NAZARBAYEV UNIVERSITY

School of Sciences and Humanities

Department of Mathematics

Thesis

Solving Linear-Quadratic Regulator Problem with Average-Value-at-Risk Criteria using Approximate Dynamic Programming

Author: Arailym Raikhankyzy

Supervisor: Kerem Ugurlu

21 April, 2024

Contents

0.1 Abstract

This master's thesis explores the intersection of optimal control theory and risk-sensitive decision-making by addressing the finite-horizon discrete-time linear quadratic regulator (LQR) problem with a focus on the average-value-at-risk (AVaR) criteria. The study aims to mathematically formalize the LQR-AVaR problem within the dynamic programming framework and develop a computational algorithm based on approximate dynamic programming techniques to solve it. The algorithm's effectiveness is rigorously assessed through the analysis of experiment results and plot evaluations. The experiment results indicate that the approximate dynamic programming algorithm, when applied properly, performs well for the problem, with experiments suggesting high accuracy.

1 Overview

1.1 Introduction

Optimal control problems have been widely studied and applied in various fields, such as robotics, aerospace, and finance. The linear quadratic regulator (LQR) is a classical control method that has been widely used to solve optimal control problems with quadratic cost or reward functions. However, in real-world applications, the system dynamics and cost functions are often uncertain or stochastic, which can lead to suboptimal performance or even failure of the control system.

To address this challenge, the average value at risk (AVaR) has been proposed as a risk measure to provide robustness to uncertainty and unexpected events. In this thesis, we have chosen to use "dynamic" AVaR instead of a simple AVaR methodology. Dynamic AVaR is preferred due to its ability to take into account the time-varying nature of financial markets and better handle changes in market conditions. Additionally, it provides more flexibility in terms of the range of data that can be incorporated, allowing for a more accurate capture of complex risk patterns. Dynamic AVaR also offers a more accurate estimation of tail risk, which may be missed by a simple AVaR model that assumes a symmetrical distribution of returns. Ultimately, our decision to use dynamic AVaR is based on the belief that it is a more appropriate methodology for providing a robust and accurate estimation of risk in financial markets.

The LQR-AVaR problem is an extension of the classical LQR problem that includes the AVaR risk measure as a constraint. The LQR-AVaR problem can be solved using traditional dynamic programming and optimisation techniques, but the computational complexity can be high, especially for high-dimensional systems.

In this thesis, approximate dynamic programming (ADP) was the method of choice for this optimal control problem due to its discrete and nonlinear nature. For the application of machine learning-based approaches, our problem lacked the data required to train machine learning algorithms. Bellman's principle functions are complex and nonlinear with the inclusion of a new risk measure, making the old optimisation techniques impracticable. These functions could not be minimised by optimisation techniques; hence, a method that could manage the complex dynamics of the issue had to be used. ADP offers a principled framework for addressing the complexities of our problem domain and arriving at precise solutions, since there is no straightforward optimisation path. This thesis aims to address the LQR-AVaR problem by presenting an ADP algorithm and implementing a computer program to solve the problem.

1.2 Literature Review

In recent years, there has been a growing interest in using machine learning techniques for solving optimal control problems. Several studies have proposed different approaches and algorithms to tackle this problem, and in this review, we will summarize some of the related work in this field.

N. Báuerle and J. Ott's study, presented in [1], explores into the problem of minimizing the AVaR of discounted costs across both limited and infinite horizon scenarios. By reducing the complexity of the issue to a standard Markov Decision Process (MDP) and creating the necessary conditions for the existence of an ideal policy, their approach expands the state space as needed. On the basis of this work, N. Báuerle and U. Rieder expand on the research in [2] by examining situations in which exponential utility is employed for risk-sensitive evaluations rather than AVaR.

K. Ugurlu[3] makes more progress by formulating the LQR-AVaR problem, which deals with situations where costs may be unbounded across an indefinite horizon. The presence of an optimal policy is shown by suitable state aggregation and heuristic selection of a global variable s.

Properties of the AVaR and dynamic AVaR are studied by Y. Yoshida in [4] and by Y. Yoshida and S. Kumamoto in [5]. Through dynamic programming, an optimality equation for the optimal average value-at-risks across time is formulated by Y. Yoshida[5]. The study provides optimal portfolio compositions and their associated average value-at-risks as solutions to this equation.

There are currently two popular machine learning methods for approximating the Hamilton-Jacobi-Bellman equation: deep learning and reinforcement learning. The deep learning approach to solving high-dimensional partial differential equations, including the Hamilton-Jacobi-Bellman equation, is studied and implemented using Python by M. R. Rothe[6] for her master thesis. Deep learning approach based on Monte-Carlo sampling for solving stochastic control problems is presented by J. Han and W. E in [7]. Another article written by J. Blechschmidt and O. G. Ernst[8] presents three neural network based method to solve partial differential equations such as Hamilton-Jacobi-Bellman equation. The reinforcement learning method for solving the problem of risk-sensitive Markov Decision Processes is studied by X. Yu[9]. In this paper, they consider maximizing reward problem instead of minimizing risk. The algorithm that they present is developed using deep Q-learning framework. Finally, an approximate dynamic programming algorithm for solving the problem of the curse of dimensionality in large and stochastic optimization problems as the LQR-AVaR problem is presented by M. Mes and A. P. Rivera[10].

1.3 Linear Quadratic Regulator Control Problem

We consider a controlled Markov Decision Process (x_t) in discrete time and a non-negative cost process (C_t). The initial state at time 0 is given by $x_0 = x$. The action (a_t) is chosen from the given controlled constrained action set A. For discrete time $t \in [0, T]$ the next state is given by a transition function $X(x_t, a_t, w_t)$, that is

$$x_{t+1} = X(x_t, a_t, w_t)$$

where $a(x_t, a_t, w_t)$ is a real valued function, $a_t \in A$ is an action at time t. The problem is to minimize the cost

$$C_T^u = \sum_{t=0}^{T-1} c(x_t, a_t) + g(x_T),$$

where $x_0 = x$ is an initial state, $c(x_t, a_t)$ is a cost function at time t and $g(x_T)$ is a terminal cost at time T.

Embed the problem into finding

$$Q(t, x_0) = \inf_{a_t \in A} \Big[\sum_{k=t}^{T-1} c(x_t, a_t) + g(x_T) \Big],$$

where $x_{k+1} = x_k + a_k$, $x_t = x$, $t \leq k \leq T$.

Proposition 1.1 (Hamilton-Jacobi-Bellman equation) *For all* (t, x), $x \in \mathbb{R}$ *and* $0 \leq t \leq T$,

$$\begin{aligned} Q(t, x) &= \inf_{a \in A} \Big[c(x, a, t) + Q(t + 1, X(x, a, t) \Big] \\ Q(T + 1, x) &= g(x). \end{aligned} \tag{1.3.1}$$

For more information about dynamic programming and control problems, please refer to [11].

In our case, we consider Linear Quadratic Regulator control problem (LQR problem) defined as follows:

Definition 1.1 *For a discrete-time* **linear** *system given by*

$$x_{t+1} = Ax_t + Ba_t + w_t, \quad x_0 = x, \quad where \quad t \in [0, 1, 2, ..., T], \quad x \in \mathbb{R}$$

with a noise w_t *(i.i.d.) and a* **quadratic** *cost function defined as*

$$J(0, x) = E\Big[x_T^T Q_T x_T + \sum_{t=0}^{T-1} \big(x_t^T Q x_t + a_t^T R a_t \big) \Big].$$

The goal is to find the optimal control sequence minimizing the cost function.

1.4 Average Value at Risk

Instead of minimizing the expected value of the cost function we will use Average-Value-at-Risk which is a more comprehensive measure of risk that measures the expected value of the worst-case scenario.

Definition 1.2 *Let X be a real-valued random variable and let α be a discount factor such that $\alpha \in (0, 1)$.*

The Average-Value-at-Risk of X at level α, denoted by $AVaR_\alpha(X)$ is defined by

$$AVaR_\alpha(X) = \mathbb{E}\big[X|X \geq VaR_\alpha(X)\big],$$

where $VaR_\alpha(X)$ is the Value-at-Risk of X at level α, defined by

$$VaR_\alpha(X) = \inf\big\{x \in \mathbb{R} : \mathbb{P}(X \leq x) \geq \alpha\big\}.$$

To reduce the complexity of computing AVaR both in the code and in the experimental problems, we represent it as the solution of a convex optimization problem, as shown in the lemma given by R. T. Rockafellar and S. Uryasev[12].

Lemma 1.1 *Let X be a real-valued random variable and let $\alpha \in (0, 1)$. Then*

$$AVaR_\alpha(X) = \min_{\forall s \in \mathbb{R}} \left\{ s + \frac{1}{1-\alpha}\mathbb{E}[(X-s)^+] \right\} \tag{1.4.1}$$

and the minimum is given by

$$s^* = VaR_\alpha(X) = inf\big\{x \in \mathbb{R} : P(X \leq x) \geq \alpha\big\}. \tag{1.4.2}$$

The following properties of AVaR is given in [4].

Lemma 1.2 *For $\alpha \in [0,1]$ and real-valued random variables X and Y, the Average-Value-at-Risk has the following properties:*

1. *Coherence: sub-additive*

$$AVaR_\alpha(\sum_{i=1}^{n} X_i) \leq \sum_{i=1}^{n} AVaR_\alpha(X_i)$$

   and translation-invariant

$$AVaR_\alpha(X + c) = AVaR_\alpha(X) + c, \text{ for } c \in \mathbb{R}.$$

2. *Monotonicity: if $X \leq Y$, then*

$$AVaR_\alpha(X) \leq AVaR_\alpha(Y).$$

3. *Positive homogeneity:*

$$AVaR_\alpha(X) + AVaR_\alpha(Y) \leq AVaR_\alpha(X + Y).$$

The dynamic AVaR, which is AVaR evaluated with respect to conditional expectation, has the following properties[5]:

Lemma 1.3 *Let $\alpha \in [0,1]$ and X, Y and Z be real-valued random variables. Assume X and Z are independent. Then*

1. $AVaR_\alpha(X|Z) = AVaR_\alpha(X)$.
2. $AVaR_\alpha(Y|Z) = Y$.
3. $AVaR_\alpha(X + Y|Z) = AVaR_\alpha(X) + Y$.

1.5 LQR-AVaR problem

The main objective of the problem is to find the optimal control, denoted by a_t^* for $t \in \{0, ..., T\}$, for the problem

$$\min_{a_t} AVaR_\alpha(c(x_t, a_t)|x_t, a_t), \text{ for } 0 \leq t \leq T,$$

where

$$x_t = Ax_t + Ba_t + w_t, \quad x_0 = x, \quad t \in \{0, 1, 2, ..., T\},$$
$$c(x_t, a_t) = \sum_{t=0}^{T} \left(x_t^T Q x_t + a_t^T R a_t \right),$$

given a set of admissible actions A and a random variable w_t. Here A, B and Q, R are parameters of choice for different problems.

2 Methods

2.1 Approximate Dynamic Programming

Dynamic programming breaks down complex Markov Decision Processes (MDPs) based optimal control problems into smaller, easier to handle subproblems. The goal is to solve these smaller problems in order to find the best possible policy or set of actions for the MDP overall. But because of the infamous "curse of dimensionality," calculating the exact solution—which is often accomplished through backward dynamic programming—proves difficult and sometimes impossible for large-scale issues. To address this, Approximate Dynamic Programming (ADP) is introduced as a modelling paradigm based on the MDP framework, providing a range of methods to overcome the dimensionality problem in large-scale, multi-period stochastic optimisation problems.

ADP is a method used to solve complex stochastic optimization problems, particularly in the field of control theory. It is an iterative approach that seeks to find an optimal solution by breaking down the problem into smaller subproblems and solving each one in a recursive manner. The term "approximate" in ADP indicates that the method is not always guaranteed to find the exact optimal solution, but rather a solution that is close enough to the optimal solution within a specified tolerance level.

One of the key advantages of ADP is its ability to handle large-scale optimization problems that would be computationally intensive to solve exactly. By breaking the problem down into smaller subproblems and solving them iteratively, ADP can provide near-optimal solutions in a more manageable amount of time. This is especially beneficial when dealing with systems that have a large number of states or inputs, or when the system dynamics are complex and difficult to model.

The ADP framework is particularly suitable for problems with a finite horizon, such as the finite horizon discrete-time linear quadratic regulator (LQR) problem. In the context of LQR, ADP is used to find the optimal control input sequence that minimizes a cost function over a finite time horizon, subject to the system dynamics. The cost function typically includes terms for state deviation, input size, and final state deviation, and the goal is to minimize the total cost over the entire time horizon.

2.2 Algorithm Description and Implementation

In this section, we will delve into the intricacies of the approximate dynamic programming (ADP) algorithm as implemented within this thesis. Originally proposed by M. Mes and A. P. Rivera[10], the ADP algorithm represents a value-based approach tailored to tackle stochastic optimization problems effectively.

The ADP algorithm operates on the principle of iteratively solving Bellman's equations for individual states at each stage. It accomplishes this by utilizing estimates of downstream values and conducting iterative updates to refine these estimations. The algorithm takes as input the initial state x_t, the admissible set of actions A, the set of random variables w_t, the discount factor or risk averseness α, and the terminal time T. Additionally, it allows the learning of the hyperparameters such as the number of iterations N and the learning rate β.

At its core, the algorithm aims to yield the optimal actions and corresponding values for each time step $t \in \{0, 1, ..., T\}$. Notably, the code incorporates a built-in function capable of computing both the expected cost value when $\alpha = 0$ and the average-value-at-risk (AVaR) for varying α values.

The ADP algorithm consists of two main stages: the forward pass and the backward pass. During the forward pass, random actions a_t and random variables w_t are selected to construct a sample path, which is then stored as states $x_{t+1} = x_t + a_t + w_t$. Subsequently, in the backward pass, these generated sample paths are utilized to iteratively update the values at each iteration, refining the approximation of the optimal solution.

This approach not only facilitates efficient exploration of the solution space but also enables the algorithm to adapt and learn from the dynamics of the system, ultimately yielding robust and effective solutions to stochastic optimization problems.

Algorithm 1: ADP algorithm for solving LQR-AVaR Problem

Input : $x_0, A, w_t, \alpha, T, \beta, N$
Output: $J(t, x_t)$ for $t \in \{0, 1, ..., T\}$

Step 0: Initialization

Step 0a: Choose an initial approximation $J(t, x_t)$ for $t \in \{0, ..., T\}$.
Step 0b: Choose the number of iterations N.
Step 0c: Set the initial state to x_0.

for $n = 1, 2, ..., N$ **do**
  Step 1: Forward Pass
  for $t = 0, ..., T$ **do**
    Step 1a: Create a sample path by choosing random (a_t, w_t) and update the states $x_{t+1} = x_t + a_t + w_t$;
  end
  Step 2: Backward Pass
  for $t = T, T-1, ..., 1$ **do**
    Step 2a: Compute $\tilde{J}(t, x_t)$ using the state x_t from the forward pass:

$$\tilde{J}(t, x_t) = c(x_t, a_t) + AVaR(\tilde{J}(t+1, x_{t+1})), \text{ with } \tilde{J}(T+1, x_{T+1}) = 0;$$

    Step 2b: Update the approximation $J(t, x_t)$ for state x_t using

$$J(t, x_t) = (1-\beta) * \tilde{J}(t, x_t) + \beta * J(t, x_t);$$

  end
end

Step 3: Return $J(t, x_t)$ for $t \in \{0, 1, ..., T\}$

3 Experiments

In this section, we will solve problems using experimental scenarios using dynamic programming techniques and compare the findings with the results returned by the code in order to assess its performance, since there are no data or accuracy metrics available to support the numerical outcomes offered by the algorithm. We will also do plot analysis to show that the suggested ADP algorithm validates the expected dynamics and trends in the LQR-AVaR problem.

3.1 Experimental Problems

In the following calculations we will use formulas 1.4.1 and 1.4.2 to calculate the $AVaR_\alpha$,

$$AVaR_\alpha(X) = \min_{\forall s \in \mathbb{R}} \left\{ s + \frac{1}{1-\alpha} \mathbb{E}[(X-s)^+] \right\},$$

$$s^* = VaR_\alpha(X) = inf\big\{x \in \mathbb{R} : P(X \leq x) \geq \alpha\big\},$$

and the following representation of the Bellman's principle, given by proposition 1.3.1, will be used for easier calculations:

$$J(t, x_t) = \inf_{a_t} Q(t, x_t, a_t),$$

$$\begin{aligned} Q(t, x_t, a_t) =& c(x_t, a_t) + AVaR_\alpha\big(J(t+1, x_{t+1})|x_t, a_t\big) = \\ =& c(x_t, a_t) + \inf_{s \in \mathbb{R}} \left\{ s + \frac{1}{1-\alpha} \mathbb{E}[(J(t+1, x_{t+1}) - s)^+ | x_t, a_t] \right\} = \\ =& c(x_t, a_t) + s^* + \frac{1}{1-\alpha} \mathbb{E}[(J(t+1, x_{t+1}) - s^*)^+ | x_t, a_t]. \end{aligned}$$

Problem 1: LQR-AVaR Problem with $\alpha = 0.25$. Given a linear transition function $x_{t+1} = x_t + a_t + w_t$ and a quadratic cost function $c(x_t, a_t) = x_t^2 + a_t^2$, where the random variable w_t is Bernoulli and given by

$$w_t = \begin{cases} 1, \text{ with } p = 0.5 \\ -1, \text{ with } p = 0.5 \end{cases}$$

with the initial state $x_0 = 1$, the set of admissible actions $A = \{-1, -0.5, 0, 0.5, 1\}$. Minimize $AVaR_\alpha\Big(\sum_{t=0}^T c(x_t, a_t)\Big)$ over $a_t \in A$ when $\alpha = 0$.

Solution: Note that $AVaR_{\alpha=0}\Big(\sum_{t=0}^T c(x_t, a_t)\Big) = \mathbb{E}\Big[\sum_{t=0}^T c(x_t, a_t)\Big]$.

When T=1:

For $t = 1$,

$$J(1, x_1) = \inf_{a_1} \big\{\mathbb{E}\big[x_1^2 + a_1^2 | x_1, a_1\big]\big\} = \inf_{a_1} \big\{x_1^2 + a_1^2\big\} = x_1^2,\ a_1^* = 0.$$

Here, the conditional expectation $\mathbb{E}\big[x_1^2 + a_1^2 | x_1, a_1\big]$ simplifies to $x_1^2 + a_1^2$ because, given the fixed values of x_1 and a_1, they act as constants in the computation. Also, a_1^* is the optimal action at time $t = 1$.

For $t = 0$,

$$\begin{aligned} J(0, x_0) =& \inf_{a_0} \big\{x_0^2 + a_0^2 + \mathbb{E}\big[(x_0 + a_0 + w)^2 | x_0, a_0\big]\big\} = \\ =& \inf_{a_0} \Big\{x_0^2 + a_0^2 + \frac{1}{2}(x_0 + a_0 + 1)^2 + \frac{1}{2}(x_0 + a_0 - 1)^2\Big\} = \\ =& \inf_{a_0} \big\{2x_0^2 + 2a_0^2 + 2x_0 a_0 + 1\big\} = \\ =& 2x_0^2 + 1 + 2\inf_{a_0} \big\{a_0^2 + x_0 a_0\big\}. \end{aligned}$$

Here, we need to find the infimum of the function $\phi(a_0) = a_0^2 + x_0 a_0$. The function attains its infimum point $\phi(a_0) = -0.25$ when $a_0^* = -0.5$.

$$J(0, x_0 = 1) = 2 * 1^2 + 1 + 2 * (-0.25) = 2.5.$$

In summary,

$$J(1, -1) = 1, J(1, -0.5) = 0.25, J(1, 0) = 0, J(1, 0.5) = 0.25, J(1, 1) = 1,$$
$$J(1, 1.5) = 2.25, J(1, 2) = 4, J(1, 2.5) = 6.25, J(1, 3) = 9,$$
$$J(0, 1) = \mathbf{2.5}.$$

When T=2:

For $t = 2$,

$$J(2, x_2) = \inf_{a_2} \left\{ \mathbb{E}\left[x_2^2 + a_2^2 | x_2, a_2\right] \right\} = \inf_{a_2} \left\{ x_2^2 + a_2^2 \right\} = x_2^2,\ a_2^* = 0.$$

For $t = 1$,

$$\begin{aligned} J(1, x_1) &= \inf_{a_1} \left\{ x_1^2 + a_1^2 + \mathbb{E}\left[(x_1 + a_1 + w)^2 | x_1, a_1\right] \right\} = \\ &= \inf_{a_1} \left\{ x_1^2 + a_1^2 + \frac{1}{2}(x_1 + a_1 + 1)^2 + \frac{1}{2}(x_1 + a_1 - 1)^2 \right\} = \\ &= \inf_{a_1} \left\{ 2x_1^2 + 2a_1^2 + 2x_1 a_1 + 1 \right\} = 2x_1^2 + 1 + 2 \inf_{a_1} \left\{ a_1^2 + x_1 a_1 \right\}. \end{aligned}$$

Here, we need to find the infimum of the function $\phi(a_1) = a_1^2 + x_1 a_1$. By analysing each possible cases graphically we derive the followings:

$$x_1 \in [-1, -0.5),\ a_1^* = 0.5,\ J(1, x_1) = 2x_1^2 + x_1 + 1.5$$
$$x_1 \in [-0.5, 0.5),\ a_1^* = 0,\ J(1, x_1) = 2x_1^2 + 1.$$
$$x_1 \in [0.5, 1.5),\ a_1^* = -0.5,\ J(1, x_1) = 2x_1^2 - x_1 + 1.5.$$
$$x_1 \in [1.5, 3],\ a_1^* = -1,\ J(1, x_1) = 2x_1^2 - 2x_1 + 3.$$

For t=0,

$$
\begin{aligned}
&Q(0,1,-1) = 1^2 + (-1)^2 + \frac{1}{2}\big(J(1,1) + J(1,-1)\big) = 4.5.\\
&Q(0,1,-0.5) = 1^2 + (-0.5)^2 + \frac{1}{2}\big(J(1,1.5) + J(1,-0.5)\big) = \mathbf{4.25}.\\
&Q(0,1,0) = 1^2 + 0^2 + \frac{1}{2}\big(J(1,2) + J(1,0)\big) = 5.\\
&Q(0,1,0.5) = 1^2 + 0.5^2 + \frac{1}{2}\big(J(1,2.5) + J(1,0.5)\big) = 7.25.\\
&Q(0,1,1) = 1^2 + 1^2 + \frac{1}{2}\big(J(1,2.5) + J(1,0.5)\big) = 10.75.
\end{aligned}
$$

The minimizing action for time $t = 0$ is $a_0^* = -0.5$ and the optimal value is $J(0,1) = 4.25$.

In summary,

$$
\begin{aligned}
&J(2,-3) = 9, J(2,-2.5) = 6.25, J(2,-2) = 4, J(2,-1.5) = 2.25, J(2,-1) = 1,\\
&J(2,-0.5) = 0.25, J(2,0) = 0, J(2,0.5) = 0.25, J(2,1) = 1, J(2,1.5) = 2.25,\\
&J(2,2) = 4, J(2,2.5) = 6.25, J(2,3) = 9, J(2,3.5) = 12.25, J(2,4) = 16,\\
&J(2,4.5) = 20.25, J(2,5) = 25,\\
&J(1,-1) = 2.5, J(1,-0.5) = 1.5, J(1,0) = 1, J(1,0.5) = 1.5, J(1,1) = 2.5,\\
&J(1,1.5) = 4.5, J(1,2) = 7, J(1,2.5) = 10.5, J(1,3) = 15,\\
&J(0,1) = \mathbf{4.25}.
\end{aligned}
$$

The numerical results achieved by this calculation perfectly matches with the outputs generated by the code. The dynamics of the optimal values given by code over the iterations are shown in Figure 1.

Problem 2: LQR-AVaR Problem with $\alpha = 0.25$. Given a linear transition function $x_{t+1} = x_t + a_t + w_t$ and a quadratic cost function $c(x_t, a_t) = x_t^2 + a_t^2$, where

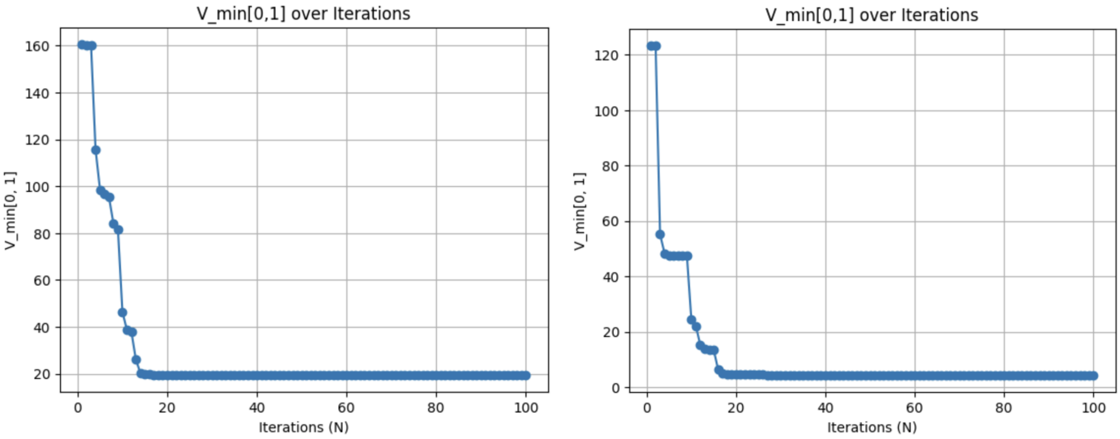

Figure 1: The dynamics of the optimal value $J(0,1)$ over $N = 100$ iterations for Problem 1 and 2.

the random variable w_t is Bernoulli and given by

$$w_t = \begin{cases} 1, \text{ with } p = 0.5 \\ -1, \text{ with } p = 0.5 \end{cases}$$

with the initial state $x_0 = 1$, the set of admissible actions $A = \{-2, 2\}$, and the terminal time $T = 2$. Minimize $AVaR_\alpha\Big(\sum_{t=0}^{T} c(x_t, a_t)\Big)$ over $a_t \in A$ when $\alpha = 0.25$.

Solution:

For $t = 2$,

$$J(2, x_2) = \inf_{a_2} \left\{AVaR_{0.25}\big(x_2^2 + a_2^2 | x_2, a_2\big)\right\} = \inf_{a_2} \left\{x_2^2 + a_2^2\right\} = x_2^2 + 4, a_2^* = -2 \text{ or } 2.$$

For $t = 1$,

$$\begin{aligned} Q(1, x_1, a_1) =& x_1^2 + a_1^2 + s^* + \frac{1}{1 - 0.25}\mathbb{E}[((x_1 + a_1 + w_1)^2 + 4 - s^*)^+ | x_1, a_1] = \\ =& x_1^2 + a_1^2 + s^* + \frac{4}{3}\bigg(\frac{1}{2}((x_1 + a_1 + 1)^2 + 4 - s^*)^+ + \\ &+ \frac{1}{2}((x_1 + a_1 - 1)^2 + 4 - s^*)^+\bigg). \end{aligned}$$

Given $x_0 = 1$ and $A = \{-2, 2\}$, the set of possible states for x_1 is $\{-2, 0, 2, 4\}$.

Now, we will find the optimal values at time $t = 1$ separately for each cases of x_1.

For $x_1 = -2$,

$$a_1 = -2,\ s^* = VaR_{0.25}((-2-2+w)^2+4) = 17,\ Q(1,-2,-2) = 33,$$
$$a_1 = 2,\ s^* = VaR_{0.25}((-2+2+w)^2+4) = 5,\ Q(1,-2,2) = \mathbf{13}.$$

For $x_1 = 0$,

$$a_1 = -2,\ s^* = VaR_{0.25}((0-2+w)^2+4) = 7,\ Q(1,0,-2) = \mathbf{15},$$
$$a_1 = 2,\ s^* = VaR_{0.25}((0+2+w)^2+4) = 7,\ Q(1,0,2) = \mathbf{15}.$$

For $x_1 = 2$,

$$a_1 = -2,\ s^* = VaR_{0.25}((2-2+w)^2+4) = 5,\ Q(1,2,-2) = \mathbf{13},$$
$$a_1 = 2,\ s^* = VaR_{0.25}((2+2+w)^2+4) = 17,\ Q(1,2,2) = 33.$$

For $x_1 = 4$,

$$a_1 = -2,\ s^* = VaR_{0.25}((4-2+w)^2+4) = 7,\ Q(1,4,-2) = \mathbf{31},$$
$$a_1 = 2,\ s^* = VaR_{0.25}((4+2+w)^2+4) = 35,\ Q(1,4,2) = 67.$$

For t=0,

$$J(0,x_0) = \inf_{a_0}\left\{x_0^2 + a_0^2 + AVaR_{0.25}\big(J(1,x_1)|x_0,a_0\big)\right\},\ x_0 = 1.$$
$$a_0 = -2,\ s^* = VaR_{0.25}(J(1,1+2+w)) = 17.5,\ Q(0,1,2) = 31.5$$
$$a_0 = 2,\ s^* = VaR_{0.25}(J(1,1-2+w)) = 13.5,\ Q(0,1,-2) = \mathbf{19.5}$$

In summary,

$J(2,-5)=29, J(2,-3)=13, J(2,-1)=5, J(2,1)=5, J(2,3)=13, J(2,5)=29,$

$J(1,-2)=13, J(1,0)=15, J(1,2)=13, J(1,4)=31,$

$J(0,1)=\mathbf{19.5}.$

For this problem the results obtained matches exactly with the outputs of the code as well.

Problem 3: LQR-AVaR with the standard normal random variable w_t and $\alpha=0$. Given the linear transition function $x_{t+1}=x_t+a_t+w_t$ and a quadratic cost function $c(x_t,a_t)=x_t^2+a_t^2$, where the random variable w_t is standard normal and given by

$$w_t=\begin{cases}-0.14, \text{ with } p=1/3\\ -0.17, \text{ with } p=1/3\\ -0.11, \text{ with } p=1/3\end{cases}$$

with the initial state $x_0=1$, the set of admissible actions $A=\{-1,0,1\}$, and the terminal time $T=2$. Minimize $AVaR_\alpha\Big(\sum_{t=0}^{T}c(x_t,a_t)\Big)$ over $a_t\in A$ when $\alpha=0$.

Solution: For $t=2$,

$$J(2,x_2)=\inf_{a_2}\left\{\mathbb{E}\left[x_2^2+a_2^2|x_2,a_2\right]\right\}=\inf_{a_2}\left\{x_2^2+a_2^2\right\}=x_2^2, a_2=0.$$

x_1	-0.14	-0.17	-0.11	0.83	0.86	0.89	1.83	1.86	1.89
$J(1, x_1)$	0.10	0.13	0.07	1.17	1.26	1.35	4.83	4.98	5.13

Table 1: Values of $J(1, x_1)$ for each x_1

For $t = 1$,

$$
\begin{aligned}
J(1, x_1) &= \inf_{a_1} \big\{x_1^2 + a_1^2 + \mathbb{E}\big[(x_1 + a_1 + w)^2 | x_1, a_1\big]\big\} = \\
&= \inf_{a_1} \bigg\{x_1^2 + a_1^2 + \frac{1}{3}(x_1 + a_1 - 0.14)^2 + \frac{1}{3}(x_1 + a_1 - 0.17)^2 + \\
&+ \frac{1}{3}(x_1 + a_1 - 0.11)^2\bigg\} = \\
&= \inf_{a_1} \big\{2x_1^2 + 2a_1^2 + 2x_1a_1 - 0.28x_1 - 0.28a_1 + 0.0202\big\} = \\
&= 2x_1^2 - 0.28x_1 + 0.0202 + 2 * \inf_{a_1} \big\{a_1^2 + x_1a_1 - 0.14a_1\big\}.
\end{aligned}
$$

Here, we need to find the infimum of the function $\phi(a_1) = a_1^2 + x_1a_1 - 0.14a_1$. By analysing each possible cases we get the followings:

$$
\begin{aligned}
&x_1 \in [-0.17, 1.14),\ a_1 = 0,\ J(1, x_1) = 2x_1^2 - 0.28x_1 + 0.0202. \\
&x_1 \in [1.41, 1.89],\ a_1 = -1,\ J(1, x_1) = 2x_1^2 - 0.28x_1 + 2.3002.
\end{aligned}
$$

For t=0,

$$
\begin{aligned}
Q(0, 1, -1) &= 1^2 + (-1)^2 + \frac{1}{3}\big(J(1, -0.11) + J(1, -0.14) + J(1, -0.17)\big) = \mathbf{2.1}. \\
Q(0, 1, 0) &= 1^2 + 0^2 + \frac{1}{3}\big((J(1, 0.89) + J(1, 0.86) + J(1, 0.83)\big) = 2.26. \\
Q(0, 1, 1) &= 1^2 + 1^2 + \frac{1}{3}\big((J(1, 1.89) + J(1, 1.86) + J(1, 1.83)\big) = 6.98.
\end{aligned}
$$

Thus, the optimal value is $J(0, 1) = \mathbf{2.1}$.

The results of this problem demonstrate the effectiveness of the proposed code, even when applied to standard normal random variables instead of Bernoulli distributions. The evolution of optimal values over the course of iterations is illustrated in

Figure 2. As depicted in this graph, larger random variables require a greater number of iterations to converge to an optimal value compared to the previous problems.

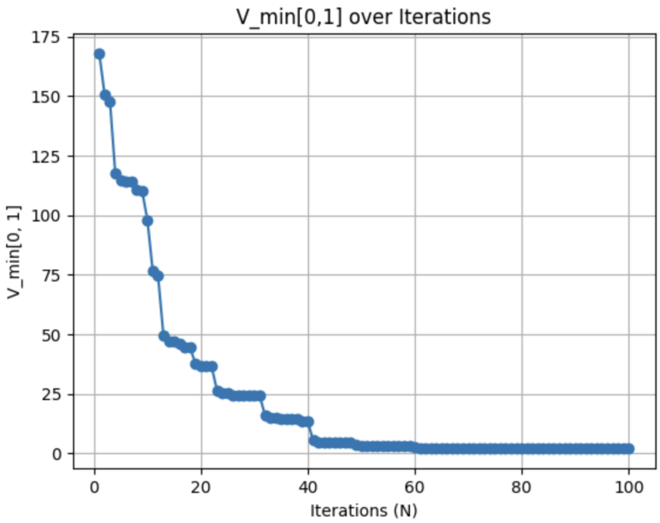

Figure 2: The dynamics of the optimal value $J(0,1)$ over $N = 100$ iterations for Problem 3.

3.2 Plot Analysis

In this plot analysis, our aim is to validate four crucial trends present in the LQR-AVaR problem, as demonstrated by the output results obtained from the proposed ADP algorithm-based code.

1. As the risk aversion parameter α increases, the optimal value $J(0,1)$ is expected to increase correspondingly.
2. With an increase in the terminal time T, the optimal values $J(0,1)$ should also rise.
3. As the available action set A expands, the optimal values $J(0,1)$ are anticipated to decrease.
4. As the terminal time T increases, a larger number of iterations N is necessary for the convergence of the optimal values $J(0,1)$.

In Figure 3, the plot illustrates problem 2 for different values of α. It is evident that as α rises, the optimal value $J(0,1)$ steadily increases. This trend can be ascribed to the growing inclination towards risk reduction, which encourages the selection of strategies that offer greater protection against potential losses. Consequently, this cautious approach tends to favor actions associated with higher expected values, resulting in an overall increase in the optimal value.

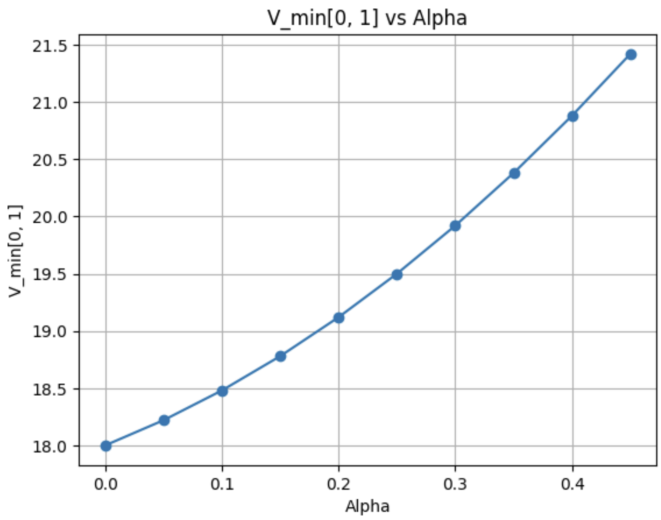

Figure 3: Optimal values $J(0,1)$ across varying α values when $T = 2$

In Figure 4, two graphs illustrate the variation of optimal values $J(0,1)$ as the terminal time changes for problem 2, with a fixed number of iterations $N = 100$. The initial graph depicts a linear increase in the optimal value $J(0,1)$ up to $T = 8$. In the subsequent graph, we observe that as time progresses, the linear trend begins to fluctuate. This deviation occurs because, with increasing time, the number of iterations N required for convergence also increases. Therefore, with $N = 100$, the number of iterations is insufficient for convergence of the values beyond $T = 8$, leading to the observed fluctuations in the line graph. This phenomenon is further demonstrated in Figure 6.

In Figure 5, we present the optimal values of problem 2, showcasing various samples of the action set A drawn from the interval $[-1,1]$, with different step sizes

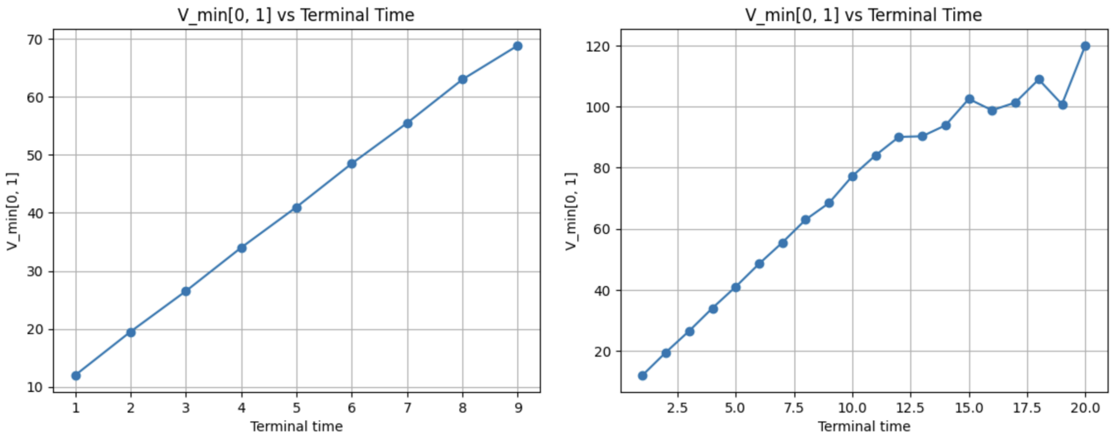

Figure 4: Optimal values $J(0, 1)$ across different terminal time T

denoted by state_delta. As state_delta increases, the action set A becomes smaller.

For the state_delta $= 0.25$, the action set $A = \{-1, -0.75, -0.5, 0, 0.5, 0.75, 1\}$

For the state_delta $= 0.5$, the action set $A = \{-1, -0.5, 0, 0.5, 1\}$

For the state_delta $= 1$, the action set $A = \{-1, 0, 1\}$

For the state_delta $= 2$, the action set $A = \{-1, 1\}$

For the state_delta $= 0.25$, we have the largest set A, whereas for state_delta $=$ 2, the set reduces to its smallest form $\{-1, 1\}$. Notably, the optimal values $J(0, 1)$ exhibit a decreasing trend as state_delta decreases and the action set A expands.

Figure 6 illustrates how the number of iterations N required for convergence steadily grows as the terminal time T increases from 1 to 6. However, at $T = 6$, a jump appears in the interval $t \in (20, 40)$, which is marked by a red arrow. The unusual behaviour arises from the insufficiency of iterations: at the terminal time $T = 6$, the convergence of the optimal values $J(0, 1)$ is not achieved at $N = 100$. Thus, in this case, it is necessary to increase the number of iterations N for the convergence of the values.

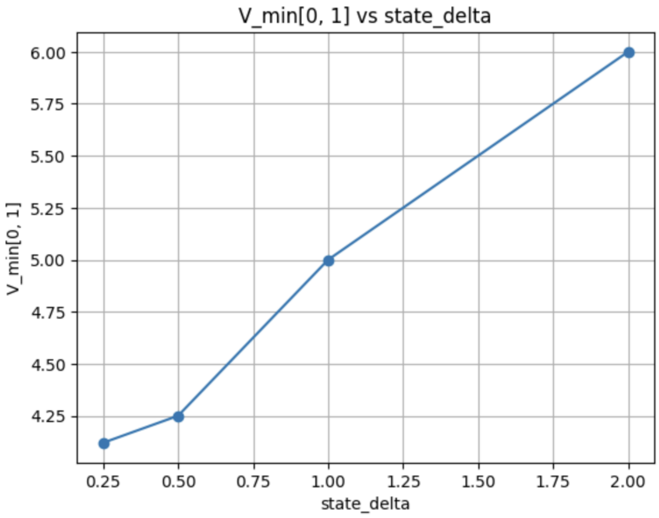

Figure 5: Optimal values $J(0,1)$ across different action set A values when $T = 2$

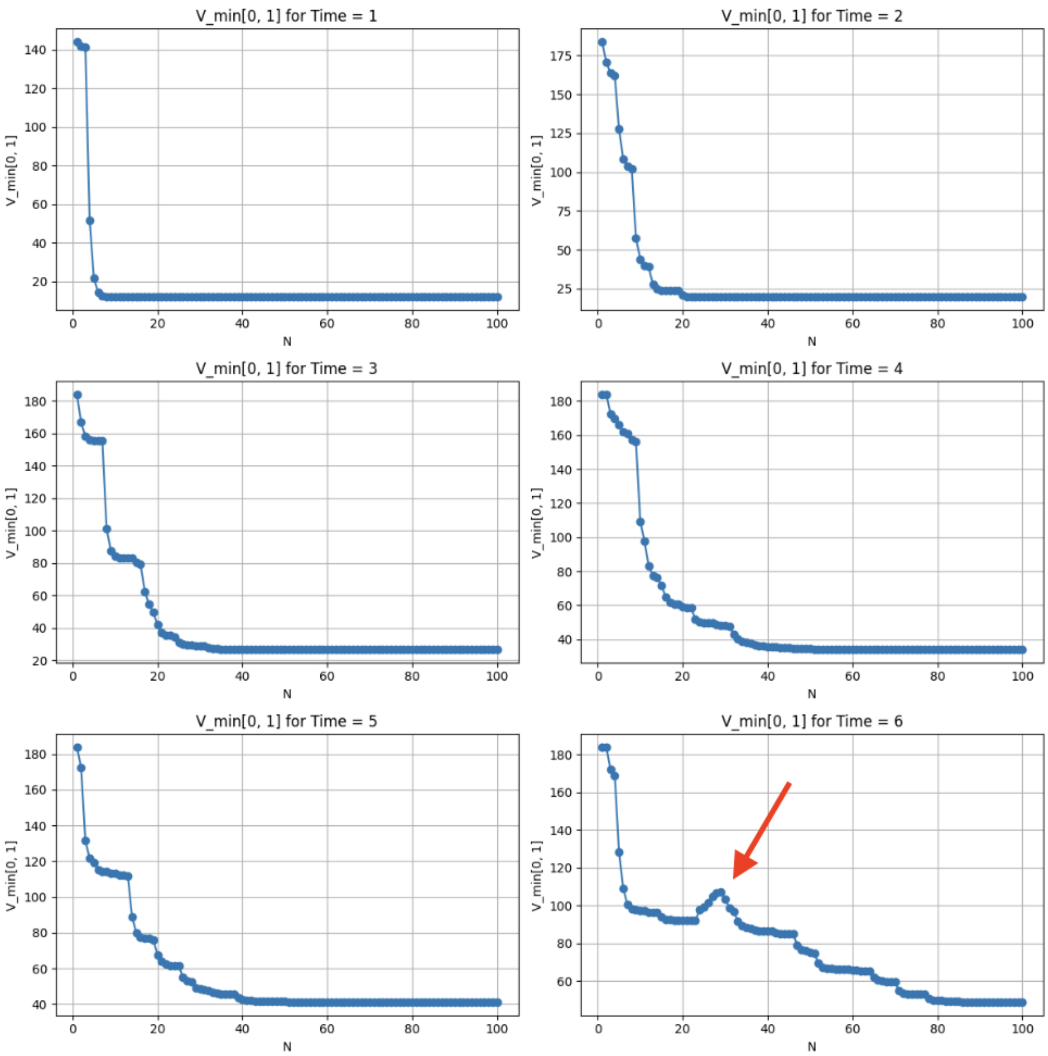

Figure 6: Optimal values $J(0,1)$ across different terminal times T when the maximum iteration number is $N = 100$ and $\alpha = 0.25$

4 Conclusion

In this thesis, we delved into addressing the linear quadratic regulator problem with average-value-at-risk criteria (LQR-AVaR) through the application of approximate dynamic programming (ADP). Initially, we formulated the problem using dynamic programming principles and solved several experimental problems manually. Subsequently, using Python, we developed an ADP algorithm-based code and did a comparison analysis of the outcomes with those of the experimental cases. The thorough analysis indicated a remarkably high level of accuracy, with the graphical representations closely mirroring the anticipated trends and dynamics present in the LQR-AVaR problem.

References

[1] Bäuerle, N. and Ott, J. "Markov Decision Processes with Average-Value-at-Risk criteria". In: *Mathematical Methods of Operations Research* 74 (2011), pp. 361–379. DOI: `http://dx.doi.org/10.1007/s00186-011-0367-0`.

[2] Bäuerle, N. and Rieder, U. "More Risk-Sensitive Markov Decision Processes". In: *Mathematical Methods of Operations Research* 39(1) (2014), pp. 105–120. DOI: `http://dx.doi.org/10.1287/moor.2013.0601`.

[3] K. Uğurlu. "Controlled Markov Decision Processes with AVaR criteria for unbounded costs". In: *Journal of Computational and Applied Mathematics 319* (2016), pp. 24–37.

[4] Y. Yoshida. "An Ordered Weighted Average with a Truncation Weight on Intervals". In: *Springer-Verlag Berlin Heidelberg* (2012), pp. 45–55.

[5] Y. Yoshida and S. Kumamoto. "Dynamic Average Value-at-Risk Allocation on Worst Scenarios in Asset Management". In: *Springer Nature Switzerland* (2019), pp. 674–683.

[6] M. R. Rothe. "Numerical Experiments on a Deep Learning Approach to Solving High Dimensional Partial Differential Equations". In: *Norwegian University of Science and Technology, Master's thesis in Applied Physics and Mathematics* (2020).

[7] Jiequn Han and Weinan E. "Deep Learning Approximation for Stochastic Control Problems". In: *Beijing Institute of Big Data Research* (2016).

[8] J. Blechschmidt and O. J. Ernst. "Three ways to solve partial differential equations with neural networks—Areview". In: *Gesellschaft für Angewandte Mathematik und Mechanik* (2021). DOI: http://dx.doi.org/10.1002/gamm.202100006.

[9] X. Yu and S. Shen. "Risk-Averse Reinforcement Learning via Dynamic Time-Consistent Risk Measures". In: *IEEE Conference Paper* (2022). DOI: http://dx.doi.org/10.1109/CDC51059.2022.9992450.

[10] Martijn Mes and Arturo Perez Rivera. "Approximate Dynamic Programming by Practical Examples". In: *Beta Working Paper series 495* (2016).

[11] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. 2017.

[12] R. Tyrrell Rockafellar and Stanislav Uryasev. "Conditional value-at-risk for general loss distributions". In: *Journal of Banking and Finance* (2002).

5 Appendix

The Code

```
import numpy as np
import tensorflow as tf
import time
import random
import matplotlib.pyplot as plt

def cost(state, action):
  cost = (state ** 2) + (action ** 2)
  cost = np.round(cost, 2)

  return cost

def random_next_element(state, action, ran):
  next_state = round(state + action + ran, 2)

  return next_state

def V_hash(V_table, time, state, value):
  key = (time, state)
  V_table[key] = value

  return V_table

def V_lookup(V_table, time, state):
  key = (time, state)
  value = V_table[key]
```

```python

  return value

def init_cost(state, time):
  max_action = 2
  max_random = 2
  terminal = 5
  val = state**2 + max_action**2
  for i in range(0, terminal - time + 1):
    next_state = state + max_random + max_action
    tmp = next_state**2 + max_action**2
    val+= tmp

  return val

def avar(V_table, alpha, state, s, action, time, terminal, ran_arr): #use V_table sil Q_table_val
  # if terminal calculate the terminal value AND hash the value for that c(state,...)+ aggr - s for that (state, aggr) CHANGE

  if time == terminal:
    avar_val = np.round(cost(state, action),2)

    return (avar_val, V_table)
  #otherwise calculate next stage and next aval avar
  else:
    ran_len = len(ran_arr)
    next_time = time + 1
    s_arr = np.array([])
    tmp = 0
```

```
avar_val = 0
for ran in ran_arr:
#find the avar for the next space for the given fixed action
  next_state = random_next_element(state, action, ran)  # F(x_t, a_t, \xi_t) # real valued
  next_time = time + 1
  next_key = (next_time, next_state)
  val_2 = 0
  if next_key in V_table: # in V_table_val
    print('KEY {} found next_time: {}, next_state: {} with used action: {}'.format(next_key, next_time, next_state, action))
    val_2 = V_lookup(V_table, next_time, next_state) #V lookup val without act
    print('val_2 by looking up: ', val_2)
  else:
    print('KEY {} NOT found next_time: {}, next_state: {} with used action: {}'.format(next_key, next_time, next_state, action))
    val_2 = init_cost(state, time)
    print('init cost val2: ', val_2)
    #hash that void value to that KEY
    V_table = V_hash( V_table, next_time, next_state, val_2) ##V hash val without act
  s_arr = np.append(s_arr, val_2)
s_q = np.quantile(s_arr, alpha, interpolation='linear')  # real valued quantile for s_q
s_q = np.round(s_q, 2)
for elt in s_arr:
  avar_val += (1 / ran_len) * max(elt - s_q, 0) #expected value part
avar_val = s_q + (1 / (1 - alpha)) * avar_val #the remaining arithmetic operations
```

```
    avar_val = cost(state, action) + avar_val
    avar_val = np.round(avar_val,2 )

    print('time, state, action: ', time, state,  action)

    return (avar_val, V_table)

def Running_Bellman(alpha, terminal, action_arr, random_arr, N, s, x_0, beta): #remove Q_table argument
  save_values = []
  output_dim = len(action_arr)

  path = {} #time and state, aggr and min action dictionary
  V_table = {} #optimal value and state, aggr  dictionary
  cntr = 0

  # composed of two passes: one forward one backward pass
  while cntr < N:

    x_init = x_0
    state = x_init
    len_random_arr = len(random_arr)
    len_action_arr = len(action_arr)
    for t in range(0,terminal): #forward pass.. no value assignment in forward loop
      rand_act_index = random.randint(0, len_action_arr -1 ) #choose an action index
      sample_action = action_arr[rand_act_index] #choose a random action

```

```
  rand_index = random.randint(0, len_random_arr -1 ) #choose  a random index
  sample_rand = random_arr[rand_index] #sample randomness

  path[t] = state # at time t  record the state and the min_action NO STORE FOR ACTION SIL

  print('path[time]: ', t, path[t])
  print('Current state: {} at t: {}'.format(state, t))
  state = round(state + sample_action + sample_rand, 2) #create next state
  print('next state: {} at time: {} using action {}: '.format(state, t+1, sample_action))
  # no value hashed in the first loop

# use the state from the loop above to store state at terminal
path[terminal] = state
print('Forward pass ended')

print('Backward pass started')
for t in range(terminal, -1, -1): # for loop up to and including zero
    state = path[t] #go backwards using the states and optimal actions sampled at each time t < terminal
    print('path[{}]:{}'.format(t, state))

    if t == terminal:
      min_term_val = float('inf') #assign min terminal val for (state,aggr) pair
      min_term_action = float('inf') #assign min terminal action for (state,aggr) pair
```

```python
    print('finding minimal action loop')
    for action in action_arr: #assign q table for terminal
      print('state, action: ', state, action)
      val = np.round(cost(state, action),2)
      print('current val: ', val)
      if val < min_term_val:
        min_term_val = val
        min_term_action = action
    print('Found in terminal: {}, state: {}, the optimal action: {}, the optimal value: {} '.format(terminal, state, min_term_action, min_term_val))
    #hash the minimum value at terminal
    V_table = V_hash(V_table, terminal, state, min_term_val)
    print('V_table: ', V_table)
  else:
    v_tilde = float('inf')
    min_action = float('inf')
    for action in action_arr: # find the minimum among the actions
      (tmp_val, V_table) = avar(V_table, alpha, state, s, action, t, terminal, random_arr)
      if tmp_val < v_tilde:
        min_action = action
        v_tilde = tmp_val
    # PREVIOUS V_lookup store to approximate with beta lookup and from below 1-beta v_tilde CHANGE
    if (t, state) in V_table: #if it is in v table
      V_lookup_val = V_lookup(V_table, t, state)
    else:
      V_lookup_val = v_tilde # just count on bellman principle
    print('t, Before Update: V_table: ', V_table)
```

```
          print('t, Before Update: V_lookup_val: '.format(t, state), V_lookup_val)
          print('t, v_tilde: ', t, v_tilde)

          # arrangement for learning rate or not
          val = np.round(((1-beta) * v_tilde) + (beta*V_lookup_val),2)

          ##print('time, value to be hashed: ', t, val)
          #                V_table, time, state, aggr, value
          print('t, state, min_action, val:', t, state, min_action, val)
          V_table = V_hash(V_table, t, state, val) #update the  table val for that time, state
          print('V_table: ', V_table)

    cntr += 1
    print('Backward pass ended')
    save_values.append(V_table[0, 1])

  return V_table, save_values

def simulate(x_0, t_0, T, action_arr, alpha, random_arr, N, s, beta):

  print('s in simulate: ', s)

  V_table, save_values = Running_Bellman(alpha, T, action_arr, random_arr, N, s, x_0, beta)

  return  V_table, save_values

x_0 = 1
```

```
t_0 = 0
terminal = 2
alpha = 0.0
N = 100

s_array = np.array([0])

max_action = 1
max_random = 1
delta = 1

action_arr = np.round(np.arange(-max_action, max_action + delta, delta), 2)
random_arr = np.round(np.arange(-max_random, max_random + delta, delta), 2)

key = (t_0, x_0)
min_value = float('inf')
V_min = {}
for r_a in s_array:
  V_table, save_values = simulate(x_0, t_0, terminal, action_arr, alpha,
   ↪  random_arr, N, r_a, 0.1)
  tmp = V_table[key]
  if tmp < min_value:
    #if the initial value is minimum then assign the value function as the
     ↪  minimum
    V_min = V_table
    min_value = tmp
    s_min = r_a

print('(V_min[(time,state, aggr)]: min_value)', V_min)
print('s_min: ', s_min)
```

```python
print('V_min[({}, {})]: {}'.format(t_0, x_0, min_value))
```

Toy Problem 1: $\alpha = 0$

```python
x_0 = 1
t_0 = 0
terminal = 2
alpha = 0.0
N = 100

action_arr = np.array([-1, -0.5, 0, 0.5, 1])
random_arr = np.array([-1, 1])
s_array = np.array([0])

key = (t_0, x_0)
min_value = float('inf')
V_min = {}
for r_a in s_array:
  V_table, save_values = simulate(x_0, t_0, terminal, action_arr, alpha,
    ↪ random_arr, N, r_a, 0.1)
  tmp = V_table[key]
  if tmp < min_value:
    V_min = V_table
    min_value = tmp
    s_min = r_a

print('(V_min[(time,state, aggr)]: min_value)', V_min)
print('s_min: ', s_min)
print('V_min[({}, {})]: {}'.format(t_0, x_0, min_value))
```

```
N = list(range(1, 101))

# Plotting
plt.plot(N, save_values, marker='o', linestyle='-')
plt.title('V_min[0,1] over Iterations')
plt.xlabel('Iterations (N)')
plt.ylabel('V_min[0, 1]')
plt.grid(True)
plt.show()
```

Toy Problem 2: $\alpha = 0.25$

```
x_0 = 1
t_0 = 0
terminal = 2
alpha = 0.25
N = 100

action_arr = np.array([-2, 2])
random_arr = np.array([-1, 1])
s_array = np.array([0])

key = (t_0, x_0)
min_value = float('inf')
V_min = {}
for r_a in s_array:
  V_table, save_values = simulate(x_0, t_0, terminal, action_arr, alpha,
  ↪  random_arr, N, r_a, 0.1)
  tmp = V_table[key]
  if tmp < min_value:
```

```
        V_min = V_table
        min_value = tmp
        s_min = r_a

print('(V_min[(time,state, aggr)]: min_value)', V_min)
print('s_min: ', s_min)
print('V_min[({}, {})]: {}'.format(t_0, x_0, min_value))

N = list(range(1, 101))

# Plotting
plt.plot(N, save_values, marker='o', linestyle='-')
plt.title('V_min[0,1] over Iterations')
plt.xlabel('Iterations (N)')
plt.ylabel('V_min[0, 1]')
plt.grid(True)
plt.show()
```

---

1) As alpha increases, the optimal values must increase

```
x_0 = 1
t_0 = 0
terminal = 2
alpha = [x * 0.01 for x in list(range(0, 50, 5))]
N = 100

action_arr = np.array([-2, 2])
random_arr = np.array([-1, 1])
s_array = np.array([0])

```

```python
key = (t_0, x_0)
min_value = float('inf')
V_min = {}
save_values_2 = []
for a in alpha:
  for r_a in s_array:
    V_table, _ = simulate(x_0, t_0, terminal, action_arr, a, random_arr, N,
     ↪ r_a, 0.25)
    tmp = V_table[key]
    save_values_2.append(tmp)
    if tmp < min_value:
      V_min = V_table
      min_value = tmp
      s_min = r_a

# Plotting
plt.plot(alpha, save_values_2, marker='o', linestyle='-')
plt.title('V_min[0, 1] vs Alpha')
plt.xlabel('Alpha')
plt.ylabel('V_min[0, 1]')
plt.grid(True)
plt.show()
```

---

2) As time increases, the optimal values must also increase

```python
x_0 = 1
t_0 = 0
terminal = list(range(1, 10, 1))
alpha = 0.25
N = 100
```

```

action_arr = np.array([-2, 2])
random_arr = np.array([-1, 1])
s_array = np.array([0])

key = (t_0, x_0)
min_value = float('inf')
V_min = {}
save_values_3 = []
for time in terminal:
  for r_a in s_array:
    V_table, _ = simulate(x_0, t_0, time, action_arr, alpha, random_arr, N,
    ↪  r_a, 0.25)
    tmp = V_table[key]
    save_values_3.append(tmp)
    if tmp < min_value:
      V_min = V_table
      min_value = tmp
      s_min = r_a

# Plotting
plt.plot(terminal, save_values_3, marker='o', linestyle='-')
plt.title('V_min[0, 1] vs Terminal Time')
plt.xlabel('Terminal time')
plt.ylabel('V_min[0, 1]')
plt.grid(True)
plt.show()
```

3) As the available action set increases, the optimal values must decrease

```python
x_0 = 1
t_0 = 0
terminal = 2
alpha = 0
N = 200

random_arr = np.array([-1, 1])
s_array = np.array([0])
max_action = 1
state_deltas = [1/(2**x) for x in list(range(-1, 3))]

key = (t_0, x_0)
min_value = float('inf')
V_min = {}
save_values_4 = []
for state_delta in state_deltas:
  for r_a in s_array:
    action_arr = np.round(np.arange(-max_action, max_action + state_delta,
     ↪ state_delta), 2)
    V_table, _ = simulate(x_0, t_0, terminal, action_arr, alpha,
     ↪ random_arr, N, r_a, 0.25)
    tmp = V_table[key]
    save_values_4.append(tmp)
    if tmp < min_value:
      V_min = V_table
      min_value = tmp
      s_min = r_a

# Plotting
```

```
plt.plot(state_deltas, save_values_4, marker='o', linestyle='-')
plt.title('V_min[0, 1] vs state_delta')
plt.xlabel('state_delta')
plt.ylabel('V_min[0, 1]')
plt.grid(True)
plt.show()
```

4) As the terminal time increases, more iterations is needed for convergence of the optimal values

```
x_0 = 1
t_0 = 0
terminal = list(range(1, 11, 1))
alpha = 0.25
N = 100

action_arr = np.array([-2, 2])
random_arr = np.array([-1, 1])
s_array = np.array([0])

key = (t_0, x_0)
min_value = float('inf')
V_min = {}
save_values_5 = np.zeros((10, 100))
for time in terminal:
  for r_a in s_array:
    V_table, values = simulate(x_0, t_0, time, action_arr, alpha,
    ↪ random_arr, N, r_a, 0.25)
    tmp = V_table[key]
    save_values_5[time-1] = np.array(values)
```

```
    if tmp < min_value:
      V_min = V_table
      min_value = tmp
      s_min = r_a

N_iter = list(range(1, 101, 1))
num_plots = len(terminal)
num_cols = 2
num_rows = (num_plots + num_cols - 1) // num_cols

fig, axs = plt.subplots(num_rows, num_cols, figsize=(12, 20))

for i, ax in zip(terminal, axs.flat):
    values = list(save_values_5[i-1])
    ax.plot(N_iter, values, marker='o', linestyle='-')
    ax.set_title('V_min[0, 1] for Time = '+str(i))
    ax.set_xlabel('N')
    ax.set_ylabel('V_min[0, 1]')
    ax.grid(True)

plt.tight_layout()
plt.show()
```

Standard Normal Randomness with Toy Problem 2

```
import random

random.seed(42)
random_arr = [round(random.gauss(0, 1), 2) for _ in range(3)]

```

```
x_0 = 1
t_0 = 0
terminal = 2
alpha = 0
N = 100

s_array = np.array([0])

max_action = 1
state_delta = 1
action_arr = np.round(np.arange(-max_action, max_action + state_delta,
 ↪ state_delta), 2)

key = (t_0, x_0)
min_value = float('inf')
V_min = {}
for r_a in s_array:
  V_table, save_values_5 = simulate(x_0, t_0, terminal, action_arr, alpha,
   ↪ random_arr, N, r_a, 0.1)
  tmp = V_table[key]
  if tmp < min_value:
    V_min = V_table
    min_value = tmp
    s_min = r_a

print('(V_min[(time,state, aggr)]: min_value)', V_min)
print('s_min: ', s_min)
print('V_min[({}, {})]: {}'.format(t_0, x_0, min_value))

N = list(range(1, 101))
```

```python

# Plotting
plt.plot(N, save_values_5, marker='o', linestyle='-')
plt.title('V_min[0,1] over Iterations')
plt.xlabel('Iterations (N)')
plt.ylabel('V_min[0, 1]')
plt.grid(True)
plt.show()
```