal value of a storage unit decreases when the capacity increases. For the "symmetric" price probability (black) the marginal values are higher than the marginal values for the price probability matrix which tends to be increasing (red). These information can be used to choose an appropriate and optimal storage capacity for the energy storage problem being modeled in this chapter.

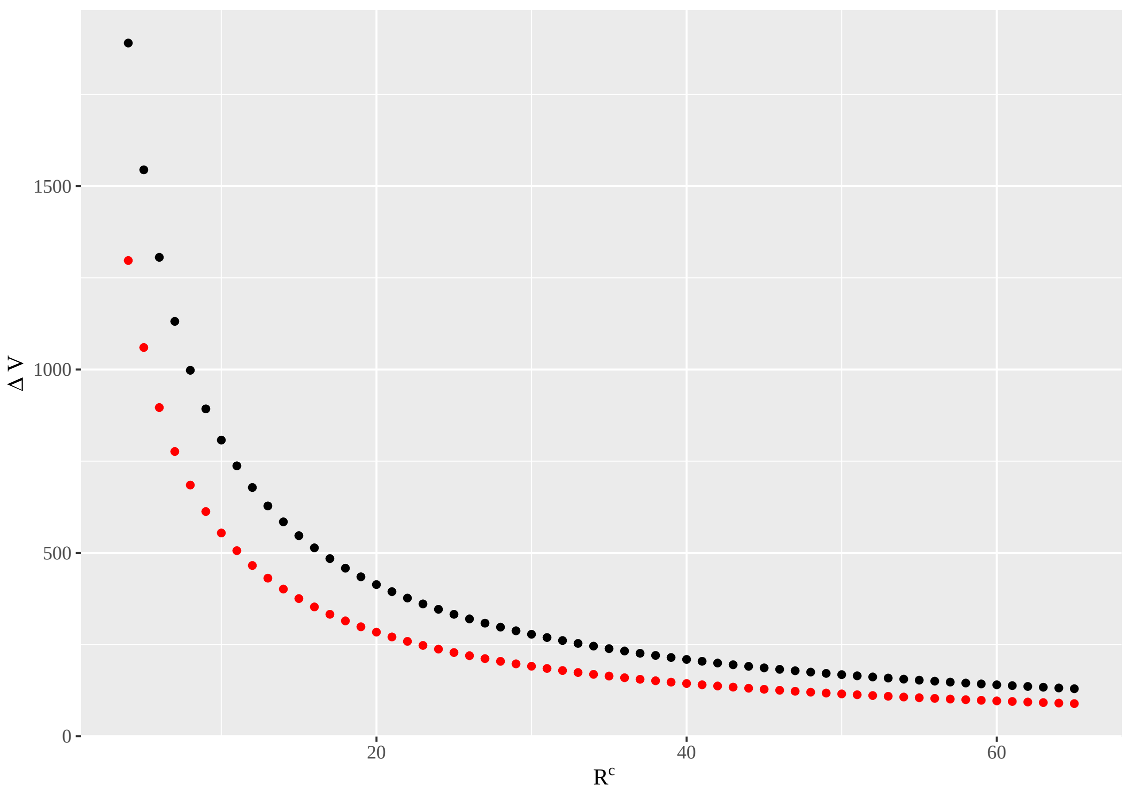


Figure 6 Experiment 1 - Marginal Values

4.3 Experiment 2 - Learning Durations

As previously described the discretization of a continuous state space is typically a trade-off decision. A fine space with many points gives a better approximation of the continuous space but leads to a costly computation in terms of time. A second experiment should elaborate on this objective. For that several execution of the policy iteration were conducted with different discretization steps for the resource variable which in the energy storage case is the battery level. Therefore, the resource space of 0 to $R^c = 16$ was divided into the following parts and the time for finding the optimal policy was monitored.

$$\{11, 21, 31, 41, 51, 61, 71, 81, 91, 101, 111, 121, 131, 141, 151, 161, 171, 181, 191, 201\}$$

The market price in turn can take on the values $MP = \{1, 2, 3, 4, 5\}$. To compute the total amount of states the part size of the resource can simply be multiplied with |MP|.

In one timestep the agent can decide on two different actions. The discretization of the action space with $|a_t^{RG}| = |a_t^{GR}| = 21$ different discrete steps per action results in 441 different actions. A transfer loss of $\eta = 0.8$ was chosen and the discount ration γ was set to 0.9. A max transfer rate of $\delta_{in} = \delta_{out} = 2.5$ was chosen as well as the approximation threshold $\epsilon = 0.01$. Table 4 provides the price transition matrix that was used throughout this experiment.

Figure 7 shows the result of this elaboration. It represents the number of states for one execution on the x-axis as well as the duration in seconds it takes to find the optimal policy for the specific number of states on the y-axis. For 1005 different states which is a relatively small amount of states, it takes about 628 seconds to find the best policy. Moreover, the duration seems to be growing quadratic, hence a quadratic regression model was applied and drawn as a straight line into the same figure.

	1.0	2.0	3.0	4.0	5.0
1.0	0.40	0.30	0.20	0.10	0.00
2.0	0.20	0.40	0.25	0.10	0.05
3.0	0.10	0.20	0.40	0.20	0.10
4.0	0.05	0.10	0.25	0.40	0.20
5.0	0.00	0.10	0.20	0.30	0.40

Table 4

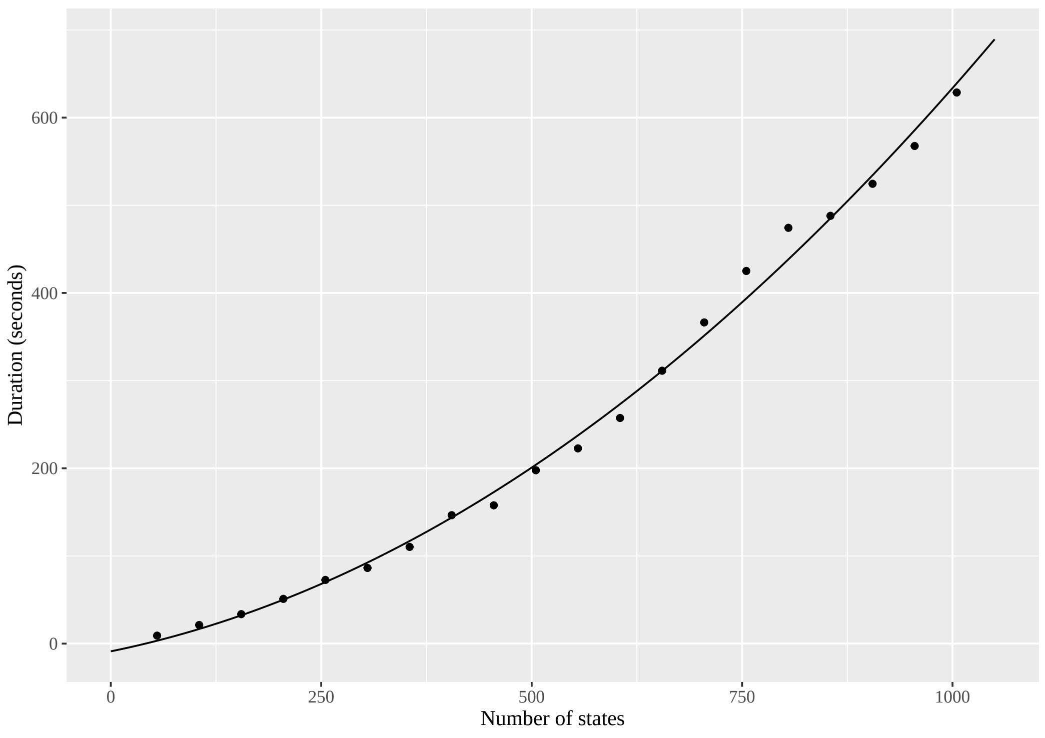


Figure 7 Experiment 2 - Duration for Finding the Optimal Policies

The provided figure gives a glimpse about the applicability of dynamic programming in practice. Even though dynamic programming offers a method to find an optimal policy in polynomial time, it suffers because of the curse of dimensionality. That means that by introducing other state variables the number of states often grows exponential and therefore a policy iteration execution quickly becomes not convenient for real world problems (Bellman 1961; Sutton and Barto 1998).

5 Conclusion and Outlook

The present seminar thesis offered an implementation of the policy iteration algorithm as well as its application on an energy storage problem. First the theoretical foundation of dynamic programming was laid out in chapter 2. In particular that is the Markov Decision Process where an agent decides on actions in any point in time and receives a contribution of its action from the environment it is in. The components of Markov Decision Processes were defined formally as well as the state-value and action-value functions were presented. The two functions indicate how good it is for the agent to be in a certain state respectively how good it is to be in a certain state and decide on an action.

Next, the policy iteration was presented which is an algorithm of the dynamic programming family. At first the algorithm was presented formally with its components defined in the previous chapter and the pseudocode was set up. Furthermore, the substeps policy evaluation and policy improvement were also presented in pseudocode. As a next step the algorithm was implemented to substantiate the theoretical foundation. It was taken care that the theoretical concepts are unambiguously reflected in the code such that the implementation can be taken as a general framework. Therefore it can be used as a skeleton for any other problem instances of any complexity which is solvable by dynamic programming. However, the implementation offers further potentials for optimization. For instance a lookup table for finding a specific state would decrease the execution time.

The implemented policy iteration algorithm of the previous chapter was applied on a simplified energy storage problem where the agent can decide to buy or sell energy at a market in any given point in time by utilizing a battery. At first an overview about the problem was given and the problem was defined formally with the help of a modelling framework described in a previous chapter.

In a first experiment the marginal value of one storage unit was computed. For that two different price probability matrices were used and the values recorded. The result for this was that for both matrices the values increases logarithmic when the battery is increased. Furthermore, when having a price probability matrix which is more "symmetric" the values are higher overall. Logarithmic models were applied to study the marginal values further. In principle a decision maker could use such a model to obtain an optimal battery capacity for her/his energy storage problem.

In a second experiment the duration it takes to find the optimal policy was elaborated. Here it was approved that even though dynamic programming offers a method to find an optimal policy in polynomial time the method suffers due to the curse of dimensionality. This in particular turned out to be a major drawback of a dynamic program since by introducing other state variables the number of states grows exponential and therefore policy iteration quickly becomes not convenient for real world problems.

Besides model-based algorithm like policy iteration and value iteration it might be interesting to investigate how model-free algorithms like Q-Learning works in detail and how they differ from model-based algorithm presented in this seminar thesis. Furthermore, it might be interesting to study what other techniques were used to defeat the world champion in the board game Go.

References

Adda, J., and Cooper, R. 2003. “Dynamic Economics Quantitative Methods and Applications,” *Adda, J and Cooper, RW (2003) Dynamic economics: quantitative methods and applications MIT Press, Cambridge, US ISBN 9780262012010* .

Bellman, R. 1957. *Dynamic Programming*, Rand Corporation research study, Princeton University Press.

Bellman, R. 1961. *Adaptive Control Processes: A Guided Tour*, Princeton Legacy Library, Princeton University Press.

Foster, F. G., and Howard, R. A. 1962. “Dynamic Programming and Markov Processes,” *The Mathematical Gazette* .

Judd, K. L. 1998. “Numerical Methods in Economics,” .

Krueger, P., Griffiths, T., and Russell, S. J. 2017. *Shaping Model-Free Reinforcement Learning with Model-Based Pseudorewards*, Master’s thesis, EECS Department, University of California, Berkeley.

K.Vinotha 2014. “Bellman Equation In Dynamic Programming,” *International Journal of Computing Algorithm* (3).

Littman, M. L. 2001. “Value-function reinforcement learning in Markov games,” *Cognitive Systems Research* (2:1), pp. 55–66.

Pashenkova, E., Rish, I., and Dechter, R. 1996. “Value iteration and Policy iteration algorithms for Markov decision problem,” *Relation* pp. 1–15.

Powell, W., and Meisel, S. 2015. “Tutorial on Stochastic Optimization in Energy—Part I: Modeling and Policies,” *IEEE Transactions on Power Systems* (31), pp. 1–9.

Powell, W. B. 2011. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, Wiley Series in Probability and Statistics, Wiley: Hoboken, NJ, USA, 2nd ed.

Powell, W. B., and Meisel, S. 2016. “Tutorial on Stochastic Optimization in Energy - Part II: An Energy Storage Illustration,” *IEEE Transactions on Power Systems* (31:2), pp. 1468–1475.

Puterman, M. L. 1994. *Markov decision processes: discrete stochastic dynamic programming*, Wiley Series in Probability and Statistics, Wiley-Interscience, 1st ed.

Russell, N. 2002. *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2nd ed.

Rust, J. 1996. “Numerical dynamic programming in economics,” in *Handbook of Computational Economics*, vol. 1, H. M. Amman, D. A. Kendrick, and J. Rust (eds.), Elsevier: New York, NY, USA, chap. 14, pp. 619 – 729.

Sutton, R., and Barto, A. 1998. “Reinforcement Learning: An Introduction,” *IEEE Transactions on Neural Networks* .

Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this Seminar Thesis titled "Implementation of the Policy Iteration Algorithm for an Energy Storage Problem" is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references.

Münster, 3rd July 2020

Sebastian Deisel

Consent Form

for the use of plagiarism detection software to check my thesis

Last name: Deisel **First name**: Sebastian
Student number: 394 951 **Course of study**: Information Systems
Address: Bentelerstr. 39, 48149 Münster
Title of the thesis: “Implementation of the Policy Iteration Algorithm for an Energy Storage Problem”

What is plagiarism? Plagiarism is defined as submitting someone else’s work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as “failed”. This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 3rd July 2020

Sebastian Deisel