Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

Contents

Overview

Learning Reinforcement Learning

Reinforcement Learning (RL) is emerging as a viable and powerful technique for solving a variety of complex business problems across industries that involve Sequential Optimal Decisioning under Uncertainty. Although RL is classified as a branch of Machine Learning (ML), it tends to be viewed and treated quite differently from other branches of ML (Supervised and Unsupervised Learning). Indeed, **RL seems to hold the key to unlocking the promise of AI** – machines that adapt their decisions to vagaries in observed information, while continuously steering towards the optimal outcome. It's penetration in high-profile problems like self-driving cars, robotics and strategy games points to a future where RL algorithms will have decisioning abilities far superior to humans.

But when it comes getting educated in RL, there seems to be a reluctance to jump right in because RL seems to have acquired a reputation of being mysterious and exotic. We often hear even technical people claim that RL involves "advanced math" and "complicated engineering", and so there seems to be a psychological barrier to entry. While real-world RL algorithms and implementations do get fairly elaborate and complicated in overcoming the proverbial last-mile of business problems, the foundations of RL can actually be learnt without heavy technical machinery. **The core purpose of this book is to demystify RL by finding a balance between depth of understanding and keeping technical content basic**. So now we list the key features of this book which enable this balance:

- Focus on the foundational theory underpinning RL. Our treatment of this theory is based on undergraduate-level Probability, Optimization, Statistics and Linear Algebra. **We emphasize rigorous but simple mathematical notations and formulations in developing the theory**, and encourage you to write out the equations rather than just reading from the book. Occasionally, we invoke some advanced mathematics (eg: Stochastic Calculus) but the majority of the book is based on easily understandable mathematics. In particular, two basic theory concepts - Bellman Optimality Equation and Generalized Policy Iteration - are emphasized throughout the book as they form the basis of pretty much everything we do in RL, even in the most advanced algorithms.
- Parallel to the mathematical rigor, we bring the concepts to life with simple examples and informal descriptions to help you develop an intuitive

understanding of the mathematical concepts. **We drive towards creating appropriate mental models to visualize the concepts**. Often, this involves turning mathematical abstractions into physical examples (emphasizing visual intuition). So we go back and forth between rigor and intuition, between abstractions and visuals, so as to blend them nicely and get the best of both worlds.

- Each time you learn a new mathematical concept or algorithm, we ask you to write small pieces of code (in Python) that implements the concept/algorithm. As an example, if you just learnt a surprising theorem, we'd ask you to write a simulator to simply verify the statement of the theorem. We emphasize this approach not just to bolster the theoretical and intuitive understanding with a hands-on experience, but also because there is a strong emotional effect of seeing expected results emanating from one's code, which in turn promotes long-term retention of the concepts. Most importantly, we avoid messy and complicated ML/RL/BigData tools/packages and stick to bare-bones Python/numpy as these unnecessary tools/packages are huge blockages to core understanding. We believe **coding-from-scratch is the correct approach to truly understand the concepts/algorithms**.
- Lastly, it is important to work with examples that are A) simplified versions of real-world problems in a business domain rich with applications, B) adequately comprehensible without prior business-domain knowledge, C) intellectually interesting and D) sufficiently marketable to employers. We've chosen Financial Trading applications. For each financial problem, we first cover the traditional approaches (including solutions from landmark papers) and then cast the problems in ways that can be solved with RL. **We have made considerable effort to make this book self-contained in terms of the financial knowledge required to navigate these problems**.

What you'll learn from this Book

Here is what you will specifically learn and gain from the book:

- You will learn about the simple but powerful theory of Markov Decision Processes (MDPs) – a framework for Sequential Optimal Decisioning under Uncertainty. You will firmly understand the power of Bellman Equations, which is at the heart of all Dynamic Programming as well as all RL algorithms.

- You will master Dynamic Programming (DP) Algorithms, which are a class of (in the language of AI) Planning Algorithms. You will learn about Policy Iteration, Value Iteration, Backward Induction, Approximate Dynamic Programming and the all-important concept of Generalized Policy Iteration which lies at the heart of all DP as well as all RL algorithms.

- You will gain a solid understanding of a variety of Reinforcement Learning (RL) Algorithms, starting with the basic algorithms like SARSA and Q-Learning and moving on to several important algorithms that work well in practice, including Gradient Temporal Difference, Deep Q-Network, Least-Squares Policy Iteration, Policy Gradient, Monte-Carlo Tree Search. You will learn about how to gain advantages in these algorithms with bootstrapping, off-policy learning and deep-neural-networks-based function approximation. You will also learn how to balance exploration and exploitation with Multi-Armed Bandits techniques like Upper Confidence Bounds, Thompson Sampling, Gradient Bandits and Information State-Space algorithms.

- You will exercise with plenty of "from-scratch" Python implementations of models and algorithms. Throughout the book, we emphasize healthy Python programming practices including interface design, type annotations, functional programming and inheritance-based polymorphism (always ensuring that the programming principles reflect the mathematical principles). The larger take-away from this book will be a rare (and high-in-demand) ability to blend Applied Mathematics concepts with Software Design paradigms.

- You will go deep into important Financial Trading problems, including:

  - (Dynamic) Asset-Allocation to maximize Utility of Consumption
  - Pricing and Hedging of Derivatives in an Incomplete Market
  - Optimal Exercise/Stopping of Path-Dependent American Options
  - Optimal Trade Order Execution (managing Price Impact)
  - Optimal Market-Making (Bid/Ask managing Inventory Risk)

- We treat each of the above problems as MDPs (i.e., Optimal Decisioning formulations), first going over classical/analytical solutions to these problems, then introducing real-world frictions/considerations, and tackling with DP and/or RL.

- As a bonus, we throw in a few applications beyond Finance, including a couple from Supply-Chain and Clearance Pricing in a Retail business.

- We implement a wide range of Algorithms and develop various models in this git code base that we refer to and explain in detail throughout the book. This code base not only provides detailed clarity on the algorithms/models, but also serves to educate on healthy programming patterns suitable not just for RL, but more generally for any Applied Mathematics work.

- In summary, this book blends Theory/Mathematics, Programming/Algorithms and Real-World Financial Nuances while always keeping things simple and intuitive.

Expected Background to read this Book

There is no short-cut to learning Reinforcement Learning or learning the Financial Applications content. You will need to allocate at least 50 hours of effort to learn this material (assuming you have no prior background in these topics). Also, although we have kept the Mathematics, Programming and Financial content fairly basic, this topic is only for technically-inclined readers. Below we outline the technical preparation that is required to follow the material covered in this book.

- Experience with (but not necessarily expertise in) Python is expected and a good deal of comfort with numpy is required. The nature of Python programming we do is mainly numerical algorithms. You don't need to be a professional software developer/engineer but you need to have a healthy interest in learning Python best practices associated with mathematical modeling, algorithms development and numerical programming (we teach these best practices in this book). We don't use any of the popular (but messy and complicated) Big Data/Machine Learning libraries such as Pandas, PySpark, scikit, Tensorflow, PyTorch, OpenCV, NLTK etc. (all you need to know is numpy).
- Familiarity with git and use of an Integrated Development Environment (IDE), eg: Pycharm or Emacs (with Python plugins), is recommended, but not required.
- Familiarity with LaTeX for writing equations is recommended, but not required (other typesetting tools, or even hand-written math is fine, but LaTeX is a skill that is very valuable if you'd like a future in the general domain of Applied Mathematics).
- You need to be strong in undergraduate-level Probability as it is the most important foundation underpinning RL.
- You will also need to have some preparation in undergraduate-level Numerical Optimization, Statistics, Linear Algebra.
- No background in Finance is required, but a strong appetite for Mathematical Finance is required.

Decluttering the Jargon linked to Reinforcement Learning

Machine Learning has exploded in the past decade or so, and Reinforcement Learning (treated as a branch of Machine Learning and hence, a branch of A.I.) has surfaced to the mainstream in both academia and in the industry. It is important to understand what Reinforcement Learning aims to solve, rather than the more opaque view of RL as a technique to learn from data. RL aims to solve problems that involve making *Sequential Optimal Decisions under Uncertainty*. Let us break down this jargon so as to develop an intuitive (and high-level) understanding of the features pertaining to the problems RL solves.

Firstly, let us understand the term *Uncertainty*. This means the problems under consideration involve random variables that evolve over time. The technical term for this is *Stochastic Processes*. We will cover this in detail later in this book, but for now, it's important to recognize that evolution of random variables over time is very common in nature (eg: weather) and in business (eg: customer demand or stock prices), but modeling and navigating such random evolutions can be enormously challenging.

The next term is *Optimal Decisions*, which refers to the technical term *Optimization*. This means there is a well-defined quantity to be maximized (the "goal"). The quantity to be maximized might be financial (like investment value or business profitability), or it could be a safety or speed metric (such as health of customers or time to travel), or something more complicated like a blend of multiple objectives rolled into a single objective.

The next term is *Sequential*, which refers to the fact that as we move forward in time, the relevant random variables' values evolve, and the optimal decisions have to be adjusted to the "changing circumstances". Due to this non-static nature of the optimal decisions, the term *Dynamic Decisions* is often used in the literature covering this subject.

Putting together the three notions of (Uncertainty/Stochastic, Optimization, Sequential/Dynamic Decisions), these problems (that RL tackles) have the common feature that one needs to *overpower the uncertainty by persistent steering towards the goal*. This brings us to the term *Control* (in references to *persistent steering*). These problems are often (aptly) characterized by the technical term *Stochastic Control*. So you see that there is indeed a lot of jargon here. All of this jargon will become amply clear after the first few chapters in this book where we develop mathematical formalism to understand these concepts precisely (and also write plenty of code to internalized these concepts). For now, we just wanted to familiarize you with the range of jargon linked to Reinforcement Learning.

This jargon overload is due to the confluence of terms from Control Theory (emerging from Engineering disciplines), from Operations Research, and from Artificial Intelligence (emerging from Computer Science). For simplicity, we prefer to refer to the class of problems RL aims to solve as *Stochastic Control* problems. Reinforcement Learning is a class of algorithms that are used to solve Stochastic Control problems. We should point out here that there are other disciplines (beyond Control Theory, Operations Research and Artificial Intelligence) with a rich history of developing theory and techniques within the general space of Stochastic Control. Figure 0.1 (a popular image on the internet) illustrates the many faces of Stochastic Control, which has recently been refered to as "The many faces of Reinforcement Learning".

It is also important to recognize that Reinforcement Learning is considered to be a branch of Machine Learning. While there is no crisp definition for *Machine Learning* (ML), ML generally refers to the broad set of techniques to infer mathematical models/functions by acquiring ("learning") knowledge of patterns and properties in the presented data. In this regard, Reinforcement Learning does fit this definition. However, unlike the other branches of ML (Supervised

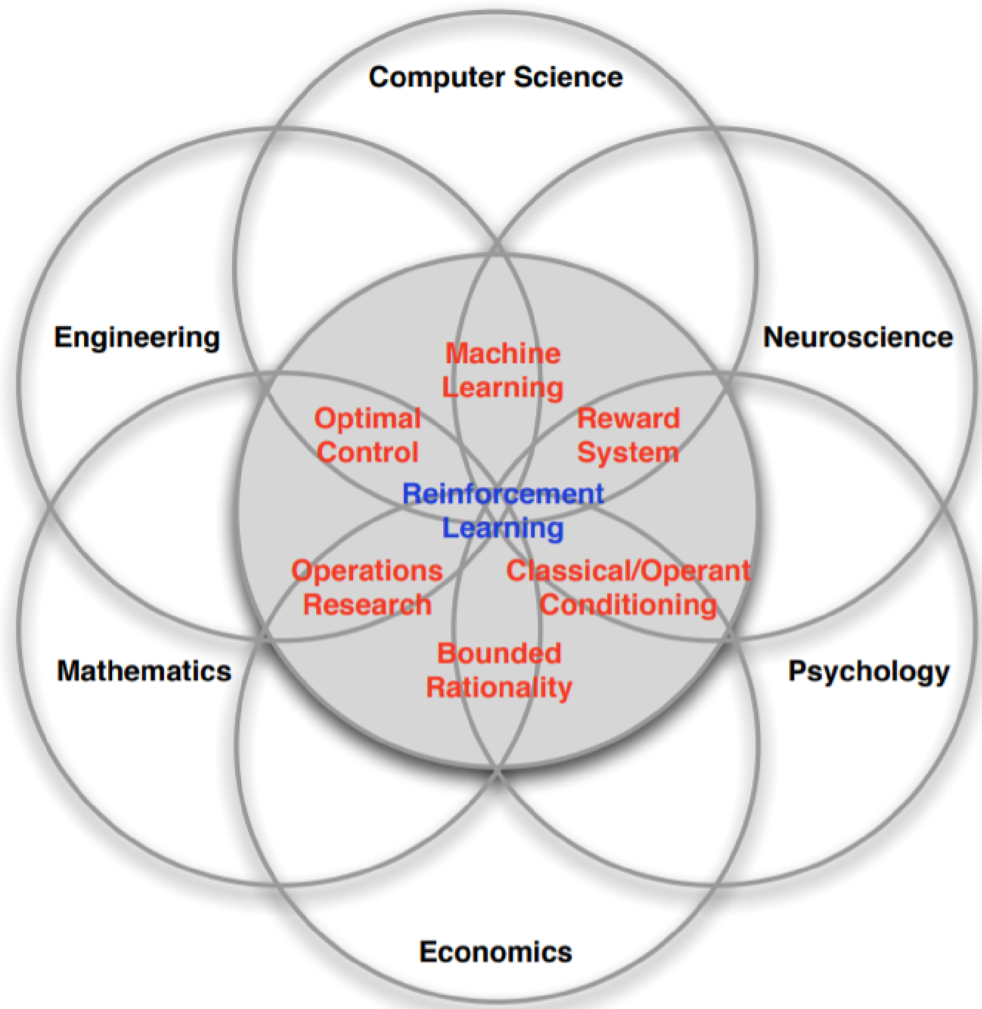

Figure 0.1.: Many Faces of Reinforcement Learning

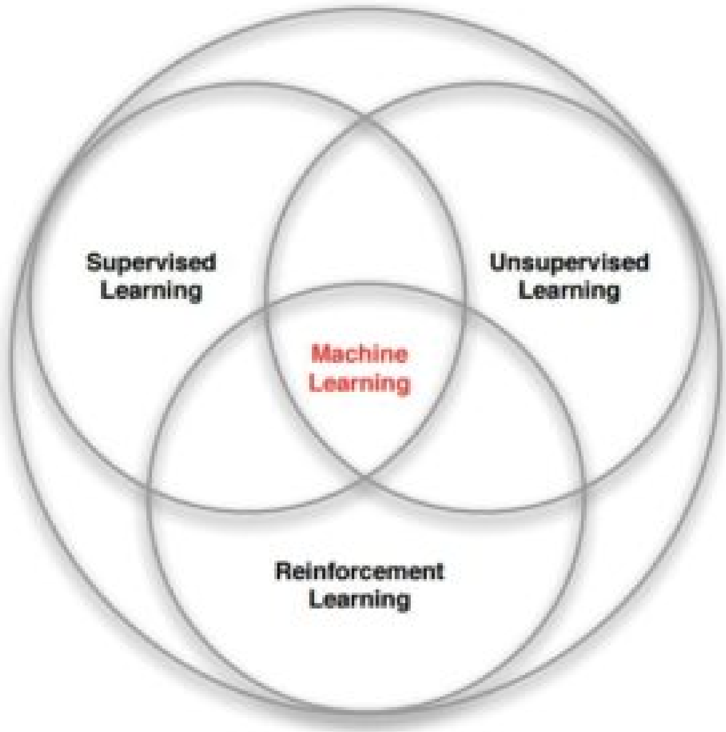

Figure 0.2.: Branches of Machine Learning

Learning and Unsupervised Learning), Reinforcement Learning is a lot more ambitious - it not only learns the patterns and properties of the presented data (internally building a model of the data), it also learns about the appropriate behaviors to be exercised (appropriate decisions to be made) so as to drive towards the optimization objective. It is sometimes said that Supervised Learning and Unsupervised learning are about "minimization" (i.e., they minimize the fitting error of a model to the presented data), while Reinforcement Learning is about "maximization" (i.e., RL also identifies the suitable decisions to be made to maximize a well-defined objective). Figure 0.2 depicts the in-vogue classification of Machine Learning.

More importantly, the class of problems RL aims to solve can be described with a simple yet powerful mathematical framework known as *Markov Decision Processes* (abbreviated as MDPs). We have an entire chapter dedicated to deep coverage of MDPs, but we provide a quick high-level introduction to MDPs in the next section.

Introduction to the Markov Decision Process (MDP) framework

The framework of a Markov Decision Process is depicted in Figure 0.3. As the Figure indicates, the *Agent* and the *Environment* interact in a time-sequenced

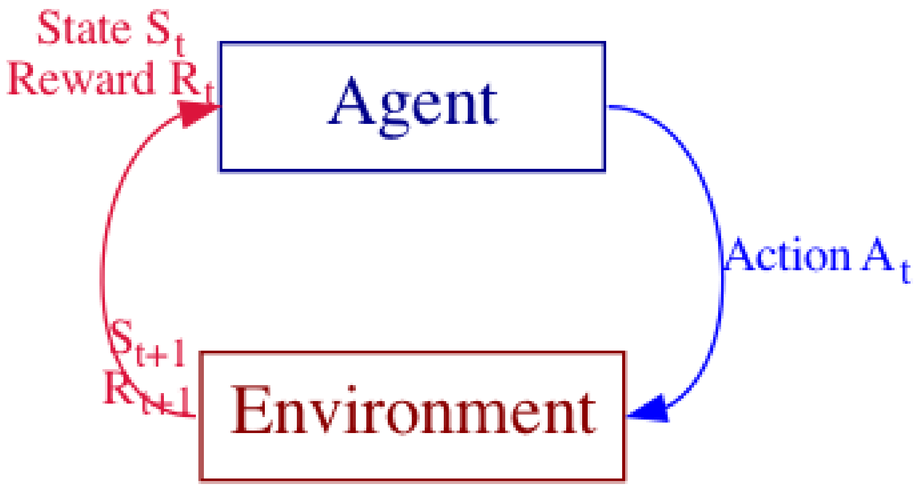

Figure 0.3.: The MDP Framework

loop. The term *Agent* refers to an algorithm (AI algorithm) and the term *Environment* refers to an abstract entity that serves up uncertain outcomes to the Agent. It is important to note that the Environment is indeed abstract in this framework and can be used to model all kinds of real-world situations such as the financial market serving up random stock prices or customers of a company serving up random demand or a chess opponent serving up random moves (from the perspective of the Agent), or really anything at all you can imagine that serves up something random at each time step (it is up to us to model an Environment appropriately to fit the MDP framework).

As the Figure indicates, at each time step t, the Agent observes an abstract piece of information (which we call *State*) and a numerical (real number) quantity that we call *Reward*. Note that the concept of *State* is indeed completely abstract in this framework and we can model *State* to be any data type, as complex or elaborate as we'd like. This flexibility in modeling *State* permits us to model all kinds of real-world situations as an MDP. Upon observing a *State* and a *Reward* at time step t, the *Agent* responds by taking an *Action*. Again, the concept of *Action* is completely abstract and is meant to represent an activity performed by an AI algorithm. It could be a purchase or sale of a stock responding to market stock price movements, or it could be movement of inventory from a warehouse to a store in response to large sales at the store, or it could be a chess move in response to the opponent's chess move (opponent is *Environment*), or really anything at all you can imagine that responds to observations (*State* and *Reward*) served by the *Environment*.

Upon receiving an *Action* from the *Agent* at time step t, the *Environment* responds (with time ticking over to $t + 1$) by serving up the next time step's random *State* and random *Reward*. A technical detail (that we shall explain in detail later) is that the *State* is assumed to have the *Markov Property*, which means:

- The next *State*/*Reward* depends only on Current *State* (for a given *Action*).

- The current *State* encapsulates all relevant information from the history of the interaction between the *Agent* and the *Environment*.
- The current *State* is a sufficient statistic of the future (for a given *Action*).

The goal of the *Agent* at any point in time is to maximize the *Expected Sum* of all future *Reward*s by controlling (at each time step) the *Action* as a function of the observed *State* (at that time step). This function from a *State* to *Action* at any time step is known as the *Policy* function. So we say that the agent's job is exercise control by determining the optimal *Policy* function. Hence, this is a dynamic (i.e., time-sequenced) control system under uncertainty. If the above description was too terse, don't worry - we will explain all of this in great detail in the coming chapters. For now, we just wanted to provide a quick flavor for what the MDP framework looks like. Now we sketch the above description with some (terse) mathematical notation to provide a bit more of the overview of the MDP framework. The following notation is for discrete time steps (continuous time steps notation is analogous, but technically more complicated to describe here):

We denote time steps as $t = 1, 2, 3, \ldots$. Markov State at time t is denoted as $S_t \in \mathcal{S}$ where $\mathcal{S}$ is refered to as the *State Space*. Action at time t is denoted as $A_t \in \mathcal{A}$ where $\mathcal{A}$ is refered to as the *Action Space*. Reward at time t is denoted as $R_t \in \mathbb{R}$ (representing the numerical feedback served by the Environment, along with the State, at each time step t).

We represent the transition probabilities from one time step to the next with the following notation:

$$p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a]$$

$\gamma \in [0, 1]$ is known as the discount factor used to discount Rewards when accumulating Rewards, as follows:

$$\text{Return } G_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots$$

The discount factor γ allows us to model situations where a future reward is less desirable than a current reward of the same quantity.

The goal is to find a *Policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maximizes $\mathbb{E}[G_t|S_t = s]$ for all $s \in \mathcal{S}$. In subsequent chapters, we clarify that the MDP framework actually considers more general policies than described here - policies that are stochastic, i.e., functions that take as input a state and output a probability distribution of actions (rather than a single action). However, for ease of understanding of the core concepts, in this chapter, we stick to deterministic policies $\pi : \mathcal{S} \rightarrow \mathcal{A}$.

The intuition here is that the two entities *Agent* and *Environment* interact in a time-sequenced loop wherein the *Environment* serves up next states and rewards based on the transition probability function p and the *Agent* exerts control over the vagaries of p by exercising the policy π in a way that optimizes the Expected "accumulated rewards" (i.e., Expected Return) from any state.

It's worth pointing out that the MDP framework is inspired by how babies (*Agent*) learn to perform tasks (i.e., take *Action*s) in response to the random

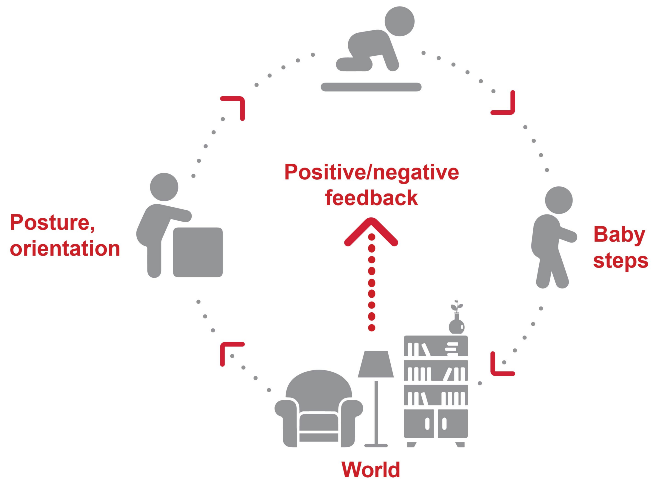

Figure 0.4.: Baby Learning MDP

activities and events (*States* and *Rewards*) they observe as being served up from the world (*Environment*) around them. Figure 0.4 illustrates this - at the bottom of the Figure (labeled "World", i.e., *Environment*), we have a room in a house with a vase atop a bookcase. At the top of the Figure is a baby (learning *Agent*) on the other side of the room who wants to make her way to the bookcase, reach for the vase, and topple it - doing this efficiently (i.e., in quick time and quietly) would mean a large *Reward* for the baby. At each time step, the baby finds herself in a certain posture (eg: lying on the floor, or sitting up, or trying to walk etc.) and observes various visuals around the room - her posture and her visuals would constitute the *State* for the baby at each time step. The baby's *Actions* are various options of physical movements to try to get to the other side of the room (assume the baby is still learning how to walk). The baby tries one physical movement, but is unable to move forward with that movement. That would mean a negative *Reward* - the baby quickly learns that this movement is probably not a good idea. Then she tries a different movement, perhaps trying to stand on her feet and start walking. She makes a couple of good steps forward (positive *Rewards*), but then falls down and hurts herself (that would be a big negative *Reward*). So by trial and error, the baby learns about the consequences of different movements (different actions). Eventually, the baby learns that by holding on to the couch, she can walk across, and then when she reaches the bookcase, she learns (again by trial and error) a technique to climb the bookcase that is quick yet quiet (so she doesn't raise her mom's attention). This means the baby learns of the optimal policy (best actions for each of the states she finds herself in) after essentially what is a "trial and error" method of learning what works and what doesn't. This example is essentially generalized in the MDP framework, and the baby's "trial and error" way of learning is essentially

Figure 0.5.: Self-driving Car MDP

a special case of the general technique of Reinforcement Learning.

Real-world problems that fit the MDP framework

As you might imagine by now, all kinds of problems in nature and in business (and indeed, in our personal lives) can be modeled as Markov Decision Processes. Here is a sample of such problems:

- Self-driving vehicle (Actions constitute speed/steering to optimize safety/time).
- Game of Chess (Actions constitute moves of the pieces to optimize chances of winning the game).
- Complex Logistical Operations, such as those in a Warehouse (Actions constitute inventory movements to optimize throughput/time).
- Making a humanoid robot walk/run on a difficult terrain (Actions are walking movements to optimize time to destination).
- Manage an investment portfolio (Actions are trades to optimize long-term investment gains).
- Optimal decisions during a football game (Actions are strategic game calls to optimize chances of winning the game).
- Strategy to win an election (Actions constitute political decisions to optimize chances of winning the election).

Figure 0.5 illustrates the MDP for a self-driving car. At the top of the figure is the *Agent* (the car's driving algorithm) and at the bottom of the figure is the *Environment* (constituting everything the car faces when driving - other vehicles, traffic signals, road conditions, weather etc.). The *State* consists of the car's location, velocity, and all of the information picked up by the car's sensors/cameras. The *Action* consists of the steering, acceleration and brake. The *Reward* would be a combination of metrics on ride comfort and safety, as well as the negative of each time step (because maximizing the accumulated Reward would then amount to minimizing time taken to reach the destination).

The inherent difficulty in solving MDPs

"Solving" an MDP refers to identifying the optimal policy with an algorithm. This section paints an intuitive picture of why solving a general MDP is fundamentally a hard problem. Often, the challenge is simply that the *State Space* is very large (involving many variables) or complex (elaborate data structure), and hence, is computationally intractable. Likewise, sometimes the *Action Space* can be quite large or complex.

But the main reason for why solving an MDP is inherently difficult is the fact that there is no direct feedback on what the "correct" Action is for a given State. What we mean by that is that unlike a supervised learning framework, the MDP framework doesn't give us anything other than a *Reward* feedback to indicate if an Action is the *right one* or not. A large Reward might encourage the Agent, but it's not clear if one just got lucky with the large Reward or if there could be an even larger Reward if the Agent tries the Action again. The linkage between Actions and Rewards is further complicated by the fact that there is time-sequenced complexity in an MDP, meaning an Action can influence future States, which in turn influences future Actions. Consequently, we sometimes find that Actions can have delayed consequences, i.e., the Rewards for a good Action might come after many time steps (eg: in a game of Chess, a brilliant move leads to a win after several further moves).

The other problem one encounters in real-world situations is that the Agent often doesn't know the *Model* of the *Environment*. By *Model*, we are refering to the probabilities of state-transitions and rewards, i.e., the function p we defined above. This means the Agent has to simultaneously learn the Model (from the real-world data stream) and solve for the Optimal Policy.

Lastly, when there are many actions, the Agent needs to try them all to check if there are some hidden gems (great actions that haven't been tried yet), which in turn means one could end up wasting effort on "duds" (bad actions). So the agent has to find the balance between *exploitation* (retrying actions which have yielded good rewards so far) and *exploration* (trying actions that have either not been tried enough or not been tried at all).

All of this seems to indicate that we don't have much hope in solving MDPs in a reliable and efficient manner. But it turns out that with some clever mathematics, we can indeed make some good inroads. We outline the core idea of this "clever mathematics" in the next section.

Value Function, Bellman Equation, Dynamic Programming and RL

Perhaps the most important concept we want to highlight in this entire book is the idea of a *Value Function* and how we can compute it in an efficient manner with either Planning or Learning algorithms. The Value Function $V^\pi : \mathcal{S} \rightarrow \mathbb{R}$ for a given policy π is defined as:

$$V^\pi(s) = \mathbb{E}[G_t | S_t = s] \text{ for all } s \in \mathcal{S}$$

The intuitive way to understand Value Function is that it tells us how much "accumulated future reward" (i.e., *Return*) we expect to obtain from a given state. The randomness under the expectation comes from the uncertain future states and rewards the Agent is going to see upon taking future actions prescribed by the policy π. The key in evaluating the Value Function for a given policy is that it can be expressed recursively, in terms of the Value Function for the next time step's states. In other words,

$$V^\pi(s) = \sum_{s',r} p(s', r | s, \pi(s)) \cdot (r + \gamma \cdot V^\pi(s')) \text{ for all } s \in \mathcal{S}$$

This equation says that when the Agent is in a given state s, it takes an action $a = \pi(s)$, then sees a random next state s' and a random reward r, so $V^\pi(s)$ can be broken into the expectation of r (first step's expected reward) and the remainder of the future expected accumulated rewards (which can be written in terms of the expectation of $V^\pi(s')$). We won't get into the details of how to solve this recursive formulation in this chapter (will cover this in great detail in future chapters), but it's important for you to recognize for now that this recursive formulation is the key to evaluating the Value Function for a given policy.

However, evaluating the Value Function for a given policy is not the end goal - it is simply a means to the end goal of evaluating the *Optimal Value Function* (from which we obtain the *Optimal Policy*). The Optimal Value Function $V^* : \mathcal{S} \rightarrow \mathbb{R}$ is defined as:

$$V^*(s) = \max_\pi V^\pi(s) \text{ for all } s \in \mathcal{S}$$

The good news is that even the Optimal Value Function can be expressed recursively, as follows:

$$V^*(s) = \max_a \sum_{s',r} p(s', r | s, a) \cdot (r + \gamma \cdot V^*(s')) \text{ for all } s \in \mathcal{S}$$

Furthermore, we can prove that there exists an Optimal Policy π^* achieving $V^*(s)$ for all $s \in \mathcal{S}$ (the proof is constructive, which gives a simple method to obtain the function π^* from the function V^*)$. Specifically, this means that the Value Function obtained by following the optimal policy π^* is the same as the Optimal Value Function V^*, i.e.,

$$V^{\pi^*}(s) = V^*(s) \text{ for all } s \in \mathcal{S}$$

There is a bit of terminology here to get familiar with. The problem of calculating $V^\pi(s)$ (Value Function for a give policy) is known as the *Prediction* problem (since this amounts of statistical estimation of the expected returns from any given state when following a policy π). The problem of calculating the Optimal Value Function V^* (and hence, Optimal Policy π^*), is known as the *Control* problem (since this requires steering of the policy such that we obtain the maximum expected return from any state). Solving the Prediction problem is typically a stepping stone towards solving the (harder) problem of Control. These recursive equations for V^π and V^* are known as the (famous) Bellman Equations (which you will hear a lot about in future chapters). In a continuous-time formulation, the Bellman Equation is refered to as the famous *Hamilton-Jacobi-Bellman* (*HJB*) equation.

The algorithms to solve the prediction and control problems based on Bellman equations are broadly classified as:

- Dynamic Programming, a class of (in the language of A.I.) *Planning* algorithms.
- Reinforcement Learning, a class of (in the language of A.I.) *Learning* algorithms.

Now let's talk a bit about the difference between Dynamic Programming and Reinforcement Learning algorithms. Dynamic Programming algorithms (which we cover a lot of in this book) assume that the agent knows of the transition probabilities p and the algorithm takes advantage of the knowledge of those probabilities (leveraging the Bellman Equation to efficiently calculate the Value Function). Dynamic Programming algorithms are considered to be *Planning* and not *Learning* (in the language of A.I.) because the algorithm doesn't need to interact with the Environment and doesn't need to learn from the (states, rewards) data stream coming from the Environment. Rather, armed with the transition probabilities, the algorithm can reason about future probabilistic outcomes and perform the requisite optimization calculation to infer the Optimal Policy. So it *plans* it's path to success, rather than *learning* about how to succeed.

However, in typical real-word situations, one doesn't really know the transition probabilities p. This is the realm of Reinforcement Learning (RL). RL algorithms interact with the Environment, learn with each new (state, reward) pair received from the Environment, and incrementally figure out the Optimal Value Function (with the "trial and error" approach that we outlined earlier). However, note that the Environment interaction could be *real* interaction or *simulated* interaction. In the latter case, we do have a model of the transitions but the structure of the model is so complex that we only have access to samples of the next state and reward (rather than an explicit representation of the probabilities). This is known as a *Sampling Model* of the Environment. With access to such a sampling model of the environment (eg: a robot learning on a simulated terrain), we can employ the same RL algorithm that we would have used when interacting with a real environment (eg: a robot learning on an actual terrain). In fact, most RL algorithms in practice learn from simulated models of the environment. As we explained earlier, RL is essentially a "trial and error" learning approach and hence, is quite laborious and fundamentally inefficient. The recent progress in RL is coming from more efficient ways of learning the Optimal Value Function, and better ways of approximating the Optimal Value Function. One of the key challenges for RL in the future is to identify better ways of finding the balance between "exploration" and "exploitation" of actions. In any case, one of the key reasons RL has started doing well lately is due to the assistance it has obtained from Deep Learning (typically Deep Neural Networks are used to approximate the Value Function and/or to approximate the Value Function). RL with such deep learning approximations is known by the catchy modern term *Deep RL*.

We believe the current promise of A.I. is dependent on the success of Deep RL. The next decade will be exciting as RL research will likely yield improved algorithms and it's pairing with Deep Learning will hopefully enable us to solve fairly complex real-world stochastic control problems.

Outline of Chapters

The chapters in this book are organized into 3 modules as follows:

- Module I: Processes and Planning Algorithms (Chapters 1-4)
- Module II: Modeling Financial Applications (Chapters 5-8)
- Module III: Reinforcement Learning Algorithms (Chapters 9-15)

Before covering the contents of the chapters in these 3 modules, the book starts with 2 unnumbered chapters. The first of these unnumbered chapters is *this chapter* (the one you are reading) which serves as an *Overview*, covering the pedagogical aspects of learning RL (and more generally Applied Math), outline of the learnings to be acquired from this book, background required to read this book, a high-level overview of Stochastic Control, MDP, Value Function, Bellman Equation and RL, and finally the outline of chapters in this book. The second unnumbered chapter is called *Design Paradigms for Applied Mathematics implementations in Python*. Since this book makes heavy use of Python code for developing mathematical models and for algorithms implementations, we cover the requisite Python background (specifically the design paradigms we use in our Python code) in this chapter. To be clear, this chapter is not a full Python tutorial – the reader is expected to have some background in Python already. It is a tutorial of some key techniques and practices in Python (that many readers of this book might not be accustomed to) that we use heavily in this book and that are also highly relevant to programming in the broader area of Applied Mathematics. We cover the topics of Type Annotations, List and Dict Comprehensions, Functional Programming, Interface Design with Abstract Base Classes, Generics Programming and Generators.

The remaining chapters in this book are organized in the 3 modules we listed above.

Module I: Processes and Planning Algorithms

The first module of this book covers the theory of Markov Decision Processes (MDP), Dynamic Programming (DP) and Approximate Dynamics Programming (ADP) across Chapters 1-4.

In order to understand the MDP framework, we start with the foundations of *Markov Processes* (sometimes refered to as Markov Chains) in Chapter 1. Markov Processes do not have any Rewards or Actions, they only have states and states transitions. We believe spending a lot of time on this simplified framework of Markov Processes is excellent preparation before getting to MDPs. Chapter 1 then builds upon Markov Processes to include the concept of Reward (but not Action) - the inclusion of Reward yields a framework known as Markov Reward Process. With Markov Reward Processes, we can talk about Value Functions and Bellman Equation, which serve as great preparation for understanding Value Function and Bellman Equation later in the context of MDPs. Chapter 1 motivates these concepts with examples of stock prices and with a simple inventory example that serves first as a Markov Process and then as a Markov Reward Process. There is also a significant amount of programming content in this chapter to develop comfort as well as depth in these concepts.

Chapter 2 on *Markov Decision Processes* lays the foundational theory underpinning RL – the framework for representing problems dealing with sequential optimal decisioning under uncertainty (Markov Decision Process). You will learn about the relationship between Markov Decision Processes and Markov Reward Processes, about the Value Function and the Bellman Equations. Again, there is a considerable amount of programming exercises in this chapter. The heavy investment in this theory together with hands-on programming will put you in a highly advantaged position to learn the following chapters in a very clear and speedy manner.

Chapter 3 on *Dynamic Programming* covers the Planning technique of Dynamic Programming (DP), which is an important class of foundational algorithms that can be an alternative to RL if the MDP is not too large or too complex. Also, learning these algorithms provides important foundations to be able to understand subsequent RL algorithms more deeply. You will learn about several important DP algorithms by the end of the chapter and you will learn about why DP gets difficult in practice which draws you to the motivation behind RL. Again, we cover plenty of programming exercises that are quick to implement and will aid considerably in internalizing the concepts. Finally, we emphasize a special algorithm - Backward Induction - for solving finite-horizon Markov Decision Processes, which is the setting for the financial applications we cover in this book.

The Dynamic Programming algorithms covered in Chapter 3 suffer from the two so-called curses: Curse of Dimensionality and Curse of Modeling. These curses can be cured with a combination of sampling and function approximation. Module III covers the sampling cure (using Reinforcement Learning). Chapter 4 on *Function Approximation and Approximate Dynamic Programming* covers the topic of function approximation and shows how an intermediate cure - Approximate Dynamic Programming (function approximation without sampling) - is often quite viable and can be suitable for some problems. As part of this chapter, we implement linear function approximation and approximation with deep neural networks (forward and back propagation algorithms) so we can use these approximations in Approximate Dynamic Programming algorithms and later also in RL.

Module II: Modeling Financial Applications

The second module of this book covers the background on Utility Theory and 5 financial applications of Stochastic Control across Chapter 5-8.

We begin this module with Chapter 5 on *Utility Theory* which covers a very important Economics concept that is a pre-requisite for most of the Financial Applications we cover in subsequent chapters. This is the concept of risk-aversion (i.e., how people want to be compensated for taking risk) and the related concepts of risk-premium and Utility functions. The remaining chapters in this module cover not only the 5 financial applications, but also great detail on how to model them as MDPs, develop DP/ADP algorithms to solve them, and write plenty of code to implement the algorithms, which helps internalize the learnings quite well. Note that in practice these financial applications can get fairly complex and DP/ADP algorithms don't quite scale, which means we need to tap into RL algorithms to solve them. So we revisit these financial applications in Module III when we cover RL algorithms.

Chapter 6 is titled *Dynamic Asset Allocation and Consumption*. This chapter covers the first of the 5 Financial Applications. This problem is about how to adjust the allocation of one's wealth to various investment choices in response to changes in financial markets. The problem also involves how much wealth to consume in each interval over one's lifetime so as to obtain the best utility from wealth consumption. Hence, it is the joint problem of (dynamic) allocation of wealth to financial assets and appropriate consumption of one's wealth over a period of time. This problem is best understood in the context of Merton's landmark paper in 1969 where he stated and solved this problem. This chapter is mainly focused on the mathematical derivation of Merton's solution of this problem with Dynamic Programming. You will also learn how to solve the asset allocation problem in a simple setting with Approximate Backward Induction (an ADP algorithm covered in Chapter 4).

Chapter 7 covers a very important topic in Mathematical Finance: *Pricing and Hedging of Derivatives*. Full and rigorous coverage of derivatives pricing and hedging is a fairly elaborate and advanced topic, and beyond the scope of this book. But we have provided a way to understand the theory by considering a very simple setting - that of a single-period with discrete outcomes and no provision for rebalancing of the hedges, that is typical in the general theory. Following the coverage of the foundational theory, we cover the problem of optimal pricing/hedging of derivatives in an *incomplete market* and the problem of optimal exercise of American Options (both problems are modeled as MDPs). In this chapter, you will learn about some highly important financial foundations such as the concepts of arbitrage, replication, market completeness, and the all-important risk-neutral measure. You will learn the proofs of the two fundamental theorems of asset pricing in this simple setting. We also provide an overview of the general theory (beyond this simple setting). Next you will learn about how to price/hedge derivatives incorporating real-world frictions by modeling this problem as an MDP. In the final module of this chapter, you will learn how to model the more general problem of optimal stopping as an MDP. You will learn how to use Backward Induction (a DP algorithm we learnt in Chapter 3) to solve this problem when the state-space is not too big. By the end of this chapter, you would have developed significant expertise in pricing and hedging complex derivatives, a skill that is in high demand in the finance industry.

Chapter 8 on *Order-Book Algorithms* covers the remaining two Financial Applications, pertaining to the world of Algorithmic Trading. The current practice in Algorithmic Trading is to employ techniques that are rules-based and heuristic. However, Algorithmic Trading is quickly transforming into Machine Learning-based Algorithms. In this chapter, you will be first introduced to the mechanics of trade order placements (market orders and limit orders), and then introduced to a very important real-world problem – how to submit a large-sized market order by splitting the shares to be transacted and timing the splits optimally in order to overcome "price impact" and gain maximum proceeds. You will learn about the classical methods based on Dynamic Programming. Next you will learn about the market frictions and the need to tackle them with RL. In the second half of this chapter, we cover the Algorithmic-Trading twin of the Optimal Execution problem – that of a market-maker having to submit dynamically-changing bid and ask limit orders so she can make maximum gains. You will learn about how market-makers (a big and thriving industry) operate. Then you will learn about how to formulate this problem as an MDP. We will do a thorough coverage of the classical Dynamic Programming solution by Avellaneda and Stoikov. Finally, you will be exposed to the real-world nuances of this problem, and hence, the need to tackle with a market-calibrated simulator and RL.

Module III: Reinforcement Learning Algorithms

The third module of this book covers Reinforcement Learning algorithms across Chapter 9-15.

Chapter 9 on *Monte-Carlo and Temporal-Difference methods for Prediction* starts a new phase in this book - our entry into the world of RL algorithms. To understand the basics of RL, we start this chapter by restricting the RL problem to a very simple one – one where the state space is small and manageable as a table enumeration (known as tabular RL) and one where we only have to calculate the Value Function for a Fixed Policy (this problem is known as the

Prediction problem, versus the optimization problem which is known as the Control problem). The restriction to Tabular Prediction is important because it makes it much easier to understand the core concepts of Monte-Carlo (MC) and Temporal-Difference (TD) in this simplified setting. The later part of this chapter extends Tabular Prediction to Prediction with Function Approximation (leveraging the function approximation foundations we had developed in Chapter 4 in the context of ADP). The remaining chapters will build upon this chapter by adding more complexity and more nuances, while retaining much of the key core concepts developed in this chapter. As ever, you will learn by coding plenty of MC and TD algorithms from scratch.

Chapter 10 on *Monte-Carlo and Temporal-Difference for Control* makes the natural extension from Prediction to Control, while initially remaining in the tabular setting. The investments made in understanding the core concepts of MC and TD in Chapter 9 bear fruit here as important Control Algorithms such as SARSA and Q-learning can now be learnt with enormous clarity. In this chapter, we implement both SARSA and Q-Learning from scratch in Python. This chapter also introduces a very important concept for the future success of RL in the real-world: off-policy learning (Q-Learning is the simplest off-policy learning algorithm and it has had good success in various applications). The later part of this chapter extends Tabular Control to Control with Function Approximation (leveraging the function approximation foundations we had developed in Chapter 4).

Chapter 11 on *Experience Replay, Least-Squares Policy Iteration and Gradient TD* moves on from basic and more traditional RL algorithms to recent innovations in RL. We start this chapter with the important idea of *Experience Replay* which makes more efficient use of data by storing data as it comes and re-using it throughout the learning process of the algorithm. We also emphasize a simple but important linear function approximation algorithm that learns and improves through batches of data (versus the traditional algorithms that learn incrementally upon each new piece of data) - this *Batch Algorithm* is known as Least-Squares Temporal Difference (to solve the Prediction problem) and it's extension to solve the Control problem is known as Least-Squares Policy Iteration. We will discuss these algorithms in the context of Financial Applications that were covered in Module II. In the later part of this chapter, we provide deeper insights into the core mathematics underpinning RL algorithms (back to the basics of Bellman Equation). Understanding *Value Function Geometry* will place you in a highly advantaged situation in terms of truly understanding what is it that makes some Algorithms succeed in certain situations and fail in other situations. This chapter also explains how to break out of the so-called Deadly Triad (when bootstrapping, function approximation and off-policy are employed together, RL algorithms tend to fail). The state-of-the-art Gradient TD Algorithm resists the deadly triad and we dive deep into its inner workings to understand how and why.

Chapter 12 on *Policy Gradient Algorithms* introduces a very different class of RL algorithms that are based on improving the policy using the gradient of the policy function approximation (rather than the usual policy improvement based on explicit argmax on Q-Value Function). When action spaces are large or continuous, Policy Gradient tends to be the only option and so, this chapter is useful to overcome many real-world challenges (including those in many financial applications) where the action space is indeed large. You will learn about the mathematical proof of the elegant Policy Gradient Theorem and implement a couple of Policy Gradient Algorithms from scratch. You will learn about state-of-the-art Actor-Critic methods. Lastly, you will also learn about Evolutionary Strategies, an algorithm that looks quite similar to Policy Gradient Algorithms, but is technically not an RL Algorithm. However, learning about Evolutionary Strategies is important because some real-world applications, including Financial Applications, can indeed be tackled well with Evolutionary Strategies.

Chapter 13 on *Learning versus Planning* brings the various pieces of Planning and Learning concepts learnt in this book together. You will learn that in practice, one needs to be creative about blending planning and learning concepts (a technique known as Model-based RL). In practice, many problems are indeed tackled using Model-based RL. You will also get familiar with an algorithm (Monte Carlo Tree Search) that was highly popularized when it solved the Game of GO, a problem that was thought to be insurmountable by present AI technology.

Chapter 14 on *Multi-Armed Bandits: Exploration versus Exploitation* is a deep-dive into the topic of balancing exploration and exploitation, a topic of great importance in RL algorithms. Exploration versus Exploitation is best understood in the simpler setting of the Multi-Armed Bandit (MAB) problem. You will learn about various state-of-the-art MAB algorithms, implement them in Python, and draw various graphs to understand how they perform versus each other in various problem settings. You will then be exposed to Contextual Bandits which is a popular approach in optimizing choices of Advertisement placements. Finally, you will learn how to apply the MAB algorithms within RL.

Chapter 15 is the concluding chapter titled *RL in Real-World Finance: Reality versus Hype, Present versus Future*. The purpose of this chapter is to put the entire book's content into perspective relative to the current state of the financial industry, and the practical challenges in adoption of RL. We also provide some guidance on how to go about building an end-to-end system for financial applications based on RL. The reader will be guided on reality versus hype in the current "AI-First" landscape. You will also gain a perspective of where RL stands today and what the future holds.

Short Appendix Chapters

Finally, we have 6 short Appendix chapters at the end of this book. The first appendix is on *Moment Generating Functions* and it's use in various calculations across this book. The second appendix is a technical perspective of *Function Approximations as Vector Spaces*, which helps develop a deeper mathematical understanding of function approximations. The second appendix is on *Portfolio Theory* covering the mathematical foundations of balancing return versus risk in portfolios and the much-celebrated Capital Asset Pricing Model (CAPM). The third appendix covers the basics of *Stochastic Calculus* as we need some of this theory (*Ito Integral*, *Ito's Lemma* etc.) in the derivations in a couple of the chapters in Module II. The fourth appendix is on the *HJB Equation*, which as a key part of the derivation of the closed-form solutions for 2 of the 5 financial applicatons we cover in Module II. The fifth and final appendix covers the derivation of the famous *Black-Scholes Equation* (and it's solution for Call/Put Options).

Design Paradigms for Applied Mathematics implementations in Python

This chapter is coming soon ...

Part I.

Processes and Planning Algorithms

1. Markov Processes

This book is about "Sequential Decisioning under Sequential Uncertainty". In this chapter, we will ignore the "sequential decisioning" aspect and focus just on the "sequential uncertainty" aspect.

The Concept of *State* in a Process

For a gentle introduction to the concept of *State*, we start with an informal notion of the terms *Process* and *State* (this will be formalized later in this chapter). Informally, think of a Process as producing a sequence of random outcomes at discrete time steps that we'll index by a time variable $t = 0, 1, 2, \ldots$. The random outcomes produced by a process might be key financial/trading/business metrics one cares about, such as prices of financial derivatives or the value of a portfolio held by an investor. To understand and reason about the evolution of these random outcomes of a process, it is beneficial to focus on the internal representation of the process at each point in time t, that is fundamentally responsible for driving the outcomes produced by the process. We refer to this internal representation of the process at time t as the (random) *state* of the process at time t and denote it as S_t. Specifically, we are interested in the probability of the next state S_{t+1}, given the present state S_t and the past states $S_0, S_1, \ldots, S_{t-1}$, i.e., $\mathbb{P}[S_{t+1}|S_t, S_{t-1}, \ldots, S_0]$. So to clarify, we distinguish between the internal representation (*state*) and the output (outcomes) of the Process. The *state* could be any data type - it could be something as simple as the daily closing price of a single stock, or it could be something quite elaborate like the number of shares of each publicly traded stock held by each bank in the U.S., as noted at the end of each week.

Understanding Markov Property from Stock Price Examples

We will be learning about Markov Processes in this chapter and these processes have *States* that possess a property known as the *Markov Property*. So we will now learn about the *Markov Property of States*. Let us develop some intuition for this property with some examples of random evolution of stock prices over time.

To aid with the intuition, let us pretend that stock prices take on only integer values and that it's acceptable to have zero or negative stock prices. Let us denote the stock price at time t as X_t. Let us assume that from time step t to

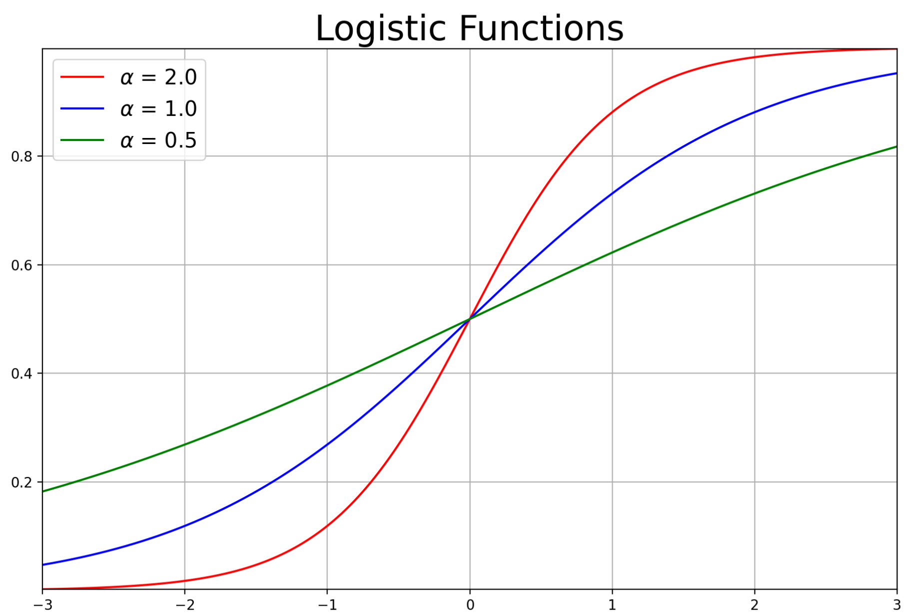

Figure 1.1.: Logistic Curves

the next time step $t+1$, the stock price can either go up by 1 or go down by 1, i.e., the only two outcomes for X_{t+1} are $X_t + 1$ or $X_t - 1$. To understand the random evolution of the stock prices in time, we just need to quantify the probability of an up-move $\mathbb{P}[X_{t+1} = X_t + 1]$ since the probability of a down-move $\mathbb{P}[X_{t+1} = X_t - 1] = 1 - \mathbb{P}[X_{t+1} = X_t + 1]$. We will consider 3 different processes for the evolution of stock prices. These 3 processes will prescribe $\mathbb{P}[X_{t+1} = X_t + 1]$ in 3 different ways.

Process 1:

$$\mathbb{P}[X_{t+1} = X_t + 1] = \frac{1}{1 + e^{-\alpha_1 (L - X_t)}}$$

where L is an arbitrary reference level and $\alpha_1 \in \mathbb{R}_{\geq 0}$ is a "pull strength" parameter. Note that this probability is defined as a logistic function of $L - X_t$ with the steepness of the logistic function controlled by the parameter α_1 (see Figure 1.1)

The way to interpret this logistic function of $L - X_t$ is that if X_t is greater than the reference level L (making $\mathbb{P}[X_{t+1} = X_t + 1] < 0.5$), then there is more of a down-pull than an up-pull. Likewise, if X_t is less than L, then there is more of an up-pull. The extent of the pull is controlled by the magnitude of the parameter α_1. We refer to this behavior as *mean-reverting behavior*, meaning the stock price tends to revert to the "mean" (i.e., to the reference level L).

We can model the state $S_t = X_t$ and note that the probabilities of the next state S_{t+1} depend only on the current state S_t and not on the previous states $S_0, S_1, \ldots, S_{t-1}$. Informally, we phrase this property as: "The future is independent of the past given the present". Formally, we can state this property of the states as:

$$\mathbb{P}[S_{t+1}|S_t, S_{t-1}, \ldots, S_0] = \mathbb{P}[S_{t+1}|S_t] \text{ for all } t \geq 0$$

This is a highly desirable property since it helps make the mathematics of such processes much easier and the computations much more tractable. We call this the *Markov Property* of States, or simply that these are *Markov States*.

Let us now code this up. First, we will create a dataclass to represent the dynamics of this process. As you can see in the code below, the dataclass `Process1` contains two attributes `level_param: int` and `alpha1: float = 0.25` to represent L and α_1 respectively. It contains the method `up_prob` to calculate $\mathbb{P}[X_{t+1} = X_t + 1]$ and the method `next_state`, which samples from a Bernoulli distribution (whose probability is obtained from the method `up_prob`) and creates the next state S_{t+1} from the current state S_t. Also, note the nested dataclass `State` meant to represent the state of Process 1 (it's only attribute `price: int` reflects the fact that the state consists of only the current price, which is an integer).

```
import numpy as np

@dataclass
class Process1:
    @dataclass
    class State:
        price: int

    level_param: int  # level to which price mean-reverts
    alpha1: float = 0.25  # strength of mean-reversion (non-negative value)

    def up_prob(self, state: State) -> float:
        return 1. / (1 + np.exp(-self.alpha1 * (self.level_param - state.price)))

    def next_state(self, state: State) -> State:
        up_move: int = np.random.binomial(1, self.up_prob(state), 1)[0]
        return Process1.State(price=state.price + up_move * 2 - 1)
```

Next, we write a simple simulator using Python's generator functionality (using `yield`) as follows:

```
def simulation(process, start_state):
    state = start_state
    while True:
        yield state
        state = process.next_state(state)
```

Now we can use this simulator function to generate sampling traces. In the following code, we generate `num_traces` number of sampling traces over `time_steps` number of time steps starting from a price X_0 of `start_price`. The use of Python's generator feature lets us do this "lazily" (on-demand) using the `itertools.islice` function.

```python
import itertools

def process1_price_traces(
    start_price: int,
    level_param: int,
    alpha1: float,
    time_steps: int,
    num_traces: int
) -> np.ndarray:
    process = Process1(level_param=level_param, alpha1=alpha1)
    start_state = Process1.State(price=start_price)
    return np.vstack([
        np.fromiter((s.price for s in itertools.islice(
            simulation(process, start_state),
            time_steps + 1
        )), float) for _ in range(num_traces)])
```

The entire code is in the file rl/chapter2/stock_price_simulations.py. We encourage you to play with this code with different `start_price`, `level_param`, `alpha1`, `time_steps`, `num_traces`. You can plot graphs of sampling traces of the stock price, or plot graphs of the terminal distributions of the stock price at various time points (both of these plotting functions are made available for you in this code).

Now let us consider a different process.

Process 2:

$$\mathbb{P}[X_{t+1} = X_t + 1] = \begin{cases} 0.5(1 - \alpha_2(X_t - X_{t-1})) & \text{if } t > 0 \\ 0.5 & \text{if } t = 0 \end{cases}$$

where α_2 is a "pull strength" parameter in the closed interval $[0, 1]$. The intuition here is that the direction of the next move $X_{t+1} - X_t$ is biased in the reverse direction of the previous move $X_t - X_{t-1}$ and the extent of the bias is controlled by the parameter α_2.

We note that if we model the state S_t as X_t, we won't satisfy the Markov Property because the probabilities of X_{t+1} depend on not just X_t but also on X_{t-1}. However, we can perform a little trick here and create an augmented state S_t consisting of the pair $(X_t, X_t - X_{t-1})$. In case $t = 0$, the state S_0 can assume the value $(X_0, Null)$ where $Null$ is just a symbol denoting the fact that there have been no stock price movements thus far. With the state S_t as this pair $(X_t, X_t - X_{t-1})$, we can see that the Markov Property is indeed satisfied:

$$\mathbb{P}[(X_{t+1}, X_{t+1} - X_t)|(X_t, X_t - X_{t-1}), (X_{t-1}, X_{t-1} - X_{t-2}), \ldots, (X_0, Null)]$$

$$= \mathbb{P}[(X_{t+1}, X_{t+1} - X_t)|(X_t, X_t - X_{t-1})] = 0.5(1 - \alpha_2(X_{t+1} - X_t)(X_t - X_{t-1}))$$

Note that if we had modeled the state S_t as the entire stock price history $(X_0, X_1, \ldots, X_t)$, then the Markov Property would be satisfied trivially. The

Markov Property would also be satisfied if we had modeled the state S_t as the pair (X_t, X_{t-1}) for $t > 0$ and S_0 as $(X_0, Null)$. However, our choice of $S_t := (X_t, X_t - X_{t-1})$ is the "simplest/minimal" internal representation. In fact, in this entire book, our endeavor in modeling states for various processes will be to ensure the Markov Property with the "simplest/minimal" representation for the state.

The corresponding dataclass for Process 2 is shown below:

```
handy_map: Mapping[Optional[bool], int] = {True: -1, False: 1, None: 0}

@dataclass
class Process2:
   @dataclass
   class State:
       price: int
       is_prev_move_up: Optional[bool]

   alpha2: float = 0.75  # strength of reverse-pull (value in [0,1])

   def up_prob(self, state: State) -> float:
       return 0.5 * (1 + self.alpha2 * handy_map[state.is_prev_move_up])

   def next_state(self, state: State) -> State:
       up_move: int = np.random.binomial(1, self.up_prob(state), 1)[0]
       return Process2.State(
           price=state.price + up_move * 2 - 1,
           is_prev_move_up=bool(up_move)
       )
```

The code for generation of sampling traces of the stock price is almost identical to the code we wrote for Process 1.

```
def process2_price_traces(
    start_price: int,
    alpha2: float,
    time_steps: int,
    num_traces: int
) -> np.ndarray:
    process = Process2(alpha2=alpha2)
    start_state = Process2.State(price=start_price, is_prev_move_up=None)
    return np.vstack([
        np.fromiter((s.price for s in itertools.islice(
            simulation(process, start_state),
            time_steps + 1
        )), float) for _ in range(num_traces)])
```

Now let us look at a more complicated process.

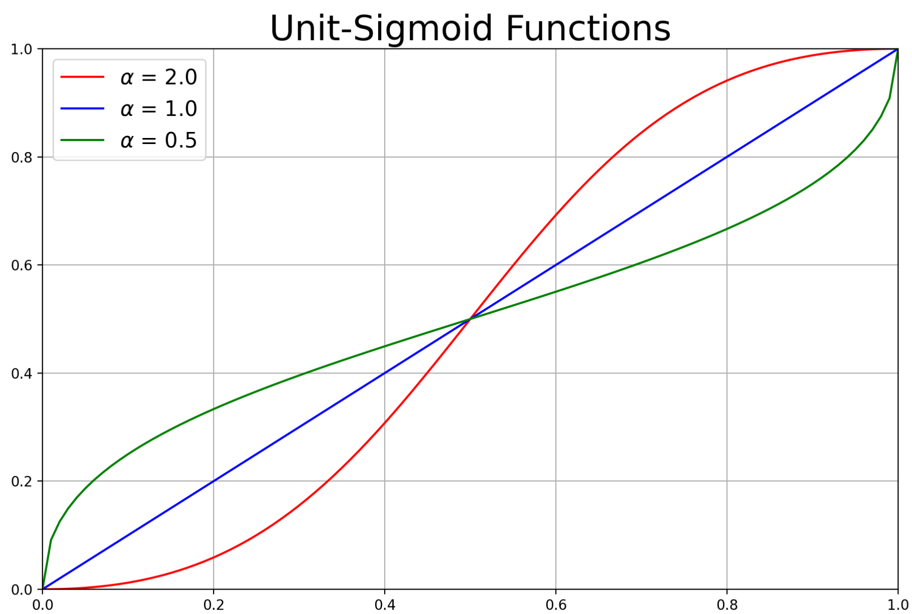

Figure 1.2.: Unit-Sigmoid Curves

Process 3: This is an extension of Process 2 where the probability of next movement depends not only on the last movement, but on all past movements. Specifically, it depends on the number of past up-moves (call it $U_t = \sum_{i=1}^{t} \max(X_i - X_{i-1}, 0)$) relative to the number of past down-moves (call it $D_t = \sum_{i=1}^{t} \max(X_{i-1} - X_i, 0)$) in the following manner:

$$\mathbb{P}[X_{t+1} = X_t + 1] = \begin{cases} \frac{1}{1+(\frac{U_t+D_t}{D_t}-1)^{\alpha_3}} & \text{if } t > 0 \\ 0.5 & \text{if } t = 0 \end{cases}$$

where $\alpha_3 \in \mathbb{R}_{\geq 0}$ is a "pull strength" parameter. Let us view the above probability expression as:

$$f(\frac{D_t}{U_t + D_t}; \alpha_3)$$

where $f : [0,1] \to [0,1]$ is a sigmoid-shaped function

$$f(x; \alpha) = \frac{1}{1 + (\frac{1}{x} - 1)^\alpha}$$

whose steepness at $x = 0.5$ is controlled by the parameter α (note: values of $\alpha < 1$ will produce an inverse sigmoid as seen in Figure 1.2 which shows unit-sigmoid functions f for different values of α).

The probability of next up-movement is fundamentally dependent on the quantity $\frac{D_t}{U_t+D_t}$ (the function f simply serves to control the extent of the "reverse pull"). $\frac{D_t}{U_t+D_t}$ is the fraction of past time steps when there was a down-move.

So, if number of down-moves in history are greater than number of up-moves in history, then there will be more of an up-pull than a down-pull for the next price movement $X_{t+1} - X_t$ (likewise, the other way round when $U_t > D_t$). The extent of this "reverse pull" is controlled by the "pull strength" parameter α_3 (governed by the sigmoid-shaped function f).

Again, note that if we model the state S_t as X_t, we won't satisfy the Markov Property because the probabilities of next state $S_{t+1} = X_{t+1}$ depends on the entire history of stock price moves and not just on the current state $S_t = X_t$. However, we can again do something clever and create a compact enough state S_t consisting of simply the pair (U_t, D_t). With this representation for the state S_t, the Markov Property is indeed satisfied:

$$\mathbb{P}[(U_{t+1}, D_{t+1})|(U_t, D_t), (U_{t-1}, D_{t-1}), \ldots, (U_0, D_0)] = \mathbb{P}[(U_{t+1}, D_{t+1})|(U_t, D_t)]$$

$$= \begin{cases} f(\frac{D_t}{U_t+D_t}; \alpha_3) & \text{if } U_{t+1} = U_t + 1, D_{t+1} = D_t \\ f(\frac{U_t}{U_t+D_t}; \alpha_3) & \text{if } U_{t+1} = U_t, D_{t+1} = D_t + 1 \end{cases}$$

It is important to note that unlike Processes 1 and 2, the stock price X_t is actually not part of the state S_t in Process 3. This is because U_t and D_t together contain sufficient information to capture the stock price X_t (since $X_t = X_0 + U_t - D_t$, and noting that X_0 is provided as a constant).

The corresponding dataclass for Process 2 is shown below:

```
@dataclass
class Process3:
    @dataclass
    class State:
        num_up_moves: int
        num_down_moves: int

    alpha3: float = 1.0  # strength of reverse-pull (non-negative value)

    def up_prob(self, state: State) -> float:
        total = state.num_up_moves + state.num_down_moves
        if total == 0:
            return 0.5
        elif state.num_down_moves == 0:
            return state.num_down_moves ** self.alpha3
        else:
            return 1. / (1 + (total / state.num_down_moves - 1) ** self.alpha3)

    def next_state(self, state: State) -> State:
        up_move: int = np.random.binomial(1, self.up_prob(state), 1)[0]
        return Process3.State(
            num_up_moves=state.num_up_moves + up_move,
            num_down_moves=state.num_down_moves + 1 - up_move
        )
```

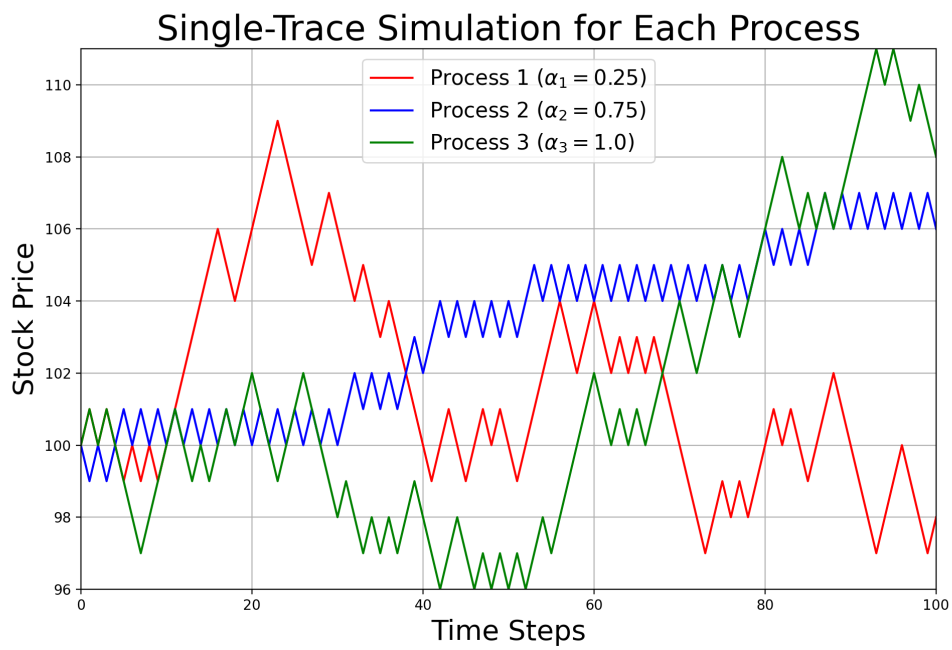

Figure 1.3.: Single Sampling Trace

The code for generation of sampling traces of the stock price is shown below:

```
def process3_price_traces(
    start_price: int,
    alpha3: float,
    time_steps: int,
    num_traces: int
) -> np.ndarray:
    process = Process3(alpha3=alpha3)
    start_state = Process3.State(num_up_moves=0, num_down_moves=0)
    return np.vstack([
        np.fromiter((start_price + s.num_up_moves - s.num_down_moves
                    for s in itertools.islice(simulation(process, start_state),
                                              time_steps + 1)), float)
        for _ in range(num_traces)])
```

As suggested for Process 1, you can plot graphs of sampling traces of the stock price, or plot graphs of the probability distributions of the stock price at various terminal time points T for Processes 2 and 3, by playing with this code.

Figure 1.3 shows a single sampling trace of stock prices for each of the 3 processes. Figure 1.4 shows the probability distribution of the stock price at terminal time $T = 100$ over 1000 traces.

Having developed the intuition for the Markov Property of States, we are now ready to formalize the notion of Markov Processes (some of the literature refers

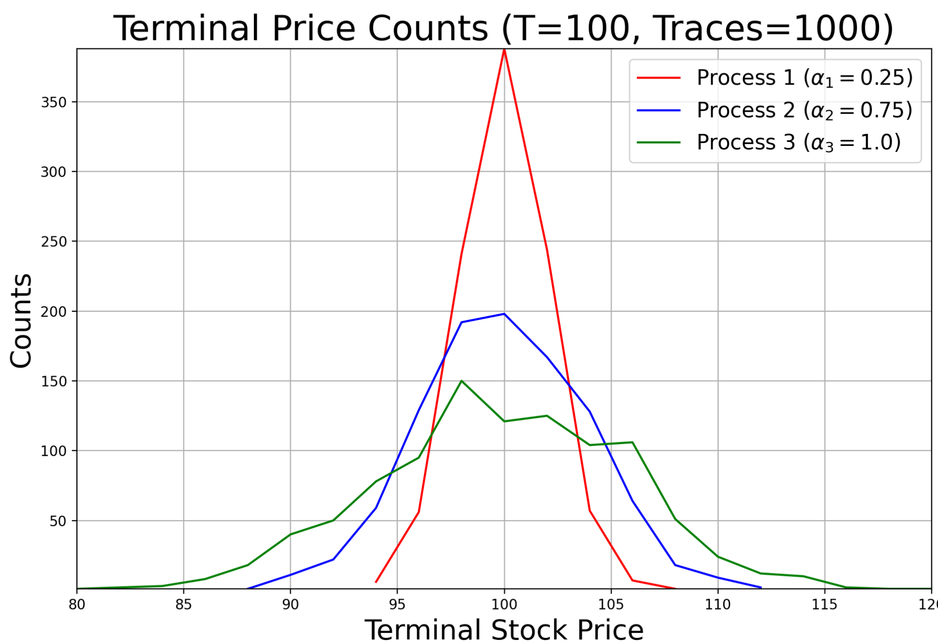

Figure 1.4.: Terminal Distribution

to Markov Processes as Markov Chains, but we will stick with the term Markov Processes).

Formal Definitions for Markov Processes

Our formal definitions in this book will be restricted to Discrete-Time Markov Processes, where time moves forward in discrete time steps $t = 0, 1, 2, \ldots$. Also for ease of exposition, our formal definitions in this book will be restricted to sets of states that are countable. A countable set can be either a finite set or an infinite set of the same cardinality as the set of natural numbers, i.e., a set that is enumerable (uncountable sets are those with cardinality larger than the set of natural numbers, eg: the set of real numbers, which are not enumerable). This book will cover examples of continuous-time Markov Processes, where time is a continuous variable (this leads to stochastic calculus, which is the foundation of some of the ground-breaking work in Mathematical Finance). This book will also cover examples of sets of states that are uncountable. However, for ease of exposition, our definitions and development of the theory in this book will be restricted to discrete-time and countable sets of states. The definitions and theory can be analogously extended to continuous-time or to uncountable sets of states (we request you to self-adjust the definitions and theory accordingly when you encounter continuous-time or uncountable sets of states in this book).

Definition 1.0.1. A *Markov Process* consists of:

- A countable set of states $\mathcal{S}$ (known as the State Space) and a set $\mathcal{T} \subseteq \mathcal{S}$ (known as the set of Terminal States)
- A time-indexed sequence of random states $S_t \in \mathcal{S}$ for time steps $t = 0, 1, 2, \ldots$ with each state transition satisfying the Markov Property: $\mathbb{P}[S_{t+1}|S_t, S_{t-1}, \ldots, S_0] = \mathbb{P}[S_{t+1}|S_t]$ for all $t \geq 0$.
- Termination: If an outcome for S_T (for some time step T) is a state in the set $\mathcal{T}$, then this sequence outcome terminates at time step T.

We refer to $\mathbb{P}[S_{t+1}|S_t]$ as the transition probabilities for time t.

Definition 1.0.2. A *Stationary Markov Process* is a Markov Process with the additional property that $\mathbb{P}[S_{t+1}|S_t]$ is independent of t.

This means, the dynamics of a Stationary Markov Process can be fully specified with the function

$$\mathcal{P} : (\mathcal{S} - \mathcal{T}) \times \mathcal{S} \to [0, 1]$$

such that $\mathcal{P}(s, s') = \mathbb{P}[S_{t+1} = s'|S_t = s]$ for all $s \in \mathcal{S} - \mathcal{T}, s' \in \mathcal{S}$. Hence, $\sum_{s' \in \mathcal{S}} \mathcal{P}(s, s') = 1$ for all $s \in \mathcal{S} - \mathcal{T}$. We refer to the function $\mathcal{P}$ as the transition probability function of the Stationary Markov Process, with the first argument to $\mathcal{P}$ to be thought of as the "source" state and the second argument as the "destination" state.

Note that this specification is devoid of the time index t (hence, the term *Stationary* which means "time-invariant"). Moreover, note that a non-Stationary Markov Process can be converted to a Stationary Markov Process by augmenting all states with the time index t. This means if the original state space of a non-Stationary Markov Process was $\mathcal{S}$, then the state space of the corresponding Stationary Markov Process is $\mathbb{Z}_{\geq 0} \times \mathcal{S}$ (where $\mathbb{Z}_{\geq 0}$ denotes the domain of the time index). This is because each time step has it's own unique set of (augmented) states, which means the entire set of states in $\mathbb{Z}_{\geq 0} \times \mathcal{S}$ can be covered by time-invariant transition probabilities, thus qualifying as a Stationary Markov Process. Therefore, henceforth, any time we say *Markov Process*, assume we are refering to a *Discrete-time Stationary Markov Process with a Countable State Space* (unless explicitly specified otherwise), which in turn will be characterized by the transition probability function $\mathcal{P}$. Note that the stock price examples (all 3 of the Processes we covered) are examples of a (Stationary) Markov Process, even without requiring augmenting the state with the time index.

The classical definitions and theory of Markov Processes model "termination" with the idea of *Absorbing States*. A state s is called an absorbing state if $\mathcal{P}(s, s) = 1$. This means, once we reach an absorbing state, we are "trapped" there, hence capturing the notion of "termination". So the classical definitions and theory of Markov Processes typically don't include an explicit specification of states as terminal and non-terminal. However, when we get to Markov Reward Processes and Markov Decision Processes (frameworks that are extensions of Markov Processes), we will need to explicitly specify states as terminal and non-terminal states, rather than model the notion of termination with absorbing states. So, for consistency in definitions and in the development of the theory, we are going with a framework where states in a Markov Process are explicitly specified as terminal or non-terminal states. We won't consider an absorbing state as a terminal state as the Markov Process keeps moving forward in time forever when it gets to an absorbing states. We will refer to $\mathcal{S} - \mathcal{T}$ as the set of Non-Terminal States $\mathcal{N}$ (and we will refer to a state in $\mathcal{N}$ as a non-terminal state). The sequence $S_0, S_1, S_2, \ldots$ terminates at time step $t = T$ if $S_T \in \mathcal{T}$.

Starting States

Now it's natural to ask the question: How do we "start" the Markov Process (in the stock price examples, this was the notion of the start state)? More generally, we'd like to specify a probability distribution of start states so we can perform simulations and (let's say) compute the probability distribution of states at specific future time steps. While this is a relevant question, we'd like to separate the following two specifications:

- Specification of the transition probability function $\mathcal{P}$
- Specification of the probability distribution of start states (denote this as $\mu : \mathcal{S} \rightarrow [0, 1]$)

We say that a Markov Process is fully specified by $\mathcal{P}$ in the sense that this gives us the transition probabilities that govern the complete dynamics of the Markov Process. A way to understand this is to relate specification of $\mathcal{P}$ to the specification of rules in a game (such as chess or monopoly). These games are specified with a finite (in fact, fairly compact) set of rules that is easy for a newbie to the game to understand. However, when we want to *actually play* the game, we need to specify the starting position (one could start these games at arbitrary, but legal, starting positions and not just at some canonical starting position). The specification of the start state of the game is analogous to the specification of μ. Given μ together with $\mathcal{P}$ enables us to generate sampling traces of the Markov Process (analogously, *play* games like chess or monopoly). These sampling traces typically result in a wide range of outcomes due to sampling and long-running of the Markov Process (versus compact specification of transition probabilities). These sampling traces enable us to answer questions such as probability distribution of states at specific future time steps or expected time of first occurrence of a specific state etc., given a certain starting probability distribution μ.

Thinking about the separation between specifying the rules of the game versus actually playing the game helps us understand the need to separate the notion of dynamics specification $\mathcal{P}$ (fundamental to the stationary character of the Markov Process) and the notion of starting distribution μ (required to perform sampling traces). Hence, the separation of concerns between $\mathcal{P}$ and μ is key to the conceptualization of Markov Processes. Likewise, we separate concerns in our code design as well, as evidenced by how we separated the `next_state` method in the Process dataclasses and the `simulation` function.

Terminal States

Games are examples of Markov Processes that terminate at specific states (based on rules for winning or losing the game). In general, in a Markov Process, termination might occur after a variable number of time steps (like in the games examples), or like we will see in many financial application examples, termination might occur after a fixed number of time steps, or like in the stock price examples we saw earlier, there is in fact no termination.

If all random sequences of states (sampling traces) reach a terminal state, then we say that these random sequences of the Markov Process are *Episodic* (otherwise we call these sequences as *Continuing*). The notion of episodic sequences is important in Reinforcement Learning since some Reinforcement Learning algorithms require episodic sequences.

When we cover some of the financial applications later in this book, we will find that the Markov Process terminates after a fixed number of time steps, say T. In these applications, the time index t is part of the state representation, each state with time index $t = T$ is labeled a terminal state, and all states with time index $t < T$ will transition to states with time index $t + 1$.

Now we are ready to write some code for Markov Processes, where we will illustrate how to specify that certain states are terminal states.

We create an abstract class `MarkovProcess` parameterized by a generic type (`TypeVar('S')`) representing a generic state space `Generic[S]`. The abstract class has an `@abstractmethod` called `transition` that is meant to specify the transition probability distribution of next states, given a current state. Note the return type of `transition`. It's `Optional[Distribution[S]]`. This means it's meant to return `None` if there is no next state (i.e., when you want to specify that `state` is a terminal state) or return `Distribution[S]` to specify the probability distribution of next states when `state` is a non-terminal state. We also have a convenience method `is_terminal` to query if a given state is terminal or not. We also have a method `simulate` that enables us to generate a sequence of sampled states starting from a specified `start_state_distribution: Distribution[S]` (from which we sample the starting state). The sampling of next states relies on the implementation of the `sample()` method in the `Distribution[S]` object produced by the `transition` method (note that the Distribution class hierarachy was covered in the chapter on *Design Paradigms for Applied Mathematics Implementations in Python*). This is the full body of the abstract class `MarkovProcess`:

```
from abc import ABC, abstractmethod
from rl.distribution import Distribution

S = TypeVar('S')

class MarkovProcess(ABC, Generic[S]):

    @abstractmethod
    def transition(self, state: S) -> Optional[Distribution[S]]:
        pass
```

```python
    def is_terminal(self, state: S) -> bool:
        return self.transition(state) is None

    def simulate(
        self,
        start_state_distribution: Distribution[S]
    ) -> Iterable[S]:
        state: S = start_state_distribution.sample()
        while True:
            yield state
            next_states = self.transition(state)
            if next_states is None:
                return

            state = next_states.sample()
```

Stock Price Examples modeled as Markov Processes

So if you have a mathematical specification of the transition probabilities of a Markov Process, all you need to do is to create a concrete class that implements the interface of the abstract class `MarkovProcess` (specifically by implementing the `@abstractmethod transition`) in a manner that captures your mathematical specification of the transition probabilities. Let us write this for the case of Process 3 (the 3rd example of stock price transitions we covered in the previous section). We will name the concrete class as `StockPriceMP3` (note that it's a `@dataclass` for convenience and simplicity). Note that the generic state space `S` is now replaced with a specific state space represented by the type `@dataclass StateMP3`. The code should be self-explanatory since we implemented this process as a standalone in the previous section. Note the use of the `Categorical` distribution in the `transition` method to capture the 2-outcomes probability distribution of next states (for movements up or down).

```python
from rl.distribution import Categorical
from rl.gen_utils.common_funcs import get_unit_sigmoid_func

@dataclass
class StateMP3:
    num_up_moves: int
    num_down_moves: int


@dataclass
class StockPriceMP3(MarkovProcess[StateMP3]):
```

```python
    alpha3: float = 1.0  # strength of reverse-pull (non-negative value)

    def up_prob(self, state: StateMP3) -> float:
        total = state.num_up_moves + state.num_down_moves
        return get_unit_sigmoid_func(self.alpha3)(
            state.num_down_moves / total
        ) if total else 0.5

    def transition(self, state: StateMP3) -> Categorical[StateMP3]:
        up_p = self.up_prob(state)

        return Categorical({
            StateMP3(state.num_up_moves + 1, state.num_down_moves): up_p,
            StateMP3(state.num_up_moves, state.num_down_moves + 1): 1 - up_p
        })
```

To generate sampling traces, we write the following function:

```python
from rl.distribution import Constant
import numpy as np

def process3_price_traces(
    start_price: int,
    alpha3: float,
    time_steps: int,
    num_traces: int
) -> np.ndarray:
    mp = StockPriceMP3(alpha3=alpha3)
    start_state_distribution = Constant(
        StateMP3(num_up_moves=0, num_down_moves=0)
    )
    return np.vstack([np.fromiter(
        (start_price + s.num_up_moves - s.num_down_moves for s in
         itertools.islice(
             mp.simulate(start_state_distribution),
             time_steps + 1
         )),
        float
    ) for _ in range(num_traces)])
```

We leave it to you as an exercise to similarly implement Stock Price Processes 1 and 2 that we had covered in the previous section. The complete code along with the driver to set input parameters, run all 3 processes and create plots is in the file rl/chapter2/stock_price_mp.py. We encourage you to change the input parameters in `__main__` and get an intuitive feel for how the simulation results vary with the changes in parameters.

Finite Markov Processes

Now let us consider Markov Processes with a finite state space. So we can represent the state space as $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$. Assume the set of non-terminal states $\mathcal{N}$ has $m \leq n$ states. Let us refer to Markov Processes with finite state spaces as Finite Markov Processes. Since Finite Markov Processes are a subclass of Markov Processes, it would make sense to create a concrete class `FiniteMarkovProcess` that implements the interface of the abstract class `MarkovProcess` (specifically implement the `@abstractmethod transition`). But first let's think about the data structure required to specify an instance of a `FiniteMarkovProcess` (i.e., the data structure we'd pass to the `__init__` method of `FiniteMarkovProcess`). One choice is a $m \times n$ 2D numpy array representation, i.e., matrix elements representing transition probabilities

$$\mathcal{P} : \mathcal{N} \times \mathcal{S} \rightarrow [0, 1]$$

However, we often find that this matrix can be sparse since one often transitions from a given state to just a few set of states. So we'd like a sparse representation and we can accomplish this by conceptualizing $\mathcal{P}$ in an equivalent curried form as follows:

$$\mathcal{N} \rightarrow (\mathcal{S} \rightarrow [0, 1])$$

With this curried view, we can represent the outer $\rightarrow$ as a map (in Python, as a dictionary of type `Mapping`) whose keys are the states in $\mathcal{S}$. A terminal-state key will map to `None` (since there are no transitions from a terminal state) and a non-terminal-state key maps to a `FiniteDistribution[S]` type that represents the inner $\rightarrow$, i.e. a finite probability distribution of the next states transitioned to from the non-terminal state (note: `FiniteDistribution` type was covered in the Chapter on *Design Paradigms for Applied Mathematics Implementations in Python*). Let us create an alias for this `Mapping` (called `Transition`) since we will use this data structure often:

```
Transition = Mapping[S, Optional[FiniteDistribution[S]]]
```

When the key in the `Mapping` is a non-terminal state, the `FiniteDistribution[S]` it maps to will only contain the set of states transitioned to from the non-terminal state with non-zero probability. To make things concrete, here's a toy `Transition` type example of a city with highly unpredictable weather outcomes from one day to the next (note: `Categorical` type inherits from `FiniteDistribution` type in the code at rl/distribution.py):

```
{
  "Rain": Categorical({"Rain": 0.3, "Nice": 0.7}),
  "Snow": Categorical({"Rain": 0.4, "Snow": 0.6}),
  "Nice": Categorical({"Rain": 0.2, "Snow": 0.3, "Nice": 0.5})
}
```

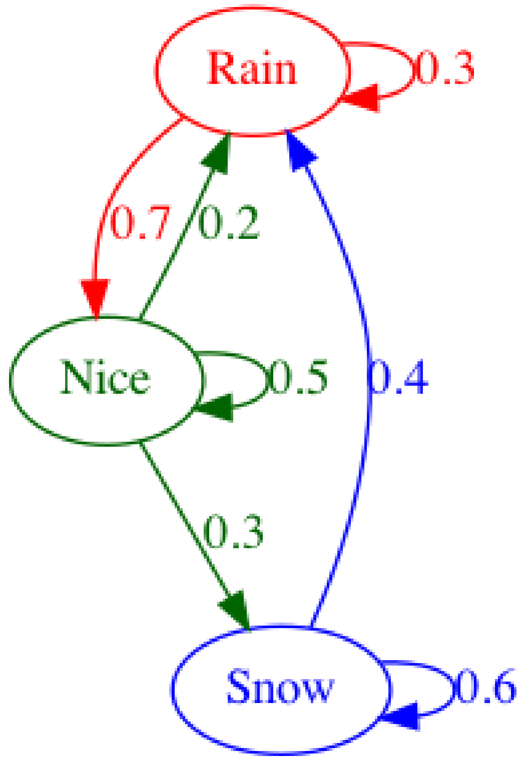

Figure 1.5.: Weather Markov Process

It is common to view this as a directed graph, as depicted in Figure 1.5. The nodes are the states and the directed edges are the probabilistic state transitions, with the transition probabilities labeled on them.

Now we are ready to write the code for the `FiniteMarkovProcess` class. The `__init__` method (constructor) takes as argument a `transition_map: Transition[S]` as we had described above. Along with the attribute `transition_map`, we also have an attribute `non_terminal_states: Sequence[S]` that is an ordered sequence of the non-terminal states. We implement the `transition` method by simply returning the `Optional[FiniteDistribution]` the given `state: S` maps to in the attribute `self.transition_map: Transition[S]`. Note that along with the `transition` method, we have implemented the `__repr__` method for a well-formatted display of `self.transition_map` and a method `states` that returns an `Iterable` enumerating the set of finite states in the `FiniteMarkovProcess`.

```
class FiniteMarkovProcess(MarkovProcess[S]):

    non_terminal_states: Sequence[S]
    transition_map: Transition[S]

    def __init__(self, transition_map: Transition[S]):
        self.non_terminal_states = [s for s, v in transition_map.items()
                                    if v is not None]
        self.transition_map = transition_map
```

```python
    def __repr__(self) -> str:
        display = ""

        for s, d in self.transition_map.items():
            if d is None:
                display += f"{s} is a Terminal State\n"
            else:
                display += f"From State {s}:\n"
                for s1, p in d:
                    display += f"  To State {s1} with Probability {p:.3f}\n"

        return display

    def transition(self, state: S) -> Optional[FiniteDistribution[S]]:
        return self.transition_map[state]

    def states(self) -> Iterable[S]:
        return self.transition_map.keys()
```

The above code is in the file rl/markov_process.py.

Simple Inventory Example

To help conceptualize Finite Markov Processes, let us consider a simple example of changes in inventory at a store. Assume you are the store manager and that you are tasked with controlling the ordering of inventory from a supplier. Let us focus on the inventory of a particular type of bicycle. Assume that each day there is random (non-negative integer) demand for the bicycle with the probabilities of demand following a Poisson distribution (with Poisson parameter $\lambda \in \mathbb{R}_{\geq 0}$), i.e. demand i for each $i = 0, 1, 2, \ldots$ occurs with probability

$$f(i) = \frac{e^{-\lambda}\lambda^i}{i!}$$

Denote $F : \mathbb{Z}_{\geq 0} \rightarrow [0, 1]$ as the poisson cumulative distribution function, i.e.,

$$F(i) = \sum_{j=0}^{i} f(j)$$

Assume you have storage capacity for at most $C \in \mathbb{Z}_{\geq 0}$ bicycles in your store. Each evening at 6pm when your store closes, you have the choice to order a certain number of bicycles from your supplier (including the option to not order any bicycles, on a given day). The ordered bicycles will arrive 36 hours later (at 6am the day after the day after you order - we refer to this as *delivery lead time* of 36 hours). Denote the *State* at 6pm store-closing each day as (α, β), where α

is the inventory in the store (refered to as On-Hand Inventory at 6pm) and β is the inventory on a truck from the supplier (that you had ordered the previous day) that will arrive in your store the next morning at 6am (β is refered to as On-Order Inventory at 6pm). Due to your storage capacity constraint of at most C bicycles, your ordering policy is to order $C - (\alpha + \beta)$ if $\alpha + \beta < C$ and to not order if $\alpha + \beta \geq C$. The precise sequence of events in a 24-hour cycle is:

- Observe the (α, β) *State* at 6pm store-closing (call this state S_t)
- Immediately order according to the ordering policy described above
- Receive bicycles at 6am if you had ordered 36 hours ago
- Open the store at 8am
- Experience random demand from customers according to demand probabilities stated above (number of bicycles sold for the day will be the minimum of demand on the day and inventory at store opening on the day)
- Close the store at 6pm and observe the state (this state is S_{t+1})

If we let this process run for a while, in steady-state we ensure that $\alpha+\beta \leq C$. So to model this process as a Finite Markov Process, we shall only consider the steady-state (finite) set of states

$$\mathcal{S} = \{(\alpha, \beta) : 0 \leq \alpha + \beta \leq C\}$$

So restricting ourselves to this finite set of states, our order quantity equals $C - (\alpha + \beta)$ when the state is (α, β).

If current state S_t is (α, β), there are only $\alpha + \beta + 1$ possible next states S_{t+1} as follows:

$$(\alpha + \beta - i, C - (\alpha + \beta)) \text{ for } i = 0, 1, \ldots, \alpha + \beta$$

with transition probabilities governed by the Poisson probabilities of demand as follows:

$$\mathcal{P}((\alpha, \beta), (\alpha + \beta - i, C - (\alpha + \beta))) = f(i) \text{ for } 0 \leq i \leq \alpha + \beta - 1$$

$$\mathcal{P}((\alpha, \beta), (0, C - (\alpha + \beta))) = \sum_{j=\alpha+\beta}^{\infty} f(j) = 1 - F(\alpha + \beta - 1)$$

Note that the next state's (S_{t+1}) On-Hand can be zero resulting from any of infinite possible demand outcomes greater than or equal to $\alpha + \beta$.

So we are now ready to write code for this simple inventory example as a Markov Process. All we have to do is to create a derived class inherited from `FiniteMarkovProcess` and write a method to construct the `transition_map: Transition`. Note that the generic state `S` is replaced here with the `@dataclass InventoryState` consisting of the pair of On-Hand and On-Order inventory quantities comprising the state of this Finite Markov Process.

```
from rl.distribution import Categorical
from scipy.stats import poisson
```

```python
@dataclass(frozen=True)
class InventoryState:
    on_hand: int
    on_order: int

    def inventory_position(self) -> int:
        return self.on_hand + self.on_order

class SimpleInventoryMPFinite(FiniteMarkovProcess[InventoryState]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float
    ):
        self.capacity: int = capacity
        self.poisson_lambda: float = poisson_lambda

        self.poisson_distr = poisson(poisson_lambda)
        super().__init__(self.get_transition_map())

    def get_transition_map(self) -> Transition[InventoryState]:
        d: Dict[InventoryState, Categorical[InventoryState]] = {}
        for alpha in range(self.capacity + 1):
            for beta in range(self.capacity + 1 - alpha):
                state = InventoryState(alpha, beta)
                ip = state.inventory_position()
                beta1 = self.capacity - ip
                state_probs_map: Mapping[InventoryState, float] = {
                    InventoryState(ip - i, beta1):
                    (self.poisson_distr.pmf(i) if i < ip else
                     1 - self.poisson_distr.cdf(ip - 1))
                    for i in range(ip + 1)
                }
                d[InventoryState(alpha, beta)] = Categorical(state_probs_map)
        return d
```

Let us utilize the __repr__ method written previously to view the transition probabilities for the simple case of $C = 2$ and $\lambda = 1.0$ (this code is in the file rl/chapter2/simple_inventory_mp.py)

```python
user_capacity = 2
user_poisson_lambda = 1.0

si_mp = SimpleInventoryMPFinite(
    capacity=user_capacity,
```

```
    poisson_lambda=user_poisson_lambda
)

print(si_mp)
```

The output we get is nicely displayed as:

```
From State InventoryState(on_hand=0, on_order=0):
  To State InventoryState(on_hand=0, on_order=2) with Probability 1.000
From State InventoryState(on_hand=0, on_order=1):
  To State InventoryState(on_hand=1, on_order=1) with Probability 0.368
  To State InventoryState(on_hand=0, on_order=1) with Probability 0.632
From State InventoryState(on_hand=0, on_order=2):
  To State InventoryState(on_hand=2, on_order=0) with Probability 0.368
  To State InventoryState(on_hand=1, on_order=0) with Probability 0.368
  To State InventoryState(on_hand=0, on_order=0) with Probability 0.264
From State InventoryState(on_hand=1, on_order=0):
  To State InventoryState(on_hand=1, on_order=1) with Probability 0.368
  To State InventoryState(on_hand=0, on_order=1) with Probability 0.632
From State InventoryState(on_hand=1, on_order=1):
  To State InventoryState(on_hand=2, on_order=0) with Probability 0.368
  To State InventoryState(on_hand=1, on_order=0) with Probability 0.368
  To State InventoryState(on_hand=0, on_order=0) with Probability 0.264
From State InventoryState(on_hand=2, on_order=0):
  To State InventoryState(on_hand=2, on_order=0) with Probability 0.368
  To State InventoryState(on_hand=1, on_order=0) with Probability 0.368
  To State InventoryState(on_hand=0, on_order=0) with Probability 0.264
```

For a graphical view of this Markov Process, see Figure 1.6. The nodes are the states, labeled with their corresponding α and β values. The directed edges are the probabilistic state transitions from 6pm on a day to 6pm on the next day, with the transition probabilities labeled on them.

We can perform a number of interesting experiments and calculations with this simple Markov Process and we encourage you to play with this code by changing values of the capacity C and poisson mean λ, performing simulations and probabilistic calculations of natural curiosity for a store owner.

There is a rich and interesting theory for Markov Processes. However, we will not get into this theory as our coverage of Markov Processes so far is a sufficient building block to take us to the incremental topics of Markov Reward Processes and Markov Decision Processes. However, before we move on, we'd like to show just a glimpse of the rich theory with the calculation of *Stationary Probabilities* and apply it to the case of the above simple inventory Markov Process.

Stationary Distribution of a Markov Process

Definition 1.0.3. The *Stationary Distribution* of a (Stationary) Markov Process with state space $\mathcal{S} = \mathcal{N}$ and transition probability function $\mathcal{P} : \mathcal{N} \times \mathcal{N} \rightarrow [0, 1]$

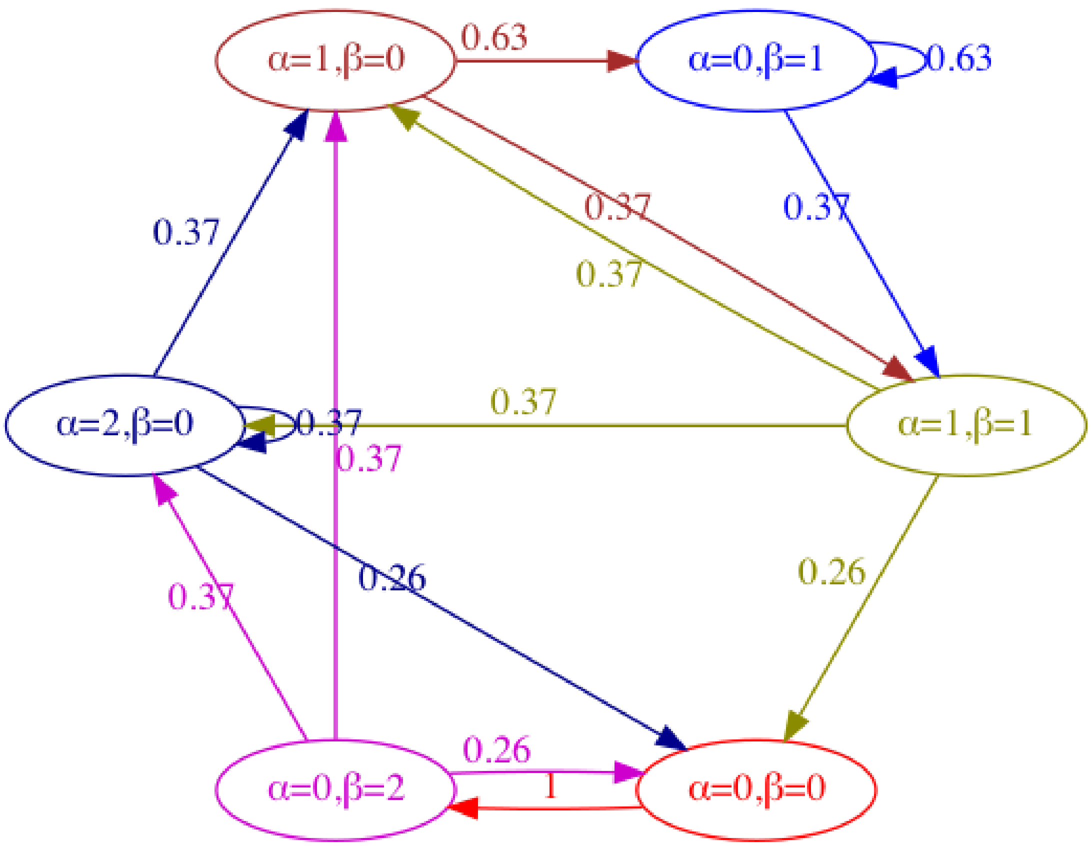

Figure 1.6.: Simple Inventory Markov Process

is a probability distribution function $\pi : \mathcal{N} \rightarrow [0, 1]$ such that:

$$\pi(s) = \sum_{s' \in \mathcal{N}} \pi(s) \cdot \mathcal{P}(s', s) \text{ for all } s \in \mathcal{N}$$

The intuitive view of the stationary distribution π is that (under specific conditions we are not listing here) if we let the Markov Process run forever, then in the long run the states occur at specific time steps with relative frequencies (probabilities) given by a distribution π that is independent of the time step. The probability of occurrence of a specific state s at a time step (asymptotically far out in the future) should be equal to the sum-product of probabilities of occurrence of all the states at the previous time step and the transition probabilities from those states to s. But since the states' occurrence probabilities are invariant in time, the π distribution for the previous time step is the same as the π distribution for the time step we considered. This argument holds for all states s, and that is exactly the statement of the definition of *Stationary Distribution* formalized above.

If we specialize this definition of *Stationary Distribution* to Finite-States Stationary Markov Processes with state space $\mathcal{S} = \{s_1, s_2, \ldots, s_n\} = \mathcal{N}$, then we can express the Stationary Distribution π as follows:

$$\pi(s_j) = \sum_{i=1}^{n} \pi(s_i) \cdot \mathcal{P}(s_i, s_j) \text{ for all } j = 1, 2, \ldots n$$

Below we use bold-face notation to represent functions as vectors and matrices (since we assume finite states). So, $\boldsymbol{\pi}$ is a column vector of length n and $\boldsymbol{\mathcal{P}}$ is the $n \times n$ transition probability matrix (rows are source states, columns are destination states with each row summing to 1). Then, the statement of the above definition can be succinctly expressed as:

$$\boldsymbol{\pi}^T = \boldsymbol{\pi}^T \cdot \boldsymbol{\mathcal{P}}$$

which can be re-written as:

$$\boldsymbol{\mathcal{P}}^T \cdot \boldsymbol{\pi} = \boldsymbol{\pi}$$

But this is simply saying that $\boldsymbol{\pi}$ is an eigenvector of $\boldsymbol{\mathcal{P}}^T$ with eigenvalue of 1. So then, it should be easy to obtain the stationary distribution $\boldsymbol{\pi}$ from an eigenvectors and eigenvalues calculation of $\boldsymbol{\mathcal{P}}^T$.

Let us write code to compute the stationary distribution. We shall add two methods in the `FiniteMarkovProcess` class, one for setting up the transition probability matrix $\boldsymbol{\mathcal{P}}$ (`get_transition_matrix` method) and another to calculate the stationary distribution $\boldsymbol{\pi}$ (`get_stationary_distribution`) from the transition probability matrix. Note that $\boldsymbol{\mathcal{P}}$ is restricted to $\mathcal{N} \times \mathcal{N} \rightarrow [0, 1]$ (rather than $\mathcal{N} \times \mathcal{S} \rightarrow [0, 1]$) because these probability transitions suffice for all the calculations we will be performing for Finite Markov Processes. Here's the code for the two methods (the full code for `FiniteMarkovProcess` is in the file `rl/markov_process.py`):

```python
import numpy as np
from rl.distribution import Categorical

    def get_transition_matrix(self) -> np.ndarray:
        sz = len(self.non_terminal_states)
        mat = np.zeros((sz, sz))

        for i, s1 in enumerate(self.non_terminal_states):
            for j, s2 in enumerate(self.non_terminal_states):
                mat[i, j] = self.transition(s1).probability(s2)

        return mat

    def get_stationary_distribution(self) -> FiniteDistribution[S]:
        eig_vals, eig_vecs = np.linalg.eig(self.get_transition_matrix().T)
        index_of_first_unit_eig_val = np.where(
            np.abs(eig_vals - 1) < 1e-8)[0][0]
        eig_vec_of_unit_eig_val = np.real(
            eig_vecs[:, index_of_first_unit_eig_val])
        return Categorical({
            self.non_terminal_states[i]: ev
            for i, ev in enumerate(eig_vec_of_unit_eig_val /
                                   sum(eig_vec_of_unit_eig_val))
        })
```

We will skip the theory that tells us about the conditions under which a stationary distribution is well-defined, or the conditions under which there is a unique stationary distribution. Instead, we will just go ahead with this calculation here assuming this Markov Process satisfies those conditions (it does!). So, we simply seek the index of the `eig_vals` vector with eigenvalue equal to 1 (accounting for floating-point error). Next, we pull out the column of the `eig_vecs` matrix at the `eig_vals` index calculated above, and convert it into a real-valued vector (eigenvectors/eigenvalues calculations are, in general, complex numbers calculations - see the reference for the `np.linalg.eig` function). So this gives us the real-valued eigenvector with eigenvalue equal to 1. Finally, we have to normalize the eigenvector so it's values add up to 1 (since we want probabilities), and return the probabilities as a `Categorical` distribution).

Running this code for the simple case of capacity $C = 2$ and poisson mean $\lambda = 1.0$ (instance of `SimpleInventoryMPFinite`) produces the following output for the stationary distribution π:

```
{InventoryState(on_hand=2, on_order=0): 0.162,
 InventoryState(on_hand=0, on_order=0): 0.117,
 InventoryState(on_hand=1, on_order=0): 0.162,
 InventoryState(on_hand=0, on_order=1): 0.279,
 InventoryState(on_hand=0, on_order=2): 0.117,
 InventoryState(on_hand=1, on_order=1): 0.162}
```

This tells us that On-Hand of 0 and On-Order of 1 is the state occurring most frequently (28% of the time) when the system is played out indefinitely.

Let us summarize the 3 different representations we've covered:

- Functional Representation: as given by the `transition` method, i.e., given a state, the `transition` method returns a probability distribution of next states (or `None`). This representation is valuable when performing simulations by sampling the next state from the returned probability distribution of the next state. This is applicable to the general case of Markov Processes (including infinite state spaces).
- Sparse Data Structure Representation: as given by `transition map: Transition`, which is convenient for compact storage and useful for visualization (eg: `__repr__` method display or as a directed graph figure). This is applicable only to Finite Markov Processes.
- Dense Data Structure Representation: as given by the `get_transition_matrix` 2D numpy array, which is useful for performing linear algebra that is often required to calculate mathematical properties of the process (eg: to calculate the stationary distribution). This is applicable only to Finite Markov Processes.

Now we are ready to move to our next topic of *Markov Reward Processes*. We'd like to finish this section by stating that the Markov Property owes its name to a mathematician from a century ago - Andrey Markov. Although the Markov Property seems like a simple enough concept, the concept has had profound implications on our ability to compute or reason with systems involving time-sequenced uncertainty in practice.

Formalism of Markov Reward Processes

As we've said earlier, the reason we covered Markov Processes is because we want to make our way to Markov Decision Processes (the framework for Reinforcement Learning algorithms) by adding incremental features to Markov Processes. Now we cover an intermediate framework between Markov Processes and Markov Decision Processes, known as Markov Reward Processes. We essentially just include the notion of a numerical *reward* to a Markov Process each time we transition from one state to the next. These rewards will be random, and all we need to do is to specify the probability distributions of these rewards as we make state transitions.

The main purpose of Markov Reward Processes is to calculate how much reward we would accumulate (in expectation, from each of the non-terminal states) if we let the Process run indefinitely, bearing in mind that future rewards need to be discounted appropriately (otherwise the sum of rewards could blow up to ∞). In order to solve the problem of calculating expected accumulative rewards from each non-terminal state, we will first set up some formalism for Markov Reward Processes, develop some (elegant) theory on calculating rewards accumulation, write plenty of code (based on the theory), and apply the theory and code to the simple inventory example (which we will embellish with rewards equal to negative of the costs incurred at the store).

Definition 1.0.4. A *Markov Reward Process* is a Markov Process, along with a time-indexed sequence of *Reward* random variables $R_t \in \mathbb{R}$ for time steps $t = 1, 2, \ldots$, satisfying the Markov Property (including Rewards): $\mathbb{P}[(R_{t+1}, S_{t+1}) | S_t, S_{t-1}, \ldots, S_0] = \mathbb{P}[(R_{t+1}, S_{t+1}) | S_t]$ for all $t \geq 0$.

It pays to emphasize again (like we emphasized for Markov Processes), that the definitions and theory of Markov Reward Processes are for discrete-time, for countable state spaces and countable set of pairs of next state and reward transitions (with the knowledge that the definitions and theory are analogously extensible to continuous-time and uncountable spaces/transitions). Since we commonly assume Stationarity of Markov Processes, we shall also (by default) assume Stationarity for Markov Reward Processes, i.e., $\mathbb{P}[(R_{t+1}, S_{t+1}) | S_t]$ is independent of t.

With the default assumption of stationarity, the transition probabilities of a Markov Reward Process can, in the most general case, be expressed as a transition probability function:

$$\mathcal{P}_R : \mathcal{N} \times \mathbb{R} \times \mathcal{S} \rightarrow [0, 1]$$

defined as:

$$\mathcal{P}_R(s, r, s') = \mathbb{P}[(R_{t+1} = r, S_{t+1} = s') | S_t = s]$$

such that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathbb{R}} \mathcal{P}_R(s, r, s') = 1 \text{ for all } s \in \mathcal{N}$$

The subsection on *Start States* we had covered for Markov Processes naturally applies to Markov Reward Processes as well. So we won't repeat the section here, rather we will simply highlight that when it comes to simulations, we need a separate specification of the probability distribution of start states. Also, by inheriting from our framework of Markov Processes, we model the notion of a "process termination" by explicitly specifying states as terminal states or non-terminal states. The sequence $S_0, R_1, S_1, R_2, S_2, \ldots$ terminates at time step $t = T$ if $S_T \in \mathcal{T}$, with R_T being the final reward in the sequence.

If all random sequences of states in a Markov Reward Process terminate, we refer to it as *episodic* sequences (otherwise, we refer to it as *continuing* sequences).

Let's write some code that captures this formalism. We create a derived `@abstractclass MarkovRewardProcess` that inherits from the `@abstractclass MarkovProcess`. Analogous to `MarkovProcess`'s `@abstractmethod transition` (that represents $\mathcal{P}$), `MarkovRewardProcess` has an `@abstractmethod transition_reward` that represents $\mathcal{P}_R$. Note that the return type of `transition_reward` is `Optional[Distribution[Tuple[S, float]]]` which means it returns `None` for a terminal `state: S` and it returns `Distribution[Tuple[S, float]]` for a non-terminal `state: S`, representing the probability distribution of (next state, reward) pairs transitioned to.

Also, analogous to `MarkovProcess`'s `simulate` method, `MarkovRewardProcess` has the method `simulate_reward` which generates a stream of `TransitionStep` objects. Each `TransitionStep` object consists of a 3-tuple: (state, next state, reward) representing the sampled transitions from the states visited in the generated sampling trace. Here's the actual code:

```
@dataclass(frozen=True)
class TransitionStep(Generic[S]):
    state: S
    next_state: S
    reward: float

class MarkovRewardProcess(MarkovProcess[S]):

    @abstractmethod
    def transition_reward(self, state: S)\
            -> Optional[Distribution[Tuple[S, float]]]:
        pass

    def simulate_reward(
        self,
        start_state_distribution: Distribution[S]
    ) -> Iterable[TransitionStep[S]]:
        state: S = start_state_distribution.sample()
        reward: float = 0.

        while True:
            next_distribution = self.transition_reward(state)
            if next_distribution is None:
                return

            next_state, reward = next_distribution.sample()
            yield TransitionStep(state, next_state, reward)

            state = next_state
```

So the idea is that if someone wants to model a Markov Reward Process, they'd simply have to create a concrete class that implements the interface of the `@abstractclass MarkovRewardProcess` (specifically implement the `@abstractmethod transition_reward`). But note that the `@abstractmethod transition` of `MarkovProcess` also needs to be implemented to make the whole thing concrete. However, we don't have to implement it in the concrete class implementing the interface of `MarkovRewardProcess` - in fact, we can implement it in the `MarkovRewardProcess` class itself by tapping the method `transition_reward`. Here's the code for the `transition` method in `MarkovRewardProcess`:

```
from rl.distribution import SampledDistribution
```

```python
    def transition(self, state: S) -> Optional[Distribution[S]]:
        distribution = self.transition_reward(state)
        if distribution is None:
            return None

        def next_state(distribution=distribution):
            next_s, _ = distribution.sample()
            return next_s

        return SampledDistribution(next_state)
```

Note that since the `transition_reward` method is abstract in `MarkovRewardProcess`, the only thing the `transition` method can do is to tap into the `sample` method of the abstract `Distribution` object produced by `transition_reward` and return a `SampledDistribution`. The full code for the `MarkovRewardProcess` class shown above is in the file rl/markov_process.py.

Now let us develop some more theory. Given a specification of $\mathcal{P}_R$, we can extract:

- The transition probability function $\mathcal{P} : \mathcal{N} \times \mathcal{S} \rightarrow [0, 1]$ of the implicit Markov Process defined as:

$$\mathcal{P}(s, s') = \sum_{r \in \mathbb{R}} \mathcal{P}_R(s, r, s')$$

- The reward transition function:

$$\mathcal{R}_T : \mathcal{N} \times \mathcal{S} \rightarrow \mathbb{R}$$

defined as:

$$\mathcal{R}_T(s, s') = \mathbb{E}[R_{t+1} | S_{t+1} = s', S_t = s] = \sum_{r \in \mathbb{R}} \frac{\mathcal{P}_R(s, r, s')}{\mathcal{P}(s, s')} \cdot r = \sum_{r \in \mathbb{R}} \frac{\mathcal{P}_R(s, r, s')}{\sum_{r \in \mathbb{R}} \mathcal{P}_R(s, r, s')} \cdot r$$

The Rewards specification of most Markov Reward Processes we encounter in practice can be directly expressed as the reward transition function $\mathcal{R}_T$ (versus the more general specification of $\mathcal{P}_R$). Lastly, we want to highlight that we can transform either of $\mathcal{P}_R$ or $\mathcal{R}_T$ into a "more compact" reward function that is sufficient to perform key calculations involving Markov Reward Processes. This reward function

$$\mathcal{R} : \mathcal{N} \rightarrow \mathbb{R}$$

is defined as:

$$\mathcal{R}(s) = \mathbb{E}[R_{t+1} | S_t = s] = \sum_{s' \in \mathcal{S}} \mathcal{P}(s, s') \cdot \mathcal{R}_T(s, s') = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathbb{R}} \mathcal{P}_R(s, r, s') \cdot r$$

We've created a bit of notational clutter here. So it would be a good idea for you to take a few minutes to pause, reflect and internalize the differences between $\mathcal{P}_R$, $\mathcal{P}$ (of the implicit Markov Process), $\mathcal{R}_T$ and $\mathcal{R}$. This notation will analogously re-appear when we learn about Markov Decision Processes in Chapter 2. Moreover, this notation will be used considerably in the rest of the book, so it pays to get comfortable with their semantics.

Simple Inventory Example as a Markov Reward Process

Now we return to the simple inventory example and embellish it with a reward structure to turn it into a Markov Reward Process (business costs will be modeled as negative rewards). Let us assume that your store business incurs two types of costs:

- Holding cost of h for each bicycle that remains in your store overnight. Think of this as "interest on inventory" - each day your bicycle remains unsold, you lose the opportunity to gain interest on the cash you paid to buy the bicycle. Holding cost also includes the cost of upkeep of inventory.
- Stockout cost of p for each unit of "missed demand", i.e., for each customer wanting to buy a bicycle that you could not satisfy with available inventory, eg: if 3 customers show up during the day wanting to buy a bicycle each, and you have only 1 bicycle at 8am (store opening time), then you lost two units of demand, incurring a cost of $2p$. Think of the cost of p per unit as the lost revenue plus disappointment for the customer. Typically $p \gg h$.

Let us go through the precise sequence of events, now with incorporation of rewards, in each 24-hour cycle:

- Observe the (α, β) *State* at 6pm store-closing (call this state S_t)
- Immediately order according to the ordering policy given by: Order quantity = $\max(C - (\alpha + \beta), 0)$
- Record any overnight holding cost incurred as described above
- Receive bicycles at 6am if you had ordered 36 hours ago
- Open the store at 8am
- Experience random demand from customers according to the specified poisson probabilities (poisson mean = λ)
- Record any stockout cost due to missed demand as described above
- Close the store at 6pm, register the reward R_{t+1} as the negative sum of overnight holding cost and the day's stockout cost, and observe the state (this state is S_{t+1})

Since the customer demand on any day can be an infinite set of possibilities (poisson distribution over the entire range of non-negative integers), we have an infinite set of pairs of next state and reward we could transition to from a given current state. Let's see what the probabilities of each of these transitions looks like. For a given current state $S_t := (\alpha, \beta)$, if customer demand for the day is i, then the next state S_{t+1} is:

$$(\max(\alpha + \beta - i, 0), \max(C - (\alpha + \beta), 0))$$

and the reward R_{t+1} is:

$$-h \cdot \alpha - p \cdot \max(i - (\alpha + \beta), 0)$$

Note that the overnight holding cost applies to each unit of on-hand inventory at store closing ($= \alpha$) and the stockout cost applies only to any units of "missed demand" ($= \max(i - (\alpha + \beta), 0)$). Since two different values of demand $i \in \mathbb{Z}_{\geq 0}$ do not collide on any unique pair (s', r) of next state and reward, we can express the transition probability function $\mathcal{P}_R$ for this Simple Inventory Example as a Markov Reward Process as:

$$\mathcal{P}_R((\alpha, \beta), -h \cdot \alpha - p \cdot \max(i - (\alpha + \beta), 0), (\max(\alpha + \beta - i, 0), \max(C - (\alpha + \beta), 0)))$$
$$= \frac{e^{-\lambda} \lambda^i}{i!} \text{ for all } i = 0, 1, 2, \ldots$$

Now let's write some code to implement this simple inventory example as a Markov Reward Process as described above. All we have to do is to create a concrete class implementing the interface of the abstract class `MarkovRewardProcess` (specifically implement the `@abstractmethod transition_reward`). The code below in `transition_reward` method in `class SimpleInventoryMRP` samples the customer demand from a Poisson distribution, uses the above formulas for the pair of next state and reward as a function of the customer demand sample, and returns an instance of `SampledDistribution`. Note that the generic state `S` is replaced here with the `@dataclass InventoryState` to represent a state of this Markov Reward Process, comprising of the On-Hand and On-Order inventory quantities.

```
from rl.distribution import SampledDistribution
import numpy as np

@dataclass(frozen=True)
class InventoryState:
    on_hand: int
    on_order: int

    def inventory_position(self) -> int:
        return self.on_hand + self.on_order


class SimpleInventoryMRP(MarkovRewardProcess[InventoryState]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float,
        holding_cost: float,
        stockout_cost: float
    ):
        self.capacity = capacity
```

```python
        self.poisson_lambda: float = poisson_lambda
        self.holding_cost: float = holding_cost
        self.stockout_cost: float = stockout_cost

    def transition_reward(
        self,
        state: InventoryState
    ) -> SampledDistribution[Tuple[InventoryState, float]]:

        def sample_next_state_reward(state=state) ->\
                Tuple[InventoryState, float]:
            demand_sample: int = np.random.poisson(self.poisson_lambda)
            ip: int = state.inventory_position()
            next_state: InventoryState = InventoryState(
                max(ip - demand_sample, 0),
                max(self.capacity - ip, 0)
            )
            reward: float = - self.holding_cost * state.on_hand\
                - self.stockout_cost * max(demand_sample - ip, 0)
            return next_state, reward

        return SampledDistribution(sample_next_state_reward)
```

The above code can be found in the file rl/chapter2/simple_inventory_mrp.py. We leave it as an exercise for you to use the `simulate_reward` method inherited by `SimpleInventoryMRP` to perform simulations and analyze the statistics produced from the sampling traces.

Finite Markov Reward Processes

Certain calculations for Markov Reward Processes can be performed easily if:

- The state space is finite ($\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$), and
- The set of unique pairs of next state and reward transitions from each of the states in $\mathcal{N}$ is finite

If we satisfy the above two characteristics, we refer to the Markov Reward Process as a Finite Markov Reward Process. So let us write some code for a Finite Markov Reward Process. We create a concrete class `FiniteMarkovRewardProcess` that primarily inherits from `FiniteMarkovProcess` (a concrete class) and secondarily implements the interface of the abstract class `MarkovRewardProcess`. Our first task is to think about the data structure required to specify an instance of `FiniteMarkovRewardProcess` (i.e., the data structure we'd pass to the `__init__` method of `FiniteMarkovRewardProcess`). Analogous to how we curried $\mathcal{P}$ for a Markov Process as $\mathcal{N} \rightarrow (\mathcal{S} \rightarrow [0, 1])$ (where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ and $\mathcal{N}$ has

$m \leq n$ states), here we curry $\mathcal{P}_R$ as:

$$\mathcal{N} \rightarrow (\mathcal{S} \times \mathbb{R} \rightarrow [0, 1])$$

Since $\mathcal{S}$ is finite and since the set of unique pairs of next state and reward transitions are also finite, this leads to the analog of the `Transition` data type for the case of Finite Markov Reward Processes (named `RewardTransition`) as follows:

```
StateReward = FiniteDistribution[Tuple[S, float]]
RewardTransition = Mapping[S, Optional[StateReward[S]]]
```

With this as input to `__init__` (input named `transition_reward_map: RewardTransition[S]`), the `FiniteMarkovRewardProcess` class has three responsibilities:

- It needs to implement the `transition_reward` method analogous to the implementation of the `transition` method in `FiniteMarkovProcess`
- It needs to create a `transition_map: Transition` (extracted from `transition_reward_map: RewardTransition`) in order to instantiate its concrete parent `FiniteMarkovProcess`.
- It needs to compute the reward fuction $\mathcal{R} : \mathcal{N} \rightarrow \mathbb{R}$ from the transition probability function $\mathcal{P}_R$ (i.e. from `transition_reward_map: RewardTransition`) based on the expectation calculation we specified above (as mentioned earlier, $\mathcal{R}$ is key to the relevant calculations we shall soon be performing on Finite Markov Reward Processes). To perform further calculations with the reward function $\mathcal{R}$, we need to produce it as a 1D numpy array (i.e., a vector) attribute of the class (we name it as `reward_function_vec`).

Here's the code that fulfils the above three responsibilities:

```
import numpy as np
from rl.distribution import Categorical
from collections import defaultdict

class FiniteMarkovRewardProcess(FiniteMarkovProcess[S],
                                MarkovRewardProcess[S]):

    transition_reward_map: RewardTransition[S]
    reward_function_vec: np.ndarray

    def __init__(self, transition_reward_map: RewardTransition[S]):

        transition_map: Dict[S, Optional[FiniteDistribution[S]]] = {}

        for state, trans in transition_reward_map.items():
            if trans is None:
                transition_map[state] = None
            else:
                probabilities: Dict[S, float] = defaultdict(float)
```

```
                for (next_state, _), probability in trans:
                    probabilities[next_state] += probability

                transition_map[state] = Categorical(probabilities)

        super().__init__(transition_map)

        self.transition_reward_map = transition_reward_map

        self.reward_function_vec = np.array([
            sum(probability * reward for (_, reward), probability in
                transition_reward_map[state])
            for state in self.non_terminal_states
        ])

    def transition_reward(self, state: S) -> Optional[StateReward[S]]:
        return self.transition_reward_map[state]
```

The above code for `FiniteMarkovRewardProcess` (and more) is in the file rl/markov_process.py.

Simple Inventory Example as a Finite Markov Reward Process

Now we'd like to model the simple inventory example as a Finite Markov Reward Process so we can take advantage of the algorithms that apply to Finite Markov Reward Processes. As we've noted previously, our ordering policy ensures that in steady-state, the sum of On-Hand (denote as α) and On-Order (denote as β) won't exceed the capacity C. So we will constrain the set of states such that this condition is satisfied: $0 \leq \alpha + \beta \leq C$ (i.e., finite number of states). Although the set of states is finite, there are an infinite number of pairs of next state and reward outcomes possible from any given current state. This is because there are an infinite set of possibilities of customer demand on any given day (resulting in infinite possibilities of stockout cost, i.e., negative reward, on any day). To qualify as a Finite Markov Reward Process, we'll need to model in a manner such that we have a finite set of pairs of next state and reward outcomes from a given current state. So what we'll do is that instead of considering (S_{t+1}, R_{t+1}) as the pair of next state and reward, we will model the pair of next state and reward to instead be $(S_{t+1}, \mathbb{E}[R_{t+1}|(S_t, S_{t+1})])$ (we know $\mathcal{P}_R$ due to the Poisson probabilities of customer demand, so we can actually calculate this conditional expectation of reward). So given a state s, the pairs of next state and reward would be: $(s', \mathcal{R}_T(s, s'))$ for all the s' we transition to from s. Since the set of possible next states s' are finite, these newly-modeled rewards associated with the transitions ($\mathcal{R}_T(s, s')$) are also finite and hence, the set of pairs of next state and reward from any current state are also finite. Note that this creative alteration of the reward definition is purely to reduce this Markov Reward Process into a Finite Markov Reward Process. Let's now work out the calculation of the reward transition function $\mathcal{R}_T$.

When the next state's (S_{t+1}) On-Hand is greater than zero, it means all of the day's demand was satisfied with inventory that was available at store-opening ($= \alpha + \beta$), and hence, each of these next states S_{t+1} correspond to no stockout cost and only an overnight holding cost of $h\alpha$. Therefore,

$$\mathcal{R}_T((\alpha, \beta), (\alpha + \beta - i, C - (\alpha + \beta))) = -h\alpha \text{ for } 0 \leq i \leq \alpha + \beta - 1$$

When next state's (S_{t+1}) On-Hand is equal to zero, there are two possibilities:

1. The demand for the day was exactly $\alpha + \beta$, meaning all demand was satisifed with available store inventory (so no stockout cost and only overnight holding cost), or
2. The demand for the day was strictly greater than $\alpha + \beta$, meaning there's some stockout cost in addition to overnight holding cost. The exact stockout cost is an expectation calculation involving the number of units of missed demand under the corresponding poisson probabilities of demand exceeding $\alpha + \beta$.

This calculation is shown below:

$$\mathcal{R}_T((\alpha, \beta), (0, C - (\alpha + \beta))) = -h\alpha - p(\sum_{j=\alpha+\beta+1}^{\infty} f(j) \cdot (j - (\alpha + \beta)))$$

$$= -h\alpha - p(\lambda(1 - F(\alpha + \beta - 1)) - (\alpha + \beta)(1 - F(\alpha + \beta)))$$

So now we have a specification of $\mathcal{R}_T$, but when it comes to our coding interface, we are expected to specify $\mathcal{P}_R$ as that is the interface through which we create a `FiniteMarkovRewardProcess`. Fear not - a specification of $\mathcal{P}_R$ is easy once we have a specification of $\mathcal{R}_T$. We simply create 4-tuples (s, r, s', p) for all $s \in \mathcal{N}, s' \in \mathcal{S}$ such that $r = \mathcal{R}_T(s, s')$ and $p = \mathcal{P}(s, s')$ (we know $\mathcal{P}$ along with $\mathcal{R}_T$), and the set of all these 4-tuples (for all $s \in \mathcal{N}, s' \in \mathcal{S}$) constitute the specification of $\mathcal{P}_R$, i.e., $\mathcal{P}_R(s, r, s') = p$. This turns our reward-definition-altered mathematical model of a Finite Markov Reward Process into a programming model of the `FiniteMarkovRewardProcess` class. This reward-definition-altered model enables us to gain from the fact that we can leverage the algorithms we'll be writing for Finite Markov Reward Processes (including some simple and elegant linear-algebra-based solutions). The downside of this reward-definition-altered model is that it prevents us from generating samples of the specific rewards encountered when transitioning from one state to another (because we no longer capture the probabilities of individual reward outcomes). Note that we can indeed generate sampling traces, but each transition step in the sampling trace will only show us the "mean reward" (specifically, the expected reward conditioned on current state and next state).

In fact, most Markov Processes you'd encounter in practice can be modeled as a combination of $\mathcal{R}_T$ and $\mathcal{P}$, and you'd simply follow the above $\mathcal{R}_T$ to $\mathcal{P}_R$

representation transformation drill to present this information in the form of $\mathcal{P}_R$ to instantiate a `FiniteMarkovRewardProcess`. We designed the interface to accept $\mathcal{P}_R$ as input since that is the most general interface for specifying Markov Reward Processes.

So now let's write some code for the simple inventory example as a Finite Markov Reward Process as described above. All we have to do is to create a derived class inherited from `FiniteMarkovRewardProcess` and write a method to construct the `transition_reward_map: RewardTransition` (i.e., $\mathcal{P}_R$) that the `__init__` constuctor of `FiniteMarkovRewardProcess` requires as input. Note that the generic state `S` is replaced here with the `@dataclass InventoryState` to represent the inventory state, comprising of the On-Hand and On-Order inventory quantities.

```
from scipy.stats import poisson

@dataclass(frozen=True)
class InventoryState:
    on_hand: int
    on_order: int

    def inventory_position(self) -> int:
        return self.on_hand + self.on_order

class SimpleInventoryMRPFinite(FiniteMarkovRewardProcess[InventoryState]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float,
        holding_cost: float,
        stockout_cost: float
    ):
        self.capacity: int = capacity
        self.poisson_lambda: float = poisson_lambda
        self.holding_cost: float = holding_cost
        self.stockout_cost: float = stockout_cost

        self.poisson_distr = poisson(poisson_lambda)
        super().__init__(self.get_transition_reward_map())

    def get_transition_reward_map(self) -> RewardTransition[InventoryState]:
        d: Dict[InventoryState, Categorical[Tuple[InventoryState, float]]] = {}
        for alpha in range(self.capacity + 1):
            for beta in range(self.capacity + 1 - alpha):
                state = InventoryState(alpha, beta)
                ip = state.inventory_position()
```

```python
            beta1 = self.capacity - ip
            base_reward = - self.holding_cost * state.on_hand
            sr_probs_map: Dict[Tuple[InventoryState, float], float] =\
                {(InventoryState(ip - i, beta1), base_reward):
                 self.poisson_distr.pmf(i) for i in range(ip)}
            probability = 1 - self.poisson_distr.cdf(ip - 1)
            reward = base_reward - self.stockout_cost *\
                (probability * (self.poisson_lambda - ip) +
                 ip * self.poisson_distr.pmf(ip))
            sr_probs_map[(InventoryState(0, beta1), reward)] = probability
            d[state] = Categorical(sr_probs_map)
    return d
```

The above code is in the file rl/chapter2/simple_inventory_mrp.py). We encourage you to play with the inputs to `SimpleInventoryMRPFinite` in `__main__` and view the transition probabilities and rewards of the constructed Finite Markov Reward Process.

Value Function of a Markov Reward Process

Now we are ready to formally define the main problem involving Markov Reward Processes. As we've said earlier, we'd like to compute the "expected accumulated rewards" from any non-terminal state. However, if we simply add up the rewards in a sampling trace following time step t as $\sum_{i=t+1}^{\infty} R_i = R_{t+1} + R_{t+2} + \ldots$, the sum would often diverge to infinity. So we allow for rewards accumulation to be done with a discount factor $\gamma \in [0,1]$: We define the (random) *Return* G_t as the "discounted accumulation of future rewards" following time step t. Formally,

$$G_t = \sum_{i=t+1}^{\infty} \gamma^{i-t-1} \cdot R_i = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots$$

We use the above definition of *Return* even for a terminating sequence (say terminating at $t = T$, i.e., $S_T \in \mathcal{T}$), by treating $R_i = 0$ for all $i > T$.

Note that γ can range from a value of 0 on one extreme (called "myopic") to a value of 1 on another extreme (called "far-sighted"). "Myopic" means the Return is the same as Reward (no accumulation of future Rewards in the Return). With "far-sighted" ($\gamma = 1$), the Return calculation can diverge for continuing (non-terminating) Markov Reward Processes but "far-sighted" is indeed applicable for episodic Markov Reward Processes (where all random sequences of the process terminate). Apart from the Return divergence consideration, $\gamma < 1$ helps algorithms become more tractable (as we shall see later when we get to Reinforcement Learning). We should also point out that the reason to have $\gamma < 1$ is not just for mathematical convenience or computational tractability - there are valid modeling reasons to discount Rewards when accumulating to a Return. When Reward is modeled as a financial quantity (revenues, costs, profits etc.), as will be the case in most financial applications, it makes sense to incorporate time-value-of-money which is a fundamental concept in Economics/Finance that says there is greater benefit in receiving a dollar now versus later (which is the economic reason why interest is paid or earned). So it is common to set γ to be the discounting based on the prevailing interest rate ($\gamma = \frac{1}{1+r}$ where r is the interest rate over a single time step). Another technical reason for setting $\gamma < 1$ is that our models often don't fully capture future uncertainty and so, discounting with γ acts to undermine future rewards that might not be accurate (due to future uncertainty modeling limitations). Lastly, from an AI perspective, if we want to build machines that acts like humans, psychologists have indeed demonstrated that human/animal behavior prefers immediate reward over future reward.

As you might imagine now, we'd want to identify non-terminal states with large expected returns and those with small expected returns. This, in fact, is the main problem involving a Markov Reward Process - to compute the "Expected Return" associated with each non-terminal state in the Markov Reward Process. Formally, we are interested in computing the *Value Function*

$$V : \mathcal{N} \rightarrow \mathbb{R}$$

defined as:

$$V(s) = \mathbb{E}[G_t|S_t = s] \text{ for all } s \in \mathcal{N}, \text{ for all } t = 0, 1, 2, \ldots$$

For the rest of the book, we will assume that whenever we are talking about a Value Function, the discount factor γ is appropriate to ensure that the Expected Return from each state is finite.

Note that we are (as usual) assuming the fact that the Markov Reward Process is stationary (time-invariant probabilities of state transitions and rewards). Now we show a creative piece of mathematics due to Richard Bellman. Bellman noted that the Value Function has a recursive structure. Specifically,

$$\begin{aligned}
V(s) &= \mathbb{E}[R_{t+1}|S_t = s] + \gamma \cdot \mathbb{E}[R_{t+2}|S_t = s] + \gamma^2 \cdot \mathbb{E}[R_{t+3}|S_t = s] + \ldots \\
&= \mathcal{R}(s) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathbb{P}[S_{t+1} = s'|S_t = s] \cdot \mathbb{E}[R_{t+2}|S_{t+1} = s'] \\
&\quad + \gamma^2 \cdot \sum_{s' \in \mathcal{N}} \mathbb{P}[S_{t+1} = s'|S_t = s] \sum_{s'' \in \mathcal{N}} \mathbb{P}[S_{t+2} = s''|S_{t+1} = s'] \cdot \mathbb{E}[R_{t+3}|S_{t+2} = s''] \\
&\quad + \ldots \\
&= \mathcal{R}(s) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s, s') \cdot \mathcal{R}(s') + \gamma^2 \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s, s') \sum_{s'' \in \mathcal{N}} \mathcal{P}(s', s'') \cdot \mathcal{R}(s'') + \ldots \\
&= \mathcal{R}(s) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s, s') \cdot (\mathcal{R}(s') + \gamma \cdot \sum_{s'' \in \mathcal{N}} \mathcal{P}(s', s'') \cdot \mathcal{R}(s'') + \ldots) \\
&= \mathcal{R}(s) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s, s') \cdot V(s') \text{ for all } s \in \mathcal{N}
\end{aligned} \tag{1.1}$$

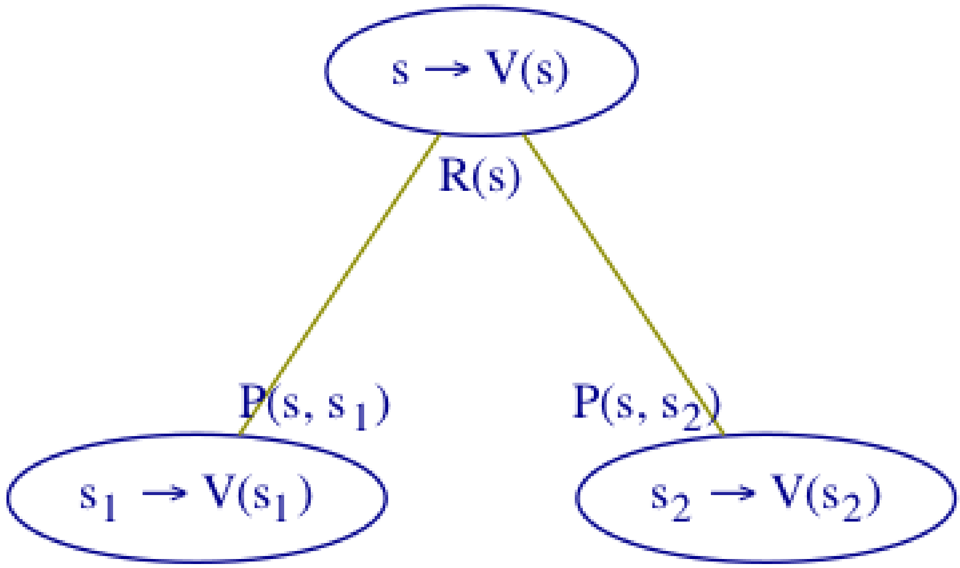

Figure 1.7.: Visualization of MRP Bellman Equation

Note that although the transitions to random states $s', s'', \ldots$ are in the state space of $\mathcal{S}$ rather than $\mathcal{N}$, the right-hand-side above sums over states $s', s'', \ldots$ only in $\mathcal{N}$ because transitions to terminal states (in $\mathcal{T} = \mathcal{S} - \mathcal{N}$) don't contribute any reward beyond the rewards produced *before reaching* the terminal state.

We refer to this recursive equation (1.1) for the Value Function as the Bellman Equation for Markov Reward Processes. Figure 1.7 is a convenient visualization aid of this important equation. In the rest of the book, we will depict quite a few of these type of state-transition visualizations to aid with creating mental models of key concepts.

For the case of Finite Markov Reward Processes, assume $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ and assume $\mathcal{N}$ has $m \leq n$ states. Below we use bold-face notation to represent functions as column vectors and matrices since we have finite states/transitions. So, $\boldsymbol{V}$ is a column vector of length m, $\boldsymbol{\mathcal{P}}$ is an $m \times m$ matrix, and $\boldsymbol{\mathcal{R}}$ is a column vector of length m (rows/columns corresponding to states in $\mathcal{N}$), so we can express the above equation in vector and matrix notation as follows:

$$\boldsymbol{V} = \boldsymbol{\mathcal{R}} + \gamma \boldsymbol{\mathcal{P}} \cdot \boldsymbol{V}$$

Therefore,

$$\Rightarrow \boldsymbol{V} = (\boldsymbol{I_m} - \gamma \boldsymbol{\mathcal{P}})^{-1} \cdot \boldsymbol{\mathcal{R}} \tag{1.2}$$

where $\boldsymbol{I_m}$ is the $m \times m$ identity matrix.

Let us write some code to implement the calculation of Equation (1.2). In the `FiniteMarkovRewardProcess` class, we implement the method `get_value_function_vec` that performs the above calculation for the Value Function V in terms of the reward function $\mathcal{R}$ and the transition probability function $\mathcal{P}$ of the implicit Markov Process. The Value Function V is produced as a 1D numpy array (i.e. a vector). Here's the code:

```
    def get_value_function_vec(self, gamma: float) -> np.ndarray:
        return np.linalg.inv(
            np.eye(len(self.non_terminal_states)) -
            gamma * self.get_transition_matrix()
        ).dot(self.reward_function_vec)
```

Invoking this `get_value_function_vec` method on `SimpleInventoryMRPFinite` for the simple case of capacity $C = 2$, poisson mean $\lambda = 1.0$, holding cost $h = 1.0$, stockout cost $p = 10.0$, and discount factor $\gamma = 0.9$ yields the following result:

```
{InventoryState(on_hand=0, on_order=0): -35.511,
 InventoryState(on_hand=1, on_order=0): -28.932,
 InventoryState(on_hand=0, on_order=1): -27.932,
 InventoryState(on_hand=0, on_order=2): -28.345,
 InventoryState(on_hand=2, on_order=0): -3

 0.345,
 InventoryState(on_hand=1, on_order=1): -29.345}
```

The corresponding values of the attribute `reward_function_vec` (i.e., $\mathcal{R}$) are:

```
{InventoryState(on_hand=0, on_order=0): -10.0,
 InventoryState(on_hand=1, on_order=0): -3.325,
 InventoryState(on_hand=0, on_order=1): -2.325,
 InventoryState(on_hand=0, on_order=2): -0.274,
 InventoryState(on_hand=2, on_order=0): -2.274,
 InventoryState(on_hand=1, on_order=1): -1.274}
```

This tells us that On-Hand of 0 and On-Order of 2 has the highest expected reward. However, the Value Function is highest for On-Hand of 0 and On-Order of 1.

This computation for the Value Function works if the state space is not too large (matrix to be inverted has size equal to number of non-terminal states). When the state space is large, this direct matrix-inversion method doesn't work and we have to resort to numerical methods to solve the recursive Bellman equation. This is the topic of Dynamic Programming and Reinforcement Learning algorithms that we shall learn in this book.

Summary of Key Learnings from this Chapter

Before we end this chapter, we'd like to highlight the two highly important concepts we learnt in this chapter:

- Markov Property: A concept that enables us to reason effectively and compute efficiently in practical systems involving sequential uncertainty

- Bellman Equation: A mathematical insight that enables us to express the Value Function recursively - this equation (and its Optimality version covered in Chapter 2) is in fact the core idea within all Dynamic Programming and Reinforcement Learning algorithms.

2. Markov Decision Processes

We've said before that this book is about "sequential decisioning" under "sequential uncertainty". In Chapter 1, we covered the "sequential uncertainty" aspect with the framework of Markov Processes, and we extended the framework to also incoporate the notion of uncertain "Reward" each time we make a state transition - we called this extended framework Markov Reward Processes. However, this framework had no notion of "sequential decisioning". In this chapter, we will further extend the framework of Markov Reward Processes to incorporate the notion of "sequential decisioning", formally known as Markov Decision Processes. Before we step into the formalism of Markov Decision Processes, let us develop some intuition and motivation for the need to have such a framework - to handle sequential decisioning. Let's do this by re-visiting the simple inventory example we covered in Chapter 1.

Simple Inventory Example: How much to Order?

When we covered the simple inventory example in Chapter 1 as a Markov Reward Process, the ordering policy was:

$$\theta = \max(C - (\alpha + \beta), 0)$$

where $\theta \in \mathbb{Z}_{\geq 0}$ is the order quantity, $C \in \mathbb{Z}_{\geq 0}$ is the space capacity (in bicycle units) at the store, α is the On-Hand Inventory and β is the On-Order Inventory ((α, β) comprising the *State*). We calculated the Value Function for the Markov Reward Process that results from following this policy. Now we ask the question: Is this Value Function good enough? More importantly, we ask the question: Can we improve this Value Function by following a different ordering policy? Perhaps by ordering less than that implied by the above formula for θ? This leads to the natural question - Can we identify the ordering policy that yields the *Optimal* Value Function (one with the highest expected returns, i.e., lowest expected accumulated costs, from each state)? Let us get an intuitive sense for this optimization problem by considering a concrete example.

Assume that instead of bicycles, we want to control the inventory of a specific type of toothpaste in the store. Assume you have space for 20 units of toothpaste on the shelf assigned to the toothpaste (assume there is no space in the backroom of the store). Asssume that customer demand follows a Poisson distribution with Poisson parameter $\lambda = 3.0$. At 6pm store-closing each evening, when you observe the *State* as (α, β), you now have a choice of ordering a quantity of toothpastes from any of the following values for the order quantity $\theta : \{0, 1, \ldots, \max(20 - (\alpha + \beta), 0)\}$. Let's say at Monday 6pm store-closing, $\alpha = 4$ and $\beta = 3$. So, you have a choice of order quantities from among the integers in the range of 0 to (20 - (4 + 3) = 13) (i.e., 14 choices). Previously, in the Markov Reward Process model, you would have ordered 13 units on Monday store-closing. This means on Wednesday morning at 6am, a truck would have arrived with 13 units of the toothpaste. If you sold say 2 units of the toothpaste on Tuesday, then on Wednesday 8am at store-opening, you'd have 4 + 3 - 2 + 13 = 18 units of toothpaste on your shelf. If you keep following this policy, you'd typically have almost a full shelf at store-opening each day, which covers almost a week worth of expected demand for the toothpaste. This means your risk of going out-of-stock on the toothpaste is extremely low, but you'd be incurring considerable holding cost (you'd have close to a full shelf of toothpastes sitting around almost each night). So as a store manager, you'd be thinking - "I can lower my costs by ordering less than that prescribed by the formula of $20 - (\alpha + \beta)$". But how much less? If you order too little, you'd start the day with too little inventory and might risk going out-of-stock. That's a risk you are highly uncomfortable with since the stockout cost per unit of missed demand (we called it p) is typically much higher than the holding cost per unit (we called it h). So you'd rather "err" on the side of having more inventory. But how much more? We also need to factor in the fact that the 36-hour lead time means a large order incurs large holding costs *two days later*. Most importantly, to find this right balance in terms of a precise mathematical optimization of the Value Function, we'd have to factor in the uncertainty of demand (based on daily Poisson probabilities) in our calculations. Now this gives you a flavor of the problem of sequential decisioning (each day you have to decide how much to order) under sequential uncertainty.

To deal with the "decisioning" aspect, we will introduce the notion of *Action* to complement the previously introduced notions of *State* and *Reward*. In the inventory example, the order quantity is our *Action*. After observing the *State*, we choose from among a set of Actions (in this case, we choose from within the set $\{0, 1, \ldots, \max(C - (\alpha + \beta), 0)\}$). We note that the Action we take upon observing a state affects the next day's state. This is because the next day's On-Order is exactly equal to today's order quantity (i.e., today's action). This in turn might affect our next day's action since the action (order quantity) is typically a function of the state (On-Hand and On-Order inventory). Also note that the Action we take on a given day will influence the Rewards after a couple of days (i.e. after the order arrives). It may affect our holding cost adversely if we had ordered too much or it may affect our stockout cost adversely if we had ordered too little and then experienced high demand.

The Difficulty of Sequential Decisioning under Uncertainty

This simple inventory example has given us a peek into the world of Markov Decision Processes, which in general, have two distinct (and inter-dependent) high-level features:

- At each time step t, an *Action* A_t is picked (from among a specified choice of actions) upon observing the *State* S_t
- Given an observed *State* S_t and a performed *Action* A_t, the probabilities of the state and reward of the next time step (S_{t+1} and R_{t+1}) are in general a function of not just the state S_t, but also of the action A_t.

We are tasked with maximizing the *Expected Return* from each state (i.e., maximizing the Value Function). This seems like a pretty hard problem in the general case because there is a cyclic interplay between:

- actions depending on state, on one hand, and

- next state/reward probabilities depending on action (and state) on the other hand.

There is also the challenge that actions might have delayed consequences on rewards, and it's not clear how to disentangle the effects of actions from different time steps on a future reward. So without direct correspondence between actions and rewards, how can we control the actions so as to maximize expected accumulated rewards? To answer this question, we will need to set up some notation and theory. Before we formally define the Markov Decision Process framework and it's associated (elegant) theory, let us set up a bit of terminology.

Using the language of AI, we say that at each time step t, the *Agent* (the algorithm we design) observes the state S_t, after which the Agent performs action A_t, after which the *Environment* (upon seeing S_t and A_t) produces a random pair: the next state state S_{t+1} and the next reward R_{t+1}, after which the *Agent* oberves this next state S_{t+1}, and the cycle repeats (until we reach a terminal state). This cyclic interplay is depicted in Figure 2.1. Note that time ticks over from t to $t+1$ when the environment sees the state S_t and action A_t.

Formal Definition of a Markov Decision Process

Similar to the definitions of Markov Processes and Markov Reward Processes, for ease of exposition, the definitions and theory of Markov Decision Processes below will be for discrete-time, for countable state spaces and countable set of pairs of next state and reward transitions (with the knowledge that the definitions and theory are analogously extensible to continuous-time and uncountable spaces, which we shall indeed encounter later in this book).

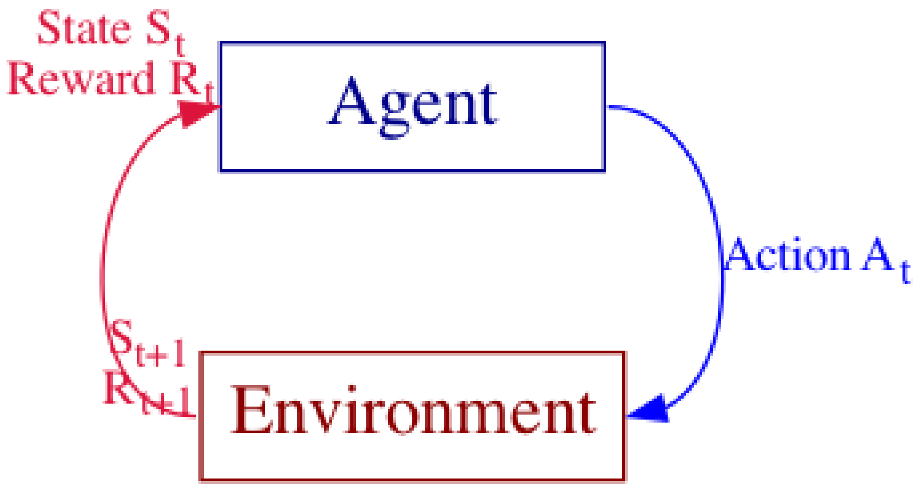

Figure 2.1.: Markov Decision Process

Definition 2.0.1. A *Markov Decision Process* comprises of:

- A countable set of states $\mathcal{S}$ (known as the State Space), a set $\mathcal{T} \subseteq \mathcal{S}$ (known as the set of Terminal States), and a countable set of actions $\mathcal{A}$

- A time-indexed sequence of environment-generated random states $S_t \in \mathcal{S}$ for time steps $t = 0, 1, 2, \ldots$, a time-indexed sequence of environment-generated *Reward* random variables $R_t \in \mathbb{R}$ for time steps $t = 1, 2, \ldots$, and a time-indexed sequence of agent-controllable actions $A_t \in \mathcal{A}$ for time steps $t = 0, 1, 2, \ldots$. (Sometimes we restrict the set of actions allowable from specific states, in which case, we abuse the $\mathcal{A}$ notation to refer to a function whose domain is $\mathcal{N}$ and range is $\mathcal{A}$, and we say that the set of actions allowable from a state $s \in \mathcal{N}$ is $\mathcal{A}(s)$.)

- Markov Property: $\mathbb{P}[(R_{t+1}, S_{t+1})|(S_t, A_t, S_{t-1}, A_{t-1}, \ldots, S_0, A_0)] = \mathbb{P}[(R_{t+1}, S_{t+1})|(S_t, A_t)]$ for all $t \geq 0$

- Termination: If an outcome for S_T (for some time step T) is a state in the set $\mathcal{T}$, then this sequence outcome terminates at time step T.

As in the case of Markov Reward Processes, we denote the set of non-terminal states $\mathcal{S} - \mathcal{T}$ as $\mathcal{N}$ and refer to any state in $\mathcal{N}$ as a non-terminal state. The sequence:

$$S_0, A_0, R_1, S_1, A_1, R_1, S_2, \ldots$$

terminates at time step T if $S_T \in \mathcal{T}$ (i.e., the final reward is R_T and the final action is A_{T-1}).

As in the case of Markov Processes and Markov Reward Processes, we shall (by default) assume Stationarity for Markov Decision Processes, i.e., $\mathbb{P}[(R_{t+1}, S_{t+1})|(S_t, A_t)]$ is independent of t. This means the transition probabilities of a Markov Decision Process can, in the most general case, be expressed as a state-reward transition probability function:

$$\mathcal{P}_R : \mathcal{N} \times \mathcal{A} \times \mathbb{R} \times \mathcal{S} \rightarrow [0, 1]$$

defined as:

$$\mathcal{P}_R(s, a, r, s') = \mathbb{P}[(R_{t+1} = r, S_{t+1} = s')|(S_t = s, A_t = a)]$$

such that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathbb{R}} \mathcal{P}_R(s, a, r, s') = 1 \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}$$

Henceforth, any time we say Markov Decision Process, assume we are referring to a Discrete-Time Stationary Markov Decision Process with countable spaces and countable transitions (unless explicitly specified otherwise), which in turn can be characterized by the state-reward transition probability function $\mathcal{P}_R$. Given a specification of $\mathcal{P}_R$, we can construct:

- The state transition probability function

$$\mathcal{P} : \mathcal{N} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$$

defined as:

$$\mathcal{P}(s, a, s') = \sum_{r \in \mathbb{R}} \mathcal{P}_R(s, a, r, s')$$

- The reward transition function:

$$\mathcal{R}_T : \mathcal{N} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$$

defined as:

$$\mathcal{R}_T(s, a, s') = \mathbb{E}[R_{t+1}|(S_{t+1} = s', S_t = s, A_t = a)]$$

$$= \sum_{r \in \mathbb{R}} \frac{\mathcal{P}_R(s, a, r, s')}{\mathcal{P}(s, a, s')} \cdot r = \sum_{r \in \mathbb{R}} \frac{\mathcal{P}_R(s, a, r, s')}{\sum_{r \in \mathbb{R}} \mathcal{P}_R(s, a, r, s')} \cdot r$$

The Rewards specification of most Markov Decision Processes we encounter in practice can be directly expressed as the reward transition function $\mathcal{R}_T$ (versus the more general specification of $\mathcal{P}_R$). Lastly, we want to highlight that we can transform either of $\mathcal{P}_R$ or $\mathcal{R}_T$ into a "more compact" reward function that is sufficient to perform key calculations involving Markov Decision Processes. This reward function

$$\mathcal{R} : \mathcal{N} \times \mathcal{A} \rightarrow \mathbb{R}$$

is defined as:

$$\mathcal{R}(s, a) = \mathbb{E}[R_{t+1}|(S_t = s, A_t = a)]$$

$$= \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') \cdot \mathcal{R}_T(s, a, s') = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathbb{R}} \mathcal{P}_R(s, a, r, s') \cdot r$$

Policy

Having understood the dynamics of a Markov Decision Process, we now move on to the specification of the *Agent*'s actions as a function of the current state. In the general case, we assume that the Agent will perform a random action A_t, according to a probability distribution that is a function of the current state S_t. We refer to this function as a *Policy*. Formally, a *Policy* is a function

$$\pi : \mathcal{N} \times \mathcal{A} \rightarrow [0, 1]$$

defined as:

$$\pi(s, a) = \mathbb{P}[A_t = a | S_t = s] \text{ for time steps } t = 0, 1, 2, \ldots, \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}$$

Note that the definition above assumes that a Policy is Markovian, i.e., the action probabilities depend only on the current state and not the history. The definition above also assumes that a Policy is Stationary, i.e., $\mathbb{P}[A_t = a | S_t = s]$ is invariant in time t. If we do encounter a situation where the policy would need to depend on the time t, we'll simply include t to be part of the state, which would make the Policy stationary (albeit at the cost of state-space bloat and hence, computational cost).

When we have a policy such that the action probability distribution for each state is concentrated on a single action, we refer to it as a deterministic policy. Formally, a deterministic policy has the property that for all $s \in \mathcal{N}$,

$$\pi(s, \pi_D(s)) = 1 \text{ and } \pi(s, a) = 0 \text{ for all } a \in \mathcal{A} \text{ with } a \neq \pi_D(s)$$

where $\pi_D : \mathcal{N} \rightarrow \mathcal{A}$.

So we shall denote deterministic policies simply with the function π_D. We shall refer to non-deterministic policies as stochastic policies (the word stochastic reflecting the fact that the agent will perform a random action according to the probability distribution specified by π). So when we use the notation π, assume that we are dealing with a stochastic (i.e., non-deterministic) policy and when we use the notation π_D, assume that we are dealing with a deterministic policy.

Let's write some code to get a grip on the concept of Policy - we start with the design of an abstract class called `Policy` that represents a general Policy, as we have articulated above. The only method it contains is an `@abstractmethod act` that accepts as input a `state: S` (as seen before in the classes `MarkovProcess` and `MarkovRewardProcess`, `S` is a generic type to represent a generic state) and produces as output `None` for a terminal `state: S` and a `Distribution[A]` for a non-terminal `state: S` representing the probability distribution of the random action as a function of the input state.

```
A = TypeVar('A')
S = TypeVar('S')
```

```
class Policy(ABC, Generic[S, A]):

    @abstractmethod
    def act(self, state: S) -> Optional[Distribution[A]]:
        pass
```

The simplest type of `Policy` is one that produces a single fixed action with 100% probability for each state. We can represent such a policy with the following `class Always`:

```
class Always(Policy[S, A]):
    action: A

    def __init__(self, action: A):
        self.action = action

    def act(self, _: S) -> Optional[Distribution[A]]:
        return Constant(self.action)
```

Now let's write some code to create some concrete policies for an example we are familiar with - the simple inventory example. We first create a concrete class `SimpleInventoryDeterministicPolicy` for deterministic inventory replenishment policies that implements the interface of the abstract class `Policy` (specifically implements the `@abstractmethod act`). Note that the generic state `S` is replaced here with the class `InventoryState` that represents a state in the inventory example, comprising of the On-Hand and On-Order inventory quantities. Also note that the generic action `A` is replaced here with the `int` type since in this example, the action is the quantity of inventory to be ordered at store-closing (which is an integer quantity). Note that since our class is meant to produce a deterministic policy, the `act` method returns a `Constant[int]` which is a probability distribution with 100% of the probability concentrated at a single `int` value (`int` represents the integer quantity of inventory to be ordered). The code in `act` implements the following deterministic policy:

$$\pi_D((\alpha, \beta)) = \max(C - (\alpha + \beta), 0)$$

where C is a parameter representing the "reorder point" (meaning, we order only when the inventory position falls below the "reorder point"), α is the On-Hand Inventory at store-closing, β is the On-Order Inventory at store-closing, and inventory position is equal to $\alpha + \beta$. In Chapter 1, we set the reorder point to be equal to the store capacity C.

```
from rl.distribution import Constant

@dataclass(frozen=True)
class InventoryState:
    on_hand: int
```

```python
    on_order: int

    def inventory_position(self) -> int:
        return self.on_hand + self.on_order

class SimpleInventoryDeterministicPolicy(Policy[InventoryState, int]):
    def __init__(self, reorder_point: int):
        self.reorder_point: int = reorder_point

    def act(self, state: InventoryState) -> Constant[int]:
        return Constant(max(self.reorder_point - state.inventory_position(),
                            0))
```

We can instantiate a specific deterministic policy with a reorder point of say 10 as:

```python
si_dp = SimpleInventoryDeterministicPolicy(reorder_point=10)
```

Now let's write some code to create stochastic policies for the inventory example. Similar to `SimpleInventoryDeterministicPolicy`, we create a concrete class `SimpleInventoryStochasticPolicy` that implements the interface of the abstract class `Policy` (specifically implements the `@abstractmethod act`). The code in `act` implements the following stochastic policy:

$$\pi((\alpha, \beta), \theta) = \frac{e^{-\lambda} \lambda^r}{r!}$$
$$\theta = \max(r - (\alpha + \beta), 0)$$

where $r \in \mathbb{Z}_{\geq 0}$ is the random re-order point with poisson probability distribution given by a specified poisson mean parameter $\lambda \in \mathbb{R}_{\geq 0}$, and $\theta \in \mathbb{Z}_{\geq 0}$ is the order quantity (action).

```python
import numpy as np
from rl.distribution import SampledDistribution

class SimpleInventoryStochasticPolicy(Policy[InventoryState, int]):
    def __init__(self, reorder_point_poisson_mean: float):
        self.reorder_point_poisson_mean: float = reorder_point_poisson_mean

    def act(self, state: InventoryState) -> SampledDistribution[int]:
        def action_func(state=state) -> int:
            reorder_point_sample: int = \
                np.random.poisson(self.reorder_point_poisson_mean)
            return max(reorder_point_sample - state.inventory_position(), 0)

        return SampledDistribution(action_func)
```

We can instantiate a specific stochastic policy with a reorder point poisson distribution mean of say 5.2 as:

```
si_sp = SimpleInventoryStochasticPolicy(reorder_point_poisson_mean=5.2)
```

We will revisit the simple inventory example in a bit after we cover the code for Markov Decision Processes, when we'll show how to simulate the Markov Decision Process for this simple inventory example, with the agent running a deterministic policy. But before we move on to the code design for Markov Decision Processes (to accompany the above implementation of Policies), we require an important insight linking Markov Decision Processes, Policies and Markov Reward Processes.

[Markov Decision Process, Policy] := Markov Reward Process

This section has an important insight - that if we evaluate a Markov Decision Process (MDP) with a fixed policy π (in general, with a fixed stochastic policy π), we get the Markov Reward Process (MRP) that is *implied* by the combination of the MDP and the policy π. Let's clarify this with notational precision. But first we need to point out that we have some notation clashes between MDP and MRP. We used $\mathcal{P}_R$ to denote the transition probability function of the MRP as well as to denote the state-reward transition probability function of the MDP. We used $\mathcal{P}$ to denote the transition probability function of the Markov Process implicit in the MRP as well as to denote the state transition probability function of the MDP. We used $\mathcal{R}_T$ to denote the reward transition function of the MRP as well as to denote the reward transition function of the MDP. We used $\mathcal{R}$ to denote the reward function of the MRP as well as to denote the reward function of the MDP. We can resolve these notation clashes by noting the arguments to $\mathcal{P}_R$, $\mathcal{P}, \mathcal{R}_T$ and $\mathcal{R}$, but to be extra-clear, we'll put a superscript of π to each of the functions $\mathcal{P}_R$, $\mathcal{P}, \mathcal{R}_T$ and $\mathcal{R}$ of the π-implied MRP so as to distinguish between these functions for the MDP versus the π-implied MRP.

Let's say we are given a fixed policy π and an MDP specified by it's state-reward transition probability function $\mathcal{P}_R$. Then the transition probability function $\mathcal{P}_R^\pi$ of the MRP implied by the evaluation of the MDP with the policy π is defined as:

$$\mathcal{P}_R^\pi(s, r, s') = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{P}_R(s, a, r, s')$$

Likewise,

$$\mathcal{P}^\pi(s, s') = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{P}(s, a, s')$$

$$\mathcal{R}_T^\pi(s, s') = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{R}_T(s, a, s')$$

$$\mathcal{R}^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{R}(s, a)$$

So any time we talk about an MDP evaluated with a fixed policy, you should know that we are effectively talking about the implied MRP. This insight is now going to be key in the design of our code to represent Markov Decision Processes.

We create an abstract class called `MarkovDecisionProcess` (code shown below) with two `@abstractmethods` - `step` and `actions`. The `step` method is key: it is meant to specify the distribution of pairs of next state and reward, given a state and action. The `actions` method's interface specifies that it takes as input a `state: S` and produces as output an `Iterable[A]` to represent the set of actions allowable for the input `state` (since the set of actions can be potentially infinite, in which case we'd have to return an `Iterator[A]`, the return type is fairly generic, i.e., `Iterable[A]`).

The `apply_policy` method takes as input a `policy: Policy[S, A]` and returns a `MarkovRewardProcess` representing the implied MRP. Let's understand the code in `apply_policy`: First, we construct a `class RewardProcess` that implements the `@abstractmethod transition_reward` of `MarkovRewardProcess`. `transition_reward` takes as input a `state: S`, creates `actions: Optional[Distribution[A]]` by applying the given `policy` on `state`, and finally uses the `apply` method of `Distribution` to transform `actions: Distribution[A]` into a `Distribution[Tuple[S, float]]` (distribution of (next state, reward) pairs) using the `@abstractmethod step`.

The `is_terminal` method takes as input a `state: S` and returns a `bool` signifying whether `state` is a terminal state or not. Since the `actions` method can returns the set of actions in the form of any `Iterable` type, the only way to check if it's an empty `Iterable` is by turning it into an `Iterator`, and checking if the `next` invocation on the `Iterator` triggers `StopIteration` (in which case, it would be an empty `Iterable`).

We also write the `simulate_actions` method that is analogous to the `simulate_reward` method we had written for `MarkovRewardProcess` for generating a sampling trace. In this case, each step in the sampling trace involves sampling an action from the given `policy` and then sampling the pair of next state and reward, given the `state` and sampled `action`. Each generated `TransitionStep` object consists of the 4-tuple: (state, action, next state, reward). Here's the actual code:

```
from rl.distribution import Distribution

@dataclass(frozen=True)
class TransitionStep(Generic[S, A]):
    state: S
    action: A
    next_state: S
    reward: float
```

```python
class MarkovDecisionProcess(ABC, Generic[S, A]):

    @abstractmethod
    def actions(self, state: S) -> Iterable[A]:
        pass

    @abstractmethod
    def step(
        self,
        state: S,
        action: A
    ) -> Optional[Distribution[Tuple[S, float]]]:
        pass

    def apply_policy(self, policy: Policy[S, A]) -> MarkovRewardProcess[S]:
        mdp = self

        class RewardProcess(MarkovRewardProcess[S]):
            def transition_reward(
                self,
                state: S
            ) -> Optional[Distribution[Tuple[S, float]]]:
                actions: Optional[Distribution[A]] = policy.act(state)
                if actions is None:
                    return None

                return actions.apply(lambda a: mdp.step(state, a))

        return RewardProcess()

    def is_terminal(self, state: S) -> bool:
        try:
            next(iter(self.actions(state)))
            return False
        except StopIteration:
            return True

    def simulate_actions(
            self,
            start_states: Distribution[S],
            policy: Policy[S, A]
    ) -> Iterable[TransitionStep[S, A]]:
        state: S = start_states.sample()
        reward: float = 0

        while True:
```

```python
        action_distribution = policy.act(state)
        if action_distribution is None:
            return

        action = action_distribution.sample()
        next_distribution = self.step(state, action)
        if next_distribution is None:
            return

        next_state, reward = next_distribution.sample()
        yield TransitionStep(state, action, next_state, reward)
        state = next_state
```

The above code for `Policy`, `Always` and `MarkovDecisionProcess` is in the file rl/markov_decision_process.py.

Simple Inventory Example with Unlimited Capacity (Infinite State/Action Space)

Now we come back to our simple inventory example. Unlike previous situations of this example, here we will assume that there is no space capacity constraint on toothpaste. This means we have a choice of ordering any (unlimited) non-negative integer quantity of toothpaste units. Therefore, the action space is infinite. Also, since the order quantity shows up as On-Order the next day and as delivered inventory the day after the next day, the On-Hand and On-Order quantities are also unbounded. Hence, the state space is infinite. Due to the infinite state and action spaces, we won't be able to take advantage of the so-called "Tabular Dynamic Programming Algorithms" we will cover in Chapter 3 (algorithms that are meant for finite state and action spaces). There is still significant value in modeling infinite MDPs of this type because we can perform simulations (by sampling from an infinite space). Simulations are valuable not just to explore various properties and metrics relevant in the real-world problem modeled with an MDP, but simulations also enable us to design approximate algorithms to calculate Value Functions for given policies as well as Optimal Value Functions (which is the ultimate purpose of modeling MDPs).

We will cover details on these approximate algorithms later in the book - for now, it's important for you to simply get familiar with how to model infinite MDPs of this type. This infinite-space inventory example serves as a great learning for an introduction to modeling an infinite (but countable) MDP.

We create a concrete class `SimpleInventoryMDPNoCap` that implements the abstract class `MarkovDecisionProcess` (specifically implements `@abstractmethod apply_policy` and `@abstractmethod actions`). The attributes `poisson_lambda`, `holding_cost` and `stockout_cost` have the same semantics as what we had covered for Markov Reward Processes in Chapter 1 (`SimpleInventoryMRP`). The step method takes as input a state: InventoryState and an order: int (representing the MDP action). We sample from the poisson probability distribution of customer demand (calling it demand_sample: int). Using order: int and demand_sample: int, we obtain a sample of the pair of next_state: InventoryState and reward: float. This sample pair is returned as a SampledDistribution object. The above sampling dynamics effectively describe the MDP in terms of this step method. The actions method returns an Iterator[int], an infinite generator of non-negative integers to represent the fact that the action space (order quantities) for any state comprise of all non-negative integers.

```
import itertools
import numpy as np
from rl.distribution import SampledDistribution

class SimpleInventoryMDPNoCap(MarkovDecisionProcess[InventoryState, int]):
    def __init__(self, poisson_lambda: float, holding_cost: float,
                 stockout_cost: float):
        self.poisson_lambda: float = poisson_lambda
        self.holding_cost: float = holding_cost
        self.stockout_cost: float = stockout_cost

    def step(
        self,
        state: InventoryState,
        order: int
    ) -> SampledDistribution[Tuple[InventoryState, float]]:

        def sample_next_state_reward(
            state=state,
            order=order
        ) -> Tuple[InventoryState, float]:
            demand_sample: int = np.random.poisson(self.poisson_lambda)
            ip: int = state.inventory_position()
            next_state: InventoryState = InventoryState(
                max(ip - demand_sample, 0),
                order
            )
            reward: float = - self.holding_cost * state.on_hand\
                - self.stockout_cost * max(demand_sample - ip, 0)
            return next_state, reward

        return SampledDistribution(sample_next_state_reward)

    def actions(self, state: InventoryState) -> Iterator[int]:
        return itertools.count(start=0, step=1)
```

We leave it to you as an exercise to run various simulations of the MRP implied by the deterministic and stochastic policy instances we had created earlier (the above code is in the file rl/chapter3/simple_inventory_mdp_nocap.py). See the method `fraction_of_days_oos` in this file as an example of a simulation to calculate the percentage of days when we'd be unable to satisfy some customer demand for toothpaste due to too little inventory at store-opening (naturally, the higher the re-order point in the policy, the lesser the percentage of days when we'd be Out-of-Stock). This kind of simulation exercise will help build intuition on the tradeoffs we have to make between having too little inventory versus having too much inventory (holding costs versus stockout costs) - essentially leading to our ultimate goal of determining the Optimal Policy (more on this later).

Finite Markov Decision Processes

Certain calculations for Markov Decision Processes can be performed easily if:

- The state space is finite ($\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$),
- The action space $\mathcal{A}(s)$ is finite for each $s \in \mathcal{N}$.
- The set of unique pairs of next state and reward transitions from each pair of current non-terminal state and action is finite

If we satisfy the above three characteristics, we refer to the Markov Decision Process as a Finite Markov Decision Process. Let us write some code for a Finite Markov Decision Process. We create a concrete class `FiniteMarkovDecisionProcess` that implements the interface of the abstract class `MarkovDecisionProcess` (specifically implements the `@abstractmethod apply_policy`). Our first task is to think about the data structure required to specify an instance of `FiniteMarkovDecisionProcess` (i.e., the data structure we'd pass to the `__init__` method of `FiniteMarkovDecisionProcess`). Analogous to how we curried $\mathcal{P}_R$ for a Markov Reward Process as $\mathcal{N} \rightarrow (\mathcal{S} \times \mathbb{R} \rightarrow [0,1])$ (where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ and $\mathcal{N}$ has $m \leq n$ states), here we curry $\mathcal{P}_R$ for the MDP as:

$$\mathcal{N} \rightarrow (\mathcal{A} \rightarrow (\mathcal{S} \times \mathbb{R} \rightarrow [0,1]))$$

Since $\mathcal{S}$ is finite, $\mathcal{A}$ is finite, and the set of next state and reward transitions for each pair of current state and action is also finite, we can represent $\mathcal{P}_R$ as a data structure of type `StateActionMapping[S, A]` as shown below:

```
StateReward = FiniteDistribution[Tuple[S, float]]
ActionMapping = Mapping[A, StateReward[S]]
StateActionMapping = Mapping[S, Optional[ActionMapping[A, S]]]
```

The constructor (`__init__` method) of `FiniteMarkovDecisionProcess` takes as input `mapping: StateActionMapping[S, A]` that represents the complete structure of the Finite MDP - it maps each non-terminal state to an action map (maps each terminal state to `None`), and it maps each action in each action map to a finite probability distribution of pairs of next state and reward (essentially the structure of the $\mathcal{P}_R$ function). Along with the attribute mapping, we also have an attribute non_terminal_states: Sequence[S] that is an ordered sequence of non-terminal states. Now let's consider the implementation of the abstract method step of MarkovDecisionProcess. It takes as input a state: S and an action: A. self.mapping[state][action] gives us an object of type FiniteDistribution[Tuple[S, float]] which represents a finite probability distribution of pairs of next state and reward, which is exactly what we want to return. On the other hand, if self.mapping[state] is None (meaning it's a terminal state), then we simply return None. This satisfies the responsibility of FiniteMarkovDecisionProcess in terms of implementing the @abstractmethod step of the abstract class MarkovDecisionProcess. The other @abstractmethod to implement is the actions method which produces an Iterable on the allowed actions $\mathcal{A}(s)$ for a given $s \in \mathcal{S}$ by invoking self.mapping[state].keys() (it returns an empty iterable for non-terminal states). The action_mapping and states methods are quite straightforward. Finally, __repr__ method pretty-prints self.mapping.

```
from rl.distribution import SampledDistribution

class FiniteMarkovDecisionProcess(MarkovDecisionProcess[S, A]):

    mapping: StateActionMapping[S, A]
    non_terminal_states: Sequence[S]

    def __init__(self, mapping: StateActionMapping[S, A]):
        self.mapping = mapping
        self.non_terminal_states = [s for s, v in mapping.items()
                                    if v is not None]

    def __repr__(self) -> str:
        display = ""
        for s, d in self.mapping.items():
            if d is None:
                display += f"{s} is a Terminal State\n"
            else:
                display += f"From State {s}:\n"
                for a, d1 in d.items():
                    display += f"  With Action {a}:\n"
                    for (s1, r), p in d1.table():
                        display += f"    To [State {s1} and "\
                            + f"Reward {r:.3f}] with Probability {p:.3f}\n"
        return display

    def step(self, state: S, action: A) -> Optional[StateReward]:
        action_map: Optional[ActionMapping[A, S]] = self.mapping[state]
        if action_map is None:
```

```python
            return None
        return action_map[action]

    def actions(self, state: S) -> Iterable[A]:
        actions = self.mapping[state]
        return iter([]) if actions is None else actions.keys()

    def action_mapping(self, state: S) -> Optional[ActionMapping[A, S]]:
        return self.mapping[state]

    def states(self) -> Iterable[S]:
        return self.mapping.keys()
```

Now that we've implemented a finite MDP, let's implement a finite policy that maps each non-terminal state to a probability distribution over a finite set of actions (and maps each terminal state to `None`). So we create a concrete class `FinitePolicy` that implements the interface of the abstract class `Policy` (specifically implements the `@abstractmethod act`). The input to the constructor (`__init__` method) is `policy_map: Mapping[S, Optional[FiniteDistribution[A]]]` since this type captures the structure of the $\pi : \mathcal{N} \times \mathcal{A} \rightarrow [0, 1]$ function in the curried form:

$$\mathcal{N} \rightarrow (\mathcal{A} \rightarrow [0, 1])$$

for the case of finite $\mathcal{S}$ and finite $\mathcal{A}$. The `act` method and `states` method are straightforward. We also implement a `__repr__` method for pretty-printing of `self.policy_map`.

```python
class FinitePolicy(Policy[S, A]):
    policy_map: Mapping[S, Optional[FiniteDistribution[A]]]

    def __init__(
        self,
        policy_map: Mapping[S, Optional[FiniteDistribution[A]]]
    ):
        self.policy_map = policy_map

    def __repr__(self) -> str:
        display = ""
        for s, d in self.policy_map.items():
            if d is None:
                display += f"{s} is a Terminal State\n"
            else:
                display += f"For State {s}:\n"
                for a, p in d:
                    display += f"  Do Action {a} with Probability {p:.3f}\n"
        return display
```

```python
    def act(self, state: S) -> Optional[FiniteDistribution[A]]:
        return self.policy_map[state]

    def states(self) -> Iterable[S]:
        return self.policy_map.keys()
```

Armed with a `FinitePolicy` class, we can now write a method `apply_finite_policy` in `FiniteMarkovDecisionProcess` that takes as input a `policy: FinitePolicy[S, A]` and returns a `FiniteMarkovRewardProcess[S]` by processing the finite structures of both of the MDP and the Policy, and producing a finite structure of the implied MRP.

```python
from collections import defaultdict
from rl.distribution import Categorical

    def apply_finite_policy(self, policy: FinitePolicy[S, A])\
            -> FiniteMarkovRewardProcess[S]:

        transition_mapping: Dict[S, Optional[StateReward[S]]] = {}

        for state in self.mapping:
            action_map: Optional[ActionMapping[A, S]] = self.mapping[state]
            if action_map is None:
                transition_mapping[state] = None
            else:
                outcomes: DefaultDict[Tuple[S, float], float]\
                    = defaultdict(float)
                actions = policy.act(state)
                if actions is not None:
                    for action, p_action in actions:
                        for outcome, p_state_reward in action_map[action]:
                            outcomes[outcome] += p_action * p_state_reward
                transition_mapping[state] = Categorical(outcomes)

        return FiniteMarkovRewardProcess(transition_mapping)
```

The above code for `FiniteMarkovRewardProcess` and `FinitePolicy` is in the file rl/markov_decision_process.py.

Simple Inventory Example as a Finite Markov Decision Process

Now we'd like to model the simple inventory example as a Finite Markov Decision Process so we can take advantage of the algorithms specifically for Finite Markov Decision Processes. To enable finite states and finite actions, we will re-introduce the constraint of space capacity C and we will apply the restriction that the order quantity (action) cannot exceed $C - (\alpha + \beta)$ where α is the On-Hand component of the State and β is the On-Order component of the State. Thus, the action space for any given state $(\alpha, \beta) \in \mathcal{S}$ is finite. Next, note that this ordering policy ensures that in steady-state, the sum of On-Hand and On-Order will not exceed the capacity C. So we will constrain the set of states to be the steady-state set of finite states

$$\mathcal{S} = \{(\alpha, \beta) : 0 \leq \alpha + \beta \leq C\}$$

Although the set of states is finite, there are an infinite number of pairs of next state and reward outcomes possible from any given pair of current state and action. This is because there are an infinite set of possibilities of customer demand on any given day (resulting in infinite possibilities of stockout cost, i.e., negative reward, on any day). To qualify as a Finite Markov Decision Process, we'll need to model in a manner such that we have a finite set of pairs of next state and reward outcomes from any given pair of current state and action. So what we'll do is that instead of considering (S_{t+1}, R_{t+1}) as the pair of next state and reward, we will model the pair of next state and reward to instead be $(S_{t+1}, \mathbb{E}[R_{t+1}|(S_t, S_{t+1}, A_t)])$ (we know $\mathcal{P}_R$ due to the Poisson probabilities of customer demand, so we can actually calculate this conditional expectation of reward). So given a state s and action a, the pairs of next state and reward would be: $(s', \mathcal{R}_T(s, a, s'))$ for all the s' we transition to from (s, a). Since the set of possible next states s' are finite, these newly-modeled rewards associated with the transitions $(\mathcal{R}_T(s, a, s'))$ are also finite and hence, the set of pairs of next state and reward from any pair of current state and action are also finite. Note that this creative alteration of the reward definition is purely to reduce this Markov Decision Process into a Finite Markov Decision Process. Let's now work out the calculation of the reward transition function $\mathcal{R}_T$.

When the next state's (S_{t+1}) On-Hand is greater than zero, it means all of the day's demand was satisfied with inventory that was available at store-opening $(= \alpha + \beta)$, and hence, each of these next states S_{t+1} correspond to no stockout cost and only an overnight holding cost of $h\alpha$. Therefore, for all α, β (with $0 \leq \alpha + \beta \leq C$) and for all order quantity (action) θ (with $0 \leq \theta \leq C - (\alpha + \beta)$):

$$\mathcal{R}_T((\alpha, \beta), \theta, (\alpha + \beta - i, \theta)) = -h\alpha \text{ for } 0 \leq i \leq \alpha + \beta - 1$$

When next state's (S_{t+1}) On-Hand is equal to zero, there are two possibilities:

1. The demand for the day was exactly $\alpha + \beta$, meaning all demand was satisifed with available store inventory (so no stockout cost and only overnight holding cost), or
2. The demand for the day was strictly greater than $\alpha + \beta$, meaning there's some stockout cost in addition to overnight holding cost. The exact stockout cost is an expectation calculation involving the number of units of missed demand under the corresponding poisson probabilities of demand exceeding $\alpha + \beta$.

This calculation is shown below:

$$\mathcal{R}_T((\alpha,\beta),\theta,(0,\theta)) = -h\alpha - p(\sum_{j=\alpha+\beta+1}^{\infty} f(j)\cdot(j-(\alpha+\beta)))$$

$$= -h\alpha - p(\lambda(1-F(\alpha+\beta-1)) - (\alpha+\beta)(1-F(\alpha+\beta)))$$

So now we have a specification of $\mathcal{R}_T$, but when it comes to our coding interface, we are expected to specify $\mathcal{P}_R$ as that is the interface through which we create a `FiniteMarkovDecisionProcess`. Fear not - a specification of $\mathcal{P}_R$ is easy once we have a specification of $\mathcal{R}_T$. We simply create 5-tuples (s,a,r,s',p) for all $s \in \mathcal{N},, s' \in \mathcal{S}, a \in \mathcal{A}$ such that $r = \mathcal{R}_T(s,a,s')$ and $p = \mathcal{P}(s,a,s')$ (we know $\mathcal{P}$ along with $\mathcal{R}_T$), and the set of all these 5-tuples (for all $s \in \mathcal{N}, s' \in \mathcal{S}, a \in \mathcal{A}$) constitute the specification of $\mathcal{P}_R$, i.e., $\mathcal{P}_R(s,a,r,s') = p$. This turns our reward-definition-altered mathematical model of a Finite Markov Decision Process into a programming model of the `FiniteMarkovDecisionProcess` class. This reward-definition-altered model enables us to gain from the fact that we can leverage the algorithms we'll be writing for Finite Markov Decision Processes (specifically, the classical Dynamic Programming algorithms - covered in Chapter 3). The downside of this reward-definition-altered model is that it prevents us from generating sampling traces of the specific rewards encountered when transitioning from one state to another (because we no longer capture the probabilities of individual reward outcomes). Note that we can indeed perform simulations, but each transition step in the sampling trace will only show us the "mean reward" (specifically, the expected reward conditioned on current state, action and next state).

In fact, most Markov Processes you'd encounter in practice can be modeled as a combination of $\mathcal{R}_T$ and $\mathcal{P}$, and you'd simply follow the above $\mathcal{R}_T$ to $\mathcal{P}_R$ representation transformation drill to present this information in the form of $\mathcal{P}_R$ to instantiate a `FiniteMarkovDecisionProcess`. We designed the interface to accept $\mathcal{P}_R$ as input since that is the most general interface for specifying Markov Decision Processes.

So now let's write some code for the simple inventory example as a Finite Markov Decision Process as described above. All we have to do is to create a derived class inherited from `FiniteMarkovDecisionProcess` and write a method to construct the `mapping: StateActionMapping` (i.e., $\mathcal{P}_R$) that the `__init__` constuctor of `FiniteMarkovRewardProcess` requires as input. Note that the generic state `S` is replaced here with the `@dataclass InventoryState` to represent the inventory state, comprising of the On-Hand and On-Order inventory quantities, and the generic action `A` is replaced here with `int` to represent the order quantity.

```
from scipy.stats import poisson
from rl.distribution import Categorical

InvOrderMapping = StateActionMapping[InventoryState, int]
```

```python
class SimpleInventoryMDPCap(FiniteMarkovDecisionProcess[InventoryState, int]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float,
        holding_cost: float,
        stockout_cost: float
    ):
        self.capacity: int = capacity
        self.poisson_lambda: float = poisson_lambda
        self.holding_cost: float = holding_cost
        self.stockout_cost: float = stockout_cost

        self.poisson_distr = poisson(poisson_lambda)
        super().__init__(self.get_action_transition_reward_map())

    def get_action_transition_reward_map(self) -> InvOrderMapping:
        d: Dict[InventoryState, Dict[int, Categorical[Tuple[InventoryState,
                                                            float]]]] = {}

        for alpha in range(self.capacity + 1):
            for beta in range(self.capacity + 1 - alpha):
                state: InventoryState = InventoryState(alpha, beta)
                ip: int = state.inventory_position()
                base_reward: float = - self.holding_cost * alpha
                d1: Dict[int, Categorical[Tuple[InventoryState, float]]] = {}

                for order in range(self.capacity - ip + 1):
                    sr_probs_dict: Dict[Tuple[InventoryState, float], float] =\
                        {(InventoryState(ip - i, order), base_reward):
                         self.poisson_distr.pmf(i) for i in range(ip)}

                    probability: float = 1 - self.poisson_distr.cdf(ip - 1)
                    reward: float = base_reward - self.stockout_cost *\
                        (probability * (self.poisson_lambda - ip) +
                         ip * self.poisson_distr.pmf(ip))
                    sr_probs_dict[(InventoryState(0, order), reward)] = \
                        probability
                    d1[order] = Categorical(sr_probs_dict)

                d[state] = d1
        return d
```

Now let's test this out with some example inputs (as shown below). We construct an instance of the SimpleInventoryMDPCap class with these inputs (named `si_mdp` below), then construct an instance of the `FinitePolicy[InventoryState, int]` class (a deterministic policy, named `fdp` below), and combine them to produce the implied MRP (an instance of the `FiniteMarkovRewardProcess[InventoryState]` class).

```
from rl.distribution import Constant

user_capacity = 2
user_poisson_lambda = 1.0
user_holding_cost = 1.0
user_stockout_cost = 10.0

si_mdp: FiniteMarkovDecisionProcess[InventoryState, int] =\
    SimpleInventoryMDPCap(
        capacity=user_capacity,
        poisson_lambda=user_poisson_lambda,
        holding_cost=user_holding_cost,
        stockout_cost=user_stockout_cost
    )

fdp: FinitePolicy[InventoryState, int] = FinitePolicy(
    {InventoryState(alpha, beta):
     Constant(user_capacity - (alpha + beta)) for alpha in
     range(user_capacity + 1) for beta in range(user_capacity + 1 - alpha)}
)

implied_mrp: FiniteMarkovRewardProcess[InventoryState] =\
    si_mdp.apply_finite_policy(fdp)
```

The above code is in the file rl/chapter3/simple_inventory_mdp_cap.py. We encourage you to play with the inputs in `__main__`, produce the resultant implied MRP, and explore it's characteristics (such as it's Reward Function and it's Value Function).

MDP Value Function for a Fixed Policy

Now we are ready to talk about the Value Function for an MDP evaluated with a fixed policy π (also known as the MDP *Prediction* problem). The term *Prediction* refers to the fact that this problem is about forecasting the expected future return when the agent follows a specific policy. Just like in the case of MRP, we define the Return G_t at time step t for an MDP as:

$$G_t = \sum_{i=t+1}^{\infty} \gamma^{i-t-1} \cdot R_i = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \dots$$

where $\gamma \in [0, 1]$ is a specified discount factor.

We use the above definition of *Return* even for a terminating sequence (say terminating at $t = T$, i.e., $S_T \in \mathcal{T}$), by treating $R_i = 0$ for all $i > T$.

The Value Function for an MDP evaluated with a fixed policy π

$$V^\pi : \mathcal{N} \to \mathbb{R}$$

is defined as:

$$V^\pi(s) = \mathbb{E}_{\pi,\mathcal{P}_R}[G_t|S_t = s] \text{ for all } s \in \mathcal{N}, \text{ for all } t = 0, 1, 2, \ldots$$

For the rest of the book, we will assume that whenever we are talking about a Value Function, the discount factor γ is appropriate to ensure that the Expected Return from each state is finite - in particular, $\gamma < 1$ for continuing (non-terminating) MDPs where the Return could otherwise diverge.

Now let's expand $\mathbb{E}_{\pi,\mathcal{P}_R}[G_t|S_t = s]$.

$$\begin{aligned}
V^\pi(s) &= \mathbb{E}_{\pi,\mathcal{P}_R}[R_{t+1}|S_t = s] + \gamma \cdot \mathbb{E}_{\pi,\mathcal{P}_R}[R_{t+2}|S_t = s] + \gamma^2 \cdot \mathbb{E}_{\pi,\mathcal{P}_R}[R_{t+3}|S_t = s] + \ldots \\
&= \sum_{a \in \mathcal{A}} \pi(s,a) \cdot \mathcal{R}(s,a) + \gamma \cdot \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \sum_{a' \in \mathcal{A}} \pi(s',a') \cdot \mathcal{R}(s',a') \\
&\quad + \gamma^2 \cdot \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a',s') \sum_{a' \in \mathcal{A}} \pi(s',a') \sum_{s'' \in \mathcal{N}} \mathcal{P}(s',a'',s'') \sum_{a'' \in \mathcal{A}} \pi(s'',a'') \cdot \mathcal{R}(s'',a'') \\
&\quad + \ldots \\
&= \mathcal{R}^\pi(s) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}^\pi(s,s') \cdot \mathcal{R}^\pi(s') \\
&\quad + \gamma^2 \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}^\pi(s,s') \sum_{s'' \in \mathcal{N}} \mathcal{P}^\pi(s',s'') \cdot \mathcal{R}^\pi(s'') + \ldots \text{ for all } s \in \mathcal{N}
\end{aligned}$$

But from Equation (1.1) in Chapter 1, we know that the last expression above is equal to the π-implied MRP's Value Function for state s. So, the Value Function V^π of an MDP evaluated with a fixed policy π is exactly the same function as the Value Function of the π-implied MRP. So we can apply the MRP Bellman Equation on V^π, i.e.,

$$\begin{aligned}
V^\pi(s) &= \mathcal{R}^\pi(s) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}^\pi(s,s') \cdot V^\pi(s') \\
&= \sum_{a \in \mathcal{A}} \pi(s,a) \cdot \mathcal{R}(s,a) + \gamma \cdot \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V^\pi(s') \qquad (2.1) \\
&= \sum_{a \in \mathcal{A}} \pi(s,a) \cdot (\mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V^\pi(s')) \text{ for all } s \in \mathcal{N}
\end{aligned}$$

As we saw in Chapter 1, for finite state spaces that are not too large, Equation (2.1) can be solved for V^π (i.e. solution to the MDP *Prediction* problem) with a linear algebra solution (Equation (1.2) from Chapter 1). More generally, Equation (2.1) will be a key equation for the rest of the book in developing various Dynamic Programming and Reinforcement Algorithms for the MDP *Prediction* problem. However, there is another Value Function that's also going to be crucial in developing MDP algorithms - one which maps a (state, action) pair to the expected return originating from the (state, action) pair when evaluated with a fixed policy. This is known as the *Action-Value Function* of an MDP evaluated with a fixed policy π:

$$Q^\pi : \mathcal{N} \times \mathcal{A} \to \mathbb{R}$$

defined as:

$$Q^\pi(s,a) = \mathbb{E}_{\pi,\mathcal{P}_R}[G_t|(S_t = s, A_t = a)] \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}, \text{ for all } t = 0,1,2,\ldots$$

To avoid terminology confusion, we refer to V^π as the *State-Value Function* (albeit often simply abbreviated to *Value Function*) for policy π, to distinguish from the *Action-Value Function* Q^π. The way to interpret $Q^\pi(s,a)$ is that it's the Expected Return from a given non-terminal state s by first taking the action a and subsequently following policy π. With this interpretation of $Q^\pi(s,a)$, we can perceive $V^\pi(s)$ as the "weighted average" of $Q^\pi(s,a)$ (over all possible actions a from a non-terminal state s) with the weights equal to probabilities of action a, given state s (i.e., $\pi(s,a)$). Precisely,

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \cdot Q^\pi(s,a) \text{ for all } s \in \mathcal{N} \tag{2.2}$$

Combining Equation (2.1) and Equation (2.2) yields:

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V^\pi(s') \text{ for all } s \in \mathcal{N}, a \in \mathcal{A} \tag{2.3}$$

Combining Equation (2.3) and Equation (2.2) yields:

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \sum_{a' \in \mathcal{A}} \pi(s',a') \cdot Q^\pi(s',a') \text{ for all } s \in \mathcal{N}, a \in \mathcal{A} \tag{2.4}$$

Equation (2.1) is known as the MDP State-Value Function Bellman Policy Equation (Figure 2.2 serves as a visualization aid for this Equation). Equation (2.4) is known as the MDP Action-Value Function Bellman Policy Equation (Figure 2.3 serves as a visualization aid for this Equation). Note that Equation (2.2) and Equation (2.3) are embedded in Figure 2.2 as well as in Figure 2.3. Equations (2.1), (2.2), (2.3) and (2.4) are collectively known as the MDP Bellman Policy Equations.

For the rest of the book, in these MDP transition figures, we shall always depict states as elliptical-shaped nodes and actions as rectangular-shaped nodes. Notice that transition from a state node to an action node is associated with a probability represented by π and transition from an action node to a state node is associated with a probability represented by $\mathcal{P}$.

Note that for finite MDPs of state space not too large, we can solve the MDP Prediction problem (solving for V^π and equivalently, Q^π) in a straightforward manner: Given a policy π, we can create the finite MRP implied by π, using the

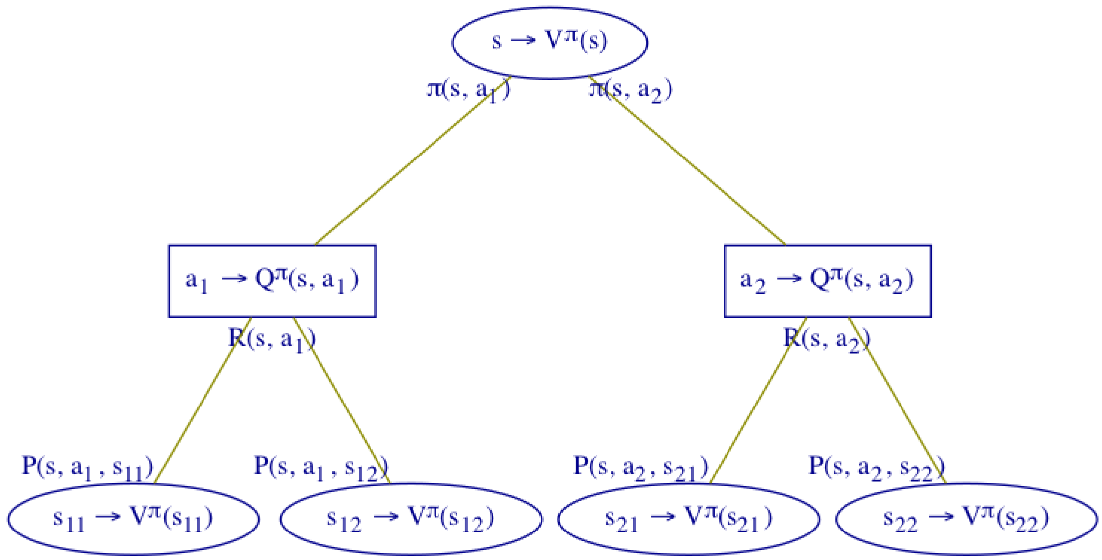

Figure 2.2.: Visualization of MDP State-Value Function Bellman Policy Equation

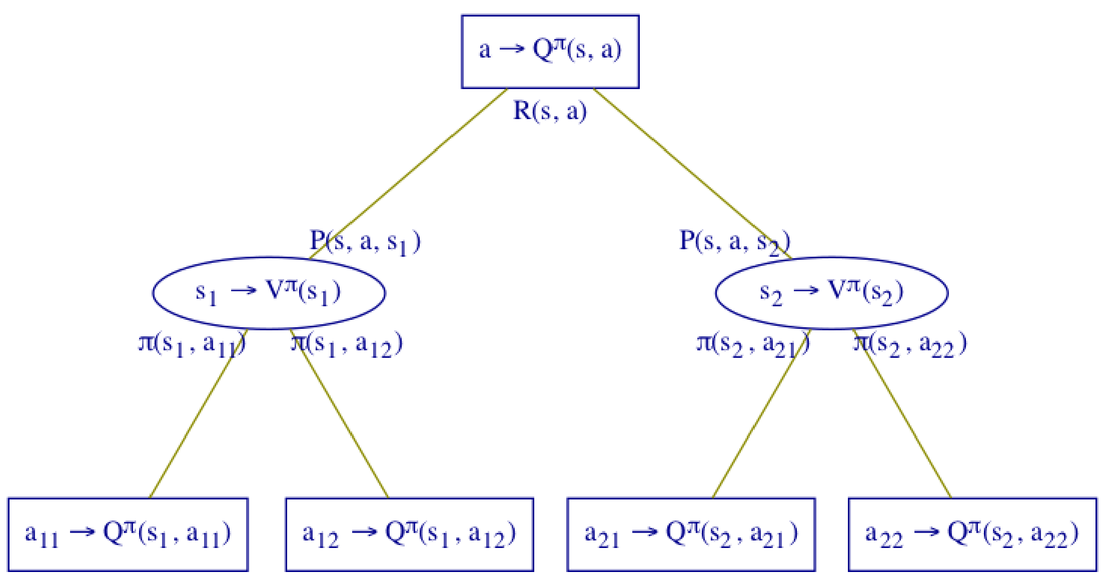

Figure 2.3.: Visualization of MDP Action-Value Function Bellman Policy Equation

method `apply_policy` in `FiniteMarkovDecisionProcess`, then use the matrix-inversion method you learnt in Chapter 1 to calculate the Value Function of the π-implied MRP. We know that the π-implied MRP's Value Function is the same as the State-Value Function V^π of the MDP which can then be used to arrive at the Action-Value Function Q^π of the MDP (using Equation (2.3)). For large state spaces, we will need to use iterative/numerical methods (Dynamic Programming and Reinforcement Learning algorithms) to solve this Prediction problem (covered later in this book).

Optimal Value Function and Optimal Policies

Finally, we arrive at the main purpose of a Markov Decision Process - to identify a policy (or policies) that would yield the Optimal Value Function (i.e., the best possible *Expected Return* from each of the non-terminal states). We say that a Markov Decision Process is "solved" when we identify its Optimal Value Function (together with its associated Optimal Policy, i.e., a Policy that yields the Optimal Value Function). The problem of identifying the Optimal Value Function and its associated Optimal Policy/Policies is known as the MDP *Control* problem. The term *Control* refers to the fact that this problem involves steering the actions (by iterative modifications of the policy) to drive the Value Function towards Optimality. Formally, the Optimal Value Function

$$V^* : \mathcal{N} \to \mathbb{R}$$

is defined as:

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s) \text{ for all } s \in \mathcal{N}$$

where Π is the set of stationary (stochastic) policies over the spaces of $\mathcal{N}$ and $\mathcal{A}$.

The way to read the above definition is that for each non-terminal state s, we consider all possible stochastic stationary policies π, and maximize $V^\pi(s)$ across all these choices of π. Note that the maximization over choices of π is done separately for each s, so it's conceivable that different choices of π might maximize $V^\pi(s)$ for different $s \in \mathcal{N}$. Thus, from the above definition of V^*, we can't yet talk about the notion of "An Optimal Policy". So, for now, let's just focus on the notion of Optimal Value Function, as defined above. Note also that we haven't yet talked about how to achieve the above-defined maximization through an algorithm - we have simply *defined* the Optimal Value Function.

Likewise, the Optimal Action-Value Function

$$Q^* : \mathcal{N} \times \mathcal{A} \to \mathbb{R}$$

is defined as:

$$Q^*(s, a) = \max_{\pi \in \Pi} Q^\pi(s, a) \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}$$

V^* is often refered to as the Optimal State-Value Function to distinguish it from the Optimal Action-Value Function Q^* (although, for succinctness, V^* is often also refered to as simply the Optimal Value Function). To be clear, if someone says, Optimal Value Function, by default, they'd be refering to the Optimal State-Value Function V^* (not Q^*).

Much like how the Value Function(s) for a fixed policy have a recursive formulation, we can create a recursive formulation for the Optimal Value Function(s). Let us start by unraveling the Optimal State-Value Function $V^*(s)$ for a given non-terminal state s - we consider all possible actions $a \in \mathcal{A}$ we can take from state s, and pick the action a that yields the best Action-Value from thereon, i.e., the action a that yields the best $Q^*(s, a)$. Formally, this gives us the following equation:

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a) \text{ for all } s \in \mathcal{N} \tag{2.5}$$

Likewise, let's think about what it means to be optimal from a given non-terminal-state and action pair (s, a), i.e, let's unravel $Q^*(s, a)$. First, we get the immediate expected reward $\mathcal{R}(s, a)$. Next, we consider all possible random states $s' \in \mathcal{S}$ we can transition to, and from each of those states which are non-terminal states, we recursively act optimally. Formally, this gives us the following equation:

$$Q^*(s, a) = \mathcal{R}(s, a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot V^*(s') \text{ for all } s \in \mathcal{N}, a \in \mathcal{A} \tag{2.6}$$

Substituting for $Q^*(s, a)$ from Equation (2.6) in Equation (2.5) gives:

$$V^*(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s, a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot V^*(s')\} \text{ for all } s \in \mathcal{N} \tag{2.7}$$

Equation (2.7) is known as the MDP State-Value Function Bellman Optimality Equation and is depicted in Figure 2.4 as a visualization aid.

Substituting for $V^*(s)$ from Equation (2.5) in Equation (2.6) gives:

$$Q^*(s, a) = \mathcal{R}(s, a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot \max_{a' \in \mathcal{A}} Q^*(s', a') \text{ for all } s \in \mathcal{N}, a \in \mathcal{A} \tag{2.8}$$

Equation (2.8) is known as the MDP Action-Value Function Bellman Optimality Equation and is depicted in Figure 2.5 as a visualization aid.

Note that Equation (2.5) and Equation (2.6) are embedded in Figure 2.4 as well as in Figure 2.5. Equations (2.7), (2.5), (2.6) and (2.8) are collectively known as the MDP Bellman Optimality Equations. We should highlight that when someone says MDP Bellman Equation or simply Bellman Equation, unless they explicit state otherwise, they'd be refering to the MDP Bellman Optimality Equations (and typically specifically the MDP State-Value Function Bellman Optimality Equation). This is because the MDP Bellman Optimality Equations address the ultimate purpose of Markov Decision Processes - to identify

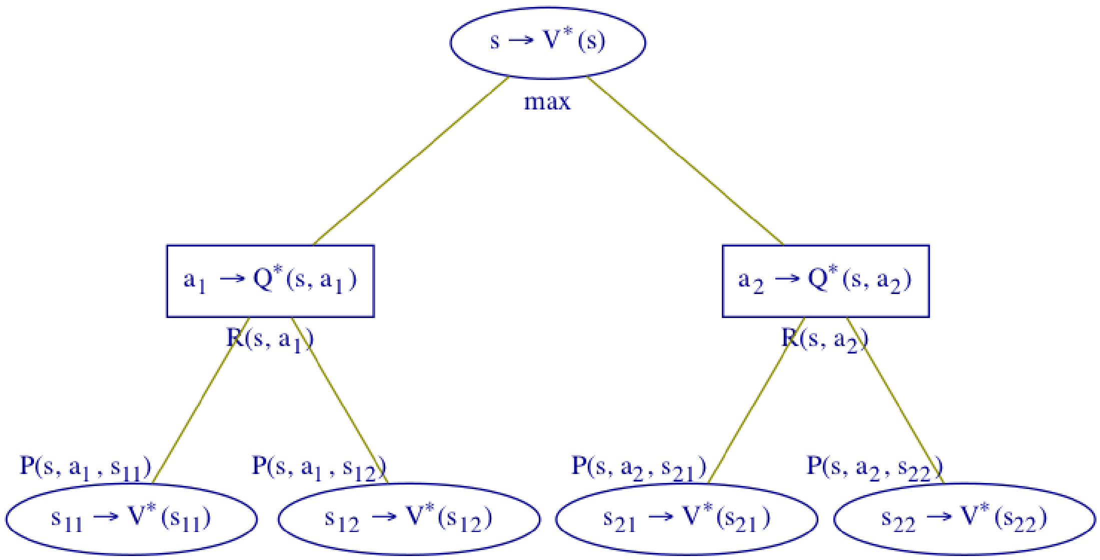

Figure 2.4.: Visualization of MDP State-Value Function Bellman Optimality Equation

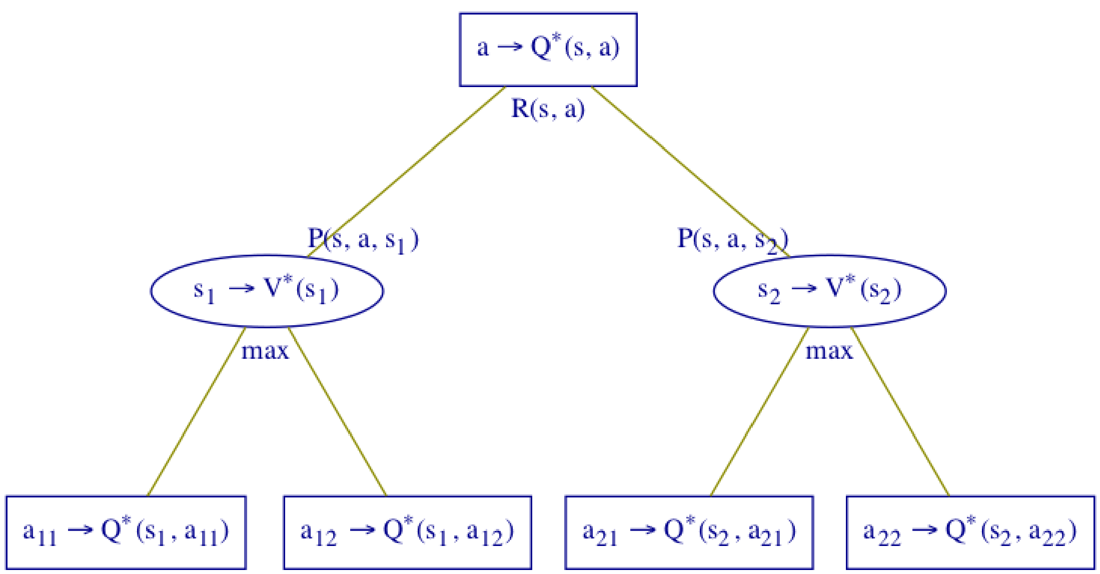

Figure 2.5.: Visualization of MDP Action-Value Function Bellman Optimality Equation

the Optimal Value Function and the associated policy/policies that achieve the Optimal Value Function (i.e., enabling us to solve the MDP *Control* problem).

Again, it pays to emphasize that the Bellman Optimality Equations don't directly give us a recipe to calculate the Optimal Value Function or the policy/policies that achieve the Optimal Value Function - they simply state a powerful mathematical property of the Optimal Value Function that (as we shall see later in this book) will help us come up with algorithms (Dynamic Programming and Reinforcement Learning) to calculate the Optimal Value Function and the associated policy/policies that achieve the Optimal Value Function.

We have been using the phrase "policy/policies that achieve the Optimal Value Function", but we haven't yet provided a clear definition of such a policy (or policies). In fact, as mentioned earlier, it's not clear from the definition of V^* if such a policy (one that would achieve V^*) exists (because it's conceivable that different policies π achieve the maximization of $V^\pi(s)$ for different states $s \in \mathcal{N}$). So instead, we define an *Optimal Policy* $\pi^* : \mathcal{N} \times \mathcal{A} \rightarrow [0, 1]$ as one that "dominates" all other policies with respect to the Value Functions for the policies. Formally,

$$\pi^* \in \Pi \text{ is an Optimal Policy if } V^{\pi^*}(s) \geq V^\pi(s) \textit{ for all } \pi \in \Pi \text{ and } \textit{for all states } s \in \mathcal{N}$$

The definition of an Optimal Policy π^* says that it is a policy that is "better than or equal to" (on the V^π metric) all other stationary policies *for all* non-terminal states (note that there could be multiple Optimal Policies). Putting this definition together with the definition of the Optimal Value Function V^*, the natural question to then ask is whether there exists an Optimal Policy π^* that maximizes $V^\pi(s)$ *for all* $s \in \mathcal{N}$, i.e., whether there exists a π^* such that $V^*(s) = V^{\pi^*}(s)$ for all $s \in \mathcal{N}$. On the face of it, this seems like a strong statement. However, this answers in the affirmative in most MDP settings of interest. The following theorem and proof is for our default setting of MDP (discrete-time, countable-states, stationary), but the statements and argument themes below apply to various other MDP settings as well (the MDP book by Martin Puterman provides rigorous proofs for a variety of settings).

Theorem 2.0.1. *For any (discrete-time, countable-states, stationary) MDP:*

- *There exists an Optimal Policy $\pi^* \in \Pi$, i.e., there exists a Policy $\pi^* \in \Pi$ such that $V^{\pi^*}(s) \geq V^\pi(s)$ for all policies $\pi \in \Pi$ and for all states $s \in \mathcal{N}$*
- *All Optimal Policies achieve the Optimal Value Function, i.e. $V^{\pi^*}(s) = V^*(s)$ for all $s \in \mathcal{N}$, for all Optimal Policies π^**
- *All Optimal Policies achieve the Optimal Action-Value Function, i.e. $Q^{\pi^*}(s, a) = Q^*(s, a)$ for all $s \in \mathcal{N}$, for all $a \in \mathcal{A}$, for all Optimal Policies π^**

Before proceeding with the proof of Theorem (2.0.1), we establish a simple Lemma.

Lemma 2.0.2. *For any two Optimal Policies π_1^* and π_2^*, $V^{\pi_1^*}(s) = V^{\pi_2^*}(s)$ for all $s \in \mathcal{N}$*

Proof. Since π_1^* is an Optimal Policy, from the Optimal Policy definition, we have: $V^{\pi_1^*}(s) \geq V^{\pi_2^*}(s)$ for all $s \in \mathcal{N}$. Likewise, since π_2^* is an Optimal Policy, from the Optimal Policy definition, we have: $V^{\pi_2^*}(s) \geq V^{\pi_1^*}(s)$ for all $s \in \mathcal{N}$. This implies: $V^{\pi_1^*}(s) = V^{\pi_2^*}(s)$ for all $s \in \mathcal{N}$. □

Now we are ready to prove Theorem (2.0.1)

Proof. As a consequence of the above Lemma, all we need to do to prove Theorem (2.0.1) is to establish an Optimal Policy that achieves the Optimal Value Function and the Optimal Action-Value Function. We construct a Deterministic Policy (as a candidate Optimal Policy) $\pi_D^* : \mathcal{N} \rightarrow \mathcal{A}$ as follows:

$$\pi_D^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s, a) \text{ for all } s \in \mathcal{N} \tag{2.9}$$

First we show that π_D^* achieves the Optimal Value Functions V^* and Q^*. Since $\pi_D^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s, a)$ and $V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a)$ for all $s \in \mathcal{N}$, we can infer for all $s \in \mathcal{N}$ that:

$$V^*(s) = Q^*(s, \pi_D^*(s))$$

This says that we achieve the Optimal Value Function from a given non-terminal state s if we first take the action prescribed by the policy π_D^* (i.e., the action $\pi_D^*(s)$), followed by achieving the Optimal Value Function from each of the next time step's states. But note that each of the next time step's states can achieve the Optimal Value Function by doing the same thing described above ("first take action prescribed by π_D^*, followed by ..."), and so on and so forth for further time step's states. Thus, the Optimal Value Function V^* is achieved if from each non-terminal state, we take the action prescribed by π_D^*. Likewise, the Optimal Action-Value Function Q^* is achieved if from each non-terminal state, we take the action a (argument to Q^*) followed by future actions prescribed by π_D^*. Formally, this says:

$$V^{\pi_D^*}(s) = V^*(s) \text{ for all } s \in \mathcal{N}$$

$$Q^{\pi_D^*}(s, a) = Q^*(s, a) \text{ for all } s \in \mathcal{N}, \text{ for all } a \in \mathcal{A}$$

Finally, we argue that π_D^* is an Optimal Policy. Assume the contradiction (that π_D^* is not an Optimal Policy). Then there exists a policy $\pi \in \Pi$ and a state $s \in \mathcal{N}$ such that $V^\pi(s) > V^{\pi_D^*}(s)$. Since $V^{\pi_D^*}(s) = V^*(s)$, we have: $V^\pi(s) > V^*(s)$ which contradicts the Optimal Value Function Definition: $V^*(s) = \max_{\pi \in \Pi} V^\pi(s)$ for all $s \in \mathcal{N}$. Hence, π_D^* must be an Optimal Policy. □

Equation (2.9) is a key construction that goes hand-in-hand with the Bellman Optimality Equations in designing the various Dynamic Programming and Reinforcement Learning algorithms to solve the MDP Control problem (i.e., to solve for V^*, Q^* and π^*). Lastly, it's important to note that unlike the Prediction problem which has a straightforward linear-algebra solution for small state spaces, the Control problem is non-linear and so, doesn't have an analogous straightforward linear-algebra solution. The simplest solutions for the Control problem (even for small state spaces) are the Dynamic Programming algorithms we will cover in Chapter 3.

Variants and Extensions of MDPs

Size of Spaces and Discrete versus Continuous

Variants of MDPs can be organized by variations in the size and type of:

- State Space
- Action Space
- Time Steps

State Space:

The definitions we've provided for MRPs and MDPs were for countable (discrete) state spaces. As a special case, we considered finite state spaces since we have pretty straightforward algorithms for exact solution of Prediction and Control problems for finite MDPs (which we shall learn about in Chapter 3). We emphasize finite MDPs because they will help you develop a sound understanding of the core concepts and make it easy to program the algorithms (known as "tabular" algorithms since we can represent the MDP in a "table", more specifically a Python data structure like `dict` or `numpy array`). However, these algorithms are practical only if the finite state space is not too large. Unfortunately, in many real-world problems, state spaces are either very large-finite or infinite (sometimes continuous-valued spaces). Large state spaces are unavoidable because phenomena in nature and metrics in business evolve in time due to a complex set of factors and often depend on history. To capture all these factors and to enable the Markov Property, we invariably end up with having to model large state spaces which suffer from two "curses":

- Curse of Dimensionality (size of state space $\mathcal{S}$)
- Curse of Modeling (size/complexity of state-reward transition probabilites $\mathcal{P}_R$)

Curse of Dimensionality is a term coined by Richard Bellman in the context of Dynamic Programming. It refers to the fact that when the number of dimensions in the state space grows, there is an exponential increase in the number of samples required to attain an adequate level of accuracy in algorithms. Consider this simple example (adaptation of an example by Bellman himself) - In a single dimension of space from 0 to 1, 100 evenly spaced sample points suffice to sample the space within a threshold distance of 0.01 between points. An equivalent sampling in 10 dimensions ($[0, 1]^{10}$) within a threshold distance of 0.01 between points will require 10^{20} points. So the 10-dimensional space requires points that are greater by a factor of 10^{18} relative to the points required in single dimension. This explosion in requisite points in the state space is known as the Curse of Dimensionality.

Curse of Modeling refers to the fact that when state spaces are large or when the structure of state-reward transition probabilities is complex, explicit modeling of these transition probabilities is very hard and often impossible (the set of probabilities can go beyond memory or even disk storage space). Even if it's possible to fit the probabilities in available storage space, estimating the actual probability values can be very difficult in complex real-world situations.

To overcome these two curses, we can attempt to contain the state space size with some dimension reduction techniques, i.e., including only the most relevant factors in the state representation. Secondly, if future outcomes depend on history, we can include just the past few time steps' values rather than the entire history in the state representation. These savings in state space size are essentially prudent approximations in the state representation. Such state space modeling considerations often require a sound understanding of the real-world problem. Recent advances in unsupervised Machine Learning can also help us contain the state space size. We won't discuss these modeling aspects in detail here - rather, rather we'd just like to emphasize for now that modeling the state space appropriately is one of the most important skills in real-world Reinforcement Learning, and we will illustrate some of these modeling aspects through a few examples later in this book.

Even after performing these modeling exercises in reducing the state space size, we often still end up with fairly large state spaces (so as to capture sufficient nuances of the real-world problem). We battle these two curses in fundamentally two (complementary) ways:

- Approximation of the Value Function - by appropriate sampling of the State Space, we collect data for the Value Function at a sample of points in the state space, and then interpolate/extrapolate/generalize the Value Function in the remainder of the State Space using an approximate representation of the Value Function (eg: by using a supervised learning representation such as a neural network, for the Value Function)
- By sampling the state-reward transition probabilities $\mathcal{P}_R$, we employ Reinforcement Learning algorithms that instead of working with transition probabilities, simply use the state-reward sample transitions to incrementally improve the estimates of the (approximated) Value Function. When state spaces are large, representing explicit transition probabilities is impossible (not enough storage space), and simply sampling from these probability distributions is our only option (and as you shall learn, is surprisingly effective).

This combination of sampling from the state space, approximation of the Value Function (with deep neural networks), sampling state-reward transitions, and clever Reinforcement Learning algorithms goes a long way in breaking both the curse of dimensionality and curse of modeling. In fact, this combination is a common pattern in the broader field of Applied Mathematics to break these curses. The combination of Sampling and Function Approximation (particularly with the modern advances in Deep Learning) are likely to pave the way for future advances in the broader fields of Real-World AI and Applied Mathematics in general. We recognize that some of this discussion is a bit premature since we haven't even started teaching Reinforcement Learning yet. But we hope that this section provides some high-level perspective and connects the learnings from this chapter to the techniques/algorithms that will come later in this book. We will also remind you of this joint-importance of sampling and function approximation once we get started with Reinforcement Learning algorithms later in this book.

Action Space:

Similar to state spaces, the definitions we've provided for MDPs were for countable (discrete) action spaces. As a special case, we considered finite action spaces (together with finite state spaces) since we have pretty straightforward algorithms for exact solution of Prediction and Control problems for finite MDPs. As mentioned above, in these algorithms, we represent the MDP in Python data structures like `dict` or `numpy array`. However, these finite-MDP algorithms are practical only if the state and action spaces are not too large. In many real-world problems, action spaces do end up as fairly large - either finite-large or infinite (sometimes continuous-valued action spaces). The large size of the action space affects algorithms for MDPs in a couple of ways:

- Large action space makes the representation, estimation and evaluation of the policy π, of the Action-Value function for a policy Q^π and of the Optimal Action-Value function Q^* difficult. We have to resort to function approximation and sampling as ways to overcome the large size of the action space.
- The Bellman Optimality Equation leads to a crucial calculation step in Dynamic Programming and Reinforcement Learning algorithms that involves identifying the action for each non-terminal state that maximizes the Action-Value Function Q. When the action space is large, we cannot afford to evaluate Q for each action for an encountered state (as is done in simple tabular algorithms). Rather, we need to tap into an optimization algorithm to perform the maximization of Q over the action space, for an encountered state. Separately, there is a special class of Reinforcement Learning algorithms called Policy Gradient Algorithms (that we shall later learn about) that are particularly valuable for large action spaces (where other types of Reinforcement Learning algorithms are not efficient and often, simply not an option). However, these techniques to deal with large action spaces require care and attention as they have their own drawbacks (more on this later).

Time Steps:

The definitions we've provided for MRP and MDP were for discrete time steps. We distinguished discrete time steps as terminating time-steps (known as terminating or episodic MRPs/MDPs) or non-terminating time-steps (known as continuing MRPs/MDPs). We talked about how the choice of γ matters in these cases ($\gamma = 1$ doesn't work for some continuing MDPs because reward accumulation can blow up to infinity). We won't cover it in this book, but there is an alternative formulation of the Value Function as expected average reward (instead of expected discounted accumulated reward) where we don't need to discount even for continuing MDPs. We had also mentioned earlier that an alternative to discrete time steps is continuous time steps, which is convenient for analytical tractability.

Sometimes, even if state space and action space components have discrete values (eg: price of a security traded in fine discrete units, or number of shares of a security bought/sold on a given day), for modeling purposes, we sometimes find it convenient to represent these components as continuous values (i.e., uncountable state space). The advantage of continuous state/action space representation (especially when paired with continuous time) is that we get considerable mathematical benefits from differential calculus as well as from properties of continuous probability distributions (eg: gaussian distribution conveniences). In fact, continuous state/action space and continuous time are very popular in Mathematical Finance since some of the groundbreaking work from Mathematical Economics from the 1960s and 1970s (Robert Merton's Portfolio Optimization formulation and solution and Black-Scholes' Options Pricing model, to name a couple) are grounded in stochastic calculus which models stock prices/portfolio value as gaussian evolutions in continuous time (more on this later in the book) and treats trades (buy/sell quantities) as also continuous variables (permitting partial derivatives and tractable partial differential equations).

When all three of state space, action space and time steps are modeled as continuous, the Bellman Optimality Equation we covered in this chapter for countable spaces and discrete-time morphs into a differential calculus formulation and is known as the famous *Hamilton-Jacobi-Bellman* (HJB) equation. The HJB Equation is commonly used to model and solve many problems in engineering, physics, economics and finance. We shall cover a couple of financial applications in this book that have elegant formulationsin terms of the HJB equation and equally elegant analytical solutions of the Optimal Value Function and Optimal Policy (tapping into stochastic calculus and differential equations).

Partially-Observable Markov Decision Processes (POMDPs)

You might have noticed in the definition of MDP that there are actually two different notions of state, which we collapsed into a single notion of state. These two notions of state are:

- The internal representation of the environment at each time step t (let's call it $S_t^{(e)}$). This internal representation of the environment is what drives the probabilistic transition to the next time step $t+1$, producing the random pair of next (environment) state $S_{t+1}^{(e)}$ and reward R_{t+1}.
- The agent state at each time step t (let's call it $S_t^{(a)}$). The agent state is what controls the action A_t the agent takes at time step t, i.e., the agent runs a policy π which is a function of the agent state $S_t^{(a)}$, producing a probability distribution of actions A_t.

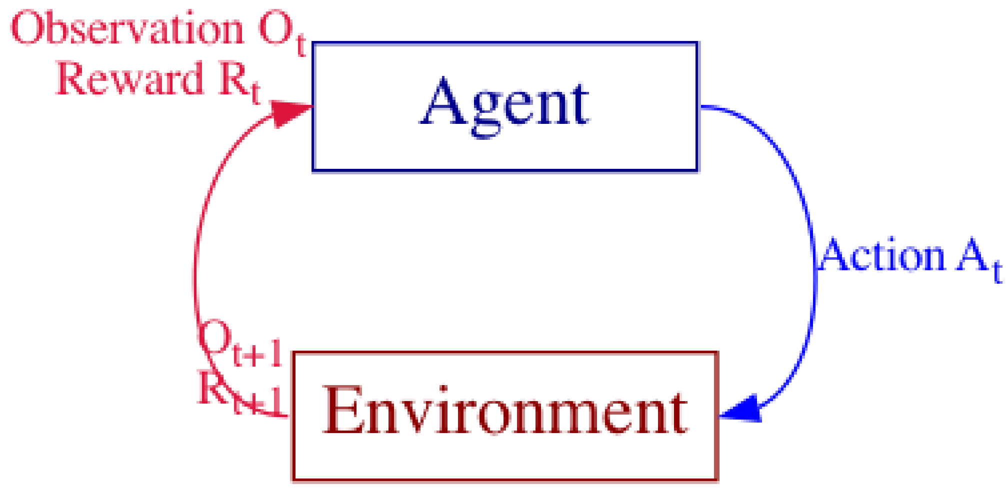

Figure 2.6.: Partially-Observable Markov Decision Process

In our definition of MDP, note that we implicitly assumed that $S_t^{(e)} = S_t^{(a)}$ at each time step t, and called it the (common) state S_t at time t. Secondly, we assumed that this state S_t is *fully observable* by the agent. To understand *full observability*, let us (first intuitively) understand the concept of *partial observability* in a more generic setting than what we had assumed in the framework for MDP. In this more generic framework, we denote O_t as the information available to the agent from the environment at time step t, as depicted in Figure 2.6. The notion of *partial observability* in this more generic framework is that from the history of observations, actions and rewards up to time step t, the agent does not have full knowledge of the environment state $S_t^{(e)}$. This lack of full knowledge of $S_t^{(e)}$ is known as *partial observability*. *Full observability*, on the other hand, means that the agent can fully construct $S_t^{(e)}$ as a function of the history of observations, actions and rewards up to time step t. Since we have the flexibility to model the exact data structures to represent observations, state and actions in this more generic framework, existence of full observability lets us re-structure the observation data at time step t to be $O_t = S_t^{(e)}$. Since we have also assumed $S_t^{(e)} = S_t^{(a)}$, we have:

$$O_t = S_t^{(e)} = S_t^{(a)} \text{ for all time steps } t = 0, 1, 2, \ldots$$

The above statement specialized the framework to that of Markov Decision Processes, which we can now name more precisely as Fully-Observable Markov Decision Processes (when viewed from the lens of the more generic framework described above, that permits partial observability or full observability).

In practice, you will often find that the agent doesn't know the true internal representation ($S_t^{(e)}$) of the environment (i.e, partial observability). Think about what it would take to know what drives a stock price from time step t to $t + 1$ - the agent would need to have access to pretty much every little detail of trading activity in the entire world, and more!). However, since the MDP framework is simple and convenient, and since we have tractable Dynamic Programming and Reinforcement Learning algorithms to solve MDPs, we often do pretend that $O_t = S_t^{(e)} = S_t^{a)}$ and carry on with our business of solving the assumed/modeled MDP. Often, this assumption of $O_t = S_t^{(e)} = S_t^{(a)}$ turns out to be a reasonable approximate model of the real-world but there are indeed situations where this assumption is far-fetched. These are situations where we have access to too little information pertaining to the key aspects of the internal state representation ($S_t^{(e)}$) of the environment. It turns out that we have a formal framework for these situations - this framework is known as Partially-Observable Markov Decision Process (POMDP for short). By default, the acronym MDP will refer to a Fully-Observable Markov Decision Process (i.e. corresponding to the MDP definition we have given earlier in this chapter). So let's now define a POMDP.

A POMDP has the usual features of an MDP (discrete-time, countable states, countable actions, countable next state-reward transition probabilities, discount factor, plus assuming stationarity), together with the notion of random observation O_t at each time step t (each observation O_t lies within the Observation Space $\mathcal{O}$) and observation probability function $\mathcal{Z} : \mathcal{S} \times \mathcal{A} \times \mathcal{O} \rightarrow [0, 1]$ defined as:

$$\mathcal{Z}(s', a, o) = \mathbb{P}[O_{t+1} = o | (S_{t+1} = s', A_t = a)]$$

It pays to emphasize that although a POMDP works with the notion of a state S_t, the agent doesn't have knowledge of S_t. It only has knowledge of observation O_t because O_t is the extent of information made available from the environment. The agent will then need to essentially "guess" (probabilistically) what the state S_t might be at each time step t in order to take the action A_t. The agent's goal in a POMDP is the same as that for an MDP: to determine the Optimal Value Function and to identify an Optimal Policy (achieving the Optimal Value Function).

Just like we have the rich theory and algorithms for MDPs, we have the theory and algorithms for POMDPs. POMDP theory is founded on the notion of *belief states*. The informal notion of a belief state is that since the agent doesn't get to see the state S_t (it only sees the observations O_t) at each time step t, the agent needs to keep track of what it thinks the state S_t might be, i.e. it maintains a probability distribution of states S_t conditioned on history. Let's make this a bit more formal.

Let us refer to the history H_t known to the agent at time t as the sequence of data it has collected up to time t. Formally, this data sequence H_t is:

$$(O_0, R_0, A_0, O_1, R_1, A_1, \ldots, O_t, R_t)$$

A Belief State $b(h)_t$ at time t is a probability distribution over states, conditioned on the history h, i.e.,

$$b(h)_t = (\mathbb{P}[S_t = s_1 | H_t = h], \mathbb{P}[S_t = s_2 | H_t = h], \ldots)$$

such that $\sum_{s \in \mathcal{S}} b(h)_t(s) = 1$ for all histories h and for each $t = 0, 1, 2, \ldots$.

Since the history H_t satisfies the Markov Property, the belief state $b(h)_t$ satisfies the Markov Property. So we can reduce the POMDP to an MDP M with the set of belief states of the POMDP as the set of states of the MDP M. Note that even if the set of states of the POMDP were finite, the set of states of the MDP M will be infinite (i.e. infinite belief states). We can see that this will almost always end up as a giant MDP M. So although this is useful for theoretical reasoning, practically solving this MDP M is often quite hard computationally. However, specialized techniques have been developed to solve POMDPs but as you might expect, their computational complexity is still quite high. So we end up with a choice when encountering a POMDP - either try to solve it with a POMDP algorithm (computationally inefficient but capturing the reality of the real-world problem) or try to approximate it as an MDP (pretending $O_t = S_t^{(e)} = S_t^{(a)}$) which will likely be computationally more efficient but might be a gross approximation of the real-world problem, which in turn means it's effectiveness in practice might be compromised. This is the modeling dilemma we often end up with: what is the right level of detail of real-world factors we need to capture in our model? How do we prevent state spaces from exploding beyond practical computational tractability? The answers to these questions typically have to do with depth of understanding of the nuances of the real-world problem and a trial-and-error process of: formulating the model, solving for the optimal policy, testing the efficacy of this policy in practice (with appropriate measurements to capture real-world metrics), learning about the drawbacks of our model, and iterating back to tweak (or completely change) the model.

Let's consider a classic example of a card game such as Poker or Blackjack as a POMDP where your objective as a player is to identify the optimal policy to maximize your expected return (Optimal Value Function). The observation O_t would be the entire set of information you would have seen up to time step t (or a compressed version of this entire information that suffices for predicting transitions and for taking actions). The state S_t would include, among other things, the set of cards you have, the set of cards your opponents have (which you don't see), and the entire set of exposed as well as unexposed cards not held by players. Thus, the state is only partially observable. With this POMDP structure, we proceed to develop a model of the transition probabilities of next state S_{t+1} and reward R_{t+1}, conditional on current state S_t and current action A_t. We also develop a model of the probabilities of next observation O_{t+1}, conditional on next state S_{t+1} and current action A_t. These probabilities are estimated from data collected from various games (capturing opponent behaviors) and knowledge of the cards-structure of the deck (or decks) used to play the game. Now let's think about what would happen if we modeled this card game as an MDP. We'd no longer have the unseen cards as part of our state. Instead, the state S_t will be limited to the information seen upto time t (i.e., $S_t = O_t$). We can still estimate the transition probabilities, but since it's much harder to estimate in this case, our estimate will likely be quite noisy and nowhere near as reliable as the probability estimates in the POMDP case. The advantage though with modeling it as an MDP is that the algorithm to arrive at the Optimal Value Function/Optimal Policy is a lot more tractable compared to the algorithm for the POMDP model. So it's a tradeoff between the reliability of the probability estimates versus the tractability of the algorithm to solve for the Optimal Value

Function/Policy.

The purpose of this subsection on POMDPs is to highlight that by default a lot of problems in the real-world are POMDPs and it can sometimes take quite a bit of domain-knowledge, modeling creativity and real-world experimentation to treat them as MDPs and make the solution to the modeled MDP successful in practice.

Summary of Key Learnings from this Chapter

- MDP Bellman Policy Equations
- MDP Bellman Optimality Equations
- Theorem (2.0.1) on the existence of an Optimal Policy, and of each Optimal Policy achieving the Optimal Value Function

3. Dynamic Programming Algorithms

As a reminder, much of this book is about algorithms to solve the MDP Control problem, i.e., to compute the Optimal Value Function (and an associated Optimal Policy). We will also cover algorithms for the MDP Prediction problem, i.e., to compute the Value Function when the agent executes a fixed policy π (which, as we know from Chapter 2, is the same as the π-implied MRP problem). Our typical approach will be to first cover algorithms to solve the Prediction problem before covering algorithms to solve the Control problem - not just because Prediction is a key component in solving the Control problem, but also because it helps understand the key aspects of the techniques employed in the Control algorithm in the simpler setting of Prediction.

Planning versus Learning

In this book, we shall look at Planning and Control from the lens of AI (and we'll specifically use the terminology of AI). We shall distinguish between algorithms that don't have a model of the MDP environment (no access to the $\mathcal{P}_R$ function) versus algorithms that do have a model of the MDP environment (meaning $\mathcal{P}_R$ is available to us either in terms of explicit probability distribution representations or available to us just as a sampling model). The former (algorithms without access to a model) are known as *Learning Algorithms* to reflect the fact that the agent will need to interact with the real-world environment (eg: a robot learning to navigate in an actual forest) and learn the Value Function from streams of data (states encountered, actions taken, rewards observed) it receives through environment interactions. The latter (algorithms with access to a model) are known as *Planning Algorithms* to reflect the fact that the agent requires no real-world environment interaction and in fact, projects (with the help of the model) probabilistic scenarios of future states/rewards for various choices of actions, and solves for the requisite Value Function with appropriate probabilistic reasoning of the projected outcomes. In both Learning and Planning, the Bellman Equation will be the fundamental concept driving the algorithms but the details of the algorithms will typically make them appear fairly different. We will only focus on Planning algorithms in this chapter, and in fact, will only focus on a subclass of Planning algorithms known as Dynamic Programming.

Usage of the term *Dynamic Programming*

Unfortunately, the term Dynamic Programming tends to be used by different fields in somewhat different ways. So it pays to clarify the history and the current usage of the term. The term *Dynamic Programming* was coined by Richard Bellman himself. Here is the rather interesting story told by Bellman about how and why he coined the term.

> "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

Bellman had coined the term Dynamic Programming to refer to the general theory of MDPs, together with the techniques to solve MDPs (i.e., to solve the Control problem). So the MDP Bellman Optimality Equation was part of this catch-all term *Dynamic Programming*. The core semantic of the term Dynamic Programming was that the Optimal Value Function can be expressed recursively - meaning, to act optimally from a given state, we will need to act optimally from each of the resulting next states (which is the essence of the Bellman Optimality Equation). In fact, Bellman used the term "Principle of Optimality" to refer to this idea of "Optimal Substructure", and articulated it as follows:

> PRINCIPLE OF OPTIMALITY. An optimal policy has the property that whatever the initial state and initial decisions are, the remaining

decisions must constitute an optimal policy with regard to the state resulting from the first decisions.

So, you can see that the term Dynamic Programming was not just an algorithm in its original usage. Crucially, Bellman laid out an iterative algorithm to solve for the Optimal Value Function (i.e., to solve the MDP Control problem). Over the course of the next decade, the term Dynamic Programming got associated with (multiple) algorithms to solve the MDP Control problem. The term Dynamic Programming was extended to also refer to algorithms to solve the MDP Prediction problem. Over the next couple of decades, Computer Scientists started refering to the term Dynamic Programming as any algorithm that solves a problem through a recursive formulation as long as the algorithm makes repeated invocations to the solutions to each subproblem (overlapping subproblem structure). A classic such example is the algorithm to compute the Fibonacci sequence by caching the Fibonacci values and re-using those values during the course of the algorithm execution. The algorithm to calculate the shortest path in a graph is another classic example where each shortest (i.e. optimal) path includes sub-paths that are optimal. However, in this book, we won't use the term Dynamic Programming in this broader sense. We will use the term Dynamic Programming to be restricted to algorithms to solve the MDP Prediction and Control problems (even though Bellman originally used it only in the context of Control). More specifically, we will use the term Dynamic Programming in the narrow context of Planning algorithms for problems with the following two specializations:

- The state space is finite, the action space is finite, and the set of pairs of next state and reward (given any pair of current state and action) are also finite.
- We have explicit knowledge of the model probabilities (either in the form of $\mathcal{P}_R$ or in the form of $\mathcal{P}$ and $\mathcal{R}$ separately).

This is the setting of the class `FiniteMarkovDecisionProcess` we had covered in Chapter 2. In this setting, Dynamic Programming algorithms solve the Prediction and Control problems *exactly* (meaning the computed Value Function converges to the true Value Function as the algorithm iterations keep increasing). There are variants of Dynamic Programming algorithms known as Asynchronous Dynamic Programming algorithms, Approximate Dynamic Programming algorithms etc. But without such qualifications, when we use just the term Dynamic Programming, we will be refering to the "classical" iterative algorithms (that we will soon describe) for the above-mentioned setting of the `FiniteMarkovDecisionProcess` class to solve MDP Prediction and Control *exactly*. Even though these classical Dynamical Programming algorithms don't scale to large state/action spaces, they are extremely vital to develop one's core understanding of the key concepts in the more advanced algorithms that will enable us to scale (i.e., the Reinforcement Learning algorithms that we shall introduce in later chapters).

Solving the Value Function as a *Fixed-Point*

We cover 3 Dynamic Programming algorithms. Each of the 3 algorithms is founded on the Bellman Equations we had covered in Chapter 2. Each of the 3 algorithms is an iterative algorithm where the computed Value Function converges to the true Value Function as the number of iterations approaches infinity. Each of the 3 algorithms is based on the concept of *Fixed-Point* and updating the computed Value Function towards the Fixed-Point (which in this case, is the true Value Function). Fixed-Point is actually a fairly generic and important concept in the broader fields of Pure as well as Applied Mathematics (also important in Theoretical Computer Science), and we believe understanding Fixed-Point theory has many benefits beyond the needs of the subject of this book. Of more relevance is the fact that the Fixed-Point view of Dynamic Programming is the best way to understand Dynamic Programming. We shall not only cover the theory of Dynamic Programming through the Fixed-Point perspective, but we shall also implement Dynamic Programming algorithms in our code based on the Fixed-Point concept. So this section will be a short primer on general Fixed-Point Theory (and implementation in code) before we get to the 3 Dynamic Programming algorithms.

Definition 3.0.1. The Fixed-Point of a function $f : \mathcal{D} \rightarrow \mathcal{D}$ (for some arbitrary domain $\mathcal{D}$) is a value $x \in \mathcal{D}$ that satisfies the equation: $x = f(x)$.

Note that for some functions, there will be multiple fixed-points and for some other functions, a fixed-point won't exist. We will be considering functions which have a unique fixed-point (this will be the case for the Dynamic Programming algorithms).

Let's warm up to the above-defined abstract concept of Fixed-Point with a concrete example. Consider the function $f(x) = \cos(x)$ defined for $x \in \mathbb{R}$ (x in radians, to be clear). So we want to solve for an x such that $x = \cos(x)$. Knowing the frequency and amplitude of cosine, we can see that the cosine curve intersects the line $y = x$ at only one point, which should be somewhere between 0 and $\frac{\pi}{2}$. But there is no easy way to solve for this point. Here's an idea: Start with any value $x_0 \in \mathbb{R}$, calculate $x_1 = \cos(x_0)$, then calculate $x_2 = \cos(x_1)$, and so on ..., i.e, $x_{i+1} = \cos(x_i)$ for $i = 0, 1, 2, \ldots$. You will find that x_i and x_{i+1} get closer and closer as i increases, i.e., $|x_{i+1} - x_i| \leq |x_i - x_{i-1}|$ for all $i \geq 1$. So it seems like $\lim_{i\to\infty} x_i = \lim_{i\to\infty} \cos(x_{i-1}) = \lim_{i\to\infty} \cos(x_i)$ which would imply that for large enough i, x_i would serve as an approximation to the solution of the equation $x = \cos(x)$. But why does this method of repeated applications of the function f (no matter what x_0 we start with) work? Why does it not diverge or oscillate? How quickly does it converge? If there were multiple fixed-points, which fixed-point would it converge to (if at all)? Can we characterize a class of functions f for which this method (repeatedly applying f, starting with any arbitrary value of x_0) would work (in terms of solving the equation $x = f(x)$)? These are the questions Fixed-Point theory attempts to answer. Can you think of problems you have solved in the past which fall into this method pattern that we've illustrated above for $f(x) = \cos(x)$? It's likely you have, because most of the root-finding and optimization methods (including multi-variate solvers) are essentially based on the idea of Fixed-Point. If this doesn't sound convincing, consider the simple Newton method:

For a differential function $g : \mathbb{R} \to \mathbb{R}$ whose root we want to solve for, the Newton method update rule is:

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

Setting $f(x) = x - \frac{g(x)}{g'(x)}$, the update rule is:

$$x_{i+1} = f(x_i)$$

and it solves the equation $x = f(x)$ (solves for the fixed-point of f), i.e., it solves the equation:

$$x = x - \frac{g(x)}{g'(x)} \Rightarrow g(x) = 0$$

Thus, we see the same method pattern as we saw above for cos(x) (repeated application of a function, starting with any initial value) enables us to solve for the root of g.

More broadly, what we are saying is that if we have a function $f : \mathcal{D} \to \mathcal{D}$ (for some arbitrary domain $\mathcal{D}$), under appropriate conditions (that we will state soon), $f(f(\ldots f(x_0)\ldots))$ converges to a fixed-point of f, i.e., to the solution of the equation $x = f(x)$ (no matter what $x_0 \in \mathcal{D}$ we start with). Now we are ready to state this formally. The statement of the following theorem is quite terse, so we will provide plenty of explanation on how to interpret it and how to use it after stating the theorem (we skip the proof of the theorem).

Theorem 3.0.1 (Banach Fixed-Point Theorem)**.** *Let $\mathcal{D}$ be a non-empty set equipped with a complete metric $d : \mathcal{D} \times \mathcal{D} \to \mathbb{R}$. Let $f : \mathcal{D} \to \mathcal{D}$ be such that there exists a $L \in [0, 1)$ such that $d(f(x_1), f(x_2)) \leq L \cdot d(x_1, x_2)$ for all $x_1, x_2 \in \mathcal{D}$ (this property of f is called a contraction, and we refer to f as a contraction function). Then,*

1. *There exists a unique Fixed-Point $x^* \in \mathcal{D}$, i.e.,*

$$x^* = f(x^*)$$

2. *For any $x_0 \in \mathcal{D}$, and sequence $[x_i | i = 0, 1, 2, \ldots]$ defined as $x_{i+1} = f(x_i)$ for all $i = 0, 1, 2, \ldots$,*

$$\lim_{i \to \infty} x_i = x^*$$

3.

$$d(x^*, x_i) \leq \frac{L^i}{1 - L} \cdot d(x_1, x_0)$$

Equivalently,

$$d(x^*, x_{i+1}) \leq \frac{L}{1 - L} \cdot d(x_{i+1}, x_i)$$

$$d(x^*, x_{i+1}) \leq L \cdot d(x^*, x_i)$$

Sorry - that was pretty terse! Let's try to understand the theorem in a simple, intuitive manner. First we need to explain the jargon *complete metric*. Let's start with the term *metric*. A metric is simply a function $d : \mathcal{D} \times \mathcal{D} \to \mathbb{R}$ that satisfies the usual "distance" properties (for any $x_1, x_2, x_3 \in \mathcal{D}$):

1. $d(x_1, x_2) = 0 \Leftrightarrow x_1 = x_2$ (meaning two different points will have a distance strictly greater than 0)
2. $d(x_1, x_2) = d(x_2, x_1)$ (meaning distance is directionless)
3. $d(x_1, x_3) \leq d(x_1, x_2) + d(x_2, x_3)$ (meaning the triangle inequality is satisfied)

The term *complete* is a bit of a technical detail on sequences not escaping the set $\mathcal{D}$ (that's required in the proof). Since we won't be doing the proof and this technical detail is not so important for the intuition, we shall skip the formal definition of *complete*. A non-empty set $\mathcal{D}$ equipped with the function d (and the technical detail of being *complete*) is known as a complete metric space.

Now we move on to the key concept of *contraction*. A function $f : \mathcal{D} \to \mathcal{D}$ is said to be a contraction function if two points in $\mathcal{D}$ get closer when they are mapped by f (the statement: $d(f(x_1), f(x_2)) \leq L \cdot d(x_1, x_2)$ for all $x_1, x_2 \in \mathcal{D}$, for some $L \in [0, 1)$).

The theorem basically says that for any contraction function f, there is not only a unique fixed-point x^*, one can arrive at x^* by repeated application of f, starting with any initial value $x_0 \in \mathcal{D}$:

$$f(f(\ldots f(x_0)\ldots)) \to x^*$$

We shall use the notation $f^i : \mathcal{D} \to \mathcal{D}$ for $i = 0, 1, 2, \ldots$ as follows:

$$f^{i+1}(x) = f(f^i(x)) \text{ for all } i = 0, 1, 2, \ldots, \text{ for all } x \in \mathcal{D}$$
$$f^0(x) = x \text{ for all } x \in \mathcal{D}$$

With this notation, the computation of the fixed-point can be expressed as:

$$\lim_{i \to \infty} f^i(x_0) = x^* \text{ for all } x_0 \in \mathcal{D}$$

The algorithm, in iterative form, is:

$$x_{i+1} = f(x_i) \text{ for all } i = 0, 2, \ldots$$

We stop the algorithm when x_i and x_{i+1} are close enough based on the distance-metric d.

Banach Fixed-Point Theorem also gives us a statement on the speed of convergence relating the distance between x^* and any x_i to the distance between any two successive x_i.

This is a powerful theorem. All we need to do is identify the appropriate set $\mathcal{D}$ to work with, identify the appropriate metric d to work with, and ensure that f is indeed a contraction function (with respect to d). This enables us to solve for the fixed-point of f with the above-described iterative process of applying f repeatedly, starting with any arbitrary value of $x_0 \in \mathcal{D}$.

We leave it to you as an exercise to verify that $f(x) = \cos(x)$ is a contraction function in the domain $\mathcal{D} = \mathbb{R}$ with metric d defined as $d(x_1, x_2) = |x_1 - x_2|$. Now let's write some code to implement the fixed-point algorithm we described above. Note that we will implement this for any generic type `X` to represent an arbitrary domain $\mathcal{D}$.

```
from typing import Iterator
X = TypeVar('X')

def iterate(step: Callable[[X], X], start: X) -> Iterator[X]:
    state = start

    while True:
        yield state
        state = step(state)
```

The above function take a function (`step: Callable[[X], X]`) and a starting value (`start: X`), and repeatedly applies the function while `yield`ing the values in the form of an `Iterator[X]`, i.e., as a stream of values. This produces an endless stream though. We need a way to specify convergence, i.e., when successive values of the stream are "close enough".

```
def converge(values: Iterator[X], done: Callable[[X, X], bool]) -> Iterator[X]:
    a = next(values)
    yield a

    for b in values:
        if done(a, b):
            break
        else:
            a = b
            yield b
```

The above function takes the generated values from `iterate` (argument `values: Iterator[X]`) and a signal to indicate convergence (argument `done: Callable[[X, X], bool]`), and produces the generated values until `done` is `True`. It is the user's responsibility to write the function `done` and pass it to `converge`. Now let's use these two functions to solve for $x = \cos(x)$.

```
import numpy as np
x = 0.0
values = converge(
    iterate(lambda y: np.cos(y), x),
    lambda a, b: np.abs(a - b) < 1e-3
)
```

```python
for i, v in enumerate(values):
    print(f"{i}: {v:.3f}")
```

This prints a trace with the index of the stream and the value at that index as the function cos is repeatedly applied. It terminates when two successive values are within 3 decimal places of each other.

```
0: 0.000
1: 1.000
2: 0.540
3: 0.858
4: 0.654
5: 0.793
6: 0.701
7: 0.764
8: 0.722
9: 0.750
10: 0.731
11: 0.744
12: 0.736
13: 0.741
14: 0.738
15: 0.740
16: 0.738
17: 0.740
18: 0.739
```

We encourage you to try other starting values (other than the one we have above: $x_0 = 0.0$) and see the trace. We also encourage you to identify other function f which are contractions in an appropriate metric. The above fixed-point code is in the file rl/iterate.py. In this file, you will find two more functions `last` and `converged` to produce the final value of the given iterator when it's values converge according to the `done` function.

Bellman Policy Operator and Policy Evaluation Algorithm

Our first Dynamic Programming algorithm is called *Policy Evaluation*. The Policy Evaluation algorithm solves the problem of calculating the Value Function of a Finite MDP evaluated with a fixed policy π (i.e., the Prediction problem for finite MDPs). We know that this is equivalent to calculating the Value Function of the π-implied Finite MRP. To avoid notation confusion, note that a superscript of π for a symbol means it refers to notation for the π-implied MRP. The precise specification of the Prediction problem is as follows:

Let the states of the MDP (and hence, of the π-implied MRP) be $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$, and without loss of generality, let $\mathcal{N} = \{s_1, s_2, \ldots, s_m\}$ be the non-terminal states. We are given a fixed policy $\pi : \mathcal{N} \times \mathcal{A} \rightarrow [0, 1]$. We are also given the π-implied MRP's transition probability function:

$$\mathcal{P}_R^\pi : \mathcal{N} \times \mathbb{R} \times \mathcal{S} \rightarrow [0, 1]$$

in the form of a data structure (since the states are finite, and the pairs of next state and reward transitions from each non-terminal state are also finite).

We know from Chapters 1 and 2 that by extracting (from $\mathcal{P}_R^\pi$) the transition probability function $\mathcal{P}^\pi : \mathcal{N} \times \mathcal{S} \rightarrow [0, 1]$ of the implicit Markov Process and the reward function $\mathcal{R}^\pi : \mathcal{N} \rightarrow \mathbb{R}$, we can perform the following calculation for the Value Function $V^\pi : \mathcal{N} \rightarrow \mathbb{R}$ (expressed as a column vector $\boldsymbol{V}^\pi \in \mathbb{R}^m$) to solve this Prediction problem:

$$\boldsymbol{V}^\pi = (\boldsymbol{I_m} - \gamma \boldsymbol{\mathcal{P}}^\pi)^{-1} \cdot \boldsymbol{\mathcal{R}}^\pi$$

where $\boldsymbol{I_m}$ is the $m \times m$ identity matrix, column vector $\boldsymbol{\mathcal{R}}^\pi \in \mathbb{R}^m$ represents $\mathcal{R}^\pi$, and $\boldsymbol{\mathcal{P}}^\pi$ is an $m \times m$ matrix representing $\mathcal{P}^\pi$ (rows and columns corresponding to the non-terminal states). However, when m is large, this calculation won't scale. So, we look for a numerical algorithm that would solve (for $\boldsymbol{V}^\pi$) the following MRP Bellman Equation (for a larger number of finite states).

$$\boldsymbol{V}^\pi = \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V}^\pi$$

We define the *Bellman Policy Operator* $\boldsymbol{B}^\pi : \mathbb{R}^m \rightarrow \mathbb{R}^m$ as:

$$\boldsymbol{B}^\pi(\boldsymbol{V}) = \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V} \text{ for any vector } \boldsymbol{V} \text{ in the vector space } \mathbb{R}^m \tag{3.1}$$

So, the MRP Bellman Equation can be expressed as:

$$\boldsymbol{V}^\pi = \boldsymbol{B}^\pi(\boldsymbol{V}^\pi)$$

which means $\boldsymbol{V}^\pi \in \mathbb{R}^m$ is the Fixed-Point of the *Bellman Policy Operator* $\boldsymbol{B}^\pi : \mathbb{R}^m \rightarrow \mathbb{R}^m$. Note that the Bellman Policy Operator can be generalized to the case of non-finite MDPs and V^π is still the Fixed-Point of that generalized definition of the Bellman Policy Operator. However, since this chapter focuses on developing algorithms for finite MDPs, we will work with the above narrower (Equation (3.1)) definition.

Note that $\boldsymbol{B}^\pi$ is a linear transformation on vectors in $\mathbb{R}^m$ and should be thought of as a generalization of a simple 1-D ($\mathbb{R} \rightarrow \mathbb{R}$) linear transformation $y = a + bx$ where the multiplier b is replaced with the matrix $\gamma \boldsymbol{\mathcal{P}}^\pi$ and the shift a is replaced with the column vector $\boldsymbol{\mathcal{R}}^\pi$.

We'd like to come up with a metric for which $\boldsymbol{B}^\pi$ is a contraction function so we can take advantage of Banach Fixed-Point Theorem and solve this Prediction problem by iterative applications of the Bellman Policy Operator $\boldsymbol{B}^\pi$. For any Value Function $\boldsymbol{V} \in \mathbb{R}^m$ (representing $V : \mathcal{N} \rightarrow \mathbb{R}$), we shall express the Value for any state $s \in \mathcal{N}$ as $\boldsymbol{V}(s)$.

Our metric $d : \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R}$ shall be the L^∞ norm defined as:

$$d(\boldsymbol{X}, \boldsymbol{Y}) = \|\boldsymbol{X} - \boldsymbol{Y}\|_\infty = \max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)|$$

$\boldsymbol{B}^\pi$ is a contraction function under L^∞ norm because for all $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$,

$$\max_{s \in \mathcal{N}} |(\boldsymbol{B}^\pi(\boldsymbol{X}) - \boldsymbol{B}^\pi(\boldsymbol{Y}))(s)| = \gamma \cdot \max_{s \in \mathcal{N}} |(\boldsymbol{\mathcal{P}}^\pi \cdot (\boldsymbol{X} - \boldsymbol{Y}))(s)| \leq \gamma \cdot \max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)|$$

So invoking Banach Fixed-Point Theorem proves the following Theorem:

Theorem 3.0.2 (Policy Evaluation Convergence Theorem)**.** *For a Finite MDP with $|\mathcal{N}| = m$, if $\boldsymbol{V}^\pi \in \mathbb{R}^m$ is the Value Function of the MDP when evaluated with a fixed policy $\pi : \mathcal{N} \times \mathcal{A} \to [0, 1]$, then $\boldsymbol{V}^\pi$ is the unique Fixed-Point of the Bellman Policy Operator $\boldsymbol{B}^\pi : \mathbb{R}^m \to \mathbb{R}^m$, and*

$$\lim_{i \to \infty} (\boldsymbol{B}^\pi)^i(\boldsymbol{V_0}) \to \boldsymbol{V}^\pi \textit{ for all starting Value Functions } \boldsymbol{V_0} \in \mathbb{R}^m$$

This gives us the following iterative algorithm (known as the *Policy Evaluation* algorithm for fixed policy $\pi : \mathcal{N} \times \mathcal{A} \to [0, 1]$):

- Start with any Value Function $\boldsymbol{V_0} \in \mathbb{R}^m$
- Iterating over $i = 0, 1, 2, \ldots$, calculate in each iteration:

$$\boldsymbol{V_{i+1}} = \boldsymbol{B}^\pi(\boldsymbol{V_i}) = \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V_i}$$

We stop the algorithm when $d(\boldsymbol{V_i}, \boldsymbol{V_{i+1}}) = \max_{s \in \mathcal{N}} |(\boldsymbol{V_i} - \boldsymbol{V_{i+1}})(s)|$ is adequately small.

It pays to emphasize that Banach Fixed-Point Theorem not only assures convergence to the unique solution $\boldsymbol{V}^\pi$ (no matter what Value Function $\boldsymbol{V_0}$ we start the algorithm with), it also assures a reasonable speed of convergence (dependent on the choice of starting Value Function $\boldsymbol{V_0}$ and the choice of γ). Now let's write the code for Policy Evaluation.

```
DEFAULT_TOLERANCE = 1e-5
V = Mapping[S, float]

def evaluate_mrp(
    mrp: FiniteMarkovRewardProcess[S],
    gamma: float
) -> Iterator[np.ndarray]:
    def update(v: np.ndarray) -> np.ndarray:
        return mrp.reward_function_vec + gamma * \
            mrp.get_transition_matrix().dot(v)

    v_0: np.ndarray = np.zeros(len(mrp.non_terminal_states))
```

```python
    return iterate(update, v_0)

def almost_equal_np_arrays(
    v1: np.ndarray,
    v2: np.ndarray,
    tolerance: float = DEFAULT_TOLERANCE
) -> bool:
    return max(abs(v1 - v2)) < tolerance


def evaluate_mrp_result(
    mrp: FiniteMarkovRewardProcess[S],
    gamma: float
) -> V[S]:
    v_star: np.ndarray = converged(
        evaluate_mrp(mrp, gamma=gamma),
        done=almost_equal_np_arrays
    )
    return {s: v_star[i] for i, s in enumerate(mrp.non_terminal_states)}
```

The code should be fairly self-explanatory. Since the Policy Evaluation problem applies to Finite MRPs, the function `evaluate_mrp` above takes as input `mrp: FiniteMarkovDecisionProcess[S]` and a `gamma: float` to produce an `Iterator` on Value Functions represented as `np.ndarray` (for fast vector/matrix calculations). The function `update` in `evaluate_mrp` represents the application of the Bellman Policy Operator $\boldsymbol{B}^\pi$. The function `evaluate_mrp_result` produces the Value Function for the given `mrp` and the given `gamma`, returning the last value function on the `Iterator` (which terminates based on the `almost_equal_np_arrays` function, considering the maximum of the absolute value differences across all states). Note that the return type of `evaluate_mrp_result` is `V[S]` which is an alias for `Mapping[S, float]`, capturing the semantic of $\mathcal{N} \rightarrow \mathbb{R}$. Note that `evaluate_mrp` is useful for debugging (by looking at the trace of value functions in the execution of the Policy Evaluation algorithm) while `evaluate_mrp_result` produces the desired output Value Function.

If the number of non-terminal states of a given MRP is m, then the running time of each iteration is $O(m^2)$. Note though that to construct an MRP from a given MDP and a given policy, we have to perform $O(m^2 \cdot k)$ operations, where $k = |\mathcal{A}|$.

Greedy Policy

We had said earlier that we will be presenting 3 Dynamic Programming Algorithms. The first (Policy Evaluation), as we saw in the previous section, solves the MDP Prediction problem. The other two (that will present in the next two sections) solve the MDP Control problem. This section is a stepping stone from

Prediction to *Control*. In this section, we define a function that is motivated by the idea of *improving a value function/improving a policy* with a "greedy" technique. Formally, the *Greedy Policy Function*

$$G : \mathbb{R}^m \rightarrow (\mathcal{N} \rightarrow \mathcal{A})$$

interpreted as a function mapping a Value Function $\boldsymbol{V}$ (represented as a vector) to a deterministic policy $\pi'_D : \mathcal{N} \rightarrow \mathcal{A}$, is defined as:

$$G(\boldsymbol{V})(s) = \pi'_D(s) = \arg\max_{a \in \mathcal{A}} \{\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot \boldsymbol{V}(s')\} \text{ for all } s \in \mathcal{N} \tag{3.2}$$

We shall use Equation (3.2) in our mathematical exposition but we require a different (but equivalent) expression for $G(\boldsymbol{V})(s)$ to guide us with our code since the interface for `FiniteMarkovDecisionProcess` operates on $\mathcal{P}_R$, rather than $\mathcal{R}$ and $\mathcal{P}$. The equivalent expression for $G(\boldsymbol{V})(s)$ is as follows:

$$G(\boldsymbol{V})(s) = \arg\max_{a \in \mathcal{A}} \{\sum_{s' \in \mathcal{S}} \sum_{r \in \mathbb{R}} \mathcal{P}_R(s, a, r, s') \cdot (r + \gamma \cdot \boldsymbol{W}(s'))\} \text{ for all } s \in \mathcal{N} \tag{3.3}$$

where $\boldsymbol{W} \in \mathbb{R}^n$ is defined as:

$$\boldsymbol{W}(s') = \begin{cases} \boldsymbol{V}(s') & \text{if } s' \in \mathcal{N} \\ 0 & \text{if } s' \in \mathcal{T} = \mathcal{S} - \mathcal{N} \end{cases}$$

Note that in Equation (3.3), because we have to work with $\mathcal{P}_R$, we need to consider transitions to all states $s' \in \mathcal{S}$ (versus transition to all states $s' \in \mathcal{N}$ in Equation (3.2)), and so, we need to handle the transitions to states $s' \in \mathcal{T}$ carefully (essentially by using the $\boldsymbol{W}$ function as described above).

Now let's write some code to create this "greedy policy" from a given value function, guided by Equation (3.3).

```
import operator

def greedy_policy_from_vf(
    mdp: FiniteMarkovDecisionProcess[S, A],
    vf: V[S],
    gamma: float
) -> FinitePolicy[S, A]:
    greedy_policy_dict: Dict[S, FiniteDistribution[A]] = {}

    for s in mdp.non_terminal_states:

        q_values: Iterator[Tuple[A, float]] = \
            ((a, mdp.mapping[s][a].expectation(
                lambda s_r: s_r[1] + gamma * vf.get(s_r[0], 0.)
            )) for a in mdp.actions(s))
```

```python
        greedy_policy_dict[s] =\
            Constant(max(q_values, key=operator.itemgetter(1))[0])

    return FinitePolicy(greedy_policy_dict)
```

As you can see above, we loop through all the non-terminal states that serve as keys in `greedy_policy_dict: Dict[S, FiniteDistribution[A]]`. Within this loop, we go through all the actions in $\mathcal{A}(s)$ and compute Q-Value $Q(s,a)$ as the sum (over all (s',r) pairs) of $\mathcal{P}_R(s,a,r,s') \cdot (r + \gamma \cdot \boldsymbol{W}(s'))$, written as $\mathbb{E}_{(s',r)\sim\mathcal{P}_R}[r + \gamma \cdot \boldsymbol{W}(s')]$. Finally, we calculate $\arg\max_a Q(s,a)$ for all non-terminal states s, and return it as a `FinitePolicy` (which is our greedy policy).

The word "Greedy" is a reference to the term "Greedy Algorithm", which means an algorithm that takes heuristic steps guided by locally-optimal choices in the hope of moving towards a global optimum. Here, the reference to *Greedy Policy* means if we have a policy π and its corresponding Value Function $\boldsymbol{V}^\pi$ (obtained say using Policy Evaluation algorithm), then applying the Greedy Policy function G on $\boldsymbol{V}^\pi$ gives us a deterministic policy $\pi'_D : \mathcal{N} \rightarrow \mathcal{A}$ that is hopefully "better" than π in the sense that $\boldsymbol{V}^{\pi'_D}$ is "greater" than $\boldsymbol{V}^\pi$. We shall now make this statement precise and show how to use the *Greedy Policy Function* to perform *Policy Improvement*.

Policy Improvement

Terms such a "better" or "improvement" refer to either Value Functions or to Policies (in the latter case, to Value Functions of an MDP evaluated with the policies). So what does it mean to say a Value Function $X : \mathcal{N} \rightarrow \mathbb{R}$ is "better" than a Value Function $Y : \mathcal{N} \rightarrow \mathbb{R}$? Here's the answer:

Definition 3.0.2 (Value Function Comparison)**.** We say $X \geq Y$ for Value Functions $X, Y : \mathcal{N} \rightarrow \mathbb{R}$ of an MDP if and only if:

$$X(s) \geq Y(s) \text{ for all } s \in \mathcal{N}$$

If we are dealing with finite MDPs (with m non-terminal states), we'd represent the Value Functions as vector $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$, and say that $\boldsymbol{X} \geq \boldsymbol{Y}$ if and only if $\boldsymbol{X}(s) \geq \boldsymbol{Y}(s)$ for all $s \in \mathcal{N}$.

So whenever you hear terms like "Better Value Function" or "Improved Value Function", you should interpret it to mean that the Value Function is *no worse for each of the states* (versus the Value Function it's being compared to).

So then, what about the claim of $\pi'_D = G(\boldsymbol{V}^\pi)$ being "better" than π? The following theorem provides the clarification:

Theorem 3.0.3 (Policy Improvement Theorem)**.** *For a finite MDP, for any policy π,*

$$\boldsymbol{V}^{\pi'_D} = \boldsymbol{V}^{G(\boldsymbol{V}^\pi)} \geq \boldsymbol{V}^\pi$$

Proof. We start by noting that applying the Bellman Policy Operator $\boldsymbol{B}^{\pi'_D}$ repeatedly, starting with the Value Function $\boldsymbol{V}^{\pi}$, will converge to the Value Function $\boldsymbol{V}^{\pi'_D}$. Formally,

$$\lim_{i \to \infty} (\boldsymbol{B}^{\pi'_D})^i(\boldsymbol{V}^{\pi}) = \boldsymbol{V}^{\pi'_D}$$

So the proof is complete if we prove that:

$$(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi}) \geq (\boldsymbol{B}^{\pi'_D})^i(\boldsymbol{V}^{\pi}) \text{ for all } i = 0, 1, 2, \ldots$$

which means we get an increasing tower of Value Functions $[(\boldsymbol{B}^{\pi'_D})^i(\boldsymbol{V}^{\pi})|i = 0, 1, 2, \ldots]$ with repeated applications of $\boldsymbol{B}^{\pi'_D}$ starting with the Value Function $\boldsymbol{V}^{\pi}$.

Let us prove this by induction. The base case (for $i = 0$) of the induction is to prove that:

$$\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^{\pi}) \geq \boldsymbol{V}^{\pi}$$

Note that:

$$\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^{\pi})(s) = \mathcal{R}(s, \pi'_D(s)) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, \pi'_D(s), s') \cdot \boldsymbol{V}^{\pi}(s') \text{ for all } s \in \mathcal{N}$$

From Equation (3.2), we know that for each $s \in \mathcal{N}$, $\pi'_D(s) = G(\boldsymbol{V}^{\pi})(s)$ is the action that maximizes $\{\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot \boldsymbol{V}^{\pi}(s')\}$. Therefore,

$$\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^{\pi})(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot \boldsymbol{V}^{\pi}(s')\} = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) \text{ for all } s \in \mathcal{N}$$

Let's compare this equation against the Bellman Policy Equation for π (below):

$$\boldsymbol{V}^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot Q^{\pi}(s, a) \text{ for all } s \in \mathcal{N}$$

We see that $\boldsymbol{V}^{\pi}(s)$ is a weighted average of $Q^{\pi}(s, a)$ (with weights equal to probabilities $\pi(s, a)$ over choices of a) while $\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^{\pi})(s)$ is the maximum (over choices of a) of $Q^{\pi}(s, a)$. Therefore,

$$\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^{\pi}) \geq \boldsymbol{V}^{\pi}$$

This establishes the base case of the proof by induction. Now to complete the proof, all we have to do is to prove:

$$\text{If } (\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi}) \geq (\boldsymbol{B}^{\pi'_D})^i(\boldsymbol{V}^{\pi}), \text{ then } (\boldsymbol{B}^{\pi'_D})^{i+2}(\boldsymbol{V}^{\pi}) \geq (\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi}) \text{ for all } i = 0, 1, 2, \ldots$$

Since $(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi}) = \boldsymbol{B}^{\pi'_D}((\boldsymbol{B}^{\pi'_D})^i(\boldsymbol{V}^{\pi}))$, from the definition of Bellman Policy Operator (Equation (3.1)), we can write the following two equations:

$$(\boldsymbol{B}^{\pi'_D})^{i+2}(\boldsymbol{V}^\pi)(s) = \mathcal{R}(s, \pi'_D(s)) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, \pi'_D(s), s') \cdot (\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^\pi)(s') \text{ for all } s \in \mathcal{N}$$

$$(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^\pi)(s) = \mathcal{R}(s, \pi'_D(s)) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, \pi'_D(s), s') \cdot (\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^\pi)(s') \text{ for all } s \in \mathcal{N}$$

Subtracting each side of the second equation from the first equation yields:

$$(\boldsymbol{B}^{\pi'_D})^{i+2}(\boldsymbol{V}^\pi)(s) - (\boldsymbol{B}^{\pi'_D})^{i+1}(s)$$

$$= \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, \pi'_D(s), s') \cdot ((\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^\pi)(s') - (\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^\pi)(s'))$$

for all $s \in \mathcal{N}$

Since $\gamma \mathcal{P}(s, \pi'_D(s), s')$ consists of all non-negative values and since the induction step assumes $(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^\pi)(s') \geq (\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^\pi)(s')$ for all $s' \in \mathcal{N}$, the right-hand-side of this equation is non-negative, meaning the left-hand-side of this equation is non-negative, i.e.,

$$(\boldsymbol{B}^{\pi'_D})^{i+2}(\boldsymbol{V}^\pi)(s) \geq (\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^\pi)(s) \text{ for all } s \in \mathcal{N}$$

This completes the proof by induction. $\square$

The way to understand the above proof is to think in terms of how each stage of further application of $\boldsymbol{B}^{\pi'_D}$ improves the Value Function. Stage 0 is when you have the Value Function $\boldsymbol{V}^\pi$ where we execute the policy π throughout the MDP. Stage 1 is when you have the Value Function $\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^\pi)$ where from each state s, we execute the policy π'_D for the first time step following s and then execute the policy π for all further time steps. This has the effect of improving the Value Function from Stage 0 ($\boldsymbol{V}^\pi$) to Stage 1 ($\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^\pi)$). Stage 2 is when you have the Value Function $(\boldsymbol{B}^{\pi'_D})^2(\boldsymbol{V}^\pi)$ where from each state s, we execute the policy π'_D for the first two time steps following s and then execute the policy π for all further time steps. This has the effect of improving the Value Function from Stage 1 ($\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^\pi)$) to Stage 2 ($(\boldsymbol{B}^{\pi'_D})^2(\boldsymbol{V}^\pi)$). And so on ... each stage applies policy π'_D instead of policy π for one extra time step, which has the effect of improving the Value Function. Note that "improve" means $\geq$ (really means that the Value Function doesn't get worse for *any* of the states). These stages are simply the iterations of the Policy Evaluation algorithm (using policy π'_D) with starting Value Function $\boldsymbol{V}^\pi$, building an increasing tower of Value Functions $[(\boldsymbol{B}^{\pi'_D})^i(\boldsymbol{V}^\pi) | i = 0, 1, 2, \ldots]$ that get closer and closer until they converge to the Value Function $\boldsymbol{V}^{\pi'_D}$ that is $\geq \boldsymbol{V}^\pi$ (hence, the term *Policy Improvement*).

The Policy Improvement Theorem yields our first Dynamic Programming algorithm to solve the MDP Control problem - known as *Policy Iteration*

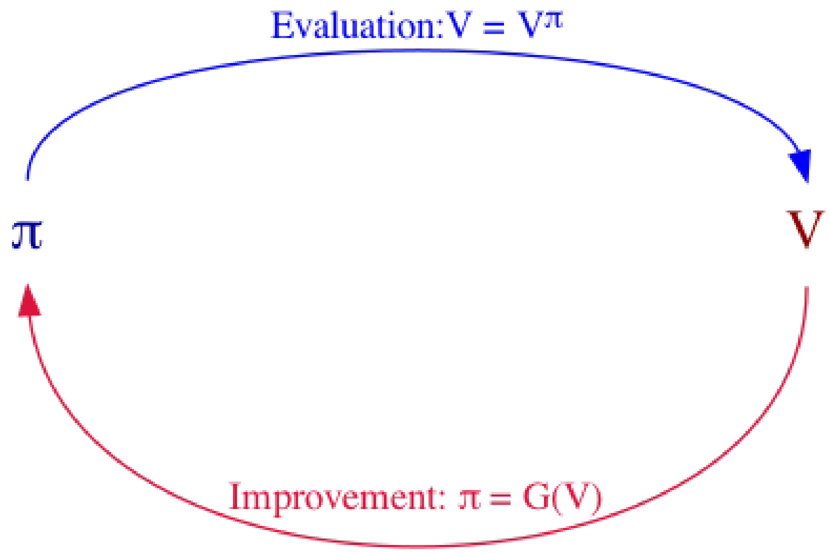

Figure 3.1.: Policy Iteration Loop

Policy Iteration Algorithm

The Policy Improvement algorithm above showed us how to start with the Value Function $\boldsymbol{V}^\pi$ (for a policy π), perform a greedy policy improvement to create a policy $\pi'_D = G(\boldsymbol{V}^\pi)$, and then perform Policy Evaluation (with policy π'_D) with starting Value Function $\boldsymbol{V}^\pi$, resulting in the Value Function $\boldsymbol{V}^{\pi'_D}$ that is an improvement over the Value Function $\boldsymbol{V}^\pi$ we started with. Now note that we can do the same process again to go from π'_D and $\boldsymbol{V}^{\pi'_D}$ to an improved policy π''_D and associated improved Value Function $\boldsymbol{V}^{\pi''_D}$. And we can keep going in this way to create further improved policies and associated Value Functions, until there is no further improvement. This methodology of performing Policy Improvement together with Policy Evaluation using the improved policy, in an iterative manner (depicted in Figure 3.1), is known as the Policy Iteration algorithm (shown below).

- Start with any Value Function $\boldsymbol{V_0} \in \mathbb{R}^m$
- Iterating over $j = 0, 1, 2, \ldots$, calculate in each iteration:

$$\text{Deterministic Policy } \pi_{j+1} = G(\boldsymbol{V_j})$$

$$\text{Value Function } \boldsymbol{V_{j+1}} = \lim_{i \to \infty} (\boldsymbol{B}^{\pi_{j+1}})^i(\boldsymbol{V_j})$$

We end these iterations (over j) when $\boldsymbol{V_{j+1}}$ is essentially the same as $\boldsymbol{V_j}$, i.e., when $\max_{s \in \mathcal{N}} |\boldsymbol{V_{j+1}}(s) - \boldsymbol{V_j}(s)|$ is close to 0. When this happens, the following equation should hold:

$$\boldsymbol{V_j} = (\boldsymbol{B}^{G(\boldsymbol{V_j})})^i(\boldsymbol{V_j}) = \boldsymbol{V_{j+1}} \text{ for all } i = 0, 1, 2, \ldots$$

In particular, this equation should hold for $i = 1$:

Figure 3.2.: Policy Iteration Convergence

$$\boldsymbol{V_j}(s) = \boldsymbol{B}^{G(\boldsymbol{V_j})}(\boldsymbol{V_j})(s) = \mathcal{R}(s, G(\boldsymbol{V_j})(s)) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, G(\boldsymbol{V_j})(s), s') \cdot \boldsymbol{V_j}(s') \text{ for all } s \in \mathcal{N}$$

From Equation (3.2), we know that for each $s \in \mathcal{N}$, $\pi_{j+1}(s) = G(\boldsymbol{V_j})(s)$ is the action that maximizes $\{\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot \boldsymbol{V_j}(s')\}$. Therefore,

$$\boldsymbol{V_j}(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot \boldsymbol{V_j}(s')\} \text{ for all } s \in \mathcal{N}$$

But this in fact is the MDP Bellman Optimality Equation which would mean that $\boldsymbol{V_j} = \boldsymbol{V}^*$, i.e., when V_j is close enough to V_{j+1}, Policy Iteration would have converged to the Optimal Value Function. The associated deterministic policy at the convergence of the Policy Iteration algorithm ($\pi_j : \mathcal{N} \to \mathcal{A}$) is an Optimal Policy because $\boldsymbol{V}^{\pi_j} = \boldsymbol{V_j} = \boldsymbol{V}^*$, meaning that evaluating the MDP with the deterministic policy π_j achieves the Optimal Value Function (depicted in Figure 3.2). This means Policy Iteration algorithm solves the MDP Control problem. This proves the following Theorem:

Theorem 3.0.4 (Policy Iteration Convergence Theorem)**.** *For a Finite MDP with $|\mathcal{N}| = m$, Policy Iteration algorithm converges to the Optimal Value Function $\boldsymbol{V}^* \in \mathbb{R}^m$ along with a Deterministic Optimal Policy $\pi_D^* : \mathcal{N} \to \mathcal{A}$, no matter which Value Function $\boldsymbol{V_0} \in \mathbb{R}^m$ we start the algorithm with.*

Now let's write some code for Policy Iteration Algorithm. Unlike Policy Evaluation which repeatedly operates on Value Functions (and returns a Value Function), Policy Iteration repeatedly operates on a pair of Value Function and Policy (and returns a pair of Value Function and Policy). In the code below, notice the type `Tuple[V[S], FinitePolicy[S, A]]` that represents a pair of Value Function and Policy. The function `policy_iteration` repeatedly applies the function `update` on a pair of Value Function and Policy. The `update` function, after splitting its input `vf_policy` into `vf: V[S]` and `pi: FinitePolicy[S, A]`, creates a MRP (`mrp: FiniteMarkovRewardProcess[S]`) from the combination of the input `mdp` and `pi`. Then it performs a policy evaluation on `mrp` (using the `evaluate_mrp_result` function) to produce a Value Function `policy_vf: V[S]`, and finally creates a greedy (improved) policy named `improved_pi` from `policy_vf` (using the previously-written function `greedy_policy_from_vf`). Thus the function `update` performs a Policy Evaluation followed by a Policy Improvement. Notice also that `policy_iteration` offers the option to perform the matrix-inversion-based computation of Value Function for a given policy (`get_value_function_vec` method of the `mrp` object), in case the state space is not too large. `policy_iteration` returns an `Iterator` on pairs of Value Function and Policy produced by this process of repeated Policy Evaluation and Policy Improvement. `almost_equal_vf_pis` is the function to decide termination based on the distance between two successive Value Functions produced by Policy Iteration. `policy_iteration_result` returns the final (optimal) pair of Value Function and Policy (from the `Iterator` produced by `policy_iteration`), based on the termination criterion of `almost_equal_vf_pis`.

```
DEFAULT_TOLERANCE = 1e-5

def policy_iteration(
    mdp: FiniteMarkovDecisionProcess[S, A],
    gamma: float,
    matrix_method_for_mrp_eval: bool = False
) -> Iterator[Tuple[V[S], FinitePolicy[S, A]]]:

    def update(vf_policy: Tuple[V[S], FinitePolicy[S, A]])\
            -> Tuple[V[S], FinitePolicy[S, A]]:

        vf, pi = vf_policy
        mrp: FiniteMarkovRewardProcess[S] = mdp.apply_finite_policy(pi)
        policy_vf: V[S] = {mrp.non_terminal_states[i]: v for i, v in
                           enumerate(mrp.get_value_function_vec(gamma))}\
            if matrix_method_for_mrp_eval else evaluate_mrp_result(mrp, gamma)
        improved_pi: FinitePolicy[S, A] = greedy_policy_from_vf(
            mdp,
            policy_vf,
            gamma
        )

        return policy_vf, improved_pi

    v_0: V[S] = {s: 0.0 for s in mdp.non_terminal_states}
    pi_0: FinitePolicy[S, A] = FinitePolicy(
        {s: Choose(set(mdp.actions(s))) for s in mdp.non_terminal_states}
    )
    return iterate(update, (v_0, pi_0))

def almost_equal_vf_pis(
    x1: Tuple[V[S], FinitePolicy[S, A]],
    x2: Tuple[V[S], FinitePolicy[S, A]]
) -> bool:
    return max(
        abs(x1[0][s] - x2[0][s]) for s in x1[0]
    ) < DEFAULT_TOLERANCE
```

```python
def policy_iteration_result(
    mdp: FiniteMarkovDecisionProcess[S, A],
    gamma: float,
) -> Tuple[V[S], FinitePolicy[S, A]]:
    return converged(policy_iteration(mdp, gamma), done=almost_equal_vf_pis)
```

If the number of non-terminal states of a given MDP is m and the number of actions ($|\mathcal{A}|$) is k, then the running time of Policy Improvement is $O(m^2 \cdot k)$ and we've already seen before that each iteration of Policy Evaluation is $O(m^2 \cdot k)$.

Bellman Optimality Operator and Value Iteration Algorithm

By making a small tweak to the definition of Greedy Policy Function in Equation (3.2) (changing the arg max to max), we define the *Bellman Optimality Operator*

$$\boldsymbol{B}^* : \mathbb{R}^m \rightarrow \mathbb{R}^m$$

as the following (non-linear) transformation of a vector (representing a Value Function) in the vector space $\mathbb{R}^m$

$$\boldsymbol{B}^*(\boldsymbol{V})(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{V}(s')\} \text{ for all } s \in \mathcal{N} \tag{3.4}$$

We shall use Equation (3.4) in our mathematical exposition but we require a different (but equivalent) expression for $\boldsymbol{B}^*(\boldsymbol{V})(s)$ to guide us with our code since the interface for `FiniteMarkovDecisionProcess` operates on $\mathcal{P}_R$, rather than $\mathcal{R}$ and $\mathcal{P}$. The equivalent expression for $\boldsymbol{B}^*(\boldsymbol{V})(s)$ is as follows:

$$\boldsymbol{B}^*(\boldsymbol{V})(s) = \max_{a \in \mathcal{A}} \{\sum_{s' \in \mathcal{S}} \sum_{r \in \mathbb{R}} \mathcal{P}_R(s,a,r,s') \cdot (r + \gamma \cdot \boldsymbol{W}(s'))\} \text{ for all } s \in \mathcal{N} \tag{3.5}$$

where $\boldsymbol{W} \in \mathbb{R}^n$ is defined (same as in the case of Equation (3.3)) as:

$$\boldsymbol{W}(s') = \begin{cases} \boldsymbol{V}(s') & \text{if } s' \in \mathcal{N} \\ 0 & \text{if } s' \in \mathcal{T} = \mathcal{S} - \mathcal{N} \end{cases}$$

Note that in Equation (3.5), because we have to work with $\mathcal{P}_R$, we need to consider transitions to all states $s' \in \mathcal{S}$ (versus transition to all states $s' \in \mathcal{N}$ in Equation (3.4)), and so, we need to handle the transitions to states $s' \in \mathcal{T}$ carefully (essentially by using the $\boldsymbol{W}$ function as described above).

For each $s \in \mathcal{N}$, the action $a \in \mathcal{A}$ that produces the maximization in (3.4) is the action prescribed by the deterministic policy π_D in (3.2). Therefore, if we apply the Bellman Policy Operator on any Value Function $\boldsymbol{V} \in \mathbb{R}^m$ using the

Greedy Policy $G(\boldsymbol{V})$, it should be identical to applying the Bellman Optimality Operator. Therefore,

$$\boldsymbol{B}^{G(\boldsymbol{V})}(\boldsymbol{V}) = \boldsymbol{B}^*(\boldsymbol{V}) \text{ for all } \boldsymbol{V} \in \mathbb{R}^m \tag{3.6}$$

In particular, it's interesting to observe that by specializing $\boldsymbol{V}$ to be the Value Function $\boldsymbol{V}^\pi$ for a policy π, we get:

$$\boldsymbol{B}^{G(\boldsymbol{V}^\pi)}(\boldsymbol{V}^\pi) = \boldsymbol{B}^*(\boldsymbol{V}^\pi)$$

which is a succinct representation of the first stage of Policy Evaluation with an improved policy $G(\boldsymbol{V}^\pi)$ (note how all three of Bellman Policy Operator, Bellman Optimality Operator and Greedy Policy Function come together in this equation).

Much like how the Bellman Policy Operator $\boldsymbol{B}^\pi$ was motivated by the MDP Bellman Policy Equation (equivalently, the MRP Bellman Equation), Bellman Optimality Operator $\boldsymbol{B}^*$ is motivated by the MDP Bellman Optimality Equation (expressed below):

$$\boldsymbol{V}^*(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{V}^*(s')\} \text{ for all } s \in \mathcal{N}$$

Therefore, we can express the MDP Bellman Optimality Equation succinctly as:

$$\boldsymbol{V}^* = \boldsymbol{B}^*(\boldsymbol{V}^*)$$

which means $\boldsymbol{V}^* \in \mathbb{R}^m$ is the Fixed-Point of the Bellman Optimality Operator $\boldsymbol{B}^* : \mathbb{R}^m \to \mathbb{R}^m$.

Note that the definitions of the Greedy Policy Function and of the Bellman Optimality Operator that we have provided can be generalized to non-finite MDPs, and consequently we can generalize Equation (3.6) and the statement that V^* is the Fixed-Point of the Bellman Optimality Operator would still hold. However, in this chapter, since we are focused on developing algorithms for finite MDPs, we shall stick to the definitions we've provided for the case of finite MDPs.

Much like how we proved that $\boldsymbol{B}^\pi$ is a contraction function, we want to prove that $\boldsymbol{B}^*$ is a contraction function (under L^∞ norm) so we can take advantage of Banach Fixed-Point Theorem and solve the Control problem by iterative applications of the Bellman Optimality Operator $\boldsymbol{B}^*$. So we need to prove that for all $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$,

$$\max_{s \in \mathcal{N}} |(\boldsymbol{B}^*(\boldsymbol{X}) - \boldsymbol{B}^*(\boldsymbol{Y}))(s)| \leq \gamma \cdot \max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)|$$

This proof is a bit harder than the proof we did for $\boldsymbol{B}^\pi$. Here we'd need to utilize two key properties of $\boldsymbol{B}^*$.

1. Monotonicity Property, i.e, for all $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$,

   $$\text{If } \boldsymbol{X}(s) \geq \boldsymbol{Y}(s) \text{ for all } s \in \mathcal{N}, \text{ then } \boldsymbol{B}^*(\boldsymbol{X})(s) \geq \boldsymbol{B}^*(\boldsymbol{Y})(s) \text{ for all } s \in \mathcal{N}$$

Observe that for each state $s \in \mathcal{N}$ and each action $a \in \mathcal{A}$,

$$\{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{X}(s')\} - \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{Y}(s')\}$$

$$= \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot (\boldsymbol{X}(s') - \boldsymbol{Y}(s')) \geq 0$$

Therefore for each state $s \in \mathcal{N}$,

$$\boldsymbol{B}^*(\boldsymbol{X})(s) - \boldsymbol{B}^*(\boldsymbol{Y})(s)$$

$$= \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{X}(s')\} - \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{Y}(s')\} \geq 0$$

2. Constant Shift Property, i.e., for all $\boldsymbol{X} \in \mathbb{R}^m$, $c \in \mathbb{R}$,

$$\boldsymbol{B}^*(\boldsymbol{X} + c)(s) = \boldsymbol{B}^*(\boldsymbol{X})(s) + \gamma c \text{ for all } s \in \mathcal{N}$$

In the above statement, adding a constant ($\in \mathbb{R}$) to a Value Function ($\in \mathbb{R}^m$) adds the constant point-wise to all states of the Value Function (to all dimensions of the vector representing the Value Function). In other words, a constant $\in \mathbb{R}$ might as well be treated as a Value Function with the same (constant) value for all states. Therefore,

$$\boldsymbol{B}^*(\boldsymbol{X} + c)(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot (\boldsymbol{X}(s) + c)\}$$

$$= \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{X}(s)\} + \gamma c = \boldsymbol{B}^*(\boldsymbol{X}) + \gamma c$$

With these two properties of $\boldsymbol{B}^*$ in place, let's prove that $\boldsymbol{B}^*$ is a contraction function. For given $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$, assume:

$$\max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)| = c$$

We can rewrite this as:

$$\boldsymbol{X}(s) - c \leq \boldsymbol{Y}(s) \leq \boldsymbol{X}(s) + c \text{ for all } s \in \mathcal{N}$$

Since $\boldsymbol{B}^*$ has the monotonicity property, we can apply $\boldsymbol{B}^*$ throughout the above double-inequality.

$$\boldsymbol{B}^*(\boldsymbol{X} - c)(s) \leq \boldsymbol{B}^*(\boldsymbol{Y})(s) \leq \boldsymbol{B}^*(\boldsymbol{X} + c)(s) \text{ for all } s \in \mathcal{N}$$

Since $\boldsymbol{B}^*$ has the constant shift property,

$$\boldsymbol{B}^*(\boldsymbol{X})(s) - \gamma c \leq \boldsymbol{B}^*(\boldsymbol{Y})(s) \leq \boldsymbol{B}^*(\boldsymbol{X})(s) + \gamma c \text{ for all } s \in \mathcal{N}$$

In other words,

$$\max_{s \in \mathcal{N}} |(\boldsymbol{B}^*(\boldsymbol{X}) - \boldsymbol{B}^*(\boldsymbol{Y}))(s)| \leq \gamma c = \gamma \cdot \max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)|$$

So invoking Banach Fixed-Point Theorem proves the following Theorem:

Theorem 3.0.5 (Value Iteration Convergence Theorem). *For a Finite MDP with $|\mathcal{N}| = m$, if $\boldsymbol{V}^* \in \mathbb{R}^m$ is the Optimal Value Function, then $\boldsymbol{V}^*$ is the unique Fixed-Point of the Bellman Optimality Operator $\boldsymbol{B}^* : \mathbb{R}^m \rightarrow \mathbb{R}^m$, and*

$$\lim_{i \to \infty} (\boldsymbol{B}^*)^i(\boldsymbol{V_0}) \rightarrow \boldsymbol{V}^* \textit{ for all starting Value Functions } \boldsymbol{V_0} \in \mathbb{R}^m$$

This gives us the following iterative algorithm (known as the *Value Iteration* algorithm):

- Start with any Value Function $\boldsymbol{V_0} \in \mathbb{R}^m$
- Iterating over $i = 0, 1, 2, \ldots$, calculate in each iteration:

$$\boldsymbol{V_{i+1}}(s) = \boldsymbol{B}^*(\boldsymbol{V_i})(s) \text{ for all } s \in \mathcal{N}$$

We stop the algorithm when $d(\boldsymbol{V_i}, \boldsymbol{V_{i+1}}) = \max_{s \in \mathcal{N}} |(\boldsymbol{V_i} - \boldsymbol{V_{i+1}})(s)|$ is adequately small.

It pays to emphasize that Banach Fixed-Point Theorem not only assures convergence to the unique solution $\boldsymbol{V}^*$ (no matter what Value Function $\boldsymbol{V_0}$ we start the algorithm with), it also assures a reasonable speed of convergence (dependent on the choice of starting Value Function $\boldsymbol{V_0}$ and the choice of γ).

Optimal Policy from Optimal Value Function

Note that the Policy Iteration algorithm produces a policy together with a Value Function in each iteration. So, in the end, when we converge to the Optimal Value Function $\boldsymbol{V_j} = \boldsymbol{V}^*$ in iteration j, the Policy Iteration algorithm has a deterministic policy π_j associated with $\boldsymbol{V_j}$ such that:

$$\boldsymbol{V_j} = \boldsymbol{V}^{\pi_j} = \boldsymbol{V}^*$$

and we refer to π_j as the Optimal Policy π^*, one that yields the Optimal Value Function $\boldsymbol{V}^*$, i.e.,

$$\boldsymbol{V}^{\pi^*} = \boldsymbol{V}^*$$

But Value Iteration has no such policy associated with it since the entire algorithm is devoid of a policy representation and operates only with Value Functions. So now the question is: when Value Iteration converges to the Optimal Value Function $\boldsymbol{V_i} = \boldsymbol{V}^*$ in iteration i, how do we get hold of an Optimal Policy π^* such that:

$$\boldsymbol{V}^{\pi^*} = \boldsymbol{V_i} = \boldsymbol{V}^*$$

The answer lies in the Greedy Policy function G. Consider $G(\boldsymbol{V}^*)$. Equation (3.6) told us that:

$$\boldsymbol{B}^{G(\boldsymbol{V})}(\boldsymbol{V}) = \boldsymbol{B}^*(\boldsymbol{V}) \text{ for all } \boldsymbol{V} \in \mathbb{R}^m$$

Specializing $\boldsymbol{V}$ to be $\boldsymbol{V}^*$, we get:

$$\boldsymbol{B}^{G(\boldsymbol{V}^*)}(\boldsymbol{V}^*) = \boldsymbol{B}^*(\boldsymbol{V}^*)$$

But we know that $\boldsymbol{V}^*$ is the Fixed-Point of the Bellman Optimality Operator $\boldsymbol{B}^*$, i.e., $\boldsymbol{B}^*(\boldsymbol{V}^*) = \boldsymbol{V}^*$. Therefore,

$$\boldsymbol{B}^{G(\boldsymbol{V}^*)}(\boldsymbol{V}^*) = \boldsymbol{V}^*$$

The above equation says $\boldsymbol{V}^*$ is the Fixed-Point of the Bellman Policy Operator $\boldsymbol{B}^{G(\boldsymbol{V}^*)}$. However, we know that $\boldsymbol{B}^{G(\boldsymbol{V}^*)}$ has a unique Fixed-Point equal to $\boldsymbol{V}^{G(\boldsymbol{V}^*)}$. Therefore,

$$\boldsymbol{V}^{G(\boldsymbol{V}^*)} = \boldsymbol{V}^*$$

This says that evaluating the MDP with the deterministic greedy policy $G(\boldsymbol{V}^*)$ (policy created from the Optimal Value Function $\boldsymbol{V}^*$ using the Greedy Policy Function G) in fact achieves the Optimal Value Function $\boldsymbol{V}^*$. In other words, $G(\boldsymbol{V}^*)$ is the (Deterministic) Optimal Policy π^* we've been seeking.

Now let's write the code for Value Iteration. The function `value_iteration` returns an `Iterator` on Value Functions (of type `V[S]`) produced by the Value Iteration algorithm. It uses the function `update` for application of the Bellman Optimality Operator. `update` prepares the Q-Values for a state by looping through all the allowable actions for the state, and then calculates the maximum of those Q-Values (over the actions). The Q-Value calculation is same as what we saw in `greedy_policy_from_vf`: $\mathbb{E}_{(s',r)\sim\mathcal{P}_R}[r + \gamma \cdot \boldsymbol{W}(s')]$, using the $\mathcal{P}_R$ probabilities represented in the `mapping` attribute of the `mdp` object (essentially Equation (3.5)). The function `value_iteration_result` returns the final (optimal) Value Function, together with it's associated Optimal Policy. It simply returns the last Value Function of the `Iterable[V[S]]` returned by `value_iteration`, using the termination condition specified in `almost_equal_vfs`.

```
DEFAULT_TOLERANCE = 1e-5

def value_iteration(
    mdp: FiniteMarkovDecisionProcess[S, A],
    gamma: float
) -> Iterator[V[S]]:
    def update(v: V[S]) -> V[S]:
        return {s: max(mdp.mapping[s][a].expectation(
            lambda s_r: s_r[1] + gamma * v.get(s_r[0], 0.)
        ) for a in mdp.actions(s)) for s in v}

    v_0: V[S] = {s: 0.0 for s in mdp.non_terminal_states}
    return iterate(update, v_0)

def almost_equal_vfs(
    v1: V[S],
    v2: V[S],
```

```
    tolerance: float = DEFAULT_TOLERANCE
) -> bool:
    return max(abs(v1[s] - v2[s]) for s in v1) < tolerance

def value_iteration_result(
    mdp: FiniteMarkovDecisionProcess[S, A],
    gamma: float
) -> Tuple[V[S], FinitePolicy[S, A]]:
    opt_vf: V[S] = converged(
        value_iteration(mdp, gamma),
        done=almost_equal_vfs
    )
    opt_policy: FinitePolicy[S, A] = greedy_policy_from_vf(
        mdp,
        opt_vf,
        gamma
    )

    return opt_vf, opt_policy
```

If the number of non-terminal states of a given MDP is m and the number of actions ($|\mathcal{A}|$) is k, then the running time of each iteration of Value Iteration is $O(m^2 \cdot k)$.

We encourage you to play with the above implementations of Policy Evaluation, Policy Iteration and Value Iteration (code in the file rl/dynamic_programming.py) by running it on MDPs/Policies of your choice, and observing the traces of the algorithms.

Revisiting the Simple Inventory Example

Let's revisit the simple inventory example. We shall consider the version with a space capacity since we want an example of a `FiniteMarkovDecisionProcess`. It will help us test our code for Policy Evaluation, Policy Iteration and Value Iteration. More importantly, it will help us identify the mathematical structure of the optimal policy of ordering for this store inventory problem. So let's take another look at the code we wrote in Chapter 2 to set up an instance of a `SimpleInventoryMDPCap` and a `FinitePolicy` (that we can use for Policy Evaluation).

```
user_capacity = 2
user_poisson_lambda = 1.0
user_holding_cost = 1.0
user_stockout_cost = 10.0

si_mdp: FiniteMarkovDecisionProcess[InventoryState, int] =\
    SimpleInventoryMDPCap(
```

```
        capacity=user_capacity,
        poisson_lambda=user_poisson_lambda,
        holding_cost=user_holding_cost,
        stockout_cost=user_stockout_cost
    )

fdp: FinitePolicy[InventoryState, int] = FinitePolicy(
    {InventoryState(alpha, beta):
     Constant(user_capacity - (alpha + beta)) for alpha in
     range(user_capacity + 1) for beta in range(user_capacity + 1 - alpha)}
)
```

Now let's write some code to evaluate `si_mdp` with the policy `fdp`.

```
from pprint import pprint
implied_mrp: FiniteMarkovRewardProcess[InventoryState] =\
    si_mdp.apply_finite_policy(fdp)
user_gamma = 0.9
pprint(evaluate_mrp_result(implied_mrp, gamma=user_gamma))
```

This prints the following Value Function.

```
{InventoryState(on_hand=2, on_order=0): -30.345029758390766,
 InventoryState(on_hand=0, on_order=0): -35.510518165628724,
 InventoryState(on_hand=1, on_order=0): -28.932174210147306,
 InventoryState(on_hand=0, on_order=1): -27.932174210147306,
 InventoryState(on_hand=0, on_order=2): -28.345029758390766,
 InventoryState(on_hand=1, on_order=1): -29.345029758390766}
```

Next, let's run Policy Iteration.

```
opt_vf_pi, opt_policy_pi = policy_iteration_result(
    si_mdp,
    gamma=user_gamma
)
pprint(opt_vf_pi)
print(opt_policy_pi)
```

This prints the following Optimal Value Function and Optimal Policy.

```
{InventoryState(on_hand=2, on_order=0): -29.991900091403522,
 InventoryState(on_hand=0, on_order=0): -34.89485578163003,
 InventoryState(on_hand=1, on_order=0): -28.660960231637496,
 InventoryState(on_hand=0, on_order=1): -27.660960231637496,
 InventoryState(on_hand=0, on_order=2): -27.991900091403522,
 InventoryState(on_hand=1, on_order=1): -28.991900091403522}
```

```
For State InventoryState(on_hand=0, on_order=0):
  Do Action 1 with Probability 1.000
For State InventoryState(on_hand=0, on_order=1):
  Do Action 1 with Probability 1.000
For State InventoryState(on_hand=0, on_order=2):
  Do Action 0 with Probability 1.000
For State InventoryState(on_hand=1, on_order=0):
  Do Action 1 with Probability 1.000
For State InventoryState(on_hand=1, on_order=1):
  Do Action 0 with Probability 1.000
For State InventoryState(on_hand=2, on_order=0):
  Do Action 0 with Probability 1.000
```

As we can see, the Optimal Policy is to not order if the Inventory Position (sum of On-Hand and On-Order) is greater than 1 unit and to order 1 unit if the Inventory Position is 0 or 1. Finally, let's run Value Iteration.

```
opt_vf_vi, opt_policy_vi = value_iteration_result(si_mdp, gamma=user_gamma
)
pprint(opt_vf_vi)
print(opt_policy_vi)
```

You'll see the output from Value Iteration matches the output produced from Policy Iteration - this is a good validation of our code correctness. We encourage you to play around with `user_capacity`, `user_poisson_lambda`, `user_holding_cost`, `user_stockout_cost` and `user_gamma` (code in `__main__` in rl/chapter3/simple_inventory_mdp_ca
As a valuable exercise, using this code, discover the mathematical structure of the Optimal Policy as a function of the above inputs.

Generalized Policy Iteration

In this section, we dig into the structure of the Policy Iteration algorithm and show how this structure can be generalized. Let us start by looking at a 2-dimensional layout of how the Value Functions progress in Policy Iteration from the starting Value Function $\boldsymbol{V_0}$ to the final Value Function $\boldsymbol{V^*}$.

$$\pi_1 = G(\boldsymbol{V_0}), \boldsymbol{V_0} \rightarrow \boldsymbol{B}^{\pi_1}(\boldsymbol{V_0}) \rightarrow (\boldsymbol{B}^{\pi_1})^2(\boldsymbol{V_0}) \rightarrow \ldots (\boldsymbol{B}^{\pi_1})^i(\boldsymbol{V_0}) \rightarrow \ldots \boldsymbol{V}^{\pi_1} = \boldsymbol{V_1}$$

$$\pi_2 = G(\boldsymbol{V_1}), \boldsymbol{V_1} \rightarrow \boldsymbol{B}^{\pi_2}(\boldsymbol{V_1}) \rightarrow (\boldsymbol{B}^{\pi_2})^2(\boldsymbol{V_1}) \rightarrow \ldots (\boldsymbol{B}^{\pi_2})^i(\boldsymbol{V_1}) \rightarrow \ldots \boldsymbol{V}^{\pi_2} = \boldsymbol{V_2}$$

$$\ldots$$

$$\ldots$$

$$\pi_{j+1} = G(\boldsymbol{V_j}), \boldsymbol{V_j} \rightarrow \boldsymbol{B}^{\pi_{j+1}}(\boldsymbol{V_j}) \rightarrow (\boldsymbol{B}^{\pi_{j+1}})^2(\boldsymbol{V_j}) \rightarrow \ldots (\boldsymbol{B}^{\pi_{j+1}})^i(\boldsymbol{V_j}) \rightarrow \ldots \boldsymbol{V}^{\pi_{j+1}} = \boldsymbol{V^*}$$

Each row in the layout above represents the progression of the Value Function for a specific policy. Each row starts with the creation of the policy (for that row) using the Greedy Policy Function G, and the remainder of the row consists of successive applications of the Bellman Policy Operator (using that row's policy) until convergence to the Value Function for that row's policy. So each row starts with a Policy Improvement and the rest of the row is a Policy Evaluation. Notice how the end of one row dovetails into the start of the next row with application of the Greedy Policy Function G. It's also important to recognize that Greedy Policy Function as well as Bellman Policy Operator apply to *all states* in $\mathcal{N}$. So, in fact, the entire Policy Evaluation algorithm has 3 nested loops. The outermost loop is over the rows in this 2-dimensional layout (each iteration in this outermost loop creates an improved policy). The loop within this outermost loop is over the columns in each row (each iteration in this loop applies the Bellman Policy Operator, i.e. the steps in Policy Evaluation). The innermost loop is over each state in $\mathcal{N}$ since we need to sweep through all states in updating the Value Function when the Bellman Policy Operator is applied on a Value Function (we also need to sweep through all states in applying the Greedy Policy Function to improve the policy).

A higher-level view of Policy Iteration is to think of Policy Evaluation and Policy Improvement going back and forth iteratively - Policy Evaluation takes a policy and creates the Value Function for that policy, while Policy Improvement takes a Value Function and creates a Greedy Policy from it (that is improved relative to the previous policy). This was depicted in Figure 3.1. It is important to recognize that this loop of Policy Evaluation and Policy Improvement works to make the Value Function and the Policy increasingly consistent with each other, until we reach convergence when the Value Function and Policy become completely consistent with each other (as was illustrated in Figure 3.2).

We'd also like to share a visual of Policy Iteration that is quite popular in much of the literature on Dynamic Programming. It is the visual of Figure 3.3. It's a somewhat fuzzy sort of visual, but it has it's benefits in terms of pedagogy of Policy Iteration. The idea behind this image is that the lower line represents the "policy line" indicating the progression of the policies as Policy Iteration algorithm moves along and the upper line represents the "value function line" indicating the progression of the Value Functions as Policy Iteration algorithm moves along. The arrows pointing towards the upper line ("value function line") represent a Policy Evaluation for a given policy π, yielding the point (Value Function) $\boldsymbol{V}^\pi$ on the upper line. The arrows pointing towards the lower line ("policy line") represent a Greedy Policy Improvement from a Value Function $\boldsymbol{V}^\pi$, yielding the point (policy) $\pi' = G(\boldsymbol{V}^\pi)$ on the lower line. The key concept here is that Policy Evaluation (arrows pointing to upper line) and Policy Improvement (arrows pointing to lower line) are "competing" - they "push in different directions" even as they aim to get the Value Function and Policy to be consistent with each other. This concept of simultaneously trying to compete and trying to be consistent might seem confusing and contradictory, so it deserves a proper explanation. Things become clear by noting that there are actually two notions of consistency between a Value Function $\boldsymbol{V}$ and Policy π.

1. The notion of the Value Function $\boldsymbol{V}$ being consistent with/close to the

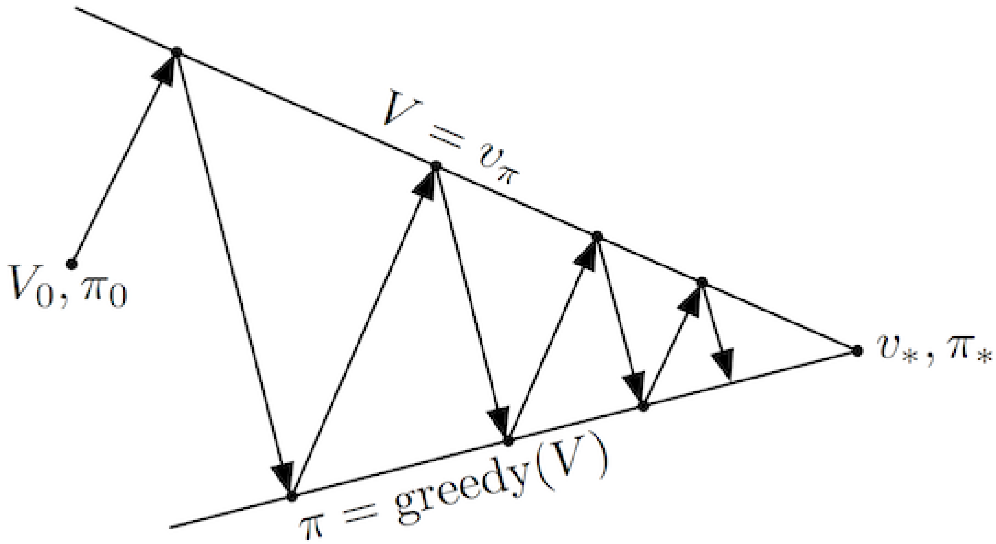

Figure 3.3.: Progression Lines of Value Function and Policy in Policy Iteration

Value Function $\boldsymbol{V}^\pi$ of the policy π.

2. The notion of the Policy π being consistent with/close to the Greedy Policy $G(\boldsymbol{V})$ of the Value Function $\boldsymbol{V}$.

Policy Evaluation aims for the first notion of consistency, but in the process, makes it worse in terms of the second notion of consistency. Policy Improvement aims for the second notion of consistency, but in the process, makes it worse in terms of the first notion of consistency. This also helps us understand the rationale for alternating between Policy Evaluation and Policy Improvement so that neither of the above two notions of consistency slip up too much (thanks to the alternating propping up of the two notions of consistency). Also, note that as Policy Iteration progresses, the upper line and lower line get closer and closer and the "pushing in different directions" looks more and more collaborative rather than competing (the gaps in consistency becomes lesser and lesser). In the end, the two lines intersect, when there is no more pushing to do for either of Policy Evaluation or Policy Improvement since at convergence, π^* and $\boldsymbol{V}^*$ have become completely consistent.

Now we are ready to talk about Generalized Policy Iteration - the idea that neither of Evaluation and Improvement steps need to go fully towards the notion of consistency they are respectively striving for. As a simple example, think of modifying Policy Evaluation (say for a policy π) to not go all the way to $\boldsymbol{V}^\pi$, but instead just perform say 3 Bellman Policy Evaluations. This means it would partially bridge the gap on the first notion of consistency (getting closer to $\boldsymbol{V}^\pi$ but not go all the way to $\boldsymbol{V}^\pi$), but it would also mean not slipping too much on the second notion of consistency. As another example, think of updating just 5 of the states (say in a large state space) with the Greedy Policy Improvement function (rather than the normal Greedy Policy Improvement function that operates on all the states). This means it would partially bridge the gap on the second notion of consistency (getting closer to $G(\boldsymbol{V}^{\pi})$ but not go all the way to $G(\boldsymbol{V}^{\pi})$), but it would also mean not slipping too much on the first notion of consistency. A concrete example of Generalized Policy Iteration is in fact Value Iteration. In Value Iteration, we apply the Bellman Policy Iterator just once before moving on to Policy Improvement. In a 2-dimensional layout, this is what Value Iteration looks like:

$$\pi_1 = G(\boldsymbol{V_0}), \boldsymbol{V_0} \rightarrow \boldsymbol{B}^{\pi_1}(\boldsymbol{V_0}) = \boldsymbol{V_1}$$
$$\pi_2 = G(\boldsymbol{V_1}), \boldsymbol{V_1} \rightarrow \boldsymbol{B}^{\pi_2}(\boldsymbol{V_1}) = \boldsymbol{V_2}$$
$$\ldots$$
$$\ldots$$
$$\pi_{j+1} = G(\boldsymbol{V_j}), \boldsymbol{V_j} \rightarrow \boldsymbol{B}^{\pi_{j+1}}(\boldsymbol{V_j}) = \boldsymbol{V}^*$$

So the greedy policy improvement step is unchanged, but Policy Evaluation is reduced to just a single Bellman Policy Operator application. In fact, pretty much all algorithms in Reinforcement Learning can be viewed as special cases of Generalized Policy Iteration. In Reinforcement Learning algorithms, we often do the evaluation for just a single state (versus for all states in usual Policy Iteration, or even in Value Iteration) and we also often do the policy improvement for just a single state. So many Reinforcement Learning algorithms are an alternating sequence of single-state evaluation and single-state policy improvement (where the single-state is the state produced by sampling or the state that is encountered in a real-world environment interaction). Figure 3.4 illustrates Generalized Policy Iteration as the red arrows (versus the black arrows which correspond to usual Policy Iteration algorithm). Note how the red arrows don't go all the way to either the "value function line"" or the "policy line" but the red arrows do go some part of the way towards the line they are meant to go towards at that stage in the algorithm.

We would go so far as to say that the Bellman Equations and the concept of Generalized Policy Iteration are the two most important concepts to internalize in the study of Reinforcement Learning, and we highly encourage you to think along the lines of these two ideas when we present several algorithms later in this book. The importance of the concept of Generalize Policy Iteration (GPI) might not be fully visible to you yet, but we hope that GPI will be your mantra by the time you finish this book. For now, let's just note the key takeaway regarding GPI - it is any algorithm to solve MDP control that alternates between *some form of* value evaluation for a policy and *some form of* policy improvement. We will bring up GPI several times later in this book.

Aysnchronous Dynamic Programming

The classical Dynamic Programming algorithms we have described in this chapter are qualified as *Synchronous* Dynamic Programming algorithms. The word

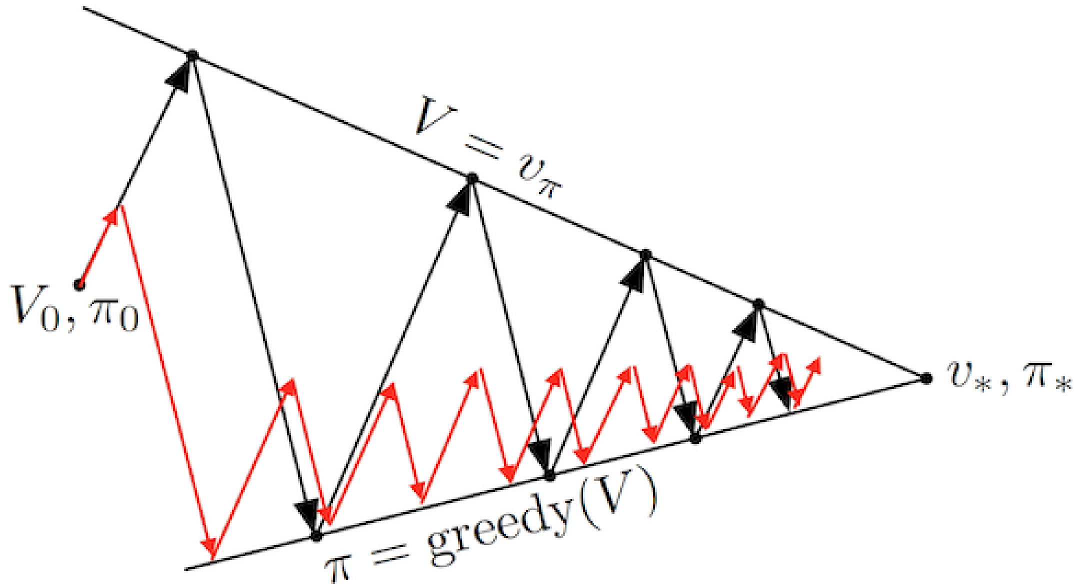

Figure 3.4.: Progression Lines of Value Function and Policy in Generalized Policy Iteration

synchronous refers to two things:

1. All states' values are updated in each iteration
2. The mathematical description of the algorithms corresponds to all the states' value updates to occur simultaneously. However, in code we write (in Python, where computation is serial and not parallel), the way to implement this simultaneous update is by creating a new copy of the Value Function vector and sweeping through all states to assign values to the new copy from the values in the old copy.

In practice, Dynamic Programming algorithms are typically implemented as *Asynchronous* algorithms, where the above two constraints (all states updated simultaneously) are relaxed. The term *asynchronous* affords a lot of flexibility - we can update a subset of states in each iteration, and we can update states in any order we like. A natural outcome of this relaxation of the synchronous constraint is that we can just maintain one vector for the value function and update the values *in-place*. This has considerable benefits as an updated value for a state is immediately available for updates of other states (note: in synchronous, with the old and new value function vectors, one has to wait for the entire states sweep to be over until an updated state value is available for another state's update). In fact, in-place updates of value function is the norm in practical implementations of algorithms to solve the MDP Control problem.

Another feature of practical asynchronous algorithms is that we can prioritize the order in which state values are updated. There are many ways in which algorithms assign priorities, and we'll just highlight a simple but effective way of prioritizing state value updates. It's known as *prioritized sweeping*. We maintain a queue of the states sorted by their "value function gaps" $g : \mathcal{N} \rightarrow \mathbb{R}$ (illustrated below as an example for Value Iteration):

$$g(s) = |V(s) - \arg\max_{a \in \mathcal{A}}(\mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V(s'))| \text{ for all } s \in \mathcal{N}$$

After each state's value is updated with the Bellman Optimality Operator, we update the Value Function Gap for all the states whose Value Function Gap does get changed as a result of this state value update. These are exactly the states from which we have a probabilistic transition to the state whose value just got updated. What this also means is that we need to maintain the reverse transition dynamics in our data structure representation. So, after each state value update, the queue of states is resorted (by their value function gaps). We always pull out the state with the largest value function gap (from the top of the queue), and update the value function for that state. This prioritizes updates of states with the largest gaps, and it ensures that we quickly get to a point where all value function gaps are low enough.

Another form of Asynchronous Dynamic Programming worth mentioning here is *Real-Time Dynamic Programming* (RTDP). RTDP means we run a Dynamic Programming algorithm *while* the agent is experiencing real-time interaction with the environment. When a state is visited during the real-time interaction, we make an update for that state's value. Then, as we transition to another state as a result of the real-time interaction, we update that new state's value, and so on. Note also that in RTDP, the choice of action is the real-time action executed by the agent, which the environment responds to. This action choice is governed by the policy implied by the value function for the encountered state at that point in time in the real-time interaction.

Finally, we need to highlight that often special types of structures of MDPs can benefit from specific customizations of Dynamic Programming algorithms (typically, Asynchronous). One such specialization is when each states is encountered not more than once in each random sequence of state occurrences when an agent plays out an MDP, and when all such random sequences of the MDP terminate. This structure can be conceptualized as a Directed Acylic Graph wherein each non-terminal node in the Directed Acyclic Graph (DAG) represents a pair of non-terminal state and action, and each terminal node in the DAG represents a terminal state (the graph edges represent probabilistic transitions of the MDP). In this specialization, the MDP Prediction and Control problems can be solved in a fairly simple manner - by walking backwards on the DAG from the terminal nodes and setting the Value Function of visited states (in the backward DAG walk) using the Bellman Optimality Equation (for Control) or Bellman Policy Equation (for Prediction). Here we don't need the "iterate to convergence" approach of Policy Evaluation or Policy Iteration or Value Iteration. Rather, all these Dynamic Programming algorithms essentially reduce to a simple back-propagation of the Value Function on the DAG. This means, states are visited (and their Value Functions set) in the order determined by the reverse sequence of a Topological Sort on the DAG. We shall make this DAG back-propagation Dynamic Programming algorithm clear for a special DAG structure - Finite-Horizon MDPs - where all random sequences of the MDP terminate after a fixed number of time steps. This special case of Finite-Horizon MDPs is fairly common in Financial Applications and so, we cover it in detail in the next section.

Finite-Horizon Dynamic Programming: Backward Induction

In Finite-Horizon MDPs, each sequence terminates within a fixed finite number of time steps, that we shall denote as T. So, all states at time-step T are terminal states and some states before time-step T could be terminal states. For all $t = 0, 1, \ldots, T$, denote the set of states for time step t as $\mathcal{S}_t$, the set of terminal states for time step t as $\mathcal{T}_t$ and the set of non-terminal states for time step t as $\mathcal{N}_t = \mathcal{S}_t - \mathcal{T}_t$ (note: $\mathcal{N}_T = \emptyset$). As mentioned previously, in these type of non-stationary situations, we augment each state to include the index of the time step so that the augmented state at time step t is (t, s_t) for $s_t \in \mathcal{S}_t$. The entire MDP's (augmented) state space $\mathcal{S}$ is:

$$\{(t, s_t) | t = 0, 1, \ldots, T, s_t \in \mathcal{S}_t\}$$

We need a Python class to represent this augmented state space.

```
@dataclass(frozen=True)
class WithTime(Generic[S]):
    state: S
    time: int = 0
```

The set of terminal states $\mathcal{T}$ is:

$$\{(t, s_t) | t = 0, 1, \ldots, T, s_t \in \mathcal{T}_t\}$$

As usual, the set of non-terminal states is denoted as $\mathcal{N} = \mathcal{S} - \mathcal{T}$.

Let us denote the allowable actions for states in $\mathcal{N}_t$ as $\mathcal{A}_t$. In a more generic setting, as we shall represent in our code, each non-terminal state (t, s_t) has it's own set of allowable actions, denoted $\mathcal{A}(s_t)$, However, for ease of exposition, here we shall treat all non-terminal states at a particular time step to have the same set of allowable actions $\mathcal{A}_t$. Let us denote the entire action space $\mathcal{A}$ of the MDP as the union of all the $\mathcal{A}_t$ over all $t = 0, 1, \ldots, T-1$.

The state-reward transition probability function

$$\mathcal{P}_R : \mathcal{N} \times \mathcal{A} \times \mathbb{R} \times \mathcal{S} \rightarrow [0, 1]$$

is given by:

$$\mathcal{P}_R((t, s_t), a_t, r, (t', s_{t'})) = \begin{cases} (\mathcal{P}_R)_t(s_t, a_t, r, s_{t'}) & \text{if } t' = t+1 \text{ and } s_{t'} \in \mathcal{S}_{t'} \\ 0 & \text{otherwise} \end{cases}$$

for all $t = 0, 1, \ldots T-1, s_t \in \mathcal{N}_t, a_t \in \mathcal{A}_t, t' = 0, 1, \ldots, T, s_{t'} \in \mathcal{S}_{t'}$, where

$$(\mathcal{P}_R)_t : \mathcal{N}_t \times \mathcal{A}_t \times \mathbb{R} \times \mathcal{S}_{t+1} \rightarrow [0, 1]$$

are the separate state-reward transition probability functions for each of the time steps $t = 0, 1, \ldots, T-1$ such that

$$\sum_{s_{t+1} \in \mathcal{S}_{t+1}} \sum_{r \in \mathbb{R}} (\mathcal{P}_R)_t(s_t, a_t, r, s_{t+1}) = 1$$

for all $t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t, a_t \in \mathcal{A}_t$.

So it is convenient to represent a finite-horizon MDP with separate state-reward transition probability functions $(\mathcal{P}_R)_t$ for each time step. Likewise, it is convenient to represent any policy of the MDP

$$\pi : \mathcal{N} \times \mathcal{A} \rightarrow [0, 1]$$

as:

$$\pi((t, s_t), a_t) = \pi_t(s_t, a_t)$$

where

$$\pi_t : \mathcal{N}_t \times \mathcal{A}_t \rightarrow [0, 1]$$

are the separate policies for each of the time steps $t = 0, 1, \ldots, T-1$

So essentially we interpret π as being composed of the sequence $(\pi_0, \pi_1, \ldots, \pi_{T-1})$.

Consequently, the Value Function for a given policy π (equivalently, the Value Function for the π-implied MRP)

$$V^\pi : \mathcal{N} \rightarrow \mathbb{R}$$

can be conveniently represented in terms of a sequence of Value Functions

$$V_t^\pi : \mathcal{N}_t \rightarrow \mathbb{R}$$

for each of time steps $t = 0, 1, \ldots, T-1$, defined as:

$$V^\pi((t, s_t)) = V_t^\pi(s_t) \text{ for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t$$

Then, the Bellman Policy Equation can be written as:

$$\begin{aligned} V_t^\pi(s_t) = \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \sum_{r \in \mathbb{R}} (\mathcal{P}_R^{\pi_t})_t(s_t, r, s_{t+1}) \cdot (r + \gamma \cdot W_{t+1}^\pi(s_{t+1})) \\ \text{for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t \end{aligned} \tag{3.7}$$

where

$$W_t^\pi(s_t) = \begin{cases} V_t^\pi(s_t) & \text{if } s_t \in \mathcal{N}_t \\ 0 & \text{if } s_t \in \mathcal{T}_t \end{cases}$$

for all $t = 1, 2, \ldots, T$ and where $(\mathcal{P}_R^{\pi_t})_t : \mathcal{N}_t \times \mathbb{R} \times \mathcal{S}_{t+1}$ for all $t = 0, 1, \ldots, T-1$ represent the π-implied MRP's state-reward transition probability functions for the time steps, defined as:

$$(\mathcal{P}_R^{\pi_t})_t(s_t, r, s_{t+1}) = \sum_{a_t \in \mathcal{A}_t} \pi_t(s_t, a_t) \cdot (\mathcal{P}_R)_t(s_t, a_t, r, s_{t+1}) \text{ for all } t = 0, 1, \ldots, T-1$$

So for a Finite MDP, this yields a simple algorithm to calculate V_t^π for all t by simply decrementing down from $t = T-1$ to $t = 0$ and using Equation (3.7) to calculate V_t^π for all $t = 0, 1, \ldots, T-1$ from the known values of W_{t+1}^π (since we are decrementing in time index t).

This algorithm is the adaptation of Policy Evaluation to the finite horizon case with this simple technique of "stepping back in time" (known as *Backward Induction*). Let's write some code to implement this algorithm. We are given a MDP over the augmented (finite) state space `WithTime[S]`, and a policy π (also over the augmented state space `WithTime[S]`). So, we can use the method `apply_finite_policy` in `FiniteMarkovDecisionProcess[WithTime[S], A]` to obtain the π-implied MRP of type `FiniteMarkovRewardProcess[WithTime[S]]`. Our first task to to "unwrap" the state-reward probability transition function $\mathcal{P}_R^\pi$ of this π-implied MRP into a time-indexed sequenced of state-reward probability transition functions $(\mathcal{P}_R^{\pi_t})_t, t = 0, 1, \ldots, T-1$. This is accomplished by the following function `unwrap_finite_horizon_MRP` (`itertools.groupby` groups the augmented states by their time step, and the function `without_time` strips the time step from the augmented states when placing the states in $(\mathcal{P}_R^{\pi_t})_t$, i.e., `Sequence[RewardTransition[S]]`).

```
from itertools import groupby

StateReward = FiniteDistribution[Tuple[S, float]]
RewardTransition = Mapping[S, Optional[StateReward[S]]]

def unwrap_finite_horizon_MRP(
    process: FiniteMarkovRewardProcess[WithTime[S]]
) -> Sequence[RewardTransition[S]]:

    def time(x: WithTime[S]) -> int:
        return x.time

    def without_time(
        arg: Optional[StateReward[WithTime[S]]]
    ) -> Optional[StateReward[S]]:
        return None if arg is None else arg.map(
            lambda s_r: (s_r[0].state, s_r[1])
        )

    return [{s.state: without_time(process.transition_reward(s))
```

```
            for s in states} for _, states in groupby(
                sorted(process.states(), key=time),
                key=time
            )][:-1]
```

Now that we have the state-reward transition functions $(\mathcal{P}_R^{\pi_t})_t$ arranged in the form of a Sequence[RewardTransition[S]], we are ready to perform backward induction to calculate V_t^π. The following function evaluate accomplishes it with a straightforward use of Equation (3.7), as described above.

```
def evaluate(
    steps: Sequence[RewardTransition[S]],
    gamma: float
) -> Iterator[V[S]]:
    v: List[Dict[S, float]] = []

    for step in reversed(steps):
        v.append({s: res.expectation(
            lambda s_r: s_r[1] + gamma * (v[-1][s_r[0]] if
                                          len(v) > 0 and s_r[0] in v[-1]
                                          else 0.)
            ) for s, res in step.items() if res is not None})

    return reversed(v)
```

If $|\mathcal{N}_t|$ is $O(m)$, then the running time of this algorithm is $O(m^2 \cdot T)$. However, note that it takes $O(m^2 \cdot k \cdot T)$ to convert the MDP to the π-implied MRP (where $|\mathcal{A}_t|$ is $O(k)$).

Now we move on to the Control problem - to calculate the Optimal Value Function and the Optimal Policy. Similar to the pattern seen so far, the Optimal Value Function

$$V^* : \mathcal{N} \to \mathbb{R}$$

can be conveniently represented in terms of a sequence of Value Functions

$$V_t^* : \mathcal{N}_t \to \mathbb{R}$$

for each of time steps $t = 0, 1, \ldots, T-1$, defined as:

$$V^*((t, s_t)) = V_t^*(s_t) \text{ for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t$$

Thus, the Bellman Optimality Equation can be written as:

$$V_t^*(s_t) = \max_{a_t \in \mathcal{A}_t} \{ \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \sum_{r \in \mathbb{R}} (\mathcal{P}_R)_t(s_t, a_t, r, s_{t+1}) \cdot (r + \gamma \cdot W_{t+1}^*(s_{t+1})) \}$$
$$\text{for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t \tag{3.8}$$

where

$$W_t^*(s_t) = \begin{cases} V_t^*(s_t) & \text{if } s_t \in \mathcal{N}_t \\ 0 & \text{if } s_t \in \mathcal{T}_t \end{cases}$$

for all $t = 1, 2, \ldots, T$.

The associated Optimal (Deterministic) Policy

$$(\pi_D^*)_t : \mathcal{N}_t \rightarrow \mathcal{A}_t$$

is defined as:

$$(\pi_D^*)_t(s_t) = \arg\max_{a_t \in \mathcal{A}_t} \{ \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \sum_{r \in \mathbb{R}} (\mathcal{P}_R)_t(s_t, a_t, r, s_{t+1}) \cdot (r + \gamma \cdot W_{t+1}^*(s_{t+1})) \}$$
$$\text{for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t \tag{3.9}$$

For the case of a Finite MDP, this yields a simple algorithm to calculate V_t^* for all t, by simply decrementing down from $t = T-1$ to $t = 0$, using Equation (3.8) to calculate V_t^*, and Equation (3.9) to calculate $(\pi_D^*)_t$ for all $t = 0, 1, \ldots, T-1$ from the known values of W_{t+1}^* (since we are decrementing in time index t).

This algorithm is the adaptation of Value Iteration to the finite horizon case with this simple technique of "stepping back in time" (known as *Backward Induction*). Let's write some code to implement this algorithm. We are given a MDP over the augmented (finite) state space `WithTime[S]`. So this MDP is of type `FiniteMarkovDecisionProcess[WithTime[S], A]`. Our first task to to "unwrap" the state-reward probability transition function $\mathcal{P}_R$ of this MDP into a time-indexed sequenced of state-reward probability transition functions $(\mathcal{P}_R)_t, t = 0, 1, \ldots, T-1$. This is accomplished by the following function `unwrap_finite_horizon_MDP` (`itertools.groupby` groups the augmented states by their time step, and the function `without_time` strips the time step from the augmented states when placing the states in $(\mathcal{P}_R)_t$, i.e., `Sequence[StateActionMapping[S, A]]`).

```
from itertools import groupby

ActionMapping = Mapping[A, StateReward[S]]
StateActionMapping = Mapping[S, Optional[ActionMapping[A, S]]]

def unwrap_finite_horizon_MDP(
    process: FiniteMarkovDecisionProcess[WithTime[S], A]
) -> Sequence[StateActionMapping[S, A]]:
    def time(x: WithTime[S]) -> int:
        return x.time

    def without_time(
        arg: Optional[ActionMapping[A, WithTime[S]]]
    ) -> Optional[ActionMapping[A, S]]:
```

```
        return None if arg is None else {
            a: sr_distr.map(lambda s_r: (s_r[0].state, s_r[1]))
            for a, sr_distr in arg.items()
        }

    return [{s.state: without_time(process.action_mapping(s))
             for s in states} for _, states in groupby(
                sorted(process.states(), key=time),
                key=time
             )][:-1]
```

Now that we have the state-reward transition functions $(\mathcal{P}_R)_t$ arranged in the form of a `Sequence[StateActionMapping[S, A]]`, we are ready to perform backward induction to calculate V_t^*. The following function `optimal_vf_and_policy` accomplishes it with a straightforward use of Equation (3.7), as described above.

```
from operator import itemgetter

def optimal_vf_and_policy(
    steps: Sequence[StateActionMapping[S, A]],
    gamma: float
) -> Iterator[Tuple[V[S], FinitePolicy[S, A]]]:
    v_p: List[Tuple[Dict[S, float], FinitePolicy[S, A]]] = []

    for step in reversed(steps):
        this_v: Dict[S, float] = {}
        this_a: Dict[S, FiniteDistribution[A]] = {}
        for s, actions_map in step.items():
            if actions_map is not None:
                action_values = ((res.expectation(
                    lambda s_r: s_r[1] + gamma * (v_p[-1][0][s_r[0]] if
                                                  len(v_p) > 0 and
                                                  s_r[0] in v_p[-1][0]
                                                  else 0.)
                ), a) for a, res in actions_map.items())
                v_star, a_star = max(action_values, key=itemgetter(0))
                this_v[s] = v_star
                this_a[s] = Constant(a_star)
        v_p.append((this_v, FinitePolicy(this_a)))

    return reversed(v_p)
```

If $|\mathcal{N}_t|$ is $O(m)$ for all t and $|\mathcal{A}_t|$ is $O(k)$, then the running time of this algorithm is $O(m^2 \cdot k \cdot T)$.

Note that these algorithms for finite-horizon finite MDPs do not require any "iterations to convergence" like we had for regular Policy Evaluation and Value

Iteration. Rather, in these algorithms we simply walk back in time and immediately obtain the Value Function for each time step from the next time step's Value Function (which is already known since we walk back in time). This technique of "backpropagation of Value Function" goes by the name of *Backward Induction* algorithms, and is quite commonplace in many Financial applications (as we shall see later in this book). The above Backward Induction code is in the file rl/finite_horizon.py.

Dynamic Pricing for End-of-Life/End-of-Season of a Product

Now we consider a rather important business application - Dynamic Pricing. We consider the problem of Dynamic Pricing for the case of products that reach their end of life or at the end of a season after which we don't want to carry the product anymore. We need to adjust the prices up and down dynamically depending on how much inventory of the product you have, how many days to go for end-of-life/end-of-season, and your expectations of customer demand as a function of price adjustments. To make things concrete, assume you own a super-market and you are T days away from Halloween. You have just received M Halloween masks from your supplier. You want to dynamically set the selling price of the Halloween masks at the start of each day in a manner that maximizes your *Expected Total Sales Revenue* for Halloween masks from today until Halloween (assume no one will buy Halloween masks after Halloween).

Assume that for each of the T days, at the start of the day, you are required to select a price for that day from one of N prices $P_1, P_2, \ldots, P_N \in \mathbb{R}$, such that your selected price will be the selling price for all masks on that day. Assume that the customer demand for number of Halloween masks on any day is governed by a Poisson probability distribution with mean $\lambda_i \in \mathbb{R}$ if you select that day's price to be P_i (where i is a choice among $1, 2, \ldots, N$). Note that on any given day, the demand could exceed the number of Halloween masks you have in the store, in which case the number of masks sold on that day will be equal to the number of Halloween masks you had at the start of that day.

A state for this MDP is given by a pair (t, I_t) where $t \in \{0, 1, \ldots, T\}$ denotes the time index and $I_t \in \{0, 1, \ldots, M\}$ denotes the inventory at time t. Using our notation from the previous section, $\mathcal{S}_t = \mathcal{N}_t = \{0, 1, \ldots, M\}$ for all $t = 0, 1, \ldots, T$ so that $I_t \in \mathcal{S}_t$. Also, the action choices at time t can be represented by the choice of integers from 1 to N. Therefore, $\mathcal{A}_t = \{1, 2, \ldots, N\}$.

Note that:

$$I_0 = M, I_{t+1} = \max(0, I_t - d_t) \text{ for } 0 \leq t < T$$

where d_t is the random demand on day t governed by a Poisson distribution with mean λ_i if the action (index of the price choice) on day t is $i \in \mathcal{A}_t$. Also, note that sales revenue R_t on day t is equal to $\min(I_t, d_t) \cdot P_i$. Therefore, the state-reward probability transition function for time index t

$$(\mathcal{P}_R)_t : \mathcal{N}_t \times \mathcal{A}_t \times \mathbb{R} \times \mathcal{S}_{t+1}$$

is defined as:

$$(\mathcal{P}_R)_t(I_t, i, R_t, I_t - k) = \begin{cases} \frac{e^{-\lambda_i}\lambda_i^k}{k!} & \text{if } k < I_t \text{ and } R_t = k \cdot P_i \\ \sum_{j=I_t}^{\infty} \frac{e^{-\lambda_i}\lambda_i^j}{j!} & \text{if } k = I_t \text{ and } R_t = k \cdot P_i \\ 0 & \text{otherwise} \end{cases}$$

for all $0 \leq t < T$

Using the definition of $(\mathcal{P}_R)_t$ and using the boundary condition $V_T^*(I_T) = 0$ for all $I_T \in \{0, 1, \ldots, M\}$, we can perform the backward induction algorithm to calculate V_t^* and associated optimal (deterministic) policy $(\pi_D^*)_t$ for all $0 \leq t < T$.

Now let's write some code to represent this Dynamic Programming problem as a `FiniteMarkovDecisionProcess` and determine it's optimal policy, i.e., the Optimal (Dynamic) Price at time step t and at any level of inventory I_t. The type $\mathcal{N}_t$ is `int` and the type $\mathcal{A}_t$ is also`int`. So we shall be creating a MDP of type `FiniteMarkovDecisionProcess[WithTime[int], int]` (since the augmented state space is `WithTime[int]`). Our first task is to construct $\mathcal{P}_R$ of type:

`Mapping[WithTime[int], Optional[Mapping[int, FiniteDistribution[Tuple[WithTime[int], float]]]]]`

In the class `ClearancePricingMDP` below, $\mathcal{P}_R$ is manufactured in `__init__` and is used to create the attribute `mdp: FiniteMarkovDecisionProces[WithTime[int], int]`. Since $\mathcal{P}_R$ is independent of time, we first create a single-step (time-invariant) MDP `single_step_mdp: FiniteMarkovDecisionProcess[int, int]` (think of this as the building-block MDP), and then use the method `finite_horizon_mdp` (from file rl/finite_horizon.py) to turn `single_step_mdp` to `mdp`. The constructor argument `initial_inventory: int` represents the initial inventory M. The constructor argument `time_steps` represents the number of time steps T. The constructor argument `price_lambda_pairs` represents $[(P_i, \lambda_i)|1 \leq i \leq N]$.

```
from scipy.stats import poisson

class ClearancePricingMDP:

    initial_inventory: int
    time_steps: int
    price_lambda_pairs: Sequence[Tuple[float, float]]
    single_step_mdp: FiniteMarkovDecisionProcess[int, int]
    mdp: FiniteMarkovDecisionProcess[WithTime[int], int]

    def __init__(
        self,
        initial_inventory: int,
        time_steps: int,
        price_lambda_pairs: Sequence[Tuple[float, float]]
    ):
        self.initial_inventory = initial_inventory
```

```
        self.time_steps = time_steps
        self.price_lambda_pairs = price_lambda_pairs
        distrs = [poisson(l) for _, l in price_lambda_pairs]
        prices = [p for p, _ in price_lambda_pairs]
        self.single_step_mdp: FiniteMarkovDecisionProcess[int, int] =\
            FiniteMarkovDecisionProcess({
                s: {i: Categorical(
                    {(s - k, prices[i] * k):
                     (distrs[i].pmf(k) if k < s else 1 - distrs[i].cdf(s - 1))
                     for k in range(s + 1)})
                    for i in range(len(prices))}
                for s in range(initial_inventory + 1)
            })
        self.mdp = finite_horizon_MDP(self.single_step_mdp, time_steps)
```

Now let's write two methods for this class:

- `get_vf_for_policy` that produces the Value Function for a given policy π, by first creating the π-implied MRP from `mdp`, then unwrapping the MRP into a sequence of state-reward transition probability functions $(\mathcal{P}_R^{\pi_t})_t$, and then performing backward induction using the previously-written function `evaluate` to calculate the Value Function.
- `get_optimal_vf_and_policy` that produces the Optimal Value Function and Optimal Policy, by first unwrapping `mdp` into a sequence of state-reward transition probability functions $(\mathcal{P}_R)_t$, and then performing backward induction using the previously-written function `optimal_vf_and_policy` to calculate the Optimal Value Function and Optimal Policy.

```
    def get_vf_for_policy(
        self,
        policy: FinitePolicy[WithTime[int], int]
    ) -> Iterator[V[int]]:
        mrp: FiniteMarkovRewardProcess[WithTime[int]] \
            = self.mdp.apply_finite_policy(policy)
        return evaluate(unwrap_finite_horizon_MRP(mrp), 1.)

    def get_optimal_vf_and_policy(self)\
            -> Iterator[Tuple[V[int], FinitePolicy[int, int]]]:
        return optimal_vf_and_policy(unwrap_finite_horizon_MDP(self.mdp), 1.)
```

Now let's create a simple instance of `ClearancePricingMDP` for $M = 12, T = 8$ and 4 price choices: "Full Price", "30% Off", "50% Off", "70% Off" with respective mean daily demand of $0.5, 1.0, 1.5, 2.5$.

```
ii = 12
steps = 8
pairs = [(1.0, 0.5), (0.7, 1.0), (0.5, 1.5), (0.3, 2.5)]
```

```
cp: ClearancePricingMDP = ClearancePricingMDP(
    initial_inventory=ii,
    time_steps=steps,
    price_lambda_pairs=pairs
)
```

Now let us calculate it's Value Function for a stationary policy that chooses "Full Price" if inventory is less than 2, otherwise "30% Off" if inventory is less than 5, otherwise "50% Off" if inventory is less than 8, otherwise "70% Off". Since we have a stationary policy, we can represent it as a single-step policy and combine it with the single-step MDP we had created above (attribute `single_step_mdp`) to create a `single_step_mrp: FiniteMarkovRewardProcess[int]`. Then we use the function `finite_horizon_mrp` (from file rl/finite_horizon.py) to create the entire (augmented state) MRP of type `FiniteMarkovRewardProcess[WithTime[int]]`. Finally, we unwrap this MRP into a sequence of state-reward transition probability functions and perform backward induction to calculate the Value Function for this stationary policy. Running the following code tells us that $V_0^\pi(12)$ is about 4.91 (assuming full price is 1), which is the Expected Revenue one would obtain over 8 days, starting with an inventory of 12, and executing this stationary policy (under the assumed demand distributions as a function of the price choices).

```
def policy_func(x: int) -> int:
    return 0 if x < 2 else (1 if x < 5 else (2 if x < 8 else 3))

stationary_policy: FinitePolicy[int, int] = FinitePolicy(
    {s: Constant(policy_func(s)) for s in range(ii + 1)}
)

single_step_mrp: FiniteMarkovRewardProcess[int] = \
    cp.single_step_mdp.apply_finite_policy(stationary_policy)

vf_for_policy: Iterator[V[int]] = evaluate(
    unwrap_finite_horizon_MRP(finite_horizon_MRP(single_step_mrp, steps)),
    1.
)
```

Now let us determine what is the Optimal Policy and Optimal Value Function for this instance of `ClearancePricingMDP`. Running `cp.get_optimal_vf_and_policy()` and evaluating the Optimal Value Function for time step 0 and inventory of 12, i.e. $V_0^*(12)$, gives us a value of 5.64, which is the Expected Revenue we'd obtain over the 8 days if we executed the Optimal Policy.

Now let us plot the Optimal Price as a function of time steps and inventory levels.

```
import matplotlib.pyplot as plt
from matplotlib import cm
```

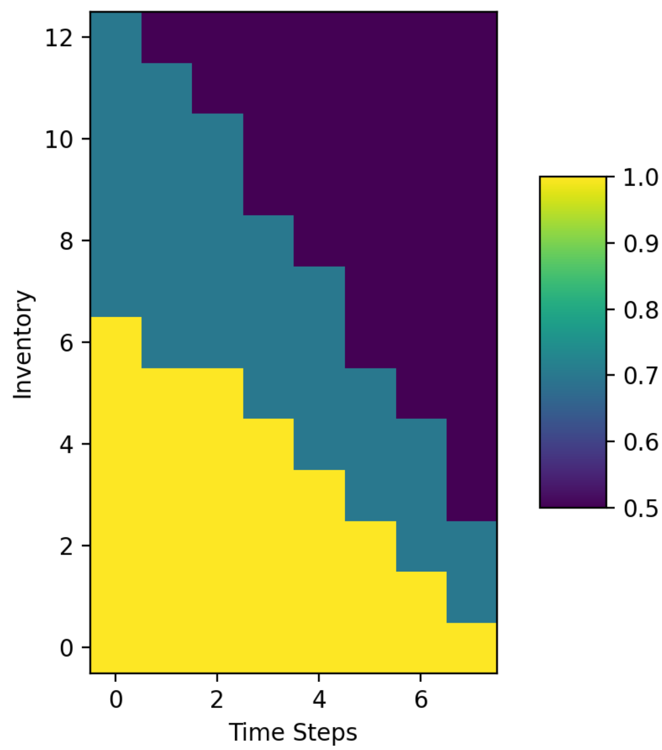

Figure 3.5.: Optimal Policy Heatmap

```python
import numpy as np

prices = [[pairs[policy.act(s).value][0] for s in range(ii + 1)]
          for _, policy in cp.get_optimal_vf_and_policy()]

heatmap = plt.imshow(np.array(prices).T, origin='lower')
plt.colorbar(heatmap, shrink=0.5, aspect=5)
plt.xlabel("Time Steps")
plt.ylabel("Inventory")
plt.show()
```

Figure 3.5 shows us the image produced by the above code. The color *Yellow* is "Full Price", the color *Blue* is "30% Off" and the color *Purple* is "50% Off". This tells us that on day 0, the Optimal Price is "30% Off" (corresponding to State 12, i.e., for starting inventory $M = I_0 = 12$). However, if the starting inventory I_0 were less than 7, then the Optimal Price is "Full Price". This makes intuitive sense because the lower the inventory, the less inclination we'd have to cut prices. We see that the thresholds for price cuts shift as time progresses (as we move horizontally in the figure). For instance, on Day 5, we set "Full Price" only if inventory has dropped below 3 (this would happen if we had a good degree of sales on the first 5 days), we set "30% Off" if inventory is 3 or 4 or 5, and we set "50% Off" if inventory is greater than 5. So even if we sold 6 units in the first 5 days, we'd offer "50% Off" because we have only 3 days remaining now and 6 units of inventory left. This makes intuitive sense. We see that the thresholds shift even further as we move to Days 6 and 7. We encourage you to play with this simple application of Dynamic Pricing by changing $M, T, N, [(P_i, \lambda_i) | 1 \leq i \leq N]$ and studying how the Optimal Value Function changes and more importantly, studying the thresholds of inventory (under optimality) for various choices of prices and how these thresholds vary as time progresses.

Extensions to Non-Tabular Algorithms

Finite MDP algorithms covered in this chapter are called "tabular" algorithms. The word "tabular" (for "table") refers to the fact that the MDP is specified in the form of a finite data structure and the Value Function is also represented as a finite "table" of non-terminal states and values. These tabular algorithms typically make a sweep through all non-terminal states in each iteration to update the value function. This is in contrast to algorithms for large state spaces or infinite state spaces where we need some function approximation for the value function. The good news is that we can modify each of these tabular algorithms such that instead of sweeping through all the non-terminal states at each step, we simply sample an appropriate subset of non-terminal states, update the Value Function for those states (with the same Bellman Operator calculations as for the case of tabular), and then create a function approximation for the Value Function using just the updated values for the sample of non-terminal states. The important point is that the fundamental structure of the algorithms and the fundamental principles (Fixed-Point and Bellman Operators) are still the same when we extend from these tabular algorithms to function approximation-based algorithms. In Chapter 4, we cover extensions of these Dynamic Programming algorithms from tabular methods to function approximation methods. We call these algorithms *Approximate Dynamic Programming*.

Summary of Key Learnings from this Chapter

Before we end this chapter, we'd like to highlight the three highly important concepts we learnt in this chapter:

- Fixed-Point of Functions and Banach Fixed-Point Theorem: The simple concept of Fixed-Point of Functions that is profound in its applications, and the Banach Fixed-Point Theorem that enables us to construct iterative algorithms to solve problems with fixed-point formulations.
- Generalized Policy Iteration: The powerful idea of alternating between improvement of a policy and evaluation of a value function, even though each of them might be partial applications. This generalized perspective unifies almost all of the algorithms that solve MDP Control problems.
- Backward Induction: A straightforward method to solve finite-horizon MDPs by simply backpropagating the Value Function from the horizon-end to the start.

4. Function Approximation and Approximate Dynamic Programming

In Chapter 3, we covered Dynamic Programming algorithms where the MDP is specified in the form of a finite data structure and the Value Function is represented as a finite "table" of states and values. These Dynamic Programming algorithms swept through all states in each iteration to update the value function. But when the state space is large (as is the case in real-world applications), these Dynamic Programming algorithm won't work because:

1. A "tabular" representation of the MDP or of the Value Function, won't fit within storage limits
2. Sweeping through all states and their transition probabilities would be time-prohibitive (or simply impossible, in the case of infinite state spaces)

Hence, when the state space is very large, we need to resort to approximation of the Value Function. The Dynamic Programming algorithms would need to be suitably modified to their Approximate Dynamic Programming (abbreviated as ADP) versions. The good news is that it's not hard to modify each of the (tabular) Dynamic Programming algorithms such that instead of sweeping through all the states in each iteration, we simply sample an appropriate subset of non-terminal states, update the Value Function for those states (with the same Bellman Operator calculations as for the case of tabular), and then construct an approximation for the Value Function using just the updated values for the sample of non-terminal states. Furthermore, if the set of transitions from a given state is large (or infinite), instead of using the explicit probabilities of those transitions, we can sample from the transitions probability distribution. The fundamental structure of the algorithms and the fundamental principles (Fixed-Point and Bellman Operators) would still be the same.

So, in this chapter, we do a quick review of function approximation, write some code for a couple for a couple of standard function approximation methods, and then utilize these function approximation methods to develop Approximate Dynamic Programming algorithms (in particular, Approximate Policy Evaluation, Approximate Value Iteration and Approximate Backward Induction). Since you are reading this book, it's highly likely that you are already familiar with the simple and standard function approximation methods such as linear function approximation and function approximation using neural networks supervised learning. So we shall go through the background on linear function approximation and neural networks supervised learning in a quick and terse manner, with the goal of developing some code for these methods that we can use not just for the ADP algorithms for this chapter, but also for RL algorithms later in the book. Note also that apart from approximation of Value Functions $\mathcal{N} \rightarrow \mathbb{R}$, these function approximation methods can also be used for approximation of Stochastic Policies $\mathcal{N} \times \mathcal{A} \rightarrow [0, 1]$ in Policy-based RL algorithms.

Function Approximation

In this section, we describe function approximation in a fairly generic setting (not specific to approximation of Value Functions or Policies). We denote the predictor variable as x, belonging to an arbitrary domain denoted $\mathcal{X}$ and the response variable as $y \in \mathbb{R}$. We treat x and y as unknown random variables and our goal is to estimate the probability distribution function f of the conditional random variable $y|x$ from data provided in the form of a sequence of (x, y) pairs. We shall consider parameterized functions f with the parameters denoted as w. The exact data type of w will depend on the specific form of function approximation. We denote the estimated probability of y conditional on x as $f(x; w)(y)$. Assume we are given data in the form of a sequence of n (x, y) pairs, as follows:

$$[(x_i, y_i)|1 \leq i \leq n]$$

The notion of estimating the conditional probability $\mathbb{P}[y|x]$ is formalized by solving for $w = w^*$ such that:

$$w^* = \arg\max_w \{\prod_{i=1}^{n} f(x_i; w)(y_i)\} = \arg\max_w \{\sum_{i=1}^{n} \log f(x_i; w)(y_i)\}$$

In other words, we shall be operating in the framework of Maximum Likelihood Estimation. We say that the data $[(x_i, y_i)|1 \leq i \leq n]$ specifies the *empirical probability distribution* D of $y|x$ and the function f (parameterized by w) specifies the *model probability distribution* M of $y|x$. With maximum likelihood estimation, we are essentially trying to reconcile the model probability distribution M with the empirical probability distribution D. Hence, maximum likelihood estimation is essentially minimization of a loss function defined as the cross-entropy $\mathcal{H}(D, M) = -\mathbb{E}_D[\log M]$ between the probability distributions D and M.

Our framework will allow for incremental estimation wherein at each iteration t of the incremental estimation (for $t = 1, 2, \ldots$), data of the form

$$[(x_{t,i}, y_{t,i})|1 \leq i \leq n_t]$$

is used to update the parameters from w_{t-1} to w_t (parameters initialized at iteration $t = 0$ to w_0). This framework can be used to update the parameters incrementally with a gradient descent algorithm, either stochastic gradient descent (where a single (x, y) pair is used for each iteration's gradient calculation) or mini-batch gradient descent (where an appropriate subset of the available data is used for each iteration's gradient calculation) or simply re-using the entire data available for each iteration's gradient calculation (and consequent, parameter update). Moreover, the flexibility of our framework, allowing for incremental estimation, is particularly important for Reinforcement Learning algorithms wherein we update the parameters of the function approximation from the new data that is generated from each state transition as a result of interaction with either the real environment or a simulated environment.

Among other things, the estimate f (parameterized by w) gives us the model-expected value of y conditional on x, i.e.

$$\mathbb{E}_M[y|x] = \mathbb{E}_{f(x;w)}[y] = \int_{-\infty}^{+\infty} y \cdot f(x;w)(y) \cdot dy$$

For the purposes of Approximate Dynamic Programming and Reinforcement Learning, the above expectation will provide an estimate of the Value Function for any state (x takes the role of the state, and y takes the role of the Value Function for that state). In the case of function approximation for policies, x takes the role of the state, and y takes the role of the action for that policy, and $f(x;w)$ will provide the probability distribution of the actions for state x (for a stochastic policy). It's also worthwhile pointing out that the broader theory of function approximations covers the case of multi-dimensional y (where y is a real-valued vector, rather than scalar) - this allows us to solve classification problems, along with regression problems. However, for ease of exposition and for sufficient coverage of function approximation applications in this book, we will only cover the case of scalar y.

Now let us write some code that captures this framework. We write an abstract base class `FunctionApprox` parameterized by `X` (to permit arbitrary data types $\mathcal{X}$), representing $f(x;w)$, with the following 3 key `@abstractmethods`, each of which will work with inputs of generic `Iterable` type (`Iterable` is any data type that we can iterate over, such as `Sequence` types or `Iterator` type):

1. `solve`: takes as input an `Iterable` of (x,y) pairs and solves for the optimal internal parameters w^* that minimizes the cross-entropy between the empirical probability distribution of the input data of (x,y) pairs and the model probability distribution $f(x;w)$. Some implementations of `solve` are iterative numerical methods and would require an additional input of `error_tolerance` that specifies the required precision of the best-fit parameters w^*. When an implementation of `solve` is an analytical solution not requiring an error tolerance, we specify the input `error_tolerance` as `None`. The output of `solve` is the `FunctionApprox` $f(x;w^*)$ (i.e., corresponding to the solved parameters w^*).
2. `update`: takes as input an `Iterable` of (x,y) pairs and updates the parameters w defining $f(x;w)$. The purpose of `update` is to perform an incremental (iterative) improvement to the parameters w, given the input data of (x,y) pairs in the current iteration. The output of `update` is the `FunctionApprox` corresponding to the updated parameters. Note that we should be able to `solve` based on an appropriate series of incremental `updates` (upto a specified `error_tolerance`).

3. `evaluate`: takes as input an `Iterable` of x values and calculates $\mathbb{E}_M[y|x] = \mathbb{E}_{f(x;w)}[y]$ for each of the input x values, and outputs these expected values in the form of a `numpy.ndarray`.

```
from abc import ABC, abstractmethod
import numpy as np

X = TypeVar('X')

class FunctionApprox(ABC, Generic[X]):

    @abstractmethod
    def evaluate(self, x_values_seq: Iterable[X]) -> np.ndarray:
        pass

    @abstractmethod
    def update(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]]
    ) -> FunctionApprox[X]:
        pass

    @abstractmethod
    def solve(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        error_tolerance: Optional[float] = None
    ) -> FunctionApprox[X]:
        pass
```

When concrete classes implementing `FunctionApprox` write the `solve` method in terms of the `update` method, they will need to check if a newly updated `FunctionApprox` is "close enough" to the previous `FunctionApprox`. So each of them will need to implement their own version of "Are two `FunctionApprox` instances within a certain `error_tolerance` of each other?". Hence, we need the following `@abstractmethod within`:

```
    @abstractmethod
    def within(self, other: FunctionApprox[X], tolerance: float) -> bool:
        pass
```

Any concrete class that implement this abstract class `FunctionApprox` will need to implement these four `abstractmethods` of `FunctionApprox`, based on the specific assumptions that the concrete class makes for f.

Next, we write some useful methods that the concrete classes implementing `FunctionApprox` can inherit and utilize. Firstly, we write a method called `iterate_updates` that takes as input a stream (`Iterator`) of `Iterable` of (x, y)

pairs, and performs a series of incremental updates to the parameters w (each using the update method), with each update done for each Iterable of (x, y) pairs in the input stream xy_seq: Iterator[Iterable[Tuple[X, float]]]. iterate_updates returns an Iterator[FunctionApprox[X]] representing the successively updated FunctionApprox instances as a consequence of the repeated invocations to update. Note the use of the standard Python function itertools.accumulate that calculates accumulated results (including intermediate results) on an Iterable, based on a provided function to govern the accumulation. In the code below, the Iterable is the input stream xy_seq_stream and the function governing the accumulation is the update method of FunctionApprox.

```
import itertools

    def iterate_updates(
        self,
        xy_seq_stream: Iterator[Iterable[Tuple[X, float]]]
    ) -> Iterator[FunctionApprox[X]]:
        return itertools.accumulate(
            xy_seq_stream,
            lambda fa, xy: fa.update(xy),
            initial=self
        )
```

Next, we write a method called rmse to calculate the Root-Mean-Squared-Error of the predictions for x (using evaluate) relative to associated (supervisory) y, given as input an Iterable of (x, y) pairs. This method will be useful in testing the goodness of a FunctionApprox estimate.

```
    def rmse(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]]
    ) -> float:
        x_seq, y_seq = zip(*xy_vals_seq)
        errors: np.ndarray = self.evaluate(x_seq) - np.array(y_seq)
        return np.sqrt(np.mean(errors * errors))
```

Finally, we write a method argmax that takes as input an Iterable of x values and returns the x value that maximizes $\mathbb{E}_{f(x;w)}[x]$.

```
    def argmax(self, xs: Iterable[X]) -> X:
        return list(xs)[np.argmax(self.evaluate(xs))]
```

The above code for FunctionApprox is in the file rl/function_approx.py.

Now we are ready to cover a concrete but simple function approximation - the case of linear function approximation.

Linear Function Approximation

We define a sequence of feature functions

$$\phi_j : \mathcal{X} \to \mathbb{R} \text{ for each } j = 1, 2, \ldots, m$$

For linear function approximation, the internal parameters w are represented as a weights vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_m) \in \mathbb{R}^m$. Linear function approximation is based on the assumption of a Gaussian distribution for $y|x$ with mean

$$\sum_{j=1}^{m} \phi_j(x) \cdot w_j$$

and constant variance σ^2, i.e.,

$$\mathbb{P}[y|x] = f(x; \boldsymbol{w})(y) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(y - \sum_{j=1}^{m} \phi_j(x) \cdot w_j)^2}{2\sigma^2}}$$

So, the cross-entropy loss function (ignoring constant terms associated with σ^2) for a given set of data points $[x_i, y_i | 1 \leq i \leq n]$ is defined as:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n} \cdot \sum_{i=1}^{n} (\sum_{j=1}^{m} \phi_j(x_i) \cdot w_j - y_i)^2$$

Note that this loss function is identical to the mean-squared-error of the linear (in $\boldsymbol{w}$) predictions $\sum_{j=1}^{m} \phi_j(x_i) \cdot w_j$ relative to the response values y_i associated with the predictor values x_i, over all $1 \leq i \leq n$.

If we include L^2 regularization (with λ as the regularization coefficient), then the regularized loss function is:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n} (\sum_{i=1}^{n} (\sum_{j=1}^{m} \phi_j(x_i) \cdot w_j - y_i)^2) + \frac{1}{2} \cdot \lambda \cdot \sum_{j=1}^{m} w_j^2$$

The gradient of $\mathcal{L}(\boldsymbol{w})$ with respect to $\boldsymbol{w}$ works out to:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \frac{1}{n} \cdot (\sum_{i=1}^{n} \boldsymbol{\phi}(x_i) \cdot (\boldsymbol{\phi}(x_i) \cdot \boldsymbol{w} - y_i)) + \lambda \cdot \boldsymbol{w}$$

where

$$\boldsymbol{\phi} : \mathcal{X} \to \mathbb{R}^m$$

is defined as:

$$\boldsymbol{\phi}(x) = (\phi_1(x), \phi_2(x), \ldots, \phi_m(x)) \text{ for all } x \in \mathcal{X}$$

We can solve for $\boldsymbol{w}^*$ by incremental estimation using gradient descent (change in $\boldsymbol{w}$ proportional to the gradient estimate of $\mathcal{L}(\boldsymbol{w})$ with respect to $\boldsymbol{w}$). If the (x_t, y_t) data at time t is:

$$[(x_{t,i}, y_{t,i}) | 1 \leq i \leq n_t]$$

,

then the gradient estimate $\mathcal{G}_{(x_t,y_t)}(\boldsymbol{w}_t)$ at time t is given by:

$$\mathcal{G}_{(x_t,y_t)}(\boldsymbol{w}_t) = \frac{1}{n} \cdot (\sum_{i=1}^{n_t} \boldsymbol{\phi}(x_{t,i}) \cdot (\boldsymbol{\phi}(x_{t,i}) \cdot \boldsymbol{w}_t - y_{t,i})) + \lambda \cdot \boldsymbol{w}_t$$

which can be interpreted as the mean (over the data in iteration t) of the feature vectors $\boldsymbol{\phi}(x_{t,i})$ weghted by the (scalar) linear prediction errors $\boldsymbol{\phi}(x_{t,i}) \cdot \boldsymbol{w} - y_{t,i}$ (plus regularization term $\lambda \cdot \boldsymbol{w}$).

Then, the update to the weights vector $\boldsymbol{w}$ is given by:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha_t \cdot \mathcal{G}_{(x_t,y_t)}(\boldsymbol{w}_t)$$

where α_t is the learning rate for the gradient descent at time t. To facilitate numerical convergence, we require α_t to be an appropriate function of time t. There are a number of numerical algorithms to achieve the appropriate time-trajectory of α_t. We shall go with one such numerical algorithm - ADAM, which we shall use not just for linear function approximation but also for the deep neural network function approximation. Before we write code for linear function approximation, we need to write some helper code to implement the ADAM gradient descent algorithm.

We create an `@dataclass Weights` to represent and update the weights (i.e., internal parameters) of a function approximation. The `Weights` dataclass has 5 attributes: `adam_gradient` that captures the ADAM parameters, including the base learning rate and the decay parameters, `time` that represents how many times the weights have been updated, `weights` that represents the weight parameters of the function approximation as a numpy array (1-D array for linear function approximation and 2-D array for each layer of deep neural network function approximation), and the two ADAM cache parameters. The `@staticmethod create` serves as a factory method to create a new instance of the `Weights` dataclass. The `update` method of this `Weights` dataclass produces an updated instance of the `Weights` dataclass that represents the updated weight parameters together with the incremented `time` and the updated ADAM cache parameters. We will follow a programming design pattern wherein we don't update anything in-place - rather, we create a new object with updated values (using the `dataclasses.replace` function). This ensures we don't get unexpected/undesirable updates in-place, which are typically the cause of bugs in numerical code. Finally, we write the `within` method which will be required to implement the `within` method in the linear function approximation class as well as in the deep neural network function approximation class.

```
SMALL_NUM = 1e-6
from dataclasses import replace

@dataclass(frozen=True)
```

```python
class AdamGradient:
    learning_rate: float
    decay1: float
    decay2: float

    @staticmethod
    def default_settings() -> AdamGradient:
        return AdamGradient(
            learning_rate=0.001,
            decay1=0.9,
            decay2=0.999
        )


@dataclass(frozen=True)
class Weights:
    adam_gradient: AdamGradient
    time: int
    weights: np.ndarray
    adam_cache1: np.ndarray
    adam_cache2: np.ndarray

    @staticmethod
    def create(
        adam_gradient: AdamGradient = AdamGradient.default_settings(),
        weights: np.ndarray,
        adam_cache1: Optional[np.ndarray] = None,
        adam_cache2: Optional[np.ndarray] = None
    ) -> Weights:
        return Weights(
            adam_gradient=adam_gradient,
            time=0,
            weights=weights,
            adam_cache1=np.zeros_like(
                weights
            ) if adam_cache1 is None else adam_cache1,
            adam_cache2=np.zeros_like(
                weights
            ) if adam_cache2 is None else adam_cache2
        )

    def update(self, gradient: np.ndarray) -> Weights:
        time: int = self.time + 1
        new_adam_cache1: np.ndarray = self.adam_gradient.decay1 * \
            self.adam_cache1 + (1 - self.adam_gradient.decay1) * gradient
        new_adam_cache2: np.ndarray = self.adam_gradient.decay2 * \
```

```python
            self.adam_cache2 + (1 - self.adam_gradient.decay2) * gradient ** 2
        corrected_m: np.ndarray = new_adam_cache1 / \
            (1 - self.adam_gradient.decay1 ** time)
        corrected_v: np.ndarray = new_adam_cache2 / \
            (1 - self.adam_gradient.decay2 ** time)

        new_weights: np.ndarray = self.weights - \
            self.adam_gradient.learning_rate * corrected_m / \
            (np.sqrt(corrected_v) + SMALL_NUM)

        return replace(
            self,
            time=time,
            weights=new_weights,
            adam_cache1=new_adam_cache1,
            adam_cache2=new_adam_cache2,
        )

    def within(self, other: Weights[X], tolerance: float) -> bool:
        return np.all(np.abs(self.weights - other.weights) <= tolerance).item()
```

Given this `Weights` dataclass, we are now ready to write the `@dataclass LinearFunctionApprox` for linear function approximation that implements the abstract base class `FunctionApprox`. It has attributes `feature_functions` that represents $\phi_j : \mathcal{X} \rightarrow \mathbb{R}$ for all $j = 1, 2, \ldots, m$, `regularization_coeff` that represents the regularization coefficient λ, `weights` which is an instance of the `Weights` class we wrote above, and `direct_solve` (which we will explain shortly. The `@staticmethod create` serves as a factory method to create a new instance of `LinearFunctionApprox`. The method `get_feature_values` takes as input an `x_values_seq: Iterable[X]` (representing a data set of $x \in \mathcal{X}$), and produces as output the corresponding feature vectors $\boldsymbol{\phi}(x) \in \mathbb{R}^m$ for each of the input x. The feature vectors are output in the form of a 2-D numpy array, with each feature vector $\boldsymbol{\phi}(x)$ (for each x in the input sequence) appearing as a row in the output 2-D numpy array (the number of rows in this numpy array is the length of the input `x_values_seq` and the number of columns is the number of feature functions). Note that often we want to include a bias term in our linear function approximation, in which case we need to prepend the sequence of feature functions we want to provide as input with an artificial feature function `lambda _: 1.` to represent the constant feature with value 1. This will ensure we have a bias weight in addition to each of the weights that serve as coefficients to the (non-artificial) feature functions. The method `evaluate` (an `@abstractmethod` in `FunctionApprox`) calculates the prediction $\mathbb{E}_M[y|x]$ for each input x as: $\boldsymbol{\phi}(x) \cdot \boldsymbol{w} = \sum_{j=1}^m \phi_j(x) \cdot w_i$. The method `regularized_loss_gradient` performs the calculation $\mathcal{G}_{(x_t,y_t)}(\boldsymbol{w}_t)$ shown above. The method `update` (`@abstractmethod` in `FunctionApprox`) invokes `regularized_loss_gradient` and returns a new instance of `LinearFunctionApprox` that contains the updated weights, along with the ADAM cache updates (invoking the `update` method of the `Weights` class to ensure there are no in-place updates).

```
from dataclasses import replace

@dataclass(frozen=True)
class LinearFunctionApprox(FunctionApprox[X]):

    feature_functions: Sequence[Callable[[X], float]]
    regularization_coeff: float
    weights: Weights
    direct_solve: bool

    @staticmethod
    def create(
        feature_functions: Sequence[Callable[[X], float]],
        adam_gradient: AdamGradient = AdamGradient.default_settings(),
        regularization_coeff: float = 0.,
        weights: Optional[Weights] = None,
        direct_solve: bool = True
    ) -> LinearFunctionApprox[X]:
        return LinearFunctionApprox(
            feature_functions=feature_functions,
            regularization_coeff=regularization_coeff,
            weights=Weights.create(
                adam_gradient=adam_gradient,
                weights=np.zeros(len(feature_functions))
            ) if weights is None else weights,
            direct_solve=direct_solve
        )

    def get_feature_values(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return np.array(
            [[f(x) for f in self.feature_functions] for x in x_values_seq]
        )

    def evaluate(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return np.dot(
            self.get_feature_values(x_values_seq),
            self.weights.weights
        )

    def within(self, other: FunctionApprox[X], tolerance: float) -> bool:
        if isinstance(other, LinearFunctionApprox):
            return self.weights.within(other.weights, tolerance)
        else:
```

```python
            return False

    def regularized_loss_gradient(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]]
    ) -> np.ndarray:
        x_vals, y_vals = zip(*xy_vals_seq)
        feature_vals: np.ndarray = self.get_feature_values(x_vals)
        diff: np.ndarray = np.dot(feature_vals, self.weights.weights) \
            - np.array(y_vals)
        return np.dot(feature_vals.T, diff) / len(diff) \
            + self.regularization_coeff * self.weights.weights

    def update(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]]
    ) -> LinearFunctionApprox[X]:
        gradient: np.ndarray = self.regularized_loss_gradient(xy_vals_seq)
        new_weights: np.ndarray = self.weights.update(gradient)
        return replace(self, weights=new_weights)
```

We also require the `within` method, that simply delegates to the `within` method of the `Weights` class.

```python
    def within(self, other: FunctionApprox[X], tolerance: float) -> bool:
        if isinstance(other, LinearFunctionApprox):
            return self.weights.within(other.weights, tolerance)
        else:
            return False
```

The only method that remains to be written now is the `solve` method. Note that for linear function approximation, we can directly solve for $\boldsymbol{w}^*$ if the number of feature functions m is not too large. If the entire provided data is $[(x_i, y_i)|1 \leq i \leq n]$, then the gradient estimate based on this data can be set to 0 to solve for $\boldsymbol{w}^*$, i.e.,

$$\frac{1}{n} \cdot (\sum_{i=1}^{n} \boldsymbol{\phi}(x_i) \cdot (\boldsymbol{\phi}(x_i) \cdot \boldsymbol{w}^* - y_i)) + \lambda \cdot \boldsymbol{w}^* = 0$$

We denote $\boldsymbol{\Phi}$ as the n rows $\times$ m columns matrix defined as $\boldsymbol{\Phi}_{i,j} = \phi_j(x_i)$ and the column vector $\boldsymbol{Y} \in \mathbb{R}^n$ defined as $\boldsymbol{Y}_i = y_i$. Then we can write the above equation as:

$$\frac{1}{n} \cdot \boldsymbol{\Phi}^T \cdot (\boldsymbol{\Phi} \cdot \boldsymbol{w}^* - \boldsymbol{Y}) + \lambda \cdot \boldsymbol{w}^* = 0$$

$$\Rightarrow (\boldsymbol{\Phi}^T \cdot \boldsymbol{\Phi} + n\lambda \cdot \boldsymbol{I_m}) \cdot \boldsymbol{w}^* = \boldsymbol{\Phi}^T \cdot \boldsymbol{Y}$$

$$\Rightarrow \boldsymbol{w}^* = (\boldsymbol{\Phi}^T \cdot \boldsymbol{\Phi} + n\lambda \cdot \boldsymbol{I_m})^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{Y}$$

where $\bm{I_m}$ is the $m \times m$ identity matrix. Note that this requires inversion of the $m \times m$ matrix $\bm{\Phi}^T \cdot \bm{\Phi} + n\lambda \cdot \bm{I_m}$ and so, this direct solution for $\bm{w}^*$ requires that m not be too large.

On the other hand, if the number of feature functions m is too large, then this direct-solve method is infeasible. In that case, we solve for $\bm{w}^*$ by repeatedly calling update. The attribute direct_solve: bool' inLinearFunctionApproxspecifies whether to perform a direct solution (linear algebra calculations shown above) or to perform a sequence of iterative (incremental)updatesto $\bm{w}$ using gradient descent. The code below for the methodsolve' does exactly this:

```
import itertools
import rl.iterate import iterate

    def solve(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        error_tolerance: Optional[float] = None
    ) -> LinearFunctionApprox[X]:
        if self.direct_solve:
            x_vals, y_vals = zip(*xy_vals_seq)
            feature_vals: np.ndarray = self.get_feature_values(x_vals)
            feature_vals_T: np.ndarray = feature_vals.T
            left: np.ndarray = np.dot(feature_vals_T, feature_vals) \
                + feature_vals.shape[0] * self.regularization_coeff * \
                np.eye(len(self.weights.weights))
            right: np.ndarray = np.dot(feature_vals_T, y_vals)
            ret = replace(
                self,
                weights=Weights.create(
                    adam_gradient=self.weights.adam_gradient,
                    weights=np.dot(np.linalg.inv(left), right)
                )
            )
        else:
            tol: float = 1e-6 if error_tolerance is None else error_tolerance

            def done(
                a: LinearFunctionApprox[X],
                b: LinearFunctionApprox[X],
                tol: float = tol
            ) -> bool:
                return a.within(b, tol)

            ret = iterate.converged(
                self.iterate_updates(itertools.repeat(xy_vals_seq)),
```

```python
            done=done
        )

    return ret
```

The above code is in the file rl/function_approx.py. Note that the `FunctionApprox` class implemented in this file has another `@abstractclass` called `representational_gradient` which we can ignore for now, and will be covered later when we get to Module III of the book. For now, assume that this `@abstractclass representational_gradient` doesn't exist in the `FunctionApprox` class as it is an unnecessary distraction for the purpose of the contents in Modules I and II.

Neural Network Function Approximation

Now we generalize the linear function approximation to accommodate non-linear functions with a simple deep neural network, specifically a feed-forward fully-connected neural network. We work with the same notation $\boldsymbol{\phi}(\cdot) = (\phi_1(\cdot), \phi_2(\cdot), \ldots, \phi_m(\cdot))$ for feature functions that we covered for the case of linear function approximation. Assume we have L hidden layers in the neural network. Layers numbered $l = 0, 1, \ldots, L-1$ carry the hidden layer neurons and layer $l = L$ carries the output layer neurons.

A couple of things to note about our notation for vectors and matrices when performing linear algebra operations: Vectors will be treated as column vectors (including gradient of a scalar with respect to a vector). When our notation expresses gradient of a vector of dimension m with respect to a vector of dimension n, we treat it as a Jacobian matrix with m rows and n columns. We use the notation $dim(\boldsymbol{V})$ to refer to the dimension of a vector $\boldsymbol{V}$.

We denote the input to layer l as vector $\boldsymbol{I_l}$ and the output to layer l as vector $\boldsymbol{O_l}$, for all $l = 0, 1, \ldots, L$. Denoting the predictor variable as $x \in \mathcal{X}$, the response variable as $y \in \mathbb{R}$, and the neural network as model M to predict the expected value of y conditional on x, we have:

$$\boldsymbol{I_0} = \boldsymbol{\phi}(x) \in \mathbb{R}^m \text{ and } \boldsymbol{O_L} = \mathbb{E}_M[y|x] \text{ and } \boldsymbol{I_{l+1}} = \boldsymbol{O_l} \text{ for all } l = 0, 1, \ldots, L-1 \tag{4.1}$$

We denote the parameters for layer l as the matrix $\boldsymbol{w_l}$ with $dim(\boldsymbol{O_l})$ rows and $dim(\boldsymbol{I_l})$ columns. Note that the number of neurons in layer l is equal to $dim(\boldsymbol{O_l})$. Since we are restricting ourselves to scalar y, $dim(\boldsymbol{O_L}) = 1$ and so, the number of neurons in the output layer is 1.

The neurons in layer l define a linear transformation from layer input $\boldsymbol{I_l}$ to a variable we denote as $\boldsymbol{S_l}$. Therefore,

$$\boldsymbol{S_l} = \boldsymbol{w_l} \cdot \boldsymbol{I_l} \text{ for all } l = 0, 1, \ldots, L \tag{4.2}$$

We denote the activation function of layer l as $g_l : \mathbb{R} \rightarrow \mathbb{R}$ for all $l = 0, 1, \ldots, L$. The activation function g_l applies point-wise on each dimension of vector $\boldsymbol{S_l}$, so we take notational liberty with g_l by writing:

$$\boldsymbol{O_l} = g_l(\boldsymbol{S_l}) \text{ for all } l = 0, 1, \ldots, L \tag{4.3}$$

Equations (4.1), (4.2) and (4.3) together define the calculation of the neural network prediction $\boldsymbol{O_L}$ (associated with the response variable y), given the predictor variable x. This calculation is known as *forward-propagation* and will define the `evaluate` method of the deep neural network function approximation class we shall soon write.

Our goal is to derive an expression for the cross-entropy loss gradient $\nabla_{\boldsymbol{w_l}} \mathcal{L}$ for all $l = 0, 1, \ldots, L$. For ease of understanding, our following exposition will be expressed in terms of the cross-entropy loss function for a single predictor variable input $x \in \mathcal{X}$ and it's associated single response variable $y \in \mathbb{R}$ (the code will generalize appropriately to the cross-entropy loss function for a given set of data points $[x_i, y_i | 1 \leq i \leq n]$).

We can reduce this problem of calculating the cross-entropy loss gradient to the problem of calculating $\boldsymbol{P_l} = \nabla_{\boldsymbol{S_l}} \mathcal{L}$ for all $l = 0, 1, \ldots, L$, as revealed by the following chain-rule calculation:

$$\nabla_{\boldsymbol{w_l}} \mathcal{L} = (\nabla_{\boldsymbol{S_l}} \mathcal{L})^T \cdot \nabla_{\boldsymbol{w_l}} \boldsymbol{S_l} = \boldsymbol{P_l}^T \cdot \nabla_{\boldsymbol{w_l}} \boldsymbol{S_l} = \boldsymbol{P_l} \cdot \boldsymbol{I_l}^T = \boldsymbol{P_l} \otimes \boldsymbol{I_l} \text{ for all } l = 0, 1, \ldots L$$

where the symbol $\otimes$ refers to the outer-product of two vectors resulting in a matrix. Note that the outer-product of the $dim(\boldsymbol{O_l})$ size vector $\boldsymbol{P_l}$ and the $dim(\boldsymbol{I_l})$ size vector $\boldsymbol{I_l}$ gives a matrix of size $dim(\boldsymbol{O_l}) \times dim(\boldsymbol{I_l})$.

If we include L^2 regularization (with λ_l as the regularization coefficient for layer l), then:

$$\nabla_{\boldsymbol{w_l}} \mathcal{L} = \boldsymbol{P_l} \otimes \boldsymbol{I_l} + \lambda_l \cdot \boldsymbol{w_l} \text{ for all } l = 0, 1, \ldots, L \tag{4.4}$$

Here's the summary of our notation:

Notation	**Description**
$\boldsymbol{I_l}$	Vector Input to layer l for all $l = 0, 1, \ldots, L$
$\boldsymbol{O_l}$	Vector Output of layer l for all $l = 0, 1, \ldots, L$
$\boldsymbol{\phi}(x)$	Feature Vector for predictor variable x
y	Response variable associated with predictor variable x
$\boldsymbol{w_l}$	Matrix of Parameters for layer l for all $l = 0, 1, \ldots, L$
$g_l(\cdot)$	Activation function for layer l for $l = 0, 1, \ldots, L$
$\boldsymbol{S_l}$	$\boldsymbol{S_l} = \boldsymbol{w_l} \cdot \boldsymbol{I_l}, \boldsymbol{O_l} = g_l(\boldsymbol{S_l})$ for all $l = 0, 1, \ldots L$
$\boldsymbol{P_l}$	$\boldsymbol{P_l} = \nabla_{\boldsymbol{S_l}} \mathcal{L}$ for all $l = 0, 1, \ldots, L$
λ_l	Regularization coefficient for layer l for all $l = 0, 1, \ldots, L$

Now that we have reduced the loss gradient calculation to calculation of $\boldsymbol{P_l}$, we spend the rest of this section deriving the analytical calculation of $\boldsymbol{P_l}$. The following theorem tells us that $\boldsymbol{P_l}$ has a recursive formulation that forms the core of the *back-propagation* algorithm for a feed-forward fully-connected deep neural network.

Theorem 4.0.1. *For all $l = 0, 1, \ldots, L-1$,*

$$\boldsymbol{P_l} = (\boldsymbol{w_{l+1}^T} \cdot \boldsymbol{P_{l+1}}) \circ g_l'(\boldsymbol{S_l})$$

where the symbol $\cdot$ represents vector-matrix multiplication and the symbol $\circ$ represents the Hadamard Product, i.e., point-wise multiplication of two vectors of the same dimension.

Proof. We start by applying the chain rule on $\boldsymbol{P_l}$.

$$\boldsymbol{P_l} = \nabla_{\boldsymbol{S_l}} \mathcal{L} = (\nabla_{\boldsymbol{S_l}} \boldsymbol{S_{l+1}})^T \cdot \nabla_{\boldsymbol{S_{l+1}}} \mathcal{L} = (\nabla_{\boldsymbol{S_l}} \boldsymbol{S_{l+1}})^T \cdot \boldsymbol{P_{l+1}} \tag{4.5}$$

Next, note that:

$$\boldsymbol{S_{l+1}} = \boldsymbol{w_{l+1}} \cdot g_l(\boldsymbol{S_l})$$

Therefore,

$$\nabla_{\boldsymbol{S_l}} \boldsymbol{S_{l+1}} = \boldsymbol{w_{l+1}} \cdot \boldsymbol{Diagonal}(g_l'(\boldsymbol{S_l}))$$

Substituting this in Equation (4.5) yields:

$$\boldsymbol{P_l} = (\boldsymbol{w_{l+1}} \cdot \boldsymbol{Diagonal}(g_l'(\boldsymbol{S_l})))^T \cdot \boldsymbol{P_{l+1}} = \boldsymbol{Diagonal}(g_l'(\boldsymbol{S_l})) \cdot \boldsymbol{w_{l+1}^T} \cdot \boldsymbol{P_{l+1}}$$

$$= g_l'(\boldsymbol{S_l}) \circ (\boldsymbol{w_{l+1}^T} \cdot \boldsymbol{P_{l+1}}) = (\boldsymbol{w_{l+1}^T} \cdot \boldsymbol{P_{l+1}}) \circ g_l'(\boldsymbol{S_l})$$

□

Now all we need to do is to calculate $\boldsymbol{P_L} = \nabla_{\boldsymbol{S_L}} \mathcal{L}$ so that we can run this recursive formulation for $\boldsymbol{P_l}$, estimate the loss gradient $\nabla_{\boldsymbol{w_l}} \mathcal{L}$ for any given data (using Equation (4.4)), and perform gradient descent to arrive at $\boldsymbol{w_l^*}$ for all $l = 0, 1, \ldots L$.

Firstly, note that $\boldsymbol{S_L}, \boldsymbol{O_L}, \boldsymbol{P_L}$ are all scalars, so let's just write them as S_L, O_L, P_L respectively (without the bold-facing) to make it explicit in the derivation that they are scalars. Specifically, the gradient

$$\nabla_{\boldsymbol{S_L}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial S_L}$$

To calculate $\frac{\partial \mathcal{L}}{\partial S_L}$, we need to assume a functional form for $\mathbb{P}[y|S_L]$. We work with a fairly generic exponential functional form for the probability distribution function:

$$p(y|\theta, \tau) = h(y, \tau) \cdot e^{\frac{\theta \cdot y - A(\theta)}{d(\tau)}}$$

where θ should be thought of as the "center" parameter (related to the mean) of the probability distribution and τ should be thought of as the "dispersion" parameter (related to the variance) of the distribution. $h(\cdot, \cdot), A(\cdot), d(\cdot)$ are general functions whose specializations define the family of distributions that can be modeled with this fairly generic exponential functional form (note that this structure is adopted from the framework of Generalized Linear Models).

For our neural network function approximation, we assume that τ is a constant, and we set θ to be S_L. So,

$$\mathbb{P}[y|S_L] = p(y|S_L, \tau) = h(y, \tau) \cdot e^{\frac{S_L \cdot y - A(S_L)}{d(\tau)}}$$

We require the scalar prediction of the neural network $O_L = g_L(S_L)$ to be equal to $\mathbb{E}_p[y|S_L]$. So the question is: What function $g_L : \mathbb{R} \rightarrow \mathbb{R}$ (in terms of the functional form of $p(y|S_L, \tau)$) would satisfy the requirement of $O_L = g_L(S_L) = \mathbb{E}_p[y|S_L]$? To answer this question, we first establish the following Lemma:

Lemma 4.0.2.

$$\mathbb{E}_p[y|S_L] = A'(S_L)$$

Proof. Since

$$\int_{-\infty}^{\infty} p(y|S_L, \tau) \cdot dy = 1,$$

the partial derivative of the left-hand-side of the above equation with respect to S_L is zero. In other words,

$$\frac{\partial\{\int_{-\infty}^{\infty} p(y|S_L, \tau) \cdot dy\}}{\partial S_L} = 0$$

Hence,

$$\frac{\partial\{\int_{-\infty}^{\infty} h(y, \tau) \cdot e^{\frac{S_L \cdot y - A(S_L)}{d(\tau)}} \cdot dy\}}{\partial S_L} = 0$$

Taking the partial derivative inside the integral, we get:

$$\int_{-\infty}^{\infty} h(y, \tau) \cdot e^{\frac{S_L \cdot y - A(S_L)}{d(\tau)}} \cdot \frac{y - A'(S_L)}{d(\tau)} \cdot dy = 0$$

$$\Rightarrow \int_{-\infty}^{\infty} p(y|S_L, \tau) \cdot (y - A'(S_L)) \cdot dy = 0$$

$$\Rightarrow \mathbb{E}_p[y|S_L] = A'(S_L)$$

□

So to satisfy $O_L = g_L(S_L) = \mathbb{E}_p[y|S_L]$, we require that

$$O_L = g_L(S_L) = A'(S_L) \tag{4.6}$$

The above equation is important since it tells us that the output layer activation function $g_L(\cdot)$ must be set to be the derivative of the $A(\cdot)$ function. In the theory of generalized linear models, the derivative of the $A(\cdot)$ function serves as the *canonical link function* for a given probability distribution of the response variable conditional on the predictor variable.

Now we are equipped to derive a simple expression for P_L.

Theorem 4.0.3.

$$P_L = \frac{\partial \mathcal{L}}{\partial S_L} = \frac{O_L - y}{d(\tau)}$$

Proof. The Cross-Entropy Loss (Negative Log-Likelihood) for a single training data point (x, y) is given by:

$$\mathcal{L} = -\log\left(h(y, \tau)\right) + \frac{A(S_L) - S_L \cdot y}{d(\tau)}$$

Therefore,

$$P_L = \frac{\partial \mathcal{L}}{\partial S_L} = \frac{A'(S_L) - y}{d(\tau)}$$

But from Equation (4.6), we know that $A'(S_L) = O_L$. Therefore,

$$P_L = \frac{\partial \mathcal{L}}{\partial S_L} = \frac{O_L - y}{d(\tau)}$$

□

At each iteration of gradient descent, we require an estimate of the loss gradient up to a constant factor. So we can ignore the constant $d(\tau)$ and simply say that $P_L = O_L - y$ (up to a constant factor). This is a rather convenient estimate of P_L for a given data point (x, y) since it represents the neural network prediction error for that data point. When presented with a sequence of data points $[(x_{t,i}, y_{t,i}) | 1 \leq i \leq n_t]$ in iteration t, we simply average the prediction errors across these presented data points. Then, beginning with this estimate of P_L, we can use the recursive formulation of $\boldsymbol{P_l}$ (Theorem 4.0.1) to calculate the gradient of the loss function (Equation (4.4)) with respect to all the parameters of the neural network (this is known as the back-propagation algorithm for a fully-connected feed-forward deep neural network).

Here are some common specializations of the functional form for the conditional probability distribution $\mathbb{P}[y|S_L]$, along with the corresponding activation function g_L of the output layer:

- Normal distribution $y \sim \mathcal{N}(\mu, \sigma^2)$: $S_L = \mu, \tau = \sigma, h(y, \tau) = \frac{e^{\frac{-y^2}{2\tau^2}}}{\sqrt{2\pi\tau}}, A(S_L) = \frac{S_L^2}{2}, d(\tau) = \tau^2$. $g_L(S_L) = \mathbb{E}[y|S_L] = S_L$, hence the output layer activation function g_L is the identity function. This means that the linear function approximation of the previous section is exactly the same as a neural network with 0 hidden layers (just the output layer) and with the output layer activation function equal to the identity function.
- Bernoulli distribution for binary-valued y, parameterized by p: $S_L = \log\left(\frac{p}{1-p}\right), \tau = 1, h(y, \tau) = 1, d(\tau) = 1, A(S_L) = \log\left(1 + e^{S_L}\right)$. $g_L(S_L) = \mathbb{E}[y|S_L] = \frac{1}{1+e^{-S_L}}$, hence the output layer activation function g_L is the logistic function. This generalizes to softmax g_L when we generalize this framework to multivariate y, which in turn enables us to classify inputs x into a finite set of categories represented by y as one-hot-encodings.
- Poisson distribution for y parameterized by λ: $S_L = \log \lambda, \tau = 1, d(\tau) = 1, h(y, \tau) = \frac{1}{y!}, A(S_L) = e^{S_L}$. $g_L(S_L) = \mathbb{E}[y|S_L] = e^{S_L}$, hence the output layer activation function g_L is the exponential function.

Now we are ready to write a class for function approximation with the deep neural network framework described above. We shall assume that the activation functions $g_l(\cdot)$ are identical for all $l = 0, 1, \ldots, L-1$ (known as the hidden layers activation function) and the activation function $g_L(\cdot)$ will be known as the output layer activation function. Note that often we want to include a bias term in the linear transformations of the layers. To include a bias term in layer 0, just like in the case of `LinearFuncApprox`, we prepend the sequence of feature functions we want to provide as input with an artificial feature function `lambda _: 1.` to represent the constant feature with value 1. This will ensure we have a bias weight in layer 0 in addition to each of the weights (in layer 0) that serve as coefficients to the (non-artificial) feature functions. Moreover, we allow the specification of a `bias` boolean variable to enable a bias term in each if the layers $l = 1, 2, \ldots L$.

Before we develop the code for forward-propagation and back-propagation, we write a `@dataclass` to hold the configuration of a deep neural network (number of `neurons` in the layers, the `bias` boolean variable, hidden layers activation function and output layer activation function).

```
@dataclass(frozen=True)
class DNNSpec:
    neurons: Sequence[int]
    bias: bool
    hidden_activation: Callable[[np.ndarray], np.ndarray]
    hidden_activation_deriv: Callable[[np.ndarray], np.ndarray]
    output_activation: Callable[[np.ndarray], np.ndarray]
    output_activation_deriv: Callable[[np.ndarray], np.ndarray]
```

`neurons` is a sequence of length L specifying $dim(O_0), dim(O_1), \ldots, dim(O_{L-1})$ (note $dim(O_L)$ doesn't need to be specified since we know $dim(O_L) = 1$). If `bias` is set to be `True`, then $dim(I_l) = dim(O_{l-1}) + 1$ for all $l = 1, 2, \ldots L$ and so in the code below, when `bias` is `True`, we'll need to prepend the matrix representing I_l with a vector consisting of all 1s (to incorporate the bias term). Note that along with specifying the hidden and output layers activation functions $g_l(\cdot)$ defined as $g_l(\boldsymbol{S_l}) = \boldsymbol{O_l}$, we also specify the hidden layers activation function derivative (`hidden_activation_deriv`) and the output layer activation function derivative (`output_activation_deriv`) in the form of functions $h_l(\cdot)$ defined as $h_l(g(\boldsymbol{S_l})) = h_l(\boldsymbol{O_l}) = g_l'(\boldsymbol{S_l})$ (as we know, this derivative is required in the back-propagation calculation). We shall soon see that in the code, $h_l(\cdot)$ is a more convenient specification than the direct specification of $g_l'(\cdot)$.

Now we write the `@dataclass DNNApprox` that implements the abstract base class `FunctionApprox`. It has attributes:

- `feature_functions` that represents $\phi_j : \mathcal{X} \to \mathbb{R}$ for all $j = 1, 2, \ldots, m$
- `dnn_spec` that specifies the neural network configuration (instance of `DNNSpec`)
- `regularization_coeff` that represents the common regularization coefficient λ for the weights across all layers

- `weights` which is a sequence of `Weights` objects (to represent and update the weights of all layers).

The method `get_feature_values` is identical to the case of `LinearFunctionApprox` producing a matrix with number of rows equal to the number of x values in it's input `x_values_seq: Iterable[X]` and number of columns equal to the number of specified `feature_functions`.

The method `forward_propagation` implements the forward-propagation calculation that was covered earlier (combining Equations (4.1) (potentially adjusted for the bias term, as mentioned above), (4.2) and (4.3)). `forward_propagation` takes as input the same data type as the input of `get_feature_values` (`x_values_seq: Iterable[X]`) and returns a list with $L+2$ numpy arrays. The last element of the returned list is a 1-D numpy array representing the final output of the neural network: $O_L = \mathbb{E}_M[y|x]$ for each of the x values in the input `x_values_seq`. The remaining $L+1$ elements in the returned list are each 2-D numpy arrays, consisting of $\boldsymbol{I_l}$ for all $l = 0, 1, \ldots L$ (for each of the x values provided as input in `x_values_seq`).

The method `evaluate` (an `@abstractmethod` in `FunctionApprox`) returns the last element ($O_L = \mathbb{E}_M[y|x]$) of the list returned by `forward_propagation`.

The method `backward_propagation` is the most important method of `DNNApprox` and deserves a detailed explanation. `backward_propagation` takes two inputs:

1. `fwd_prop: Sequence[np.ndarray]` which represents the output of the `forward_propagation` method, i.e., a sequence of $L+2$ numpy arrays with the first $L+1$ elements as the 2-D numpy arrays representing the inputs to layers $l = 0, 1, \ldots L$ (for each of an `Iterable` of x-values provided as input to the neural network), and the last element as the 1-D numpy array representing the output $O_L = \mathbb{E}_M[y|x]$ of the neural network (for each of the `Iterable` of x-values provided as input to the neural network).
2. `objective_derivative_output: Callable[[np.ndarray], np.ndarray]`, a function representing the partial derivative of an arbitrary objective function Obj with respect to O_L (the output of the neural network), i.e., a function representing $\frac{\partial Obj}{\partial O_L}$.

The output of `backward_propagation` is an estimate of $\nabla_{\boldsymbol{w_l}} Obj$ for all $l = 0, 1, \ldots, L$. If we generalize the objective function from the cross-entropy loss function $\mathcal{L}$ to an arbitrary objective function Obj and generalize $\boldsymbol{P_l}$ to be $\nabla_{\boldsymbol{S_l}} Obj$ (from $\nabla_{\boldsymbol{S_l}} \mathcal{L}$), then the output of `backward_propagation` would be equal to $\boldsymbol{P_l} \otimes \boldsymbol{I_l}$ (i.e., without the regularization term) for all $l = 0, 1, \ldots L$.

The first step in `backward_propagation` is to extract $\boldsymbol{I_l}$ as the first $L+1$ elements of `fwd_prop` and store in the variable `layer_inputs`. The next step is to construct $\boldsymbol{P_L} = \frac{\partial Obj}{\partial S_L} = \frac{\partial Obj}{\partial O_L} \cdot \frac{\partial O_L}{\partial S_L}$ (variable `deriv` is initialized to $\boldsymbol{P_L}$) which is the product of the input `objective_derivative_output` (evaluated at each value in the 1-D numpy array `fwd_prop[-1]`, representing the outputs O_L of the neural network) and the attribute `self.output_activation_deriv` representing $\frac{\partial O_L}{\partial S_L}$ as a function of O_L (also evaluated at each value in the 1-D numpy array fwd_prop[-1]). The variable deriv represents $\boldsymbol{P_l} = \nabla_{\boldsymbol{S_l}} Obj$, evaluated for each of the values made available by fwd_prop (note that deriv is updated in each iteration of the loop reflecting Theorem 4.0.1: $\boldsymbol{P_l} = (\boldsymbol{w}_{l+1}^T \cdot \boldsymbol{P_{l+1}}) \circ g_l'(\boldsymbol{S_l})$). Note also that the returned list back_prop is populated with the result of Equation (4.4): $\nabla_{\boldsymbol{w_l}} \mathcal{L} = \boldsymbol{P_l} \otimes \boldsymbol{I_l}$.

The method regularized_loss_gradient takes as input an Iterable of (x, y) pairs, invokes the forward_propagation method (to be passed as input to backward_propagation), prepares a function obj_deriv_output (to be passed as the other input to backward_propagation), invokes backward_propagation and simply adds on the regularization term $\lambda \cdot \boldsymbol{w_l}$ to the output of backward_propagation.

The method update (@abstractmethod in FunctionApprox) invokes regularized_loss_gradient and returns a new instance of DNNApprox that contains the updated weights, along with the ADAM cache updates (invoking the update method of the Weights class to ensure there are no in-place updates). Finally, the method solve (@abstractmethod in FunctionApprox) utilizes the method iterate_updates (inherited from FunctionApprox) along with the method within to perform a best-fit of the weights that minimizes the cross-entropy loss function (basically, a series of incremental updates based on gradient descent).

```
from dataclasses import replace
import itertools
import rl.iterate import iterate

@dataclass(frozen=True)
class DNNApprox(FunctionApprox[X]):

    feature_functions: Sequence[Callable[[X], float]]
    dnn_spec: DNNSpec
    regularization_coeff: float
    weights: Sequence[Weights]

    @staticmethod
    def create(
        feature_functions: Sequence[Callable[[X], float]],
        dnn_spec: DNNSpec,
        adam_gradient: AdamGradient = AdamGradient.default_settings(),
        regularization_coeff: float = 0.,
        weights: Optional[Sequence[Weights]] = None
    ) -> DNNApprox[X]:
        if weights is None:
            inputs: Sequence[int] = [len(feature_functions)] + \
                [n + (1 if dnn_spec.bias else 0)
                 for i, n in enumerate(dnn_spec.neurons)]
            outputs: Sequence[int] = dnn_spec.neurons + [1]
            wts = [Weights.create(
                weights=np.random.randn(output, inp) / np.sqrt(inp),
```

```python
                adam_gradient=adam_gradient
            ) for inp, output in zip(inputs, outputs)]
        else:
            wts = weights

        return DNNApprox(
            feature_functions=feature_functions,
            dnn_spec=dnn_spec,
            regularization_coeff=regularization_coeff,
            weights=wts
        )

    def get_feature_values(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return np.array(
            [[f(x) for f in self.feature_functions] for x in x_values_seq]
        )

    def forward_propagation(
        self,
        x_values_seq: Iterable[X]
    ) -> Sequence[np.ndarray]:
        inp: np.ndarray = self.get_feature_values(x_values_seq)
        ret: List[np.ndarray] = [inp]
        for w in self.weights[:-1]:
            out: np.ndarray = self.dnn_spec.hidden_activation(
                np.dot(inp, w.weights.T)
            )
            if self.dnn_spec.bias:
                inp = np.insert(out, 0, 1., axis=1)
            else:
                inp = out
            ret.append(inp)
        ret.append(
            self.dnn_spec.output_activation(
                np.dot(inp, self.weights[-1].weights.T)
            )[:, 0]
        )
        return ret

    def evaluate(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return self.forward_propagation(x_values_seq)[-1]

    def within(self, other: FunctionApprox[X], tolerance: float) -> bool:
        if isinstance(other, DNNApprox):
            return all(w1.within(w2, tolerance)
                       for w1, w2 in zip(self.weights, other.weights))
```

```python
        else:
            return False

    def backward_propagation(
        self,
        fwd_prop: Sequence[np.ndarray],
        objective_derivative_output: Callable[[np.ndarray], np.ndarray]
    ) -> Sequence[np.ndarray]:
        layer_inputs: Sequence[np.ndarray] = fwd_prop[:-1]
        deriv: np.ndarray = objective_derivative_output(fwd_prop[-1]) * \
            self.dnn_spec.output_activation_deriv(fwd_prop[-1])
        deriv = deriv.reshape(1, -1)
        back_prop: List[np.ndarray] = [np.dot(deriv, layer_inputs[-1]) /
                                       deriv.shape[1]]
        # L is the number of hidden layers, n is the number of points
        # layer l deriv represents dObj/dS_l where S_l = I_l . weights_l
        # (S_l is the result of applying layer l without the activation func)
        for i in reversed(range(len(self.weights) - 1)):
            # deriv_l is a 2-D array of dimension |O_l| x n
            # The recursive formulation of deriv is as follows:
            # deriv_{l-1} = (weights_l^T inner deriv_l) haddamard g'(S_{l-1}),
            # which is ((|I_l| x |O_l|) inner (|O_l| x n)) haddamard
            # (|I_l| x n), which is (|I_l| x n) = (|O_{l-1}| x n)
            # Note: g'(S_{l-1}) is expressed as hidden layer activation
            # derivative as a function of O_{l-1} (=I_l).
            deriv = np.dot(self.weights[i + 1].weights.T, deriv) * \
                self.dnn_spec.hidden_activation_deriv(layer_inputs[i + 1].T)
            # If self.dnn_spec.bias is True, then I_l = O_{l-1} + 1, in which
            # case # the first row of the calculated deriv is removed to yield
            # a 2-D array of dimension |O_{l-1}| x n.
            if self.dnn_spec.bias:
                deriv = deriv[1:]
            # layer l gradient is deriv_l inner layer_inputs_l, which is
            # of dimension (|O_l| x n) inner (n x (|I_l|) = |O_l| x |I_l|
            back_prop.append(np.dot(deriv, layer_inputs[i]) / deriv.shape[1])
        return back_prop[::-1]

    def regularized_loss_gradient(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]]
    ) -> Sequence[np.ndarray]:
        x_vals, y_vals = zip(*xy_vals_seq)
        fwd_prop: Sequence[np.ndarray] = self.forward_propagation(x_vals)

        def obj_deriv_output(out: np.ndarray) -> np.ndarray:
            return (out - np.array(y_vals)) / \
```

```python
                self.dnn_spec.output_activation_deriv(out)

        return [x + self.regularization_coeff * self.weights[i].weights
                for i, x in enumerate(self.backward_propagation(
                    fwd_prop=fwd_prop,
                    objective_derivative_output=obj_deriv_output
                ))]

    def update(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]]
    ) -> DNNApprox[X]:
        return replace(
            self,
            weights=[w.update(g) for w, g in zip(
                self.weights,
                self.regularized_loss_gradient(xy_vals_seq)
            )]
        )

    def solve(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        error_tolerance: Optional[float] = None
    ) -> DNNApprox[X]:
        tol: float = 1e-6 if error_tolerance is None else error_tolerance

        def done(
            a: DNNApprox[X],
            b: DNNApprox[X],
            tol: float = tol
        ) -> bool:
            return a.within(b, tol)

        return iterate.converged(
            self.iterate_updates(itertools.repeat(xy_vals_seq)),
            done=done
        )
```

All of the above code is in the file rl/function_approx.py.

Let us now write some code to create function approximations with `LinearFunctionApprox` and `DNNApprox`, given a stream of data from a simple data model - one that has some noise around a linear function. Here's some code to create an Iterator of (x, y) pairs (where $x = (x_1, x_2, x_3)$) for the data model:

$$y = 2 + 10x_1 + 4x_2 - 6x_3 + \mathcal{N}(0, 0.3)$$

```python
def example_model_data_generator() -> Iterator[Tuple[Triple, float]]:

    coeffs: Aug_Triple = (2., 10., 4., -6.)
    d = norm(loc=0., scale=0.3)

    while True:
        pt: np.ndarray = np.random.randn(3)
        x_val: Triple = (pt[0], pt[1], pt[2])
        y_val: float = coeffs[0] + np.dot(coeffs[1:], pt) + \
            d.rvs(size=1)[0]
        yield (x_val, y_val)
```

Next we wrap this in an Iterator that returns a certain number of (x, y) pairs upon each request for data points.

```python
def data_seq_generator(
    data_generator: Iterator[Tuple[Triple, float]],
    num_pts: int
) -> Iterator[DataSeq]:
    while True:
        pts: DataSeq = list(islice(data_generator, num_pts))
        yield pts
```

Now let's write a function to create a `LinearFunctionApprox`.

```python
def feature_functions():
    return [lambda _: 1., lambda x: x[0], lambda x: x[1], lambda x: x[2]]

def adam_gradient():
    return AdamGradient(
        learning_rate=0.1,
        decay1=0.9,
        decay2=0.999
    )

def get_linear_model() -> LinearFunctionApprox[Triple]:
    ffs = feature_functions()
    ag = adam_gradient()
    return LinearFunctionApprox.create(
         feature_functions=ffs,
         adam_gradient=ag,
         regularization_coeff=0.,
         direct_solve=True
    )
```

Likewise, let's write a function to create a `DNNApprox` with 1 hidden layer with 2 neurons and a little bit of regularization since this deep neural network is somewhat over-parameterized to fit the data generated from the linear data model with noise.

```
def get_dnn_model() -> DNNApprox[Triple]:
    ffs = feature_functions()
    ag = adam_gradient()

    def relu(arg: np.ndarray) -> np.ndarray:
        return np.vectorize(lambda x: x if x > 0. else 0.)(arg)

    def relu_deriv(res: np.ndarray) -> np.ndarray:
        return np.vectorize(lambda x: 1. if x > 0. else 0.)(res)

    def identity(arg: np.ndarray) -> np.ndarray:
        return arg

    def identity_deriv(res: np.ndarray) -> np.ndarray:
        return np.ones_like(res)

    ds = DNNSpec(
        neurons=[2],
        bias=True,
        hidden_activation=relu,
        hidden_activation_deriv=relu_deriv,
        output_activation=identity,
        output_activation_deriv=identity_deriv
    )

    return DNNApprox.create(
        feature_functions=ffs,
        dnn_spec=ds,
        adam_gradient=ag,
        regularization_coeff=0.05
    )
```

Now let's write some code to do a `direct_solve` with the `LinearFunctionApprox` based on the data from the data model we have set up.

```
training_num_pts: int = 1000
test_num_pts: int = 10000
training_iterations: int = 300
data_gen: Iterator[Tuple[Triple, float]] = example_model_data_generator()
training_data_gen: Iterator[DataSeq] = data_seq_generator(
    data_gen,
    training_num_pts
)
test_data: DataSeq = list(islice(data_gen, test_num_pts))
```

```python
direct_solve_lfa: LinearFunctionApprox[Triple] = \
    get_linear_model().solve(next(training_data_gen))
direct_solve_rmse: float = direct_solve_lfa.rmse(test_data)
```

Running the above code, we see that the Root-Mean-Squared-Error (`direct_solve_rmse`) is indeed 0.3, matching the standard deviation of the noise in the linear data model (which is used above to generate the training data as well as the test data).

Now let us perform stochastic gradient descent with instances of `LinearFunctionApprox` and `DNNApprox` and examine the Root-Mean-Squared-Errors on the two function approximations as a function of number of iterations in the gradient descent.

```python
linear_model_rmse_seq: Sequence[float] = \
    [lfa.rmse(test_data) for lfa in islice(
        get_linear_model().iterate_updates(training_data_gen),
        training_iterations
    )]

dnn_model_rmse_seq: Sequence[float] = \
    [dfa.rmse(test_data) for dfa in islice(
        get_dnn_model().iterate_updates(training_data_gen),
        training_iterations
    )]
```

The plot of `linear_model_rmse_seq` and `dnn_model_rmse_seq` is shown in Figure 4.1.

Tabular as a form of `FunctionApprox`

Now we consider a simple case where we have a fixed and finite set of x-values $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, and any data set of (x, y) made available to us needs to have it's x-values from within this finite set $\mathcal{X}$. The prediction $\mathbb{E}[y|x]$ for each $x \in \mathcal{X}$ needs to be calculated only from the y-values associated with this x within the data set of (x, y) pairs. In other words, the y-values in the data associated with other x should not influence the prediction for x. Since we'd like the prediction for x to be $\mathbb{E}[y|x]$, it would make sense for the prediction for a given x to be the average of all the y-values associated with x within the data set of (x, y) pairs seen so far. This simple case is refered to as *Tabular* because we can store all $x \in \mathcal{X}$ together with their corresponding predictions $\mathbb{E}[y|x]$ in a finite data structure (loosely refered to as a "table").

So the calculations for Tabular prediction of $\mathbb{E}[y|x]$ is particularly straightforward. What is interesting though is the fact that Tabular prediction actually fits the interface of `FunctionApprox` in terms of implementing:

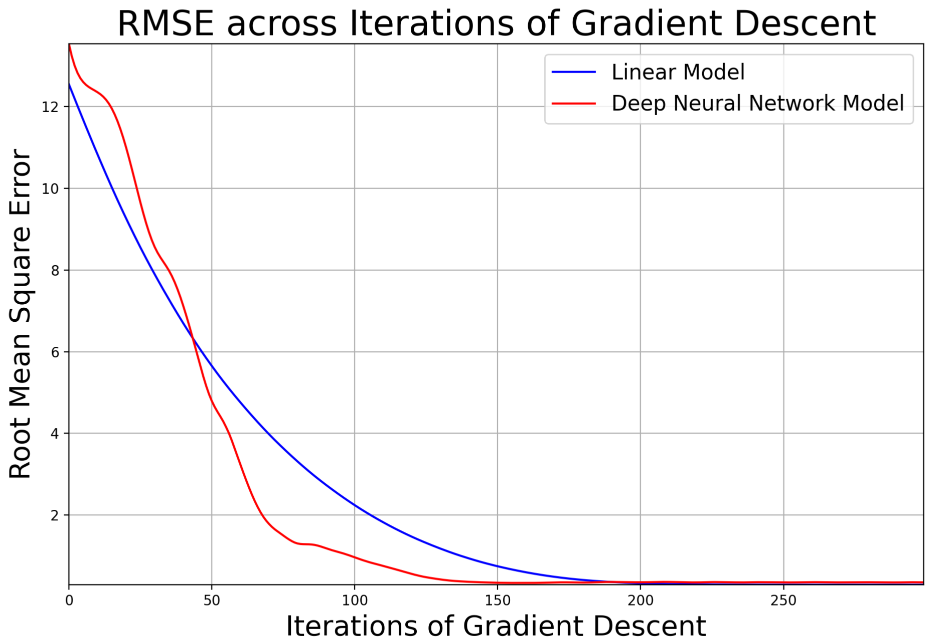

Figure 4.1.: SGD Convergence

- the `solve` method, that would simply take the average of all the y-values associated with each x in the given data set, and store those averages in a dictionary data structure.
- the `update` method, that would update the current averages in the dictionary data structure, based on the new data set of (x, y) pairs that is provided.
- the `evaluate` method, that would simply look up the dictionary data structure for the y-value averages associated with each x-value provided as input.

This view of Tabular prediction as a special case of `FunctionApprox` also permits us to cast the tabular algorithms of Dynamic Programming and Reinforcement Learning as special cases of the function approximation versions of the algorithms (using the `Tabular` class we develop below).

So now let us write the code for `@dataclass Tabular` as an implementation of the abstract base class `FunctionApprox`. The attributes of `@dataclass Tabular` are:

- `values_map` which is a dictionary mapping each x-value to the average of the y-values associated with x that have been seen so far in the data.
- `counts_map` which is a dictionary mapping each x-value to the count of y-values associated with x that have been seen so far in the data. We need to track the count of y-values associated with each x because this enables us to update `values_map` appropriately upon seeing a new y-value associated a given x.

- `count_to_weight_func` which defines a function from number of y-values seen so far (associated with a given x) to the weight assigned to the most recent y. This enables us to do a weighted average of the y-values seen so far, controlling the emphasis to be placed on more recent y-values relative to previously seen y-values (associated with a given x).

The `evaluate`, `update`, `solve` and `within` methods are now self-explanatory.

```
from dataclasses import field

@dataclass(frozen=True)
class Tabular(FunctionApprox[X]):

    values_map: Mapping[X, float] = field(default_factory=lambda: {})
    counts_map: Mapping[X, int] = field(default_factory=lambda: {})
    count_to_weight_func: Callable[[int], float] = \
        field(default_factory=lambda: lambda n: 1. / n)

    def evaluate(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return np.array([self.values_map.get(x, 0.) for x in x_values_seq])

    def update(self, xy_vals_seq: Iterable[Tuple[X, float]]) -> Tabular[X]:
        values_map: Dict[X, float] = dict(self.values_map)
        counts_map: Dict[X, int] = dict(self.counts_map)

        for x, y in xy_vals_seq:
            counts_map[x] = counts_map.get(x, 0) + 1
            weight: float = self.count_to_weight_func(counts_map.get(x, 0))
            values_map[x] = weight * y + (1 - weight) * values_map.get(x, 0.)

        return replace(
            self,
            values_map=values_map,
            counts_map=counts_map
        )

    def solve(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        error_tolerance: Optional[float] = None
    ) -> Tabular[X]:
        values_map: Dict[X, float] = {}
        counts_map: Dict[X, int] = {}
        for x, y in xy_vals_seq:
            counts_map[x] = counts_map.get(x, 0) + 1
            weight: float = self.count_to_weight_func(counts_map.get(x, 0))
```

```python
            values_map[x] = weight * y + (1 - weight) * values_map.get(x, 0.)
        return replace(
            self,
            values_map=values_map,
            counts_map=counts_map
        )

    def within(self, other: FunctionApprox[X], tolerance: float) -> bool:
        if isinstance(other, Tabular):
            return\
                all(abs(self.values_map[s] - other.values_map[s]) <= tolerance
                    for s in self.values_map)

        return False
```

Note that in the tabular Dynamic Programming algorithms, the set of finite states take the role of $\mathcal{X}$ and the Value Function for a given state $x = s$ takes the role of the "predicted" y-value associated with x. We also note that in the Dynamic Programming algorithms, in each iteration of sweeping through all the states, the Value Function for a state $x = s$ is set to the current y value (not the average of all y-values seen so far). The current y-value is simply the right-hand-side of the Bellmen Equation corresponding to the Dynamic Programmming algorithm. Consequently, for tabular Dynamic Programming, we'd need to set `count_to_weight_func` to be the function `lambda _: 1` (this is because a weight of 1 for the current y-value sets `values_map[x]` equal to the current y-value). Later, when we get to Reinforcement Learning algorithms, we will be averaging all the Returns observed for a given state. If we choose to do a plain average (equal importance for all y-values see so far, associated with a given x), then we'd need to set `count_to_weights_func` to be the function `lambda n: 1. / n`. Note that this also means tabular RL is a special case of RL with linear function approximation by setting a feature function $\phi_i(\cdot)$ for each x_i as: $\phi_i(x) = 1$ for $x = x_i$ and $\phi_(x) = 0$ for each $x \neq x_i$ (i.e., $\phi_i(x)$ is the indicator function for x_i, and the $\mathbf{\Phi}$ matrix is the identity matrix). This also means that the `count_to_weights_func` plays the role of the learning rate function (as a function of the number of iterations in stochastic gradient descent). Please do bear this in mind when we get to tabular RL.

Again, we want to emphasize that tabular algorithms are just a special case of algorithms with function approximation. However, we give special coverage in this book to tabular algorithms because they help us conceptualize the core concepts in a simple (tabular) setting without the distraction of some of the details and complications in the apparatus of function approximation.

Now we are ready to write algorithms for Approximate Dynamic Programming (ADP).

Approximate Policy Evaluation

The first ADP algorithm we cover is Approximate Policy Evaluation, i.e., evaluating the Value Function for a Markov Reward Process (MRP). Approximate Policy Evaluation is fundamentally the same as Tabular Policy Evaluation in terms of repeatedly applying the Bellman Policy Operator $\boldsymbol{B}^\pi$ on the Value Function $V : \mathcal{N} \rightarrow \mathbb{R}$. However, unlike Tabular Policy Evaluation algorithm, the Value Function $V(\cdot)$ is set up and updated as an instance of `FunctionApprox` rather than as a table of values for the non-terminal states. This is because unlike Tabular Policy Evaluation which operates on an instance of a `FiniteMarkovRewardProcess`, Approximate Policy Evaluation algorithm operates on an instance of `MarkovRewardProcess`. So we do not have an enumeration of states of the MRP and we do not have the transition probabilities of the MRP. This is typical in many real-world problems where the state space is either very large or is continuous-valued, and the transitions could be too many or could be continuous-valued transitions. So, here's what we do to overcome these challenges:

- We specify a sampling probability distribution of non-terminal states (argument `non_terminal_states_distribution` in the code below) from which we shall sample a specified number (`num_state_samples` in the code below) of non-terminal states, and construct a list of those sampled non-terminal states (`nt_states` in the code below) in each iteration.
- We sample pairs of (next state s', reward r) from a given non-terminal state s, and calculate the expectation $\mathbb{E}[r + \gamma \cdot V(s')]$ by averaging $r + \gamma \cdot V(s')$ across the sampled pairs. Note that the method `expectation` of a `Distribution` object performs a sampled expectation. $V(s')$ is obtained from the function approximation instance of `FunctionApprox` that is being updated in each iteration.
- The sampled list of non-terminal states s comprise our x-values and the associated sampled expectations described above comprise our y-values. This list of (x, y) pairs are used to update the approximation of the Value Function in each iteration (producing a new instance of `FunctionApprox` using it's `update` method).

The entire code is shown below. `evaluate_mrp` produces an `Iterator` on `FunctionApprox` instances, and the code that calls `evaluate_mrp` can decide when/how to terminate the iterations of Approximate Policy Evaluation.

```
from rl.iterate import iterate

def evaluate_mrp(
    mrp: MarkovRewardProcess[S],
    gamma: float,
    approx_0: FunctionApprox[S],
    non_terminal_states_distribution: Distribution[S],
    num_state_samples: int
) -> Iterator[FunctionApprox[S]]:
```

```
    def update(v: FunctionApprox[S]) -> FunctionApprox[S]:
        nt_states: Sequence[S] = non_terminal_states_distribution.sample_n(
            num_state_samples
        )

        def return_(s_r: Tuple[S, float]) -> float:
            s, r = s_r
            return r + gamma * v.evaluate([s]).item()

        return v.update(
            [(s, mrp.transition_reward(s).expectation(return_))
             for s in nt_states]
        )

    return iterate(update, approx_0)
```

Approximate Value Iteration

Now that we've understood and coded Approximate Policy Evaluation (to solve the Prediction problem), we can extend the same concepts to Approximate Value Iteration (to solve the Control problem). The code below in `value_iteration` is almost the same as the code above in `evaluate_mrp`, except that instead of a `MarkovRewardProcess` at each time step, here we have a `MarkovDecisionProcess` at each time step, and instead of the Bellman Policy Operator update, here we have the Bellman Optimality Operator update. Therefore, in the Value Function update, we maximize the Q-value function (over all actions a) for each non-terminal state s. Also, similar to `evaluate_mrp`, `value_iteration` produces an `Iterator` on `FunctionApprox` instances, and the code that calls `value_iteration` can decide when/how to terminate the iterations of Approximate Value Iteration.

```
from rl.iterate import iterate

def value_iteration(
    mdp: MarkovDecisionProcess[S, A],
    gamma: float,
    approx_0: FunctionApprox[S],
    non_terminal_states_distribution: Distribution[S],
    num_state_samples: int
) -> Iterator[FunctionApprox[S]]:

    def update(v: FunctionApprox[S]) -> FunctionApprox[S]:
        nt_states: Sequence[S] = non_terminal_states_distribution.sample_n(
            num_state_samples
```

```
        )

        def return_(s_r: Tuple[S, float]) -> float:
            s, r = s_r
            return r + gamma * v.evaluate([s]).item()

        return v.update(
            [(s, max(mdp.step(s, a).expectation(return_,)
                     for a in mdp.actions(s)))
             for s in nt_states]
        )

    return iterate(update, approx_0)
```

Finite-Horizon Approximate Policy Evaluation

Next, we move on to Approximate Policy Evaluation in a finite-horizon setting, meaning we will perform Approximate Policy Evaluation with a backward induction algorithm, much like how we did backward induction for finite-horizon Tabular Policy Evaluation. We will of course make the same types of adaptations from Tabular to Approximate as we did in the functions `evaluate_mrp` and `value_iteration` above.

In the `backward_evaluate` code below, the input argument `mrp_f0_mu_triples` is a list of triples, with each triple corresponding to each non-terminal time step in the finite horizon. Each triple consists of:

- An instance of `MarkovRewardProceess` - note that each time step has it's own instance of `MarkovRewardProcess` representation of transitions from non-terminal states s in a time step t to the (state s', reward r) pairs in the next time step $t+1$ (variable `mrp` in the code below).
- An instance of `FunctionApprox` to capture the approximate Value Function for the time step (variable `approx0` in the code below, representing the initial `FunctionApprox` instance).
- A sampling probability distribution of non-terminal states in the time step (variable `mu` in the code below).

The backward induction code below should be pretty self-explanatory. Note that in backward induction, we don't invoke the `update` method of `FunctionApprox` like we did in the non-finite-horizon cases - here we invoke the `solve` method which internally performs a series of `updates` on the `FunctionApprox` for a given time step (until we converge to within a specified level of `error_tolerance`). In the non-finite-horizon cases, it was okay to simply do a single `update` in each iteration because we revisit the same set of states in further iterations. Here, once we converge to an acceptable `FunctionApprox` (using `solve`) for a specific time step, we won't be performing any more updates to the Value Function for that time step (since we move on to the next time step, in reverse).

backward_evaluate returns an Iterator over FunctionApprox objects, from time step 0 to the horizon time step. We should point out that in the code below, we've taken special care to handle terminal states that occur before the end of the horizon.

```
MRP_FuncApprox_Distribution = \
    Tuple[MarkovRewardProcess[S], FunctionApprox[S], Distribution[S]]

def backward_evaluate(
    mrp_f0_mu_triples: Sequence[MRP_FuncApprox_Distribution[S]],
    gamma: float,
    num_state_samples: int,
    error_tolerance: float
) -> Iterator[FunctionApprox[S]]:
    v: List[FunctionApprox[S]] = []

    num_steps: int = len(mrp_f0_mu_triples)

    for i, (mrp, approx0, mu) in enumerate(reversed(mrp_f0_mu_triples)):

        def return_(s_r: Tuple[S, float], i=i) -> float:
            s1, r = s_r
            return r + gamma * (v[i-1].evaluate([s1]).item() if i > 0 and not
                                mrp_f0_mu_triples[num_steps - i][0].is_terminal(s1)
                                else 0.)

        v.append(
            approx0.solve(
                [(s, mrp.transition_reward(s).expectation(return_))
                 for s in mu.sample_n(num_state_samples)
                 if not mrp.is_terminal(s)],
                error_tolerance
            )
        )

    return reversed(v)
```

Finite-Horizon Approximate Value Iteration

Now that we've understood and coded finite-horizon Approximate Policy Evaluation (to solve the finite-horizon Prediction problem), we can extend the same concepts to finite-horizon Approximate Value Iteration (to solve the finite-horizon Control problem). The code below in back_opt_vf_and_policy is almost the same as the code above in backward_evaluate, except that instead of a MarkovRewardProcess, here we have a MarkovDecisionProcess. For each non-terminal time step, we maximize the Q-value function (over all actions a) for each non-terminal state s. back_opt_vf_and_policy returns an Iterator over pairs of FunctionApprox and Policy objects (representing the Optimal Value Function and the Optimal Policy respectively), from time step 0 to the horizon time step.

```
from rl.distribution import Constant
from operator import itemgetter

MDP_FuncApproxV_Distribution = Tuple[
    MarkovDecisionProcess[S, A],
    FunctionApprox[S],
    Distribution[S]
]

def back_opt_vf_and_policy(
    mdp_f0_mu_triples: Sequence[MDP_FuncApproxV_Distribution[S, A]],
    gamma: float,
    num_state_samples: int,
    error_tolerance: float
) -> Iterator[Tuple[FunctionApprox[S], Policy[S, A]]]:
    vp: List[Tuple[FunctionApprox[S], Policy[S, A]]] = []

    num_steps: int = len(mdp_f0_mu_triples)

    for i, (mdp, approx0, mu) in enumerate(reversed(mdp_f0_mu_triples)):

        def return_(s_r: Tuple[S, float], i=i) -> float:
            s1, r = s_r
            return r + gamma * (vp[i-1][0].evaluate([s1]).item() if i > 0 and not
                                mdp_f0_mu_triples[num_steps - i][0].is_terminal(s1)
                                else 0.)

        this_v = approx0.solve(
            [(s, max(mdp.step(s, a).expectation(return_)
                     for a in mdp.actions(s)))
             for s in mu.sample_n(num_state_samples)
             if not mdp.is_terminal(s)],
            error_tolerance
        )

        class ThisPolicy(Policy[S, A]):
            def act(self, state: S) -> Constant[A]:
                return Constant(max(
                    ((mdp.step(state, a).expectation(return_), a)
                     for a in mdp.actions(state)),
                    key=itemgetter(0)
```

```
                )[1])

        vp.append((this_v, ThisPolicy()))

    return reversed(vp)
```

Finite-Horizon Approximate Q-Value Iteration

The above code for Finite-Horizon Approximate Value Iteration extends the Finite-Horizon Backward Induction Value Iteration algorithm of Chapter 3 by treating the Value Function as a function approximation instead of an exact tabular representation. However, there is an alternative (and arguably simpler and more effective) way to solve the Finite-Horizon Control problem - we can perform backward induction on the optimal Action-Value (Q-value) function instead of the optimal (State-)Value Function. The key advantage of working with the optimal Action Value function is that it has all the information necessary to extract the optimal State-Value function and the optimal Policy (since we just need to perform a max / arg max over all the actions for any non-terminal state). This contrasts with the case of working with the optimal State-Value function which requires us to also avail of the transition probabilities, rewards and discount factor in order to extract the optimal policy. Performing backward induction on the optimal Q-value function means that knowledge of the optimal Q-value function for a given time step t immediately gives us the optimal State-Value function and the optimal policy for the same time step t. This contrasts with performing backward induction on the optimal State-Value function - knowledge of the optimal State-Value function for a given time step t cannot give us the optimal policy for the same time step t (for that, we need the optimal State-Value function for time step $t+1$ and furthermore, we also need the t to $t+1$ state/reward transition probabilities).

So now we develop an algorithm that works with a function approximation for the Q-Value function and steps back in time similar to the backward induction we had performed earlier for the (State-)Value function. The code below in `back_opt_qvf` is quite similar to the code above in `back_opt_vf_and_policy`. The key difference is that the `FunctionApprox` in the input to the function needs to be set up as a `FunctionApprox[Tuple[S, A]]` instead of `FunctionApprox[S]` to reflect the fact that we are approximating $Q_t^* : \mathcal{N}_t \times \mathcal{A}_t \rightarrow \mathbb{R}$ for all time steps t in the finite horizon. For each non-terminal time step, we express the Q-value function (for a set of sample non-terminal states s and for all actions a) in terms of the Q-value function approximation of the next time step. This is essentially the MDP Action-Value Function Bellman Optimality Equation for the finite-horizon case (adapted to function approximation). `back_opt_qvf` returns an Iterator over `FunctionApprox[Tuple[S, A]]` (representing the Optimal Q-Value Function), from time step 0 to the horizon time step. We can then obtain V_t^* (Optimal State-Value Function) and π_t^* for each t by simply performing a max / arg max over all actions $a \in \mathcal{A}_t$ of $Q_t^*(s,a)$ for any $s \in \mathcal{N}_t$.

```python
MDP_FuncApproxQ_Distribution = Tuple[
    MarkovDecisionProcess[S, A],
    FunctionApprox[Tuple[S, A]],
    Distribution[S]
]

def back_opt_qvf(
    mdp_f0_mu_triples: Sequence[MDP_FuncApproxQ_Distribution[S, A]],
    gamma: float,
    num_state_samples: int,
    error_tolerance: float
) -> Iterator[FunctionApprox[Tuple[S, A]]]:
    horizon: int = len(mdp_f0_mu_triples)
    qvf: List[FunctionApprox[Tuple[S, A]]] = []

    num_steps: int = len(mdp_f0_mu_triples)

    for i, (mdp, approx0, mu) in enumerate(reversed(mdp_f0_mu_triples)):

        def return_(s_r: Tuple[S, float], i=i) -> float:
            s1, r = s_r
            return r + gamma * (
                max(qvf[i-1].evaluate([(s1, a)]).item()
                    for a in mdp_f0_mu_triples[horizon - i][0].actions(s1))
                if i > 0 and
                not mdp_f0_mu_triples[num_steps - i][0].is_terminal(s1)
                else 0.
            )

        this_qvf = approx0.solve(
            [((s, a), mdp.step(s, a).expectation(return_))
             for s in mu.sample_n(num_state_samples)
             if not mdp.is_terminal(s) for a in mdp.actions(s)],
            error_tolerance
        )

        qvf.append(this_qvf)

    return reversed(qvf)
```

We should also point out here that working with the optimal Q-value function (rather than the optimal State-Value function) in the context of ADP prepares us nicely for RL because RL algorithms typically work with the optimal Q-value function instead of the optimal State-Value function.

All of the above code for Approximate Dynamic Programming (ADP) algorithms is in the file rl/approximate_dynamic_programming.py. We encourage you to create instances of `MarkovRewardProcess` and `MarkovDecisionProcess` (including finite-horizon instances) and play with the above ADP code with different choices of function approximations, non-terminal state sampling distributions, and number of samples. A simple but valuable exercise is to reproduce the tabular versions of these algorithms by using the `Tabular` implementation of `FunctionApprox` (note: the `count_to_weights_func` would need to be lambda _: 1.) in the above ADP functions.

How to Construct the Non-Terminal States Distribution

Each of the above ADP algorithms takes as input probability distribution(s) of non-terminal states. You may be wondering how one constructs the probability distribution of non-terminal states so you can feed it as input to any of these ADP algorithm. There is no simple, crisp answer to this. But we will provide some general pointers in this section on how to construct the probability distribution of non-terminal states.

Let us start with Approximate Policy Evaluation and Approximate Value Iteration algorithms. They require as input the probability distribution of non-terminal states. For Approximate Value Iteration algorithm, a natural choice would be evaluate the Markov Decision Process (MDP) with a uniform policy (equal probability for each action, from any state) to construct the implied Markov Reward Process (MRP), and then infer the stationary distribution of it's Markov Process, using some special property of the Markov Process (for instance, if it's a finite-states Markov Process, we might be able to perform the matrix calculations we covered in Chapter 1 to calculate the stationary distribution). The stationary distribution would serve as the probability distribution of non-terminal states to be used by the Approximate Value Iteration algorithm. For Approximate Policy Evaluation algorithm, we do the same stationary distribution calculation with the given MRP. If we cannot take advantage of any special properties of the given MDP/MRP, then we can run a simulation with the `simulate` method in `MarkovRewardProcess` (inherited from `MarkovProcess`) and create a `SampledDistribution` of non-terminal states based on the non-terminal states reached by the sampling trace after a sufficiently large (but fixed) number of time steps (this is essentially an estimate of the stationary distribution). If the above choices are infeasible or computationally expensive, then a simple and neutral choice is to use a uniform distribution over the states.

Next, we consider the backward induction ADP algorithms for finite-horizon MDPs/MRPs. Our job here is to infer the distribution of non-terminal states for each time step in the finite horizon. Sometimes you can take advantage of the mathematical structure of the underlying Markov Process to come up with an analytical expression (exact or approximate) for the probability distribution of non-terminal states at any time step for the underlying Markov Process of the MRP/implied-MRP. For instance, if the Markov Process is described by a stochastic differential equation (SDE) and if we are able to solve the SDE, we would know the analytical expression for the probability distribution of non-terminal states. If we cannot take advantage of any such special properties, then we can generate sampling traces by time-incrementally sampling from the state-transition probability distributions of each of the Markov Reward Processes at each time step (if we are solving a Control problem, then we create implied-MRPs by evaluating the given MDPs with a uniform policy). The states reached by these sampling traces at any fixed time step provide a `SampledDistribution` of non-terminal states for that time step. If the above choices are infeasible or computationally expensive, then a simple and neutral choice is to use a uniform distribution over the non-terminal states for each time step.

We will write some code in Chapter 6 to create a `SampledDistribution` of non-terminal states for each time step of a finite-horizon problem by stitching together samples of state transitions at each time step. If you are curious about this now, you can take a peek at the code.

Key Takeaways from this Chapter

- The Function Approximation interface involves two key methods - A) updating the parameters of the Function Approximation based on training data available from each iteration of a data stream, and B) evaluating the expectation of the response variable whose conditional probability distribution is modeled by the Function Approximation. Linear Function Approximation and Deep Neural Network Function Approximation are the two main Function Approximations we've implemented and will be using in the rest of the book.
- Tabular is a special type of Function Approximation, and Tabular RL is a special case of linear function approximation with feature functions as indicator functions for each of the states.
- All the Tabular DP algorithms can be generalized to ADP algorithms replacing tabular Value Function updates with updates to Function Approximation parameters (where the Function Approximation represents the Value Function). Sweep over all states in the tabular case is replaced by sampling states in the ADP case. Expectation calculations in Bellman Operators are handled in ADP as averages of the corresponding calculations over transition samples (versus calculations using explicit transition probabilities in the tabular algorithms).

Part II.

Modeling Financial Applications

5. Utility Theory

Introduction to the Concept of Utility

This chapter marks the beginning of Module II, where we cover a set of financial applications that can be solved with Dynamic Programming or Reinforcement Learning Algorithms. A fundamental feature of many financial applications cast as Stochastic Control problems is that the *Rewards* of the modeled MDP are Utility functions in order to capture the tradeoff between financial returns and risk. So this chapter is dedicated to the topic of *Financial Utility*. We begin with developing an understanding of what *Utility* means from a broad Economic perspective, then zoom into the concept of Utility from a financial/monetary perspective, and finally show how Utility functions can be designed to capture individual preferences of "risk-taking-inclination" when it comes to specific financial applications.

Utility Theory is a vast and important topic in Economics and we won't cover it in detail in this book - rather, we will focus on the aspects of Utility Theory that are relevant for the Financial Applications we cover in this book. But it pays to have some familiarity with the general concept of Utility in Economics. The term *Utility* (in Economics) refers to the abstract concept of an individual's preferences over choices of products or services or activities (or more generally, over choices of certain abstract entities analyzed in Economics). Let's say you are offered 3 options to spend your Saturday afternoon: A) lie down on your couch and listen to music, or B) baby-sit your neighbor's kid and earn some money, or C) play a game of tennis with your friend. We really don't know how to compare these 3 options in a formal/analytical manner. But we tend to be fairly decisive (instinctively) in picking among disparate options of this type. Utility Theory aims to formalize making choices by assigning a real number to each presented choice, and then picking the choice with the highest assigned number. The assigned real number for each choice represents the "value"/"worth" of the choice, noting that the "value"/"worth" is often an implicit/instinctive value that needs to be made explicit. In our example, the numerical value for each choice is not something concrete or precise like number of dollars earned on a choice or the amount of time spent on a choice - rather it is a more abstract notion of an individual's "happiness" or "satisfaction" associated with a choice. In this example, you might say you prefer option A) because you feel lazy today (so, no tennis) and you care more about enjoying some soothing music after a long work-week than earning a few extra bucks through baby-sitting. Thus, you are comparing different attributes like money, relaxation and pleasure. This can get more complicated if your friend is offered these options, and say your friend chooses option C). If you see your friend's choice, you might then instead choose option C) because you perceive the "collective value" (for you and your friend together) to be highest if you both choose option C).

We won't go any further on this topic of abstract Utility, but we hope the above example provides the basic intuition for the broad notion of Utility in Economics as preferences over choices by assigning a numerical value for each choice. Instead, we focus on a narrow notion of *Utility of Money* because money is what we care about when it comes to financial applications. However, Utility of Money is not so straightforward because different people respond to different levels of money in different ways. Moreover, in many financial applications, Utility functions help us determine the treadeoff between financial return and risk, and this involves (challenging) assessments of the likelihood of various outcomes. The next section develops the intuition on these concepts.

A Simple Financial Example

To warm up to the concepts associated with Financial Utility Theory, let's start with a simple financial example. Consider a casino game where your financial gain/loss is based on the outcome of tossing a fair coin (HEAD or TAIL outcomes). Let's say you will be paid \$1000 if the coin shows HEAD on the toss, and let's say you would be required to pay \$500 if the coin shows TAIL on the toss. Now the question is: How much would you be willing to pay upfront to play this game? Your first instinct might be to say: "I'd pay \$250 upfront to play this game because that's my expected payoff, based on the probability of the outcomes" ($250 = 0.5(1000) + 0.5(-500)$). But after you think about it carefully, you might alter your answer to be: "I'd pay a little less than \$250". When pressed for why the fair upfront cost for playing the game should be less than \$250, you might say: "I need to be compensated for taking the risk".

What does the word "risk" mean? It refers to the degree of variation in the outcomes (\$1000 versus -\$500). But then why would you say you need to be compensated for being exposed to this variation in outcomes? If -\$500 makes you unhappy, \$1000 should make you happy, and so, shouldn't we average out the happiness to the tune of \$250? Well, not quite. Let's understand the word "happiness" (or call it "satisfaction") - this is the notion of utility of outcomes. Let's say you did pay \$250 upfront to play the game. Then the coin toss outcome of HEAD is a net gain of \$1000 - \$250 = \$750 and the coin toss outcome of TAIL is a net gain of -\$500 - \$250 = -\$750 (i.e., net loss of \$750). Now let's say the HEAD outcome gain of \$750 gives us a level of "happiness" of 100 units. If the TAIL outcome loss of \$750 would give you a level of "unhappiness" of 100 units, then "happiness" and "unhappiness" levels cancel out, and in that case, it would be fair to pay \$250 upfront to play the game. But it turns out that for most people, the "happiness"/"satisfaction"" levels are asymmetric. If the "happiness" level of \$750 gain is 100 units, then the "unhappiness" level of \$750 loss is typically more than 100 units (let's say for you it's 120 units). This means you will pay an upfront amount X (less than \$250) such that the difference in Utilities of \$1000 and X is exactly the difference in the Utilities of X and -\$500. Let's say this X amounts of \$180. The gap of \$70 represents your compensation for taking the risk, and it really comes down to the asymmetry in your assignment of utility to the outcomes.

Note that the degree of asymmetry of utility ("happiness" versus "unhappiness" for equal gains versus losses) is fairly individualized. Your utility assignment to outcomes might be different from your friend's. Your friend might be more asymmetric in assessing utility of the two outcomes and might assign 100 units of "happiness" for the gain outcome and 150 units of "unhappiness" for the loss outcome. So then your friend would pay an upfront amount X lower than the amount of \$180 you paid upfront to play this game. Let's say the X for your friend works out to \$100, so his compensation for taking the risk is \$250 - \$100 = \$150, significantly more than your \$70 of compensation for taking the same risk.

Thus we see that each individual's asymmetry in utility assignment to different outcomes results in this psychology of "I need to be compensated for taking this risk". We refer to this individualized demand of "compensation for risk" as the attitude of *Risk-Aversion*. It means that individuals have differing degrees of discomfort with taking risk, and they want to be compensated commensurately for taking risk. The amount of compensation they seek is called *Risk-Premium*. The more Risk-Averse an individual is, the more Risk-Premium the individual seeks. In the example above, your friend was more risk-averse than you. Your risk-premium was $X = 70$ and your friend's risk-premium was $X = 150$. But the most important concept that you are learning here is that the root-cause of Risk-Aversion is the asymmetry in the assignment of utility to outcomes of opposite sign and same magnitude. We have introduced this notion of "asymmetry of utility" in a simple, intuitive manner with this example, but we will soon embark on developing the formal theory for this notion, and introduce a simple and elegant mathematical framework for Utility Functions, Risk-Aversion and Risk-Premium.

A quick note before we get into the mathematical framework - you might be thinking that a typical casino would actually charge you a bit more than \$250 upfront for playing the above game (because the casino needs to make a profit, on an expected basis), and people are indeed willing to pay this amount at a typical casino. So what about the risk-aversion we talked about earlier? The crucial point here is that people who play at casinos are looking for entertainment and excitement emanating purely from the psychological aspects of experiencing risk. They are willing to pay money for this entertainment and excitement, and this payment is separate from the cost of pure financial utility that we described above. So if people knew that the true odds of pure-chance games of the type we described above and if people did not care for entertainment and excitement value of risk-taking in these games, focusing purely on financial utility, then what they'd be willing to pay upfront to play such a game will be based on the type of calculations we outlined above (meaning for the example we described, they'd typically pay less than \$250 upfront to play the game).

The Shape of the Utility function

We seek a "valuation formula" for the amount we'd pay upfront to sign-up for situations like the simple example above, where we have uncertain outcomes with varying payoffs for the outcomes. Intuitively, we see that the amount we'd pay:

- Increases one-to-one as the Mean of the outcome increases
- Decreases as the Variance of the outcome (i.e.. Risk) increases
- Decreases as our Personal Risk-Aversion increases

The last two properties above enable us to establish the Risk-Premium. Now let us understand the nature of Utility as a function of financial outcomes. The key is to note that Utility is a non-linear function of financial payoff. We call this non-linear function as the Utility function - it represents the "happiness"/"satisfaction" as a function of money gained/lost. You should think of the concept of Utility in terms of *Utility of Consumption* of money, i.e., what exactly do the financial gains fetch you in your life or business. This is the idea of "value" (utility) derived from consuming the financial gains (or the negative utility of requisite financial recovery from monetary losses). So now let us look at another simple example to illustrate the concept of Utility of Consumption, this time not of consumption of money, but of consumption of cookies (to make the concept vivid and intuitive). Figure 5.1 shows two curves - we refer to the blue curve as the marginal satisfaction (utility) curve and the red curve as the accumulated satisfaction (utility) curve. Marginal Utility refers to the *incremental satisfaction* we gain from an additional unit of consumption and Accumulated Utility refers to the *aggregate satisfaction* obtained from a certain number of units of consumption (in continuous-space, you can think of accumulated utility function as the integral, over consumption, of marginal utility function). In this example, we are consuming (i.e., eating) cookies. The marginal satisfaction curve tells us that the first cookie we eat provides us with 100 units of satisfaction (i.e., utility). The second cookie fetches us 80 units of satisfaction, which is intuitive because you are not as hungry after eating the first cookie compared to before eating the first cookie. Also, the emotions of biting into the first cookie are extra positive because of the novelty of the experience. When you get to your 5th cookie, although you are still enjoying the cookie, you don't enjoy it as nearly as much as the first couple of cookies. The marginal satisfaction curve shows this - the 5th cookie fetches us 30 units of satisfaction, and the 10th cookie fetches us only 10 units of satisfaction. If we'd keep going, we might even find that the marginal satisfaction turns negative (as in, one might feel too full or maybe even feel like throwing up).

So, we see that the marginal utility function is a decreasing function. Hence, accumulated utility function is a concave function. The accumulated utility function is the Utility of Consumption function (call it U) that we've been discussing so far. Let us denote the number of cookies eaten as x, and so the total "satisfaction" (utility) after eating x cookies is refered to as $U(x)$. In our financial examples, x would be financial gains/losses and is typically an uncertain

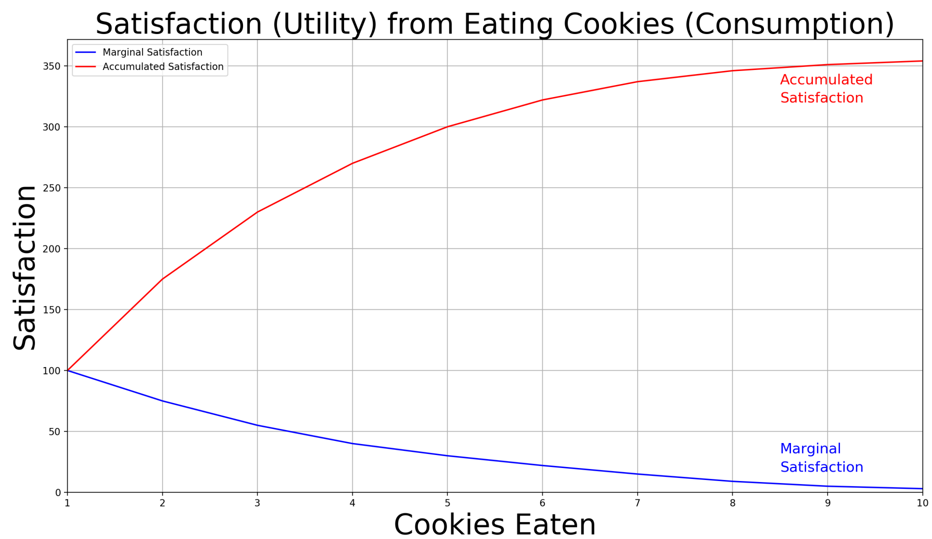

Figure 5.1.: Utility Curve

outcome, i.e., x is a random variable with an associated probability distribution. The extent of asymmetry in utility assignments for gains versus losses that we saw earlier manifests as extent of concavity of the $U(\cdot)$ function (which as we've discussed earlier, determines the extent of Risk-Aversion).

Now let's examine the concave nature of the Utility function for financial outcomes with another illustrative example. Let's say you have to pick between two situations:

- In Situation 1, you have a 10% probability of winning a million dollars (and 90% probability of winning 0).
- In Situation 2, you have a 0.1% probability of winning a billion dollars (and 99.9% probability of winning 0).

The expected winning in Situation 1 is \$10,000 and the expected winning in Situation 2 is \$1,000,000 (i.e., 10 times more than Situation 1). If you analyzed this naively as winning expectation maximization, you'd choose Situation 2. But most people would choose Situation 1. The reason for this is that the Utility of a billion dollars is nowhere close to 1000 times the utility of a million dollars (except for some very wealth people perhaps). In fact, the ratio of Utility of a billion dollars to Utility of a million dollars might be more like 10. So, the choice of Situation 1 over Situation 2 is usually quite clear - it's about Utility expectation maximization. So if the Utility of 0 dollars is 0 units, the Utility of a million dollars is say 1000 units, and the Utility of a billion dollars is say 10000 units (i.e., 10 times that of a million dollars), then we see that the Utility of financial gains is a fairly concave function.

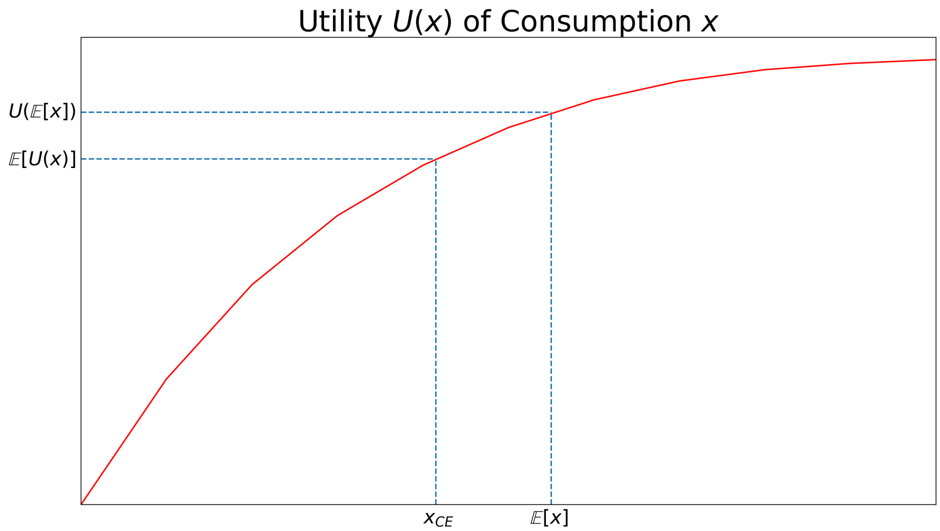

Figure 5.2.: Certainty-Equivalent Value

Calculating the Risk-Premium

Note that the concave nature of the $U(\cdot)$ function implies that:

$$\mathbb{E}[U(x)] < U(\mathbb{E}[x])$$

We define *Certainty-Equivalent Value* x_{CE} as:

$$x_{CE} = U^{-1}(\mathbb{E}[U(x)])$$

Certainty-Equivalent Value represents the certain amount we'd pay to consume an uncertain outcome. This is the amount of $180 you were willing to pay to play the casino game of the previous section.

Figure 5.2 illustrates this concept of Certainty-Equivalent Value in graphical terms. Next, we define Risk-Premium in two different conventions:

- **Absolute Risk-Premium** π_A:

$$\pi_A = \mathbb{E}[x] - x_{CE}$$

- **Relative Risk-Premium** π_R:

$$\pi_R = \frac{\pi_A}{\mathbb{E}[x]} = \frac{\mathbb{E}[x] - x_{CE}}{\mathbb{E}[x]} = 1 - \frac{x_{CE}}{\mathbb{E}[x]}$$

Now we develop mathematical formalism to derive formulas for Risk-Premia π_A and π_R in terms of the extent of Risk-Aversion and the extent of Risk itself. To lighten notation, we refer to $\mathbb{E}[x]$ as $\bar{x}$ and Variance of x as σ_x^2. We write $U(x)$

as the Taylor series expansion around $\bar{x}$ and ignore terms beyond quadratic in the expansion, as follows:

$$U(x) \approx U(\bar{x}) + U'(\bar{x}) \cdot (x - \bar{x}) + \frac{1}{2} U''(\bar{x}) \cdot (x - \bar{x})^2$$

Taking the expectation of $U(x)$ in the above formula, we get:

$$\mathbb{E}[U(x)] \approx U(\bar{x}) + \frac{1}{2} \cdot U''(\bar{x}) \cdot \sigma_x^2$$

Next, we write the Taylor-series expansion for $U(x_{CE})$ around $\bar{x}$ and ignore terms beyond linear in the expansion, as follows:

$$U(x_{CE}) \approx U(\bar{x}) + U'(\bar{x}) \cdot (x_{CE} - \bar{x})$$

Since $\mathbb{E}[U(x)] = U(x_{CE})$ (by definition of x_{CE}), the above two expressions are approximately the same. Hence,

$$U'(\bar{x}) \cdot (x_{CE} - \bar{x}) \approx \frac{1}{2} \cdot U''(\bar{x}) \cdot \sigma_x^2 \tag{5.1}$$

From Equation (5.1), Absolute Risk-Premium

$$\pi_A = \bar{x} - x_{CE} \approx -\frac{1}{2} \cdot \frac{U''(\bar{x})}{U'(\bar{x})} \cdot \sigma_x^2$$

We refer to the function:

$$A(x) = -\frac{U''(x)}{U'(x)}$$

as the *Absolute Risk-Aversion* function. Therefore,

$$\pi_A \approx \frac{1}{2} \cdot A(\bar{x}) \cdot \sigma_x^2$$

In multiplicative uncertainty settings, we focus on the variance $\sigma^2_{\frac{x}{\bar{x}}}$ of $\frac{x}{\bar{x}}$. So in multiplicative settings, we focus on the Relative Risk-Premium:

$$\pi_R = \frac{\pi_A}{\bar{x}} \approx -\frac{1}{2} \cdot \frac{U''(\bar{x}) \cdot \bar{x}}{U'(\bar{x})} \cdot \frac{\sigma_x^2}{\bar{x}^2} = -\frac{1}{2} \cdot \frac{U''(\bar{x}) \cdot \bar{x}}{U'(\bar{x})} \cdot \sigma^2_{\frac{x}{\bar{x}}}$$

We refer to the function

$$R(x) = -\frac{U''(x) \cdot x}{U'(x)}$$

as the *Relative Risk-Aversion* function. Therefore,

$$\pi_R \approx \frac{1}{2} \cdot R(\bar{x}) \cdot \sigma^2_{\frac{x}{\bar{x}}}$$

Now let's take stock of what we've learning here. We've shown that Risk-Premium is proportional to the product of:

- Extent of Risk-Aversion: either $A(\bar{x})$ or $R(\bar{x})$
- Extent of uncertainty of outcome (i.e., Risk): either σ_x^2 or $\sigma_{\frac{x}{\bar{x}}}^2$

We've expressed the extent of Risk-Aversion to be proportional to the negative ratio of:

- Concavity of the Utility function (at $\bar{x}$): $-U''(\bar{x})$
- Slope of the Utility function (at $\bar{x}$): $U'(\bar{x})$

So for typical optimization problems in financial applications, we maximize $\mathbb{E}[U(x)]$ (not $\mathbb{E}[x]$), which in turn amounts to maximization of $x_{CE} = \mathbb{E}[x] - \pi_A$. If we refer to $\mathbb{E}[x]$ as our "Expected Return on Investment" (or simply "Return" for short) and π_A as the "risk-adjustment" due to risk-aversion and uncertainty of outcomes, then x_{CE} can be conceptualized as "risk-adjusted-return". Thus, in financial applications, we seek to maximize risk-adjusted-return x_{CE} rather than just the return $\mathbb{E}[x]$. It pays to emphasize here that the idea of maximizing risk-adjusted-return is essentially the idea of maximizing expected utility, and that the utility function is a representation of an individual's risk-aversion.

Note that Linear Utility function $U(x) = a + bx$ implies *Risk-Neutrality* (i.e., when one doesn't demand any compensation for taking risk). Next, we look at typically-used Utility functions $U(\cdot)$ with:

- Constant Absolute Risk-Aversion (CARA)
- Constant Relative Risk-Aversion (CRRA)

Constant Absolute Risk-Aversion (CARA)

Consider the Utility function $U : \mathbb{R} \to \mathbb{R}$, parameterized by $a \in \mathbb{R}$, defined as:

$$U(x) = \begin{cases} \frac{1-e^{-ax}}{a} & \text{for } a \neq 0 \\ x & \text{for } a = 0 \end{cases}$$

Firstly, note that $U(x)$ is continuous with respect to a for all $x \in \mathbb{R}$ since:

$$\lim_{a \to 0} \frac{1 - e^{-ax}}{a} = x$$

Now let us analyze the function $U(\cdot)$ for any fixed a. We note that for all $a \in \mathbb{R}$:

- $U(0) = 0$
- $U'(x) = e^{-ax} > 0$ for all $x \in \mathbb{R}$
- $U''(x) = -a \cdot e^{-ax}$

This means $U(\cdot)$ is a monotonically increasing function passing through the origin, and it's curvature has the opposite sign as that of a (note: no curvature when $a = 0$).

So now we can calculate the Absolute Risk-Aversion function:

$$A(x) = \frac{-U''(x)}{U'(x)} = a$$

So we see that the Absolute Risk-Aversion function is the constant value a. Consequently, we say that this Utility function corresponds to *Constant Absolute Risk-Aversion* (*CRRA*). The parameter a is refered to as the Coefficient of CRRA. The magnitude of positive a signifies the degree of risk-aversion. $a = 0$ is the case of being Risk-Neutral. Negative values of a mean one is "risk-seeking", i.e., one will pay to take risk (the opposite of risk-aversion) and the magnitude of negative a signifies the degree of risk-seeking.

If the random outcome $x \sim \mathcal{N}(\mu, \sigma^2)$, then using Equation (A.5) from Appendix A, we get:

$$\mathbb{E}[U(x)] = \begin{cases} \frac{1-e^{-a\mu+\frac{a^2\sigma^2}{2}}}{a} & \text{for } a \neq 0 \\ \mu & \text{for } a = 0 \end{cases}$$

$$x_{CE} = \mu - \frac{a\sigma^2}{2}$$

$$\text{Absolute Risk Premium } \pi_A = \mu - x_{CE} = \frac{a\sigma^2}{2}$$

For optimization problems where we need to choose across probability distributions where σ^2 is a function of μ, we seek the distribution that maximizes $x_{CE} = \mu - \frac{a\sigma^2}{2}$. This clearly illustrates the concept of "risk-adjusted-return" because μ serves as the "return" and the risk-adjustment $\frac{a\sigma^2}{2}$ is proportional to the product of risk-aversion a and risk (i.e., variance in outcomes) σ^2.

A Portfolio Application of CARA

Let's say we are given \$1 to invest and hold for a horizon of 1 year. Let's say our portfolio investment choices are:

- A risky asset with Annual Return $\sim \mathcal{N}(\mu, \sigma^2)$, $\mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$.
- A riskless asset with Annual Return $= r \in \mathbb{R}$.

Our task is to determine the allocation π (out of the given \$1) to invest in the risky asset (so, $1 - \pi$ is invested in the riskless asset) so as to maximize the Expected Utility of Consumption of Portfolio Wealth in 1 year. Note that we allow π to be unconstrained, i.e., π can be any real number from $-\infty$ to $+\infty$. So, if $\pi > 0$, we buy the risky asset and if $\pi < 0$, we "short-sell" the risky asset. Likewise, if $1 - \pi > 0$, we lend $1 - \pi$ (and will be paid back $(1 - \pi)(1 + r)$ in 1 year), and if $1 - \pi < 0$, we borrow $1 - \pi$ (and need to pay back $(1 - \pi)(1 + r)$ in 1 year).

Portfolio Wealth W in 1 year is given by:

$$W \sim \mathcal{N}(1 + r + \pi(\mu - r), \pi^2\sigma^2)$$

We assume CARA Utility with $a \neq 0$, so:

$$U(W) = \frac{1 - e^{-aW}}{a}$$

We know that maximizing $\mathbb{E}[U(W)]$ is equivalent to maximizing the Certainty-Equivalent Value of Wealth W, which in this case (using the formula for x_{CE} in the section on CARA) is given by:

$$1 + r + \pi(\mu - r) - \frac{a\pi^2\sigma^2}{2}$$

This is a concave function of π and so, taking it's derivative with respect to π and setting it to 0 gives us the optimal investment fraction in the risky asset (π^*) as follows:

$$\pi^* = \frac{\mu - r}{a\sigma^2}$$

Constant Relative Risk-Aversion (CRRA)

Consider the Utility function $U : \mathbb{R}^+ \to \mathbb{R}$, parameterized by $\gamma \in \mathbb{R}$, defined as:

$$U(x) = \begin{cases} \frac{x^{1-\gamma}-1}{1-\gamma} & \text{for } \gamma \neq 1 \\ \log(x) & \text{for } \gamma = 1 \end{cases}$$

Firstly, note that $U(x)$ is continuous with respect to γ for all $x \in \mathbb{R}^+$ since:

$$\lim_{\gamma \to 1} \frac{x^{1-\gamma} - 1}{1 - \gamma} = \log(x)$$

Now let us analyze the function $U(\cdot)$ for any fixed γ. We note that for all $\gamma \in \mathbb{R}$:

- $U(1) = 0$
- $U'(x) = x^{-\gamma} > 0$ for all $x \in \mathbb{R}^+$
- $U''(x) = -\gamma \cdot x^{-1-\gamma}$

This means $U(\cdot)$ is a monotonically increasing function passing through $(1, 0)$, and it's curvature has the opposite sign as that of γ (note: no curvature when $\gamma = 0$).

So now we can calculate the Relative Risk-Aversion function:

$$R(x) = \frac{-U''(x) \cdot x}{U'(x)} = \gamma$$

So we see that the Relative Risk-Aversion function is the constant value γ. Consequently, we say that this Utility function corresponds to *Constant Relative Risk-Aversion* (*CRRA*). The parameter γ is refered to as the Coefficient of CRRA. The magnitude of positive γ signifies the degree of risk-aversion. $\gamma = 0$ yields the Utility function $U(x) = x - 1$ and is the case of being Risk-Neutral. Negative values of γ mean one is "risk-seeking", i.e., one will pay to take risk (the opposite of risk-aversion) and the magnitude of negative γ signifies the degree of risk-seeking.

If the random outcome x is lognormal, with $\log(x) \sim \mathcal{N}(\mu, \sigma^2)$, then making a substitution $y = \log(x)$, expressing $\mathbb{E}[U(x)]$ as $\mathbb{E}[U(e^y)]$, and using Equation (A.5) in Appendix A, we get:

$$\mathbb{E}[U(x)] = \begin{cases} \frac{e^{\mu(1-\gamma)+\frac{\sigma^2}{2}(1-\gamma)^2}-1}{1-\gamma} & \text{for } \gamma \neq 1 \\ \mu & \text{for } \gamma = 1 \end{cases}$$

$$x_{CE} = e^{\mu + \frac{\sigma^2}{2}(1-\gamma)}$$

$$\text{Relative Risk Premium } \pi_R = 1 - \frac{x_{CE}}{\bar{x}} = 1 - e^{-\frac{\sigma^2\gamma}{2}}$$

For optimization problems where we need to choose across probability distributions where σ^2 is a function of μ, we seek the distribution that maximizes $\log(x_{CE}) = \mu + \frac{\sigma^2}{2}(1-\gamma)$. Just like in the case of CARA, this clearly illustrates the concept of "risk-adjusted-return" because $\mu + \frac{\sigma^2}{2}$ serves as the "return" and the risk-adjustment $\frac{\gamma\sigma^2}{2}$ is proportional to the product of risk-aversion γ and risk (i.e., variance in outcomes) σ^2.

A Portfolio Application of CRRA

This application of CRRA is a special case of Merton's Portfolio Problem that we shall cover in its full generality in Chapter **??**. This section requires us to have some basic familiarity with Stochastic Calculus (covered in Appendix D), specifically Ito Processes and Ito's Lemma. Here we consider the single-decision version of the problem where our portfolio investment choices are:

- A risky asset, evolving in continuous time, with stochastic process S whose movements are defined by the Ito process:

  $$dS_t = \mu \cdot S_t \cdot dt + \sigma \cdot S_t \cdot dz_t$$

  where $\mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$ are given constants.
- A riskless asset, growing continuously in time, with value denoted R whose growth is defined by the ordinary differential equation:

  $$dR_t = r \cdot R_t \cdot dt$$

  where $r \in \mathbb{R}$ is a given constant.

We are given \$1 to invest over a period of 1 year. Our task is to determine the constant fraction $\pi \in \mathbb{R}$ of portfolio wealth W_t at any time t to maintain in the risky asset so as to maximize the Expected Utility of Consumption of Wealth at the end of 1 year (i.e., of Portfolio Wealth W_1). Without loss of generality, assume $W_0 = 1$. The key feature of this problem is that we are required to continuously rebalance the portfolio to maintain the same constant fraction π of investment in the risky asset. Since W_t is the portfolio wealth at time t, the value of the investment in the risky asset at time t would need to be $\pi \cdot W_t$ and the value of the investment in the riskless asset at time t would need to be $(1-\pi) \cdot W_t$. We allow π to be unconstrained, i.e., π can take any value from $-\infty$ to $+\infty$. Positive π means we have a "long" position in the risky asset and negative π means we have a "short" position in the risky asset. Likewise, positive $1-\pi$ means we are lending money at the riskless interest rate of r and negative $1-\pi$ means we are borrowing money at the riskless interest rate of r.

Therefore, the change in the value of the risky asset investment from time t to time $t + dt$ is:

$$\mu \cdot \pi \cdot W_t \cdot dt + \sigma \cdot \pi \cdot W_t \cdot dz_t$$

Likewise, the change in the value of the riskless asset investment from time t to time $t + dt$ is:

$$r \cdot (1 - \pi) \cdot W_t \cdot dt$$

Therefore, the infinitesimal change in portfolio wealth dW_t from time t to time $t + dt$ is given by:

$$dW_t = (r + \pi(\mu - r)) \cdot W_t \cdot dt + \pi \cdot \sigma \cdot W_t \cdot dz_t$$

Note that this is an Ito process defining the stochastic evolution of portfolio wealth.

We assume CRRA Utility with $\gamma \neq 0$, so:

$$U(W_1) = \begin{cases} \frac{W_1^{1-\gamma}-1}{1-\gamma} & \text{for } \gamma \neq 1 \\ \log(W_1) & \text{for } \gamma = 1 \end{cases}$$

Applying Ito's Lemma on $\log W_t$ gives us:

$$d(\log W_t) = \frac{1}{W_t} \cdot dW_t - \frac{1}{2W_t^2} \cdot (dW_t)^2$$

$$\Rightarrow d(\log W_t) = (r + \pi(\mu - r)) \cdot dt + \pi \cdot \sigma \cdot dz_t - \frac{\pi^2\sigma^2}{2} \cdot dt$$

$$\Rightarrow \log W_t = \int_0^t (r + \pi(\mu - r) - \frac{\pi^2\sigma^2}{2}) \cdot du + \int_0^t \pi \cdot \sigma \cdot dz_u$$

Using the martingale property and Ito Isometry for the Ito integral $\int_0^t \pi \cdot \sigma \cdot dz_u$ (see Appendix D), we get:

$$\log W_1 \sim \mathcal{N}(r + \pi(\mu - r) - \frac{\pi^2\sigma^2}{2}, \pi^2\sigma^2)$$

We know that maximizing $\mathbb{E}[U(W_1)]$ is equivalent to maximizing the Certainty-Equivalent Value of W_1, hence also equivalent to maximizing the log of the Certainty-Equivalent Value of W_1, which in this case (using the formula for x_{CE} from the section on CRRA) is given by:

$$r + \pi(\mu - r) - \frac{\pi^2\sigma^2}{2} + \frac{\pi^2\sigma^2(1-\gamma)}{2}$$
$$= r + \pi(\mu - r) - \frac{\pi^2\sigma^2\gamma}{2}$$

This is a concave function of π and so, taking it's derivative with respect to π and setting it to 0 gives us the optimal investment fraction in the risky asset (π^*) as follows:

$$\pi^* = \frac{\mu - r}{\gamma\sigma^2}$$

Key Takeaways from this Chapter

- An individual's financial risk-aversion is represented by the concave nature of the individual's Utility as a function of financial outcomes.
- Risk-Premium (compensation an individual seeks for taking financial risk) is roughly proportional to the individual's financial risk-aversion and the measure of uncertainty in financial outcomes.
- Risk-Adjusted-Return in finance should be thought of as the Certainty-Equivalent-Value, whose Utility is the Expected Utility across uncertain (risky) financial outcomes.

6. Dynamic Asset-Allocation and Consumption

This chapter covers the first of five financial applications of Stochastic Control covered in this book. This financial application deals with the topic of investment management for not just a financial company, but more broadly for any corporation or for any individual. The nuances for specific companies and individuals can vary considerably but what is common across these entities is the need to:

- Periodically decide how one's investment portfolio should be split across various choices of investment assets - the key being how much money to invest in more risky assets (which have potential for high returns on investment) versus less risky assets (that tend to yield modest returns on investment). This problem of optimally allocating capital across investment assets of varying risk-return profiles relates to the topic of Utility Theory we covered in Chapter 5. However, in this chapter, we deal with the further challenge of adjusting one's allocation of capital across assets, as time progresses. We refer to this feature as *Dynamic Asset Allocation* (the word dynamic refers to the adjustment of capital allocation to adapt to changing circumstances)

- Periodically decide how much capital to leave in one's investment portfolio versus how much money to consume for one's personal needs/pleasures (or for a corporation's operational requirements) by extracting money from one's investment portfolio. Extracting money from one's investment portfolio can mean potentially losing out on investment growth opportunities, but the flip side of this is the Utility of Consumption that a corporation/individual desires. Noting that ultimately our goal is to maximize total utility of consumption over a certain time horizon, this decision of investing versus consuming really amounts to the timing of consumption of one's money over the given time horizon.

Thus, this problem constitutes the dual and dynamic decisioning of asset-allocation and consumption. To gain an intuitive understanding of the challenge of this dual dynamic decisioning problem, let us consider this problem from the perspective of personal finance in a simplified setting.

Optimization of Personal Finance

Personal Finances can be very simple for some people (earn a monthly salary, spend the entire salary) and can be very complicated for some other people (eg: those who own multiple businesses in multiple countries and have complex assets and liabilities). Here we shall consider a situation that is relatively simple but includes sufficient nuances to give you a flavor for the aspects of the general problem of dynamic asset-allocation and consumption. Let's say your personal finances consist of the following aspects:

- *Receiving money*: This could include your periodic salary, which typically remains constant for a period of time, but can change if you get a promotion or if you get a new job. This also includes money you liquidate from your investment portfolio, eg: if you sell some stock, and decide not to re-invest in other investment assets. This also includes interest you earn from your savings account or from some bonds you might own. There are many other ways one can *receive money*, some fixed regular payments and some uncertain in terms of payment quantity and timing, and we won't enumerate all the different ways of *receiving money*. We just want to highlight here that *receiving money* on a periodic basis is one of the key financial aspects in one's life.
- *Consuming money*: The word "consume" refers to ""spending". Note that there is some *consumption of money* on a periodic basis that is required to satisfy basic needs like shelter, food and clothing. The rent or mortgage you pay on your house is one example - it may be a fixed amount every month, but if your mortgage rate is a floating rate, it is subject to variation. Moreover, if you move to a new house, the rent or mortgage can be different. The money you spend on food and clothing also constitutes *consuming money*. This can often be fairly stable from one month to the next, but if you have a newborn baby, it might require additional expenses of the baby's food, clothing and perhaps also toys. Then there is *consumption of money* that are beyond the "necessities" - things like eating out at a fancy restaurant on the weekend, taking a summer vacation, buying a luxury car or an expensive watch etc. One gains "satisfaction"/"happiness" (i.e., *Utility*) out of this *consumption of money*. The key point here is that we need to periodically make a decision on how much to spend (*consume money*) on a weekly or monthly basis. One faces a tension in the dynamic decision between *consuming money* (that gives us *Consumption Utility*) and *saving money* (which is the money we put in our investment portfolio in the hope of the money growing, so we can consume potentially larger amounts of money in the future).
- *Investing Money*: Let us suppose there are a variety of investment assets you can invest in - simple savings account giving small interest, exchange-traded stocks (ranging from value stocks to growth stocks, with their respective risk-return tradeoffs), real-estate (the house you bought and live in is indeed considered an investment asset), commodities such as gold,

paintings etc. We call the composition of money invested in these assets as one's investment portfolio (see Appendix **??** for a quick introduction to Portfolio Theory). Periodically, we need to decide if one should play safe by putting most of one's money in a savings account, or if we should allocate investment capital mostly in stocks, or if we should be more speculative and invest in an early-stage startup or in a rare painting. Reviewing the composition and potentially re-allocating capital (refered to as re-balancing one's portfolio) is the problem of dynamic asset-allocation. Note also that we can put some of our *received money* into our investment portfolio (meaning we choose to not consume that money right away). Likewise, we can extract some money out of our investment portfolio so we can *consume money*. The decisions of insertion and extraction of money into/from our investment portfolio is essentially the dynamic money-consumption decision we make, which goes together with the dynamic asset-allocation decision.

The above description has hopefully given you a flavor of the dual and dynamic decisioning of asset-allocation and consumption. Ultimately, our personal goal is to maximize the Expected Aggregated Utility of Consumption of Money over our lifetime (and perhaps, also include the Utility of Consumption of Money for one's spouse and children, after you die). Since investment portfolios are stochastic in nature and since we have to periodically make decisions on asset-allocation and consumption, you can see that this has all the ingredients of a Stochastic Control problem, and hence can be modeled as a Markov Decision Process (albeit typically fairly complicated, since real-life finances have plenty of nuances). Here's a rough and informal sketch of what that MDP might look like (bear in mind that we will formalize the MDP for simplified cases later in this chapter):

- States: The *State* can be quite complex in general, but mainly it consists of one's age (to keep track of the time to reach the MDP horizon), the quantities of money invested in each investment asset, the valuation of the assets invested in, and potentially also other aspects like one's job/career situation (required to make predictions of future salary possibilities).

- Actions: The *Action* is two-fold. Firstly, it's the vector of investment amounts one chooses to make at each time step (the time steps are at the periodicity at which we review our investment portfolio for potential re-allocation of capital across assets). Secondly, it's the quantity of money one chooses to consume that is *flexible/optional* (i.e., beyond the fixed payments like rent that we are committed to make).

- Rewards: The *Reward* is the Utility of Consumption of Money that we deemed as flexible/optional - it corresponds to the second part of the *Action*.

- Model: The *Model* (probabilities of next state and reward, given current state and action) can be fairly complex in most real-life situations. The hardest aspect is the prediction of what might happen tomorrow in our life and career (we need this prediction since it determines our future likelihood to receive money, consume money and invest money). Moreover, the uncertain movements of investment assets would need to be captured by our model.

Since our goal here was to simply do a rough and informal sketch, the above coverage of the MDP is very hazy but we hope you get a sense for what the MDP might look like. Now we are ready to take a simple special case of this MDP which does away with many of the real-world frictions and complexities, yet retains the key features (in particular, the dual dynamic decisioning aspect). This simple special case was the subject of Merton's Portfolio Problem which he formulated and solved in 1969 in a landmark paper. A key feature of his formulation was that time is continuous and so, *state* (based on asset prices) evolves as a stochastic process, and actions (asset-allocation and consumption) are made continuously. We cover the important parts of his paper in the next section. Note that our coverage below requires some familiarity with Stochastic Calculus (covered in Appendix D) and with the Hamilton-Jacobi-Bellman Equation (covered in Appendix E), which is the continuous-time analog of Bellman's Optimality Equation.

Merton's Portfolio Problem and Solution

Now we describe Merton's Portfolio problem and derive its analytical solution, which is one of the most elegant solutions in Mathematical Economics. The solution structure will provide tremendous intuition for how the asset-allocation and consumption decisions depend on not just the state variables but also on the problem inputs.

We denote time as t and say that current time is $t = 0$. Assume that you have just retired (meaning you won't be earning any money for the rest of your life) and that you are going to live for T more years (T is a fixed real number). So, in the language of the previous section, you will not be *receiving money* for the rest of your life, other than the option of extracting money from your investment portfolio. Also assume that you have no fixed payments to make like mortgage, subscriptions etc. (assume that you have already paid for a retirement service that provides you with your essential food, clothing and other services). This means all of your *money consumption* is flexible, i.e., you have a choice of consuming any real non-negative number at any point in time. All of the above are big (and honestly, unreasonable) assumptions but they help keep the problem simple enough for analytical tractability. In spite of these over-simplified assumptions, the problem formulation still captures the salient aspects of dual dynamic decisioning of asset-allocation and consumption while eliminating the clutter of A) *receiving money* from external sources and B) *consuming money* that is of a non-optional nature.

We define wealth at any time t (denoted W_t) as the aggregate market value of your investment assets. Note that since no external money is received and since all consumption is optional, W_t is your "net-worth". Assume there are a fixed number n of risky assets and a single riskless asset. Assume that each risky asset has a known normal distribution of returns. Now we make a couple of big assumptions for analytical tractability:

- You are allowed to buy or sell any fractional quantities of assets at any point in time (i.e., in continuous time).
- There are no transaction costs with any of the buy or sell transactions in any of the assets.

You start with wealth W_0 at time $t = 0$. As mentioned earlier, the goal is to maximize your expected lifetime-aggregated Utility of Consumption of money with the actions at any point in time being two-fold: Asset-Allocation and Consumption (Consumption being equal to the capital extracted from the investment portfolio at any point in time). Note that since there is no external source of money and since all capital extracted from the investment portfolio at any point in time is immediately consumed, you are never adding capital to your investment portfolio. The growth of the investment portfolio can happen only from growth in the market value of assets in your investment portfolio. Lastly, we will assume that Consumption Utility function is Constant Relative Risk-Aversion (CRRA), which we covered in Chapter 5.

For ease of exposition, we will formalize the problem setting and derive Merton's beautiful analytical solution for the case of $n = 1$ (i.e., only 1 risky asset). The solution generalizes in a straightforward manner to the case of $n > 1$ risky assets, so the heavier notation for n risky assets is not worth much.

Since we are operating in continuous-time, the risky asset follows a stochastic process (denoted S)- specifically an Ito process (introductory background on Ito processes and Ito's Lemma covered in D), as follows:

$$dS_t = \mu \cdot S_t \cdot dt + \sigma \cdot S_t \cdot dz_t$$

where $\mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$ are fixed constants (note that for n assets, we would instead work with a vector for μ and a matrix for σ).

The riskless asset has no uncertainty associated with it and has a fixed rate of growth in continuous-time, so the valuation of the riskless asset R_t at time t is given by:

$$dR_t = r \cdot R_t \cdot dt$$

Assume $r \in \mathbb{R}$ is a fixed constant, representing the instantaneous rate of growth of money. We denote the consumption of wealth (equal to extraction of money from the investment portfolio) per unit time (at time t) as $c(t, W_t) \geq 0$ to make it clear that the consumption (our decision at any time t) will in general depend on both time t and wealth W_t. Note that we talk about "rate of consumption in time" because consumption is assumed to be continuous in time. As mentioned earlier, we denote wealth at time t as W_t (note that W is a stochastic process too). We assume that $W_t > 0$ for all $t \geq 0$. This is a reasonable assumption to make as it manifests in constraining the consumption (extraction from investment portfolio) to ensure wealth remains positive. We denote the fraction of wealth allocated to the risky asset at time t as $\pi(t, W_t)$. Just like consumption c, risky-asset allocation fraction π is a function of time t and wealth W_t. Since there is only one risky asset, the fraction of wealth allocated to the riskless asset at time t is $1 - \pi(t, W_t)$. Unlike the constraint $c(t, W_t) > 0$, $\pi(t, W_t)$ is assumed to be unconstrained. Note that $c(t, W_t)$ and $\pi(t, W_t)$ together constitute the decision (MDP action) you'd be making at time t. To keep our notation light, we shall write c_t for $c(t, W_t)$ and π_t for $\pi(t, W_t)$, but please do recognize throughout the derivation that both are functions of wealth W_t at time t as well as of time t itself. Finally, we assume that the Utility of Consumption function is defined as the CRRA function:

$$U(x) = \begin{cases} \frac{x^{1-\gamma}}{1-\gamma} & \text{for } \gamma \neq 1 \\ \log(x) & \text{for } \gamma = 1 \end{cases}$$

As we learnt in Chapter 5, γ is the Coefficient of Constant Relative Risk-Aversion (CRRA) equal to $\frac{-x \cdot U''(x)}{U'(x)}$.

Due to our assumption of no addition of money to our investment portfolio of the risky asset S_t and riskless R_t and due to our assumption of no transaction costs of buying/selling any fractional quantities of risky as well as riskless asset, the time-evolution for wealth should be conceptualized as a continuous adjustment of the allocation π_t and continuous extraction from the portfolio (equal to continuous consumption c_t).

Since the value of the risky asset investment at time t is $\pi_t \cdot W_t$, the change in the value of the risky asset investment from time t to time $t + dt$ is:

$$\mu \cdot \pi_t \cdot W_t \cdot dt + \sigma \cdot \pi_t \cdot W_t \cdot dz_t$$

Likewise, since the value of the riskless asset investment at time t is $(1-\pi_t) \cdot W_t$, the change in the value of the riskless asset investment from time t to time $t + dt$ is:

$$r \cdot (1 - \pi_t) \cdot W_t \cdot dt$$

Therefore, the infinitesimal change in wealth dW_t from time t to time $t + dt$ is given by:

$$dW_t = ((r + \pi_t \cdot (\mu - r)) \cdot W_t - c_t) \cdot dt + \pi_t \cdot \sigma \cdot W_t \cdot dz_t \tag{6.1}$$

Note that this is an Ito process defining the stochastic evolution of wealth.

Our goal is to determine optimal $(\pi(t, W_t), c(t, W_t))$ at any time t to maximize:

$$\mathbb{E}[\int_t^T \frac{e^{-\rho(s-t)} \cdot c_s^{1-\gamma}}{1-\gamma} \cdot ds + \frac{e^{-\rho(T-t)} \cdot B(T) \cdot W_T^{1-\gamma}}{1-\gamma} \mid W_t]$$

where $\rho \geq 0$ is the utility discount rate to account for the fact that future utility of consumption might be less than current utility of consumption, and $B(\cdot)$ is known the "bequest" function (think of this as the money you will leave for your family when you die at time T). We can solve this problem for arbitrary bequest $B(T)$ but for simplicity, we shall consider $B(T) = \epsilon^\gamma$ where $0 < \epsilon \ll 1$, meaning "no bequest". We require the bequest to be ϵ^γ rather than 0 for technical reasons, that we will become apparent later. Also, we will solve this problem for $\gamma \neq 1$ ($\gamma = 1$ is easier, and we encourage you to work out the derivation for $\gamma = 1$).

We should think of this problem as a continuous-time Stochastic Control problem where the MDP is defined as below:

- The *State* at time t is (t, W_t)
- The *Action* at time t is (π_t, c_t)
- The *Reward* per unit time at time $t < T$ is:

$$U(c_t) = \frac{c_t^{1-\gamma}}{1-\gamma}$$

and the *Reward* at time T is:

$$B(T) \cdot U(W_T) = \epsilon^\gamma \cdot \frac{W_T^{1-\gamma}}{1-\gamma}$$

The *Return* at time t is the accumulated discounted *Reward*:

$$\int_t^T e^{-\rho(s-t)} \cdot \frac{c_s^{1-\gamma}}{1-\gamma} \cdot ds + \frac{e^{-\rho(T-t)} \cdot \epsilon^\gamma \cdot W_T^{1-\gamma}}{1-\gamma}$$

Our goal is to find the *Policy* : $(t, W_t) \rightarrow (\pi_t, c_t)$ that maximizes the *Expected Return*. Note the important constraint that $c_t \geq 0$, but π_t is unconstrained.

Our first step will be write out the Hamilton-Jacobi-Bellman (HJB) Equation (the analog of the Bellman Optimality Equation in continuous-time). We denote the Optimal Value Function as V^*. We write the Optimal Value at time t for wealth W_t with lighter notation $V^*(t, W_t)$ instead of the more precise notation $V^*((t, W_t))$. Appendix E provides the derivation of the general HJB formulation (Equation (E.1) in Appendix E) - this general HJB Equation specializes here to the following:

$$\max_{\pi_t, c_t} \{\mathbb{E}_t[dV^*(t, W_t) + \frac{c_t^{1-\gamma}}{1-\gamma} \cdot dt\} = \rho \cdot V^*(t, W_t) \cdot dt \tag{6.2}$$

Now use Ito's Lemma on dV^*, remove the dz_t term since it's a martingale, and divide throughout by dt to produce the HJB Equation in partial-differential form for any $0 \leq t < T$, as follows (the general form of this transformation appears as Equation (E.2) in Appendix E):

$$\max_{\pi_t, c_t} \{\frac{\partial V^*}{\partial t} + \frac{\partial V^*}{\partial W_t} + \frac{\partial^2 V^*}{\partial W_t^2} \cdot \frac{\pi_t^2 \cdot \sigma^2 \cdot W_t^2}{2} + \frac{c_t^{1-\gamma}}{1-\gamma}\} = \rho \cdot V^*(t, W_t) \tag{6.3}$$

This HJB Equation is subject to the terminal condition:

$$V^*(T, W_T) = \epsilon^\gamma \cdot \frac{W_T^{1-\gamma}}{1-\gamma}$$

Let us write Equation (6.3) more succinctly as:

$$\max_{\pi_t, c_t} \Phi(t, W_t; \pi_t, c_t) = \rho \cdot V^*(t, W_t) \tag{6.4}$$

It pays to emphasize again that we are working with the constraints $W_t > 0, c_t \geq 0$ for $0 \leq t < T$

To find optimal π_t^*, c_t^*, we take the partial derivatives of $\Phi(t, W_t; \pi_t, c_t)$ with respect to π_t and c_t, and equate to 0 (first-order conditions for Φ). The partial derivative of Φ with respect to π_t is:

$$(\mu - r) \cdot \frac{\partial V^*}{\partial W_t} + \frac{\partial^2 V^*}{\partial W_t^2} \cdot \pi_t \cdot \sigma^2 \cdot W_t = 0$$

$$\Rightarrow \pi_t^* = \frac{-\frac{\partial V^*}{\partial W_t} \cdot (\mu - r)}{\frac{\partial^2 V^*}{\partial W_t^2} \cdot \sigma^2 \cdot W_t} \tag{6.5}$$

The partial derivative of Φ with respect to c_t is:

$$-\frac{\partial V^*}{\partial W_t} + (c_t^*)^{-\gamma} = 0$$

$$\Rightarrow c_t^* = (\frac{\partial V^*}{\partial W_t})^{\frac{-1}{\gamma}} \tag{6.6}$$

Now substitute π_t^* (from Equation (6.5))and c_t^* (from Equation (6.6)) in $\Phi(t, W_t; \pi_t, c_t)$ (in Equation (6.3)) and equate to $\rho \cdot V^*(t, W_t)$. This gives us the Optimal Value Function Partial Differential Equation (PDE):

$$\frac{\partial V^*}{\partial t} - \frac{(\mu - r)^2}{2\sigma^2} \cdot \frac{(\frac{\partial V^*}{\partial W_t})^2}{\frac{\partial^2 V^*}{\partial W_t^2}} + \frac{\partial V^*}{\partial W_t} \cdot r \cdot W_t + \frac{\gamma}{1 - \gamma} \cdot (\frac{\partial V^*}{\partial W_t})^{\frac{\gamma - 1}{\gamma}} = \rho \cdot V^*(t, W_t) \tag{6.7}$$

The boundary condition for this PDE is:

$$V^*(T, W_T) = \epsilon^\gamma \cdot \frac{W_T^{1-\gamma}}{1 - \gamma}$$

The second-order conditions for Φ are satisfied under the assumptions: $c_t^* > 0, W_t > 0, \frac{\partial^2 V^*}{\partial W_t^2} < 0$ for all $0 \leq t < T$ (we will later show that these are all satisfied in the solution we derive), and for concave $U(\cdot)$, i.e., $\gamma > 0$

Next, we want to reduce the PDE (6.7) to an Ordinary Differential Equation (ODE) so we can solve the (simpler) ODE. Towards this goal, we surmise with a guess solution:

$$V^*(t, W_t) = f(t)^\gamma \cdot \frac{W_t^{1-\gamma}}{1 - \gamma} \tag{6.8}$$

Then,

$$\frac{\partial V^*}{\partial t} = \gamma \cdot f(t)^{\gamma - 1} \cdot f'(t) \cdot \frac{W_t^{1-\gamma}}{1 - \gamma} \tag{6.9}$$

$$\frac{\partial V^*}{\partial W_t} = f(t)^{\gamma} \cdot W_t^{-\gamma} \tag{6.10}$$

$$\frac{\partial^2 V^*}{\partial W_t^2} = -f(t)^{\gamma} \cdot \gamma \cdot W_t^{-\gamma-1} \tag{6.11}$$

Substituting the guess solution in the PDE, we get the simple ODE:

$$f'(t) = \nu \cdot f(t) - 1 \tag{6.12}$$

where

$$\nu = \frac{\rho - (1-\gamma) \cdot (\frac{(\mu-r)^2}{2\sigma^2\gamma} + r)}{\gamma}$$

with boundary condition $f(T) = \epsilon$.

We note that that the bequest function $B(T) = \epsilon^{\gamma}$ proves to be convenient in order to fit the guess solution for $t = T$. This means the boundary condition for this ODE is: $f(T) = \epsilon$. Consequently, this ODE together with this boundary condition has a simple enough solution, as follows:

$$f(t) = \begin{cases} \frac{1+(\nu\epsilon-1)\cdot e^{-\nu(T-t)}}{\nu} & \text{for } \nu \neq 0 \\ T - t + \epsilon & \text{for } \nu = 0 \end{cases} \tag{6.13}$$

Substituting V^* (from Equation (6.8)) and its partial derivatives (from Equations (6.9), (6.10) and (6.11)) in Equations (6.5) and (6.6), we get:

$$\pi^*(t, W_t) = \frac{\mu - r}{\sigma^2\gamma} \tag{6.14}$$

$$c^*(t, W_t) = \frac{W_t}{f(t)} = \begin{cases} \frac{\nu \cdot W_t}{1+(\nu\epsilon-1)\cdot e^{-\nu(T-t)}} & \text{for } \nu \neq 0 \\ \frac{W_t}{T-t+\epsilon} & \text{for } \nu = 0 \end{cases} \tag{6.15}$$

Finally, substituting the solution for $f(t)$ (Equation (6.13)) in Equation (6.8), we get:

$$V^*(t, W_t) = \begin{cases} \frac{(1+(\nu\epsilon-1)\cdot e^{-\nu(T-t)})^{\gamma}}{\nu^{\gamma}} \cdot \frac{W_t^{1-\gamma}}{1-\gamma} & \text{for } \nu \neq 0 \\ \frac{(T-t+\epsilon)^{\gamma} \cdot W_t^{1-\gamma}}{1-\gamma} & \text{for } \nu = 0 \end{cases} \tag{6.16}$$

Note that $f(t) > 0$ for all $0 \leq t < T$ (for all ν) ensures $W_t, c_t^* > 0$, $\frac{\partial^2 V^*}{\partial W_t^2} < 0$. This ensures the constraints $W_t > 0$ and $c_t \geq 0$ are satisfied and the second-order conditions for Φ are also satisfied. A very important lesson in solving Merton's Portfolio problem is the fact that the HJB Formulation is key and that this solution approach provides a template for similar continuous-time stochastic control problems.

Developing Intuition for the Solution to Merton's Portfolio Problem

The solution for $\pi^*(t, W_t)$ and $c^*(t, W_t)$ are surprisingly simple. $\pi^*(t, W_t)$ is a constant, i.e., it is independent of both of the state variables t and W_t. This means that no matter what wealth we carry and no matter how close we are to the end of the horizon (i.e., no matter what our age is), we should invest the same fraction of our wealth in the risky asset (likewise for the case of n risky assets). The simplifying assumptions in Merton's Portfolio problem statement did play a part in the simplicity of the solution, but the fact that $\pi^*(t, W_t)$ is a constant is still rather surprising. The simplicity of the solution means that asset allocation is straightforward - we just need to keep re-balancing to maintain this constant fraction of our wealth in the risky asset. We expect our wealth to grow over time and so, the capital in the risky asset would also grow proportionately.

The form of the solution for $c^*(t, W_t)$ is extremely intuitive - the excess return of the risky asset $(\mu - r)$ shows up in the numerator, which makes sense, since one would expect to invest a higher fraction of one's wealth in the risky asset if it gives us a higher excess return. It also makes sense that the volatility σ of the risky asset (squared) shows up in the denominator (the greater the volatility, the less we'd allocate to the risky asset, since we are typically risk-averse, i.e., $\gamma > 0$). Likewise, it makes since that the coefficient of CRRA γ shows up in the denominator since a more risk-averse individual (greater value of γ) will want to invest less in the risky asset.

The Optimal Consumption Rate $c^*(t, W_t)$ should be conceptualized in terms of the *Optimal Fractional Consumption Rate*, i.e., the Optimal Consumption Rate $c^*(t, W_t)$ as a fraction of the Wealth W_t. Note that the Optimal Fractional Consumption Rate depends only on t (it is equal to $\frac{1}{f(t)}$). This means no matter what our wealth is, we should be extracting a fraction of our wealth on a daily/monthly/yearly basis that is only dependent on our age. Note also that if $\epsilon < \frac{1}{\nu}$, the Optimal Fractional Consumption Rate increases as time progresses. This makes intuitive sense because when we have many more years to live, we'd want to consume less and invest more to give the portfolio more ability to grow, and when we get close to our death, we increase our consumption (since the optimal is "to die broke", assuming no bequest).

Now let us understand how the Wealth process evolves. Let us substitute for $\pi^*(t, W_t)$ (from Equation (6.14)) and $c^*(t, W_t)$ (from Equation (6.15)) in the Wealth process defined in Equation (6.1). This yields the following Wealth process when we asset-allocate optimally and consume optimally:

$$dW_t = (r + \frac{(\mu - r)^2}{\sigma^2 \gamma} - \frac{1}{f(t)}) \cdot W_t \cdot dt + \frac{\mu - r}{\sigma \gamma} \cdot W_t \cdot dz_t \tag{6.17}$$

The first thing to note about this Wealth process is that it is a lognormal process of the form covered in Section D of Appendix **??**. The lognormal volatility (fractional dispersion) of this wealth process is constant ($= \frac{\mu - r}{\sigma\gamma}$). The lognormal drift (fractional drift) is independent of the wealth but is dependent on time ($= r + \frac{(\mu-r)^2}{\sigma^2\gamma} - \frac{1}{f(t)}$). From the solution of the general lognormal process derived in Section D of Appendix ??, we conclude that:

$$\mathbb{E}[W_t] = W_0 \cdot e^{(r+\frac{(\mu-r)^2}{\sigma^2\gamma})t} \cdot e^{-\int_0^t \frac{du}{f(u)}} = \begin{cases} W_0 \cdot e^{(r+\frac{(\mu-r)^2}{\sigma^2\gamma})t} \cdot (1 - \frac{1-e^{-\nu t}}{1+(\nu\epsilon-1)\cdot e^{-\nu T}}) & \text{if } \nu \neq 0 \\ W_0 \cdot e^{(r+\frac{(\mu-r)^2}{\sigma^2\gamma})t} \cdot (1 - \frac{t}{T+\epsilon}) & \text{if } \nu = 0 \end{cases} \tag{6.18}$$

Since we assume no bequest, we should expect the Wealth process to keep growing up to some point in time and then fall all the way down to 0 when time runs out (i.e., when $t = T$). We shall soon write the code for Equation (6.18) and plot the graph for this rise and fall. An important point to note is that although the wealth process growth varies in time (expected wealth growth rate $= r + \frac{(\mu-r)^2}{\sigma^2\gamma} - \frac{1}{f(t)}$ as seen from Equation (6.17)), the variation (in time) of the wealth process growth is only due to the fractional consumption rate varying in time. If we ignore the fractional consumption rate ($= \frac{1}{f(t)}$), then what we get is the Expected Portfolio Annual Return of $r + \frac{(\mu-r)^2}{\sigma^2\gamma}$ which is a constant (does not depend on either time t or on Wealth W_t). Now let us write some code to calculate the time-trajectories of Expected Wealth, Fractional Consumption Rate, Expected Wealth Growth Rate and Expected Portfolio Annual Return.

The code should be pretty self-explanatory. We will just provide a few explanations of variables in the code that may not be entirely obvious: `portfolio_return` calculates the Expected Portfolio Annual Return, `nu` calculates the value of ν, `f` represents the function $f(t)$, `wealth_growth_rate` calculates the Expected Wealth Growth Rate as a function of time t. The `expected_wealth` method assumes $W_0 = 1$.

```
@dataclass(frozen=True)
class MertonPortfolio:
    mu: float
    sigma: float
    r: float
    rho: float
    horizon: float
    gamma: float
    epsilon: float = 1e-6

    def excess(self) -> float:
        return self.mu - self.r

    def variance(self) -> float:
        return self.sigma * self.sigma

    def allocation(self) -> float:
        return self.excess() / (self.gamma * self.variance())

    def portfolio_return(self) -> float:
        return self.r + self.allocation() * self.excess()
```

```python
    def nu(self) -> float:
        return (self.rho - (1 - self.gamma) * self.portfolio_return()) / \
            self.gamma

    def f(self, time: float) -> float:
        remaining: float = self.horizon - time
        nu = self.nu()
        if nu == 0:
            ret = remaining + self.epsilon
        else:
            ret = (1 + (nu * self.epsilon - 1) * exp(-nu * remaining)) / nu
        return ret

    def fractional_consumption_rate(self, time: float) -> float:
        return 1 / self.f(time)

    def wealth_growth_rate(self, time: float) -> float:
        return self.portfolio_return() - self.fractional_consumption_rate(time)

    def expected_wealth(self, time: float) -> float:
        base: float = exp(self.portfolio_return() * time)
        nu = self.nu()
        if nu == 0:
            ret = base * (1 - (1 - exp(-nu * time)) /
                          (1 + (nu * self.epsilon - 1) *
                           exp(-nu * self.horizon)))
        else:
            ret = base * (1 - time / (self.horizon + self.epsilon))
        return ret
```

The above code is in the file rl/chapter7/merton_solution_graph.py. We highly encourage you to experiment by changing the various inputs in this code $(T, \mu, \sigma, r, \rho, \gamma)$ and visualize how the results change. Doing this will help build tremendous intuition.

A rather interesting observation is that if $r + \frac{(\mu - r)^2}{\sigma^2 \gamma} > \frac{1}{f(0)}$ and $\epsilon < \frac{1}{\nu}$, then the Fractional Consumption Rate is initially less than the Expected Portfolio Annual Return and over time, the Fractional Consumption Rate becomes greater than the Expected Portfolio Annual Return. This illustrates how the optimal behavior is to consume modestly and invest more when one is younger, then to gradually increase the consumption as one ages, and finally to ramp up the consumption sharply when one is close to the end of one's life. Figure 6.1 shows the visual for this (along with the Expected Wealth Growth Rate) using the above code for input values of: $T = 20, \mu = 10\%, \sigma = 10\%, r = 2\%, \rho = 1\%, \gamma = 2.0$.

Figure 6.2 shows the time-trajectory of the expected wealth based on Equation (6.18) for the same input values as listed above. Notice how the Expected

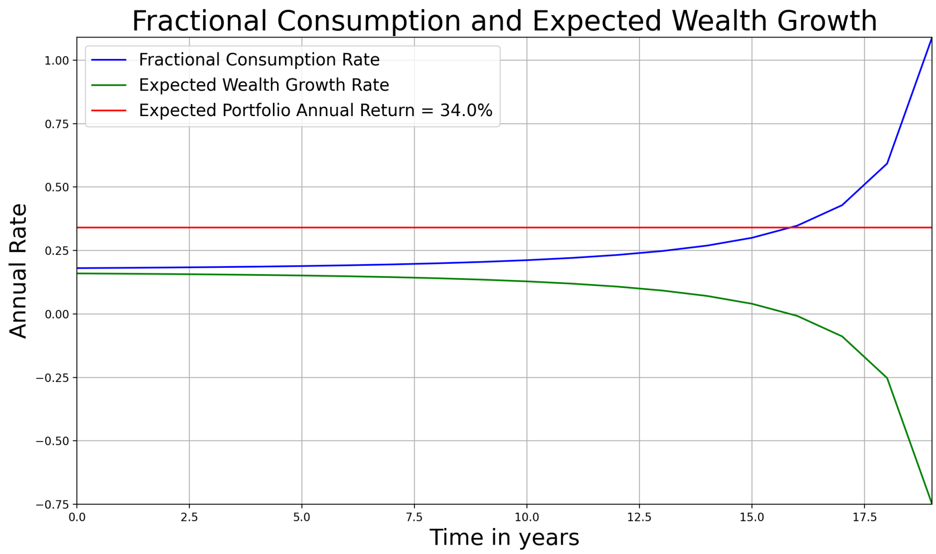

Figure 6.1.: Portfolio Return and Consumption Rate

Wealth rises in a convex shape for several years since the consumption during all these years is quite modest, and then the shape of the Expected Wealth curve turns concave at about 12 years, peaks at about 16 years (when Fractional Consumption Rate rises to equal Expected Portfolio Annual Return), and then falls precipitously in the last couple of years (as the Consumption increasingly drains the Wealth down to 0).

A Discrete-Time Asset-Allocation Example

In this section, we cover a discrete-time version of the problem that lends itself to analytical tractability, much like Merton's Portfolio Problem in continuous-time. We are given wealth W_0 at time 0. At each of discrete time steps labeled $t = 0, 1, \ldots, T-1$, we are allowed to allocate the wealth W_t at time t to a portfolio of a risky asset and a riskless asset in an unconstrained manner with no transaction costs. The risky asset yields a random return $\sim \mathcal{N}(\mu, \sigma^2)$ over each single time step (for a given $\mu \in \mathbb{R}$ and a given $\sigma \in \mathbb{R}^+$). The riskless asset yields a constant return denoted by r over each single time step (for a given $r \in \mathbb{R}$). We assume that there is no consumption of wealth at any time $t < T$, and that we liquidate and consume the wealth W_T at time t. So our goal is simply to maximize the Expected Utility of Wealth at the final time step $t = T$ by dynamically allocating $x_t \in \mathbb{R}$ in the risky asset and the remaining $W_t - x_t$ in the riskless asset for each $t = 0, 1, \ldots, T-1$. Assume the single-time-step discount factor is γ and that the Utility of Wealth at the final time step $t = T$ is given by the following CARA function:

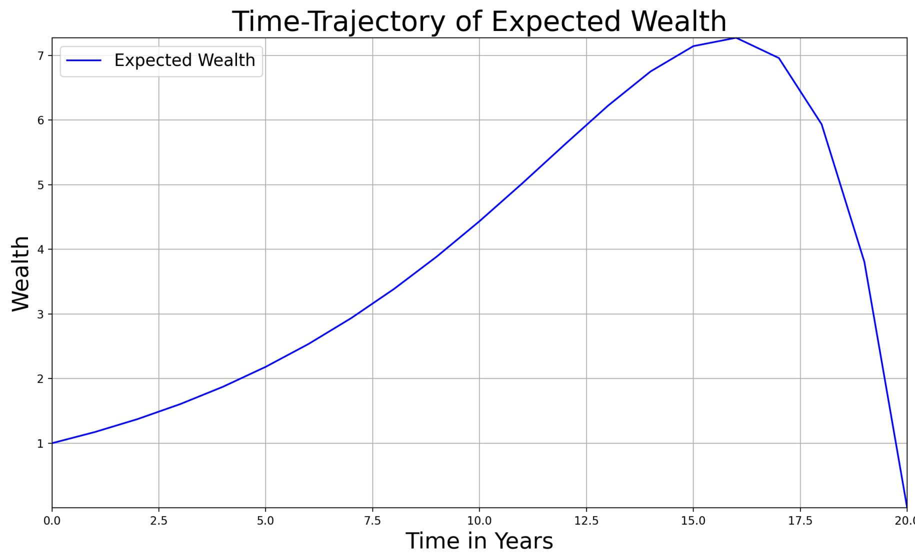

Figure 6.2.: Expected Wealth Time-Trajectory

$$U(W_T) = \frac{1 - e^{-aW_T}}{a} \text{ for some fixed } a \neq 0$$

Thus, the problem is to maximize, for each $t = 0, 1, \ldots, T-1$, over choices of $x_t \in \mathbb{R}$, the value:

$$\mathbb{E}[\gamma^{T-t} \cdot \frac{1 - e^{-aW_T}}{a} | (t, W_t)]$$

Since γ^{T-t} and a are constants, this is equivalent to maximizing, for each $t = 0, 1, \ldots, T-1$, over choices of $x_t \in \mathbb{R}$, the value:

$$\mathbb{E}[\frac{-e^{-aW_T}}{a} | (t, W_t)] \tag{6.19}$$

We formulate this problem as a *Continuous States* and *Continuous Actions* discrete-time finite-horizon MDP by specifying it's *State Transitions*, *Rewards* and *Discount Factor* precisely. The problem then is to solve the MDP's Control problem to find the Optimal Policy.

The terminal time for the finite-horizon MDP is T and hence, all the states at time $t = T$ are terminal states. We shall follow the notation of finite-horizon MDPs that we had covered in Section 3 of Chapter 3. The *State* $s_t \in \mathcal{S}_t$ at any time step $t = 0, 1, \ldots, T$ consists of the wealth W_t. The decision (*Action*) $a_t \in \mathcal{A}_t$ at any time step $t = 0, 1, \ldots, T-1$ is the quantity of investment in the risky asset $(= x_t)$. Hence, the quantity of investment in the riskless asset at time t will be $W_t - x_t$. A deterministic policy at time t (for all $t = 0, 1, \ldots T-1$) is denoted as π_t, and hence, we write: $\pi_t(W_t) = x_t$. Likewise, an optimal deterministic policy at time t (for all $t = 0, 1, \ldots, T-1$) is denoted as π^*_t, and hence, we write: $\pi^*_t(W_t) = x^*_t$.

Denote the random variable for the single-time-step return of the risky asset from time t to time $t+1$ as $Y_t \sim \mathcal{N}(\mu, \sigma^2)$ for all $t = 0, 1, \ldots T-1$. So,

$$W_{t+1} = x_t \cdot (1 + Y_t) + (W_t - x_t) \cdot (1 + r) = x_t \cdot (Y_t - r) + W_t \cdot (1 + r) \tag{6.20}$$

for all $t = 0, 1, \ldots, T-1$.

The MDP *Reward* is 0 for all $t = 0, 1, \ldots, T-1$. As a result of the simplified objective (6.19) above, the MDP *Reward* for $t = T$ is the following random quantity (conditional on state W_{T-1} and action x_{T-1}):

$$\frac{-e^{-aW_T}}{a}$$

We set the MDP discount factor to be $\gamma = 1$ (again, because of the simplified objective (6.19) above).

We denote the Value Function at time t (for all $t = 0, 1, \ldots, T-1$) for a given policy $\pi = (\pi_0, \pi_1, \ldots, \pi_{T-1})$ as:

$$V_t^\pi(W_t) = \mathbb{E}_\pi[\frac{-e^{-aW_T}}{a}|(t, W_t)]$$

We denote the Optimal Value Function at time t (for all $t = 0, 1, \ldots, T-1$) as:

$$V_t^*(W_t) = \max_\pi V_t^\pi(W_t) = \max_\pi \{\mathbb{E}_\pi[\frac{-e^{-aW_T}}{a}|(t, W_t)]\}$$

The Bellman Optimality Equation is:

$$V_t^*(W_t) = \max_{x_t} Q_t^*(W_t, x_t) = \max_{x_t} \{\mathbb{E}_{Y_t \sim \mathcal{N}(\mu, \sigma^2)}[V_{t+1}^*(W_{t+1})]\}$$

for all $t = 0, 1, \ldots, T-2$, and

$$V_{T-1}^*(W_{T-1}) = \max_{x_{T-1}} Q_{T-1}^*(W_{T-1}, x_{T-1}) = \max_{x_{T-1}} \{\mathbb{E}_{Y_{T-1} \sim \mathcal{N}(\mu, \sigma^2)}[\frac{-e^{-aW_T}}{a}]\}$$

where Q_t^* is the Optimal Action-Value Function at time t for all $t = 0, 1, \ldots, T-1$.

We make an educated guess for the functional form of the Optimal Value Function as:

$$V_t^*(W_t) = -b_t \cdot e^{-c_t \cdot W_t} \tag{6.21}$$

where b_t, c_t are independent of the wealth W_t for all $t = 0, 1, \ldots, T-1$. Next, we express the Bellman Optimality Equation using this functional form for the Optimal Value Function:

$$V_t^*(W_t) = \max_{x_t} \{\mathbb{E}_{Y_t \sim \mathcal{N}(\mu, \sigma^2)}[-b_{t+1} \cdot e^{-c_{t+1} \cdot (x_t \cdot (Y_t - r) + W_t \cdot (1+r))}]\}$$

Using Equation (6.20), we can write this as:

$$V_t^*(W_t) = \max_{x_t}\{-b_{t+1} \cdot e^{-c_{t+1}\cdot(1+r)\cdot W_t - c_{t+1}\cdot(\mu-r)\cdot x_t + c_{t+1}^2 \cdot \frac{\sigma^2}{2}\cdot x_t^2}\} \tag{6.22}$$

Since $V_t^*(W_t) = \max_{x_t} Q_t^*(W_t, x_t)$, from Equation (6.22), we can infer the functional form for $Q_t^*(W_t, x_t)$ in terms of b_{t+1} and c_{t+1}:

$$Q_t^*(W_t, x_t) = -b_{t+1} \cdot e^{-c_{t+1}\cdot(1+r)\cdot W_t - c_{t+1}\cdot(\mu-r)\cdot x_t + c_{t+1}^2 \cdot \frac{\sigma^2}{2}\cdot x_t^2} \tag{6.23}$$

Since the right-hand-side of the Bellman Optimality Equation (6.22) involves a max over x_t, we can say that the partial derivative of the term inside the max with respect to x_t is 0. This enables us to write the Optimal Allocation x_t^* in terms of c_{t+1}, as follows:

$$-c_{t+1} \cdot (\mu - r) + \sigma^2 \cdot c_{t+1}^2 \cdot x_t^* = 0$$
$$\Rightarrow x_t^* = \frac{\mu - r}{\sigma^2 \cdot c_{t+1}} \tag{6.24}$$

Next we substitute this maximizing x_t^* in the Bellman Optimality Equation (Equation (6.22)):

$$V_t^*(W_t) = -b_{t+1} \cdot e^{-c_{t+1}\cdot(1+r)\cdot W_t - \frac{(\mu-r)^2}{2\sigma^2}}$$

But since

$$V_t^*(W_t) = -b_t \cdot e^{-c_t \cdot W_t}$$

we can write the following recursive equations for b_t and c_t:

$$b_t = b_{t+1} \cdot e^{-\frac{(\mu-r)^2}{2\sigma^2}}$$
$$c_t = c_{t+1} \cdot (1 + r)$$

We can calculate b_{T-1} and c_{T-1} from the knowledge of the MDP *Reward* $\frac{-e^{-aW_T}}{a}$ (Utility of Terminal Wealth) at time $t = T$, which will enable us to unroll the above recursions for b_t and c_t for all $t = 0, 1, \ldots, T-2$.

$$V_{T-1}^*(W_{T-1}) = \max_{x_{T-1}}\{\mathbb{E}_{Y_{T-1}\sim\mathcal{N}(\mu,\sigma^2)}[\frac{-e^{-aW_T}}{a}]\}$$

From Equation (6.20), we can write this as:

$$V_{T-1}^*(W_{T-1}) = \max_{x_{T-1}}\{\mathbb{E}_{Y_{T-1}\sim\mathcal{N}(\mu,\sigma^2)}[\frac{-e^{-a(x_{T-1}\cdot(Y_{T-1}-r)+W_{T-1}\cdot(1+r))}}{a}]\}$$

Using the result in Equation (A.9) in Appendix A, we can write this as:

$$V_{T-1}^*(W_{T-1}) = \frac{-e^{-\frac{(\mu-r)^2}{2\sigma^2} - a\cdot(1+r)\cdot W_{T-1}}}{a}$$

Therefore,

$$b_{T-1} = \frac{e^{-\frac{(\mu-r)^2}{2\sigma^2}}}{a}$$

$$c_{T-1} = a \cdot (1+r)$$

Now we can unroll the above recursions for b_t and c_t for all $t = 0, 1, \ldots T-2$ as:

$$b_t = \frac{e^{-\frac{(\mu-r)^2 \cdot (T-t)}{2\sigma^2}}}{a}$$

$$c_t = a \cdot (1+r)^{T-t}$$

Substituting the solution for c_{t+1} in Equation (6.24) gives us the solution for the Optimal Policy:

$$\pi_t^*(W_t) = x_t^* = \frac{\mu - r}{\sigma^2 \cdot a \cdot (1+r)^{T-t-1}} \tag{6.25}$$

for all $t = 0, 1, \ldots, T-1$. Note that the optimal action at time step t (for all $t = 0, 1, \ldots, T-1$) does not depend on the state W_t at time t (it only depends on the time t). Hence, the optimal policy $\pi_t^*(\cdot)$ for a fixed time t is a constant deterministic policy function.

Substituting the solutions for b_t and c_t in Equation (6.21) gives us the solution for the Optimal Value Function:

$$V_t^*(W_t) = \frac{-e^{-\frac{(\mu-r)^2 (T-t)}{2\sigma^2}}}{a} \cdot e^{-a(1+r)^{T-t} \cdot W_t} \tag{6.26}$$

for all $t = 0, 1, \ldots, T-1$.

Substituting the solutions for b_{t+1} and c_{t+1} in Equation (6.23) gives us the solution for the Optimal Action-Value Function:

$$Q_t^*(W_t, x_t) = \frac{-e^{-\frac{(\mu-r)^2 (T-t-1)}{2\sigma^2}}}{a} \cdot e^{-a(1+r)^{T-t} \cdot W_t - a(\mu-r)(1+r)^{T-t-1} \cdot x_t + \frac{(a\sigma(1+r)^{T-t-1})^2}{2} \cdot x_t^2} \tag{6.27}$$

for all $t = 0, 1, \ldots, T-1$.

Porting to Real-World

We have covered a continuous-time setting and a discrete-time setting with simplifying assumptions that provide analytical tractability. The specific simplifying assumptions that enabled analytical tractability were:

- Normal distribution of asset returns
- CRRA/CARA assumptions
- Frictionless markets/trading (no transaction costs, unconstrained and continuous prices/allocation amounts/consumption)

But real-world problems involving dynamic asset-allocation and consumption are not so simple and clean. We have arbitrary, more complex asset price movements. Utility functions don't fit into simple CRRA/CARA formulas. In practice, trading often occurs in discrete space - asset prices, allocation amounts and consumption are often discrete quantities. Moreover, when we change our asset allocations or liquidate a portion of our portfolio to consume, we incur transaction costs. Furthermore, trading doesn't always happen in continuous-time - there are typically specific windows of time where one is locked-out from trading or there are trading restrictions. Lastly, many investments are illiquid (eg: real-estate) or simply not allowed to be liquidated until a certain horizon (eg: retirement funds), which poses major constraints on extracting money from one's portfolio for consumption. So even though prices/allocation amounts/consumption might be close to being continuous-variables, the other above-mentioned frictions mean that we don't get the benefits of calculus that we obtained in the simple examples we covered.

With the above real-world considerations, we need to tap into Dynamic Programming - more specifically, Approximate Dynamic Programming since real-world problems have large state spaces and large action spaces (even if these spaces are not continuous, they tend to be close to continuous). Appropriate function approximation of the Value Function is key to solving these problems. Implementing a full-blown real-world investment and consumption management system is beyond the scope of this book, but let us implement an illustrative example that provides sufficient understanding of how a full-blown real-world example would be implemented. We have to keep things simple enough and yet sufficiently general. So here is the setting we will implement:

- One risky asset and one riskless asset.
- Finite number of time steps (discrete-time setting akin to Section 6).
- No consumption (i.e., no extraction from the investment portfolio) until the end of the finite horizon, and hence, without loss of generality, we set the discount factor equal to 1.
- Arbitrary distribution of return for the risky asset, and allowing the distribution of returns to change in time (expressed as `risky_return_distributions: Sequence[Distribution[float]]` in the code below).
- Allowing the return on the riskless asset to vary in time (expressed as `riskless_returns: Sequence[float]` in the code below).
- Arbitrary Utility Function (expressed as `utility_func: Callable[[float], float]` in the code below).
- Finite number of choices of investment amounts in the risky asset at each time step (expressed as `risky_alloc_choices: Sequence[float]` in the code below).
- Arbitrary probability distribution of the initial wealth W_0 (expressed as `initial_wealth_distribution: Distribution[float]` in the code below).

The code in the class `AssetAllocDiscrete` below is fairly self-explanatory. We use the function `back_opt_qvf` covered in Section 4 of Chapter **??** to perform backward induction on the optimal Q-Value Function. Since the state space is continuous, the optimal Q-Value Function is represented as a FunctionApprox (specifically, as a DNNApprox). Moreover, since we are working with a generic distribution of returns that govern the state transitions of this MDP, we need to work with the methods of the abstract class MarkovDecisionProcess (and not the class FiniteMarkovDecisionProcess). The method backward_induction_qvf below makes the call to back_opt_qvf. Since the risky returns distribution is arbitrary and since the utility function is arbitrary, we don't have prior knowledge of the functional form of the Q-Value function. Hence, the user of the class AssetAllocDiscrete also needs to provide the set of feature functions (feature_functions in the code below) and the specification of a deep neural network to represent the Q-Value function (dnn_spec in the code below). The rest of the code below is mainly about preparing the input mdp_f0_mu_triples to be passed to back_opt_qvf. As was explained in Section 4 of Chapter **??**, mdp_f0_mu_triples is a sequence (for each time step) of the following triples:

- A MarkovDecisionProcess[float, float] object, which in the code below is prepared by the method get_mdp. *State* is the portfolio wealth (float type) and *Action* is the quantity of investment in the risky asset (also of float type). get_mdp creates a class AssetAllocMDP that implements the abstract class MarkovDecisionProcess. To do so, we need to implement the step method and the actions method. The step method returns an instance of SampledDistribution, which is based on the sr_sampler_func that returns a sample of the pair of next state (next time step's wealth) and reward, given the current state (current wealth) and action (current time step's quantity of investment in the risky asset).
- A DNNApprox[Tuple[float, float]] object, which in the code below is prepared by the method get_qvf_func_approx. This method sets up a 'DNNApprox' object that represents a neural-network function approximation for the optimal Q-Value Function. So the input to this neural network would be a Tuple[float, float] representing a (state, action) pair.
- A Distribution[float] object, which in the code below is prepared by the method get_states_distribution. This method constructs a SampledDistribution[floa representing the distribution of states (distribution of portfolio wealth) at each time step. The SampledDistribution[float] is prepared using the function states_sampler_func that generates a sampling trace by sampling the state-transitions (portfolio wealth transitions) from time 0 to the given time step in a time-incremental manner (invoking the sample method of the risky asset's return Distributions and the sample method of a uniform distribution over the action choices specified by risky_alloc_choices).

```
from rl.distribution import Distribution, SampledDistribution, Choose
from rl.function_approx import DNNSpec, AdamGradient, DNNApprox
from rl.approximate_dynamic_programming import back_opt_qvf
from operator import itemgetter
import numpy as np
```

```python
@dataclass(frozen=True)
class AssetAllocDiscrete:
    risky_return_distributions: Sequence[Distribution[float]]
    riskless_returns: Sequence[float]
    utility_func: Callable[[float], float]
    risky_alloc_choices: Sequence[float]
    feature_functions: Sequence[Callable[[Tuple[float, float]], float]]
    dnn_spec: DNNSpec
    initial_wealth_distribution: Distribution[float]

    def time_steps(self) -> int:
        return len(self.risky_return_distributions)

    def uniform_actions(self) -> Choose[float]:
        return Choose(set(self.risky_alloc_choices))

    def get_mdp(self, t: int) -> MarkovDecisionProcess[float, float]:
        distr: Distribution[float] = self.risky_return_distributions[t]
        rate: float = self.riskless_returns[t]
        alloc_choices: Sequence[float] = self.risky_alloc_choices
        steps: int = self.time_steps()
        utility_f: Callable[[float], float] = self.utility_func

        class AssetAllocMDP(MarkovDecisionProcess[float, float]):

            def step(
                self,
                wealth: float,
                alloc: float
            ) -> SampledDistribution[Tuple[float, float]]:

                def sr_sampler_func(
                    wealth=wealth,
                    alloc=alloc
                ) -> Tuple[float, float]:
                    next_wealth: float = alloc * (1 + distr.sample()) \
                        + (wealth - alloc) * (1 + rate)
                    reward: float = utility_f(next_wealth) \
                        if t == steps - 1 else 0.
                    return (next_wealth, reward)

                return SampledDistribution(
                    sampler=sr_sampler_func,
                    expectation_samples=1000
                )
```

```
            def actions(self, wealth: float) -> Sequence[float]:
                return alloc_choices

        return AssetAllocMDP()

    def get_qvf_func_approx(self) -> DNNApprox[Tuple[float, float]]:
        adam_gradient: AdamGradient = AdamGradient(
            learning_rate=0.1,
            decay1=0.9,
            decay2=0.999
        )
        return DNNApprox.create(
            feature_functions=self.feature_functions,
            dnn_spec=self.dnn_spec,
            adam_gradient=adam_gradient
        )

    def get_states_distribution(self, t: int) -> SampledDistribution[float]:
        actions_distr: Choose[float] = self.uniform_actions()

        def states_sampler_func() -> float:
            wealth: float = self.initial_wealth_distribution.sample()
            for i in range(t):
                distr: Distribution[float] = self.risky_return_distributions[i]
                rate: float = self.riskless_returns[i]
                alloc: float = actions_distr.sample()
                wealth = alloc * (1 + distr.sample()) + \
                    (wealth - alloc) * (1 + rate)
            return wealth

        return SampledDistribution(states_sampler_func)

    def backward_induction_qvf(self) -> \
            Iterator[DNNApprox[Tuple[float, float]]]:
        init_fa: DNNApprox[Tuple[float, float]] = self.get_qvf_func_approx()
        mdp_f0_mu_triples: Sequence[Tuple[
            MarkovDecisionProcess[float, float],
            DNNApprox[Tuple[float, float]],
            SampledDistribution[float]
        ]] = [(
            self.get_mdp(i),
            init_fa,
            self.get_states_distribution(i)
        ) for i in range(self.time_steps())]
```

```
        num_state_samples: int = 300
        error_tolerance: float = 1e-6

        return back_opt_qvf(
            mdp_f0_mu_triples=mdp_f0_mu_triples,
            gamma=1.0,
            num_state_samples=num_state_samples,
            error_tolerance=error_tolerance
        )
```

The above code is in the file rl/chapter7/asset_alloc_discrete.py. We encourage you to create a few different instances of AssetAllocDiscrete by varying it's inputs (try different return distributions, different utility functions, different action spaces). But how do we know the code above is correct? We need a way to test it. A good test is to specialize the inputs to fit the setting of Section 6 for which we have a closed-form solution to compare against. So let us write some code to specialize the inputs to fit this setting. Since the above code has been written with an educational motivation rather than an efficient-computation motivation, the convergence of the backward induction ADP algorithm is going to be slow. So we shall test it on a small number of time steps and provide some assistance for fast convergence (using limited knowledge from the closed-form solution in specifying the function approximation). We write code below to create an instance of AssetAllocDiscrete with time steps $T = 4$, $\mu = 13\%, \sigma = 20\%, r = 7\%$, coefficient of CARA $a = 1.0$. We set up risky_return_distributions as a sequence of identical Gaussian distributions, riskless_returns as a sequence of identical riskless rate of returns, and utility_func as a lambda parameterized by the coefficient of CARA a. We know from the closed-form solution that the optimal allocation to the risky asset for each of time steps $t = 0, 1, 2, 3$ is given by:

$$x_t^* = \frac{1.5}{1.07^{4-t}}$$

Therefore, we set risky_alloc_choices (action choices) in the range [1.0, 2.0] in increments of 0.1 to see if our code can hit the correct values within the 0.1 granularity of action choices.

To specify feature_functions and dnn_spec, we need to leverage the functional form of the closed-form solution for the Action-Value function (i.e., Equation (6.27)). We observe that we can write this as:

$$Q_t^*(W_t, x_t) = -sign(a) \cdot e^{-(\alpha_0 + \alpha_1 \cdot W_t + \alpha_2 \cdot x_t + \alpha_3 \cdot x_t^2)}$$

where

$$\alpha_0 = \frac{(\mu - r)^2 (T - t - 1)}{2\sigma^2} + \log(|a|)$$

$$\alpha_1 = a(1 + r)^{T-t}$$

$$\alpha_2 = a(\mu - r)(1 + r)^{T-t-1}$$

$$\alpha_3 = -\frac{(a\sigma(1+r)^{T-t-1})^2}{2}$$

This means, the function approximation for Q_t^* can be set up with a neural network with no hidden layers, with the output layer activation function as $g(S) = -sign(a) \cdot e^{-S}$, and with the feature functions as:

$$\phi_1((W_t, x_t)) = 1$$
$$\phi_2((W_t, x_t)) = W_t$$
$$\phi_3((W_t, x_t)) = x_t$$
$$\phi_4((W_t, x_t)) = x_t^2$$

We set `initial_wealth_distribution` to be a normal distribution with a mean of `init_wealth` (set equal to 1.0 below) and a standard distribution of `init_wealth_var` (set equal to a small value of 0.1 below).

```
from rl.distribution import Gaussian

steps: int = 4
mu: float = 0.13
sigma: float = 0.2
r: float = 0.07
a: float = 1.0
init_wealth: float = 1.0
init_wealth_var: float = 0.1

excess: float = mu - r
var: float = sigma * sigma
base_alloc: float = excess / (a * var)

risky_ret: Sequence[Gaussian] = [Gaussian(mu=mu, sigma=sigma)
                                 for _ in range(steps)]
riskless_ret: Sequence[float] = [r for _ in range(steps)]
utility_function: Callable[[float], float] = lambda x: - np.exp(-a * x) / a
alloc_choices: Sequence[float] = np.linspace(
    2 / 3 * base_alloc,
    4 / 3 * base_alloc,
    11
)
feature_funcs: Sequence[Callable[[Tuple[float, float]], float]] = \
    [
        lambda _: 1.,
        lambda w_x: w_x[0],
        lambda w_x: w_x[1],
        lambda w_x: w_x[1] * w_x[1]
    ]
```

```python
dnn: DNNSpec = DNNSpec(
    neurons=[],
    bias=False,
    hidden_activation=lambda x: x,
    hidden_activation_deriv=lambda y: np.ones_like(y),
    output_activation=lambda x: - np.sign(a) * np.exp(-x),
    output_activation_deriv=lambda y: -y
)
init_wealth_distr: Gaussian = Gaussian(
    mu=init_wealth,
    sigma=init_wealth_var
)

aad: AssetAllocDiscrete = AssetAllocDiscrete(
    risky_return_distributions=risky_ret,
    riskless_returns=riskless_ret,
    utility_func=utility_function,
    risky_alloc_choices=alloc_choices,
    feature_functions=feature_funcs,
    dnn_spec=dnn,
    initial_wealth_distribution=init_wealth_distr
)
```

Next, we perform the Q-Value backward induction, step through the returned iterator (fetching the Q-Value function for each time step from $t = 0$ to $t = T - 1$), and evaluate the Q-values at the `init_wealth` (for each time step) for all `alloc_choices`. Performing a max and arg max over the `alloc_choices` at the `init_wealth` gives us the Optimal Value function and the Optimal Policy for each time step for wealth equal to `init_wealth`.

```python
from pprint import pprint

it_qvf: Iterator[DNNApprox[Tuple[float, float]]] = \
    aad.backward_induction_qvf()

for t, q in enumerate(it_qvf):
    print(f"Time {t:d}")
    print()
    opt_alloc: float = max(
        ((q.evaluate([(init_wealth, ac)])[0], ac) for ac in alloc_choices),
        key=itemgetter(0)
    )[1]
    val: float = max(q.evaluate([(init_wealth, ac)])[0]
                     for ac in alloc_choices)
    print(f"Opt Risky Allocation = {opt_alloc:.3f}, Opt Val = {val:.3f}")
    print("Optimal Weights below:")
```

```
    for wts in q.weights:
        pprint(wts.weights)
    print()
```

This prints the following:

```
Time 0

Opt Risky Allocation = 1.200, Opt Val = -0.225
Optimal Weights below:
array([[ 0.13318188,  1.31299678,  0.07327264, -0.03000281]])

Time 1

Opt Risky Allocation = 1.300, Opt Val = -0.257
Optimal Weights below:
array([[ 0.08912411,  1.22479503,  0.07002802, -0.02645654]])

Time 2

Opt Risky Allocation = 1.400, Opt Val = -0.291
Optimal Weights below:
array([[ 0.03772409,  1.144612  ,  0.07373166, -0.02566819]])

Time 3

Opt Risky Allocation = 1.500, Opt Val = -0.328
Optimal Weights below:
array([[ 0.00126822,  1.0700996 ,  0.05798272, -0.01924149]])
```

Now let's compare these results against the closed-form solution.

```
for t in range(steps):
    print(f"Time {t:d}")
    print()
    left: int = steps - t
    growth: float = (1 + r) ** (left - 1)
    alloc: float = base_alloc / growth
    val: float = - np.exp(- excess * excess * left / (2 * var)
                          - a * growth * (1 + r) * init_wealth) / a
    bias_wt: float = excess * excess * (left - 1) / (2 * var) + \
        np.log(np.abs(a))
    w_t_wt: float = a * growth * (1 + r)
    x_t_wt: float = a * excess * growth
    x_t2_wt: float = - var * (a * growth) ** 2 / 2

    print(f"Opt Risky Allocation = {alloc:.3f}, Opt Val = {val:.3f}")
```

```
    print(f"Bias Weight = {bias_wt:.3f}")
    print(f"W_t Weight = {w_t_wt:.3f}")
    print(f"x_t Weight = {x_t_wt:.3f}")
    print(f"x_t^2 Weight = {x_t2_wt:.3f}")
    print()
```

This prints the following:

```
Time 0

Opt Risky Allocation = 1.224, Opt Val = -0.225
Bias Weight = 0.135
W_t Weight = 1.311
x_t Weight = 0.074
x_t^2 Weight = -0.030

Time 1

Opt Risky Allocation = 1.310, Opt Val = -0.257
Bias Weight = 0.090
W_t Weight = 1.225
x_t Weight = 0.069
x_t^2 Weight = -0.026

Time 2

Opt Risky Allocation = 1.402, Opt Val = -0.291
Bias Weight = 0.045
W_t Weight = 1.145
x_t Weight = 0.064
x_t^2 Weight = -0.023

Time 3

Opt Risky Allocation = 1.500, Opt Val = -0.328
Bias Weight = 0.000
W_t Weight = 1.070
x_t Weight = 0.060
x_t^2 Weight = -0.020
```

As mentioned previously, this serves as a good test for the correctness of the implementation of `AssetAllocDiscrete`.
We need to point out here that the general case of dynamic asset allocation and consumption for a large number of risky assets will involve a continuously-valued action space of high dimension. This means ADP algorithms will have challenges in performing the $\max / \arg\max$ calculation across this large and continuous action space. Even many of the RL algorithms find it challenging

to deal with very large action spaces. Sometimes we can take advantage of the specifics of the control problem to overcome this challenge. But in a general setting, these large/continuous action space require special types of RL algorithms that are well suited to tackle such action spaces. One such class of RL algorithms is Policy Gradient Algorithms that we shall learn in Chapter 12.

Key Takeaways from this Chapter

- A fundamental problem in Mathematical Finance is that of jointly deciding on A) optimal investment allocation (among risky and riskless investment assets) and B) optimal consumption, over a finite horizon. Merton, in his landmark paper from 1969, provided an elegant closed-form solution under assumptions of continuous-time, normal distribution of returns on the assets, CRRA utility, and frictionless transactions.
- In a more general setting of the above problem, we need to model it as an MDP. If the MDP is not too large and if the asset return distributions are known, we can employ finite-horizon ADP algorithms to solve it. However, in typical real-world situations, the action space can be quite large and the asset return distributions are unknown. This points to RL, and specifically RL algorithms that are well suited to tackle large action spaces (such as Policy Gradient Algorithms).

7. Derivatives Pricing and Hedging

In this chapter, we cover two applications of MDP Control regarding financial derivatives pricing and hedging (the word *hedging* refers to reducing or eliminating market risks associated with a derivative). The first application is to identify the optimal time/state to exercise an American Option (a type of financial derivative) in an idealized market setting (akin to the "frictionless" market setting of Merton's Portfolio problem from Chapter 6). Optimal exercise of an American Option is the key to determining it's fair price. The second application is to identify the optimal hedging strategy for derivatives in real-world situations (technically refered to as *incomplete markets*, a term we will define shortly). The optimal hedging strategy of a derivative is the key to determining it's fair price in the real-world (incomplete market) setting. Both of these applications can be cast as Markov Decision Processes where the Optimal Policy gives the Optimal Hedging/Optimal Exercise in the respective applications, leading to the fair price of the derivatives under consideration. Casting these derivatives applications as MDPs means that we can tackle them with Dynamic Programming or Reinforcement Learning algorithms, providing an interesting and valuable alternative to the traditional methods of pricing derivatives.

In order to understand and appreciate the modeling of these derivatives applications as MDPs, one requires some background in the classical theory of derivatives pricing. Unfortunately, thorough coverage of this theory is beyond the scope of this book and we refer you to Tomas Bjork's book on Arbitrage Theory in Continuous Time for a thorough understanding of this theory. We shall spend much of this chapter covering the very basics of this theory, and in particular explaining the key technical concepts (such as arbitrage, replication, risk-neutral measure, market-completeness etc.) in a simple and intuitive manner. In fact, we shall cover the theory for the very simple case of discrete-time with a single-period. While that is nowhere near enough to do justice to the rich continuous-time theory of derivatives pricing and hedging, this is the best we can do in a single chapter. The good news is that MDP-modeling of the two problems we want to solve - optimal exercise of american options and optimal hedging of derivatives in a real-world (incomplete market) setting - doesn't require one to have a thorough understanding of the classical theory. Rather, an intuitive understanding of the key technical and economic concepts should suffice, which we bring to life in the simple setting of discrete-time with a single-period. We start this chapter with a quick introduction to derivatives, next we describe the simple setting of a single-period with formal mathematical notation, covering the key concepts (arbitrage, replication, risk-neutral measure, market-completeness etc.), state and prove the all-important fundamental theorems of asset pricing (only for the single-period setting), and finally show

how these two derivatives applications can be cast as MDPs, along with the appropriate algorithms to solve the MDPs.

A Brief Introduction to Derivatives

If you are reading this book, you likely already have some familiarity with Financial Derivatives (or at least have heard of them, given that derivatives were at the center of the 2008 financial crisis). In this section, we sketch an overview of financial derivatives and refer you to the book by John Hull for a thorough coverage of Derivatives. The term "Derivative" is based on the word "derived" - it refers to the fact that a derivative is a financial instrument whose structure and hence, value is derived from the *performance* of an underlying entity or entities (which we shall simply refer to as "underlying"). The underlying can be pretty much any financial entity - it could be a stock, currency, bond, basket of stocks, or something more exotic like another derivative. The term *performance* also refers to something fairly generic - it could be the price of a stock or commodity, it could be the interest rate a bond yields, it could be average price of a stock over a time interval, it could be a market-index, or it could be something more exotic like the implied volatility of an option (which itself is a type of derivative). Technically, a derivative is a legal contract between the derivative buyer and seller that either:

- Entitles the derivative buyer to cashflow (which we'll refer to as derivative *payoff*) at future point(s) in time, with the payoff being contingent on the underlying's performance (i.e., the payoff is a precise mathematical function of the underlying's performance, eg: a function of the underlying's price at a future point in time). This type of derivative is known as a "lock-type" derivative.
- Provides the derivative buyer with choices at future points in time, upon making which, the derivative buyer can avail of cashflow (i.e., *payoff*) that is contingent on the underlying's performance. This type of derivative is known as an "option-type" derivative (the word "option" refering to the choice or choices the buyer can make to trigger the contingent payoff).

Although both "lock-type" and "option-type" derivatives can both get very complex (with contracts running over several pages of legal descriptions), we now illustrate both these types of derivatives by going over the most basic derivative structures. In the following descriptions, current time (when the derivative is bought/sold) is denoted as time $t = 0$.

Forwards

The most basic form of Forward Contract involves specification of:

- A future point in time $t = T$ (we refer to T as expiry of the forward contract).

- The fixed payment K to be made by the forward contract buyer to the seller at time $t = T$.

In addition, the contract establishes that at time $t = T$, the forward contract seller needs to deliver the underlying (say a stock with price S_t at time t) to the forward contract buyer. This means at time $t = T$, effectively the payoff for the buyer is $S_T - K$ (likewise, the payoff for the seller is $K - S_T$). This is because the buyer, upon receiving the underlying from the seller, can immediately sell the underlying in the market for the price of S_T and so, would have made a gain of $S_T - K$ (note $S_T - K$ can be negative, in which case the payoff for the buyer is negative).

The problem of forward contract "pricing" is to determine the fair value of K so that the price of this forward contract derivative at the time of contract creation is 0. As time t progresses, the underlying price might fluctuate, which would cause a movement away from the initial price of 0. If the underlying price increases, the price of the forward would naturally increase (and if the underlying price decreases, the price of the forward would naturally decrease). This is an example of a "lock-type" derivative since neither the buyer nor the seller of the forward contract need to make any choices. Rather, the payoff for the buyer is determined directly by the formula $S_T - K$ and the payoff for the seller is determined by the formula $K - S_T$.

European Options

The most basic forms of European Options are European Call and Put Options. The most basic European Call Option contract involves specification of:

- A future point in time $t = T$ (we refer to T as the expiry of the Call Option).
- Underlying Price K known as strike.

The contract gives the buyer (owner) of the European Call Option the right, but not the obligation, to buy the underlying at time $t = T$ for the price of K. Since the option owner doesn't have the obligation to buy, if the price S_T of the underlying at time $t = T$ ends up being equal to or below K, the rational decision for the option owner would be to not buy (at price K), which would result in a payoff of 0 (in this outcome, we say that the call option is *out-of-the-money*). However, if $S_T > K$, the option owner would make an instant profit of $S_T - K$ by *exercising* her right to buy the underlying at the price of K. Hence, the payoff in this case is $S_T - K$ (in this outcome, we say that the call option is *in-the-money*). We can combine the two cases and say that the payoff is $f(S_T) = \max(S_T - K, 0)$. Since the payoff is always non-negative, the call option owner would need to pay for this privilege. The amount the option owner would need to pay to own this call option is known as the fair price of the call option. Identifying the value of this fair price is the highly celebrated problem of *Option Pricing* (which you will learn more about as this chapter progresses).

A European Put Option is very similar to a European Call Option with the only difference being that the owner of the European Put Option has the right (but not the obligation) to *sell* the underlying at time $t = T$ for the price of K. This means that the payoff is $f(S_T) = \max(K - S_T, 0)$. Payoffs for these Call and Put Options are known as "hockey-stick" payoffs because if you plot the $f(\cdot)$ function, it is a flat line on the *out-of-the-money* side and a sloped line on the *in-the-money* side. Such European Call and Put Options are "Option-Type" (and not "Lock-Type") derivatives since they involve a choice to be made by the option owner (the choice of exercising the right to buy/sell at the strike price K). However, it is possible to construct derivatives with the same payoff as these European Call/Put Options by simply writing in the contract that the option owner will get paid $\max(S_T - K, 0)$ (in case of Call Option) or will get paid $\max(K - S_T, 0)$ (in case of Put Option) at time $t = T$. Such derivatives contracts do away with the option owner's exercise choice and hence, they are "Lock-Type" contracts. There is a subtle difference - setting these derivatives up as "Option-Type" means the option owner might act "irrationally" - the call option owner might mistakenly buy even if $S_T < K$, or the call option owner might for some reason forget/neglect to exercise her option even when $S_T > K$. Setting up such contracts as "Lock-Type" takes away the possibilities of these types of irrationalities from the option owner. However, note that the typical European Call and Put Options are set up as "Option-Type" contracts.

A more general European Derivative involves an arbitrary function $f(\cdot)$ (generalizing from the hockey-stick payoffs) and could be set up as "Option-Type" or "Lock-Type".

American Options

The term "European" above refers to the fact that the option to exercise is available only at a fixed point in time $t = T$. Even if it is set up as "Lock-Type", the term "European" typically means that the payoff can happen only at a fixed point in time $t = T$. This is in contrast to American Options. The most basic forms of American Options are American Call and Put Options. American Call and Put Options are essentially extensions of the corresponding European Call and Put Options by allowing the buyer (owner) of the American Option to exercise the option to buy (in the case of Call) or sell (in the case of Put) at any time $t \leq T$. The allowance of exercise at any time at or before the expiry time T can often be a tricky financial decision for the option owner. At each point in time when the American Option is *in-the-money* (i.e., positive payoff upon exercise), the option owner might be tempted to exercise and collect the payoff but might as well be thinking that if she waits, the option might become more *in-the-money* (i.e., prospect of a bigger payoff if she waits for a while). Hence, it's clear that an American Option is always of the "Option-Type" (and not "Lock-Type") since the timing of the decision (option) to exercise is very important in the case of an American Option. This also means that the problem of pricing an American Option (the fair price the buyer would need to pay to own an American Option) is much harder than the problem of pricing a European Option.

So what purpose do derivatives serve? There are actually many motivations for different market participants, but we'll just list two key motivations. The first reason is to protect against adverse market movements that might damage the value of one's portfolio (this is known as *hedging*). As an example, buying a put option can reduce or eliminate the risk associated with ownership of the underlying. The second reason is operational or financial convenience in trading to express a speculative view of market movements. For instance, if one thinks a stock will increase in value by 50% over the next two years, instead of paying say $100,000 to buy the stock (hoping to make $50,000 after two years), one can simply buy a call option on $100,000 of the stock (paying the option price of say $5,000). If the stock price indeed appreciates by 50% after 2 years, one makes $50,000 - $5,000 = $45,000. Although one made $5000 less than the alternative of simply buying the stock, the fact that one needs to pay $5000 (versus $50,000) to enter into the trade means the potential *return on investment* is much higher.

Next, we embark on the journey of learning how to value derivatives, i.e., how to figure out the fair price that one would be willing to buy or sell the derivative for at any point in time. As mentioned earlier, the general theory of derivatives pricing is quite rich and elaborate (based on continuous-time stochastic processes) but beyond the scope of this book. Instead, we will provide intuition for the core concepts underlying derivatives pricing theory in the context of a simple, special case - that of discrete-time with a single-period. We formalize this simple setting in the next section.

Notation for the Single-Period Simple Setting

Our simple setting involves discrete time with a single-period from $t = 0$ to $t = 1$. Time $t = 0$ has a single state which we shall refer to as the "Spot" state. Time $t = 1$ has n random outcomes formalized by the sample space $\Omega = \{\omega_1, \ldots, \omega_n\}$. The probability distribution of this finite sample space is given by the probability mass function

$$\mu : \Omega \rightarrow [0, 1]$$

such that

$$\sum_{i=1}^{n} \mu(\omega_i) = 1$$

This simple single-period setting involves $m+1$ fundamental assets $A_0, A_1, \ldots, A_m$ where A_0 is a riskless asset (i.e., it's price will evolve deterministically from $t = 0$ to $t = 1$) and $A_1, \ldots, A_m$ are risky assets. We denote the Spot Price (at $t = 0$) of A_j as $S_j^{(0)}$ for all $j = 0, 1, \ldots, m$. We denote the Price of A_j in ω_i as $S_j^{(i)}$ for all $j = 0, \ldots, m, i = 1, \ldots, n$. Assume that all asset prices are real numbers, i.e., in $\mathbb{R}$ (negative prices are typically unrealistic, but we still assume it for simplicity of exposition). For convenience, we normalize the Spot Price (at $t = 0$) of the riskless asset A_O to be 1. Therefore,

$$S_0^{(0)} = 1 \text{ and } S_0^{(i)} = 1 + r \text{ for all } i = 1, \ldots, n$$

where r represents the constant riskless rate of growth. We should interpret this riskless rate of growth as the "time value of money" and $\frac{1}{1+r}$ as the riskless discount factor corresponding to the "time value of money".

Portfolios, Arbitrage and Risk-Neutral Probability Measure

We define a portfolio as a vector $\theta = (\theta_0, \theta_1, \ldots, \theta_m) \in \mathbb{R}^{m+1}$. We interpret θ_j as the number of units held in asset A_j for all $j = 0, 1, \ldots, m$. The Spot Value (at $t = 0$) of portfolio θ denoted $V_\theta^{(0)}$ is:

$$V_\theta^{(0)} = \sum_{j=0}^{m} \theta_j \cdot S_j^{(0)} \tag{7.1}$$

The Value of portfolio θ in random outcome ω_i (at $t = 1$) denoted $V_\theta^{(i)}$ is:

$$V_\theta^{(i)} = \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)} \text{ for all } i = 1, \ldots, n \tag{7.2}$$

Next, we cover an extremely important concept in Mathematical Economics/Finance, the concept of *Arbitrage*. An Arbitrage Portfolio θ is one that "makes money from nothing". Formally, an arbitrage portfolio is a portfolio θ such that:

- $V_\theta^{(0)} \leq 0$
- $V_\theta^{(i)} \geq 0$ for all $i = 1, \ldots, n$
- There exists an $i \in \{1, \ldots, n\}$ such that $\mu(\omega_i) > 0$ and $V_\theta^{(i)} > 0$

Thus, with an Arbitrage Portfolio, we never end up (at $t = 0$) with less value than what we start with (at $t = 1$) and we end up with expected value strictly greater than what we start with. This is the formalism of the notion of *arbitrage*, i.e., "making money from nothing". Arbitrage allows market participants to make infinite returns. In an efficient market, arbitrage would disappear as soon as it appears since market participants would immediately exploit it and through the process of exploiting the arbitrage, immediately eliminate the arbitrage. Hence, Finance Theory typically assumes "arbitrage-free" markets (i.e., financial markets with no arbitrage opportunities).

Next, we describe another very important concept in Mathematical Economics/Finance, the concept of a *Risk-Neutral Probability Measure*. Consider a Probability Distribution $\pi : \Omega \to [0, 1]$ such that

$$\pi(\omega_i) = 0 \text{ if and only if } \mu(\omega_i) = 0 \text{ for all } i = 1, \ldots, n$$

Then, π is said to be a Risk-Neutral Probability Measure if:

$$S_j^{(0)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot S_j^{(i)} \text{ for all } j = 0, 1, \ldots, m \tag{7.3}$$

So for each of the $m+1$ assets, the asset spot price (at $t=0$) is the riskless rate-discounted expectation (under π) of the asset price at $t=1$. The term "risk-neutral" here is the same as the term "risk-neutral" we used in Chapter 5, meaning it's a situation where one doesn't need to be compensated for taking risk (the situation of a linear utility function). However, we are not saying that the market is risk-neutral - if that were the case, the market probability measure μ would be a risk-neutral probability measure. We are simply defining π as a *hypothetical construct* under which each asset's spot price is equal to the riskless rate-discounted expectation (under π) of the asset's price at $t=1$. This means that under the hypothetical π, there's no return in excess of r for taking on the risk of variables outcomes at $t=1$ (note: outcome probabilities are governed by the hypothetical π). Hence, we refer to π as a risk-neutral probability measure.

Before we cover the two fundamental theorems of asset pricing, we need to cover an important lemma that we will utilize in the proofs of the two fundamental theorems of asset pricing.

Lemma 7.0.1. *For any portfolio $\theta = (\theta_0, \theta_1, \ldots, \theta_m) \in \mathbb{R}^{m+1}$ and any risk-neutral probability measure $\pi : \Omega \rightarrow [0,1]$,*

$$V_\theta^{(0)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_\theta^{(i)}$$

Proof. Using Equations (7.1), (7.3) and (7.2), the proof is straightforward:

$$\begin{aligned} V_\theta^{(0)} &= \sum_{j=0}^{m} \theta_j \cdot S_j^{(0)} = \sum_{j=0}^{m} \theta_j \cdot \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot S_j^{(i)} \\ &= \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_\theta^{(i)} \end{aligned}$$

□

Now we are ready to cover the two fundamental theorems of asset pricing (sometimes, also refered to as the fundamental theorems of arbitrage and the fundamental theorems of finance!). We start with the first fundamental theorem of asset pricing, which associates absence of arbitrage with existence of a risk-neutral probability measure.

First Fundamental Theorem of Asset Pricing (1st FTAP)

Theorem 7.0.2 (First Fundamental Theorem of Asset Pricing (1st FTAP))**.** *Our simple setting of discrete time with single-period will not admit arbitrage portfolios if and only if there exists a Risk-Neutral Probability Measure.*

Proof. First we prove the easy implication - if there exists a Risk-Neutral Probability Measure π , then we cannot have any arbitrage portfolios. Let's review what it takes to have an arbitrage portfolio $\theta = (\theta_0, \theta_1, \ldots, \theta_m)$. The following are two of the three conditions to be satisfied to qualify as an arbitrage portfolio θ (according to the definition of arbitrage portfolio we gave above):

- $V_\theta^{(i)} \geq 0$ for all $i = 1, \ldots, n$
- There exists an $i \in \{1, \ldots, n\}$ such that $\mu(\omega_i) > 0$ $(\Rightarrow \pi(\omega_i) > 0)$ and $V_\theta^{(i)} > 0$

But if these two conditions are satisfied, the third condition $V_\theta^{(0)} \leq 0$ cannot be satisfied because from Lemma (7.0.1), we know that:

$$V_\theta^{(0)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_\theta^{(i)}$$

which is strictly greater than 0, given the two conditions stated above. Hence, all three conditions cannot be simultaneously satisfied which eliminates the possibility of arbitrage for any portfolio θ.

Next, we prove the reverse (harder to prove) implication - if a risk-neutral probability measure doesn't exist, there exists an arbitrage portfolio θ. We define $\mathbb{V} \subset \mathbb{R}^m$ as the set of vectors $v = (v_1, \ldots, v_m)$ such that

$$v_j = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \mu(\omega_i) \cdot S_j^{(i)} \text{ for all } j = 1, \ldots, m$$

with $\mathbb{V}$ defined as spanning over all possible probability distributions $\mu : \Omega \rightarrow [0, 1]$. $\mathbb{V}$ is a bounded, closed, convex polytope in $\mathbb{R}^m$. If a risk-neutral probability measure doesn't exist, the vector $(S_1^{(0)}, \ldots, S_m^{(0)}) \notin \mathbb{V}$. The Hyperplane Separation Theorem implies that there exists a non-zero vector $(\theta_1, \ldots, \theta_m)$ such that for any $v = (v_1, \ldots, v_m) \in \mathbb{V}$,

$$\sum_{j=1}^{m} \theta_j \cdot v_j > \sum_{j=1}^{m} \theta_j \cdot S_j^{(0)}$$

In particular, consider vectors v corresponding to the corners of $\mathbb{V}$, those for which the full probability mass is on a particular $\omega_i \in \Omega$, i.e.,

$$\sum_{j=1}^{m} \theta_j \cdot (\frac{1}{1+r} \cdot S_j^{(i)}) > \sum_{j=1}^{m} \theta_j \cdot S_j^{(0)} \text{ for all } i = 1, \ldots, n$$

Since this is a strict inequality, we will be able to choose a $\theta_0 \in \mathbb{R}$ such that:

$$\sum_{j=1}^{m} \theta_j \cdot (\frac{1}{1+r} \cdot S_j^{(i)}) > -\theta_0 > \sum_{j=1}^{m} \theta_j \cdot S_j^{(0)} \text{ for all } i = 1, \ldots, n$$

Therefore,

$$\frac{1}{1+r} \cdot \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)} > 0 > \sum_{j=0}^{m} \theta_j \cdot S_j^{(0)} \text{ for all } i = 1, \ldots, n$$

This can be rewritten in terms of the Values of portfolio $\theta = (\theta_0, \theta_1, \ldots, \theta)$ at $t = 0$ and $t = 1$, as follows:

$$\frac{1}{1+r} \cdot V_\theta^{(i)} > 0 > V_\theta^{(0)} \text{ for all } i = 1, \ldots, n$$

Thus, we can see that all three conditions in the definition of arbitrage portfolio are satisfied and hence, $\theta = (\theta_0, \theta_1, \ldots, \theta_m)$ is an arbitrage portfolio.

□

Now we are ready to move on to the second fundamental theorem of asset pricing, which associates replication of derivatives with a unique risk-neutral probability measure.

Second Fundamental Theorem of Asset Pricing (2nd FTAP)

Before we state and prove the 2nd FTAP, we need some definitions.

Definition 7.0.1. A Derivative D (in our simple setting of discrete-time with a single-period) is specified as a vector payoff at time $t = 1$, denoted as:

$$(V_D^{(1)}, V_D^{(2)}, \ldots, V_D^{(n)})$$

where $V_D^{(i)}$ is the payoff of the derivative in random outcome ω_i for all $i = 1, \ldots, n$

Definition 7.0.2. A Portfolio $\theta = (\theta_0, \theta_1, \ldots, \theta_m) \in \mathbb{R}^{m+1}$ is a *Replicating Portfolio* for derivative D if:

$$V_D^{(i)} = V_\theta^{(i)} = \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)} \text{ for all } i = 1, \ldots, n \tag{7.4}$$

The negatives of the components $(\theta_0, \theta_1, \ldots, \theta_m)$ are known as the *hedges* for D since they can be used to offset the risk in the payoff of D at $t = 1$.

Definition 7.0.3. An arbitrage-free market (i.e., a market devoid of arbitrage) is said to be *Complete* if every derivative in the market has a replicating portfolio.

Theorem 7.0.3 (Second Fundamental Theorem of Asset Pricing (2nd FTAP))**.** *A market (in our simple setting of discrete-time with a single-period) is Complete if and only if there is a unique risk-neutral probability measure.*

Proof. We will first prove that in an arbitrage-free market, if every derivative has a replicating portfolio (i.e., the market is complete), there is a unique risk-neutral probability measure. We define n special derivatives (known as *Arrow-Debreu securities*), one for each random outcome in Ω at $t = 1$. We define the time $t = 1$ payoff of *Arrow-Debreu security* D_k (for each of $k = 1, \ldots, n$) as follows:

$$V_{D_k}^{(i)} = \mathbb{I}_{i=k} \text{ for all } i = 1, \ldots, n$$

where $\mathbb{I}$ represents the indicator function. This means the payoff of derivative D_k is 1 for random outcome ω_k and 0 for all other random outcomes.

Since each derivative has a replicating portfolio, denote $\theta^{(k)} = (\theta_0^{(k)}, \theta_1^{(k)}, \ldots, \theta_m^{(k)})$ as the replicating portfolio for D_k for each $k = 1, \ldots, m$. Therefore, for each $k = 1, \ldots, m$:

$$V_{\theta^{(k)}}^{(i)} = \sum_{j=0}^{m} \theta_j^{(k)} \cdot S_j^{(i)} = V_{D_k}^{(i)} = \mathbb{I}_{i=k} \text{ for all } i = 1, \ldots, n$$

Using Lemma (7.0.1), we can write the following equation for any risk-neutral probability measure π, for each $k = 1, \ldots, m$:

$$\sum_{j=0}^{m} \theta_j^{(k)} \cdot S_j^{(0)} = V_{\theta^{(k)}}^{(0)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_{\theta^{(k)}}^{(i)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot \mathbb{I}_{i=k} = \frac{1}{1+r} \cdot \pi(\omega_k)$$

We note that the above equation is satisfied for a unique $\pi : \Omega \to [0, 1]$, defined as:

$$\pi(\omega_k) = (1+r) \cdot \sum_{j=0}^{m} \theta_j^{(k)} \cdot S_j^{(0)} \text{ for all } k = 1, \ldots, n$$

which implies that we have a unique risk-neutral probability measure.

Next, we prove the other direction of the 2nd FTAP. We need to prove that if there exists a risk-neutral probability measure π and if there exists a derivative D with no replicating portfolio, we can construct a risk-neutral probability measure different than π.

Consider the following vectors in the vector space $\mathbb{R}^n$

$$v = (V_D^{(1)}, \ldots, V_D^{(n)}) \text{ and } v_j = (S_j^{(1)}, \ldots, S_j^{(n)}) \text{ for all } j = 0, 1, \ldots, m$$

Since D does not have a replicating portfolio, v is not in the span of $v_0, v_1, \ldots, v_m$, which means $v_0, v_1, \ldots, v_m$ do not span $\mathbb{R}^n$. Hence, there exists a non-zero vector $u = (u_1, \ldots, u_n) \in \mathbb{R}^n$ orthogonal to each of $v_0, v_1, \ldots, v_m$, i.e.,

$$\sum_{i=1}^{n} u_i \cdot S_j^{(i)} = 0 \text{ for all } j = 0, 1, \ldots, n \tag{7.5}$$

Note that $S_0^{(i)} = 1 + r$ for all $i = 1, \ldots, n$ and so,

$$\sum_{i=1}^{n} u_i = 0 \tag{7.6}$$

Define $\pi' : \Omega \to \mathbb{R}$ as follows (for some $\epsilon > 0 \in \mathbb{R}$):

$$\pi'(\omega_i) = \pi(\omega_i) + \epsilon \cdot u_i \text{ for all } i = 1, \ldots, n \tag{7.7}$$

To establish π' as a risk-neutral probability measure different than π, note:

- Since $\sum_{i=1}^{n} \pi(\omega_i) = 1$ and since $\sum_{i=1}^{n} u_i = 0$, $\sum_{i=1}^{n} \pi'(\omega_i) = 1$
- Construct $\pi'(\omega_i) > 0$ for each i where $\pi(\omega_i) > 0$ by making $\epsilon > 0$ sufficiently small, and set $\pi'(\omega_i) = 0$ for each i where $\pi(\omega_i) = 0$
- From Equations (7.7), (7.3) and (7.5), we have for each $j = 0, 1, \ldots, m$:

$$\frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi'(\omega_i) \cdot S_j^{(i)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot S_j^{(i)} + \frac{\epsilon}{1+r} \cdot \sum_{i=1}^{n} u_i \cdot S_j^{(i)} = S_j^{(0)}$$

□

Together, the two FTAPs classify markets into:

- Market with arbitrage $\Leftrightarrow$ No risk-neutral probability measure
- Complete (arbitrage-free) market $\Leftrightarrow$ Unique risk-neutral probability measure
- Incomplete (arbitrage-free) market $\Leftrightarrow$ Multiple risk-neutral probability measures

The next topic is derivatives pricing that is based on the concepts of *replication of derivatives* and *risk-neutral probability measures*, and so is tied to the concepts of *arbitrage* and *completeness*.

Derivatives Pricing in Single-Period Setting

In this section, we cover the theory of derivatives pricing for our simple setting of discrete-time with a single-period. To develop the theory of how to price a derivative, first we need to define the notion of a *Position*.

Definition 7.0.4. A *Position* involving a derivative D is the combination of holding some units in D and some units in the fundamental assets $A_0, A_1, \ldots, A_m$, which can be formally represented as a vector $\gamma_D = (\alpha, \theta_0, \theta_1, \ldots, \theta_m) \in \mathbb{R}^{m+2}$ where α denotes the units held in derivative D and α_j denotes the units held in A_j for all $j = 0, 1 \ldots, m$.

Therefore, a *Position* is an extension of the Portfolio concept that includes a derivative. Hence, we can naturally extend the definition of *Portfolio Value* to *Position Value* and we can also extend the definition of *Arbitrage Portfolio* to *Arbitrage Position*.

We need to consider derivatives pricing in three market situations:

- When the market is complete
- When the market is incomplete
- When the market has arbitrage

Derivatives Pricing when Market is Complete

Theorem 7.0.4. *For our simple setting of discrete-time with a single-period, if the market is complete, then any derivative D with replicating portfolio $\theta = (\theta_0, \theta_1, \ldots, \theta_m)$ has price at time $t = 0$ (denoted as value $V_D^{(0)}$):*

$$V_D^{(0)} = V_\theta^{(0)} = \sum_{j=0}^{n} \theta_j \cdot S_j^{(i)} \tag{7.8}$$

Furthermore, if the unique risk-neutral probability measure is $\pi : \Omega \to [0,1]$, then:

$$V_D^{(0)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_D^{(i)} \tag{7.9}$$

Proof. It seems quite reasonable that since θ is the replicating portfolio for D, the value of the replicating portfolio at time $t = 0$ (equal to $V_\theta^{(0)} = \sum_{j=0}^{n} \theta_j \cdot S_j^{(i)}$) should be the price (at $t = 0$) of derivative D. However, we will formalize the proof by first arguing that any candidate derivative price for D other than $V_\theta^{(0)}$ leads to arbitrage, thus dismissing those other candidate derivative prices, and then argue that with $V_\theta^{(0)}$ as the price of derivative D, we eliminate the possibility of an arbitrage position involving D.

Consider candidate derivative prices $V_\theta^{(0)} - x$ for any positive real number x. Position $(1, -\theta_0 + x, -\theta_1, \ldots, -\theta_m)$ has value $x \cdot (1 + r) > 0$ in each of the random outcomes at $t = 1$. But this position has spot ($t = 0$) value of 0, which means this is an Arbitrage Position, rendering these candidate derivative prices invalid. Next consider candidate derivative prices $V_\theta^{(0)} + x$ for any positive real number x. Position $(-1, \theta_0 + x, \theta_1, \ldots, \theta_m)$ has value $x \cdot (1 + r) > 0$ in each of the random outcomes at $t = 1$. But this position has spot ($t = 0$) value of 0, which means this is an Arbitrage Position, rendering these candidate derivative prices invalid as well. So every candidate derivative price other than $V_\theta^{(0)}$ is invalid. Now our goal is to *establish* $V_\theta^{(0)}$ as the derivative price of D by showing that we eliminate the possibility of an arbitrage position in the market involving D if $V_\theta^{(0)}$ is indeed the derivative price.

Firstly, note that $V_\theta^{(0)}$ can be expressed as the riskless rate-discounted expectation (under π) of the payoff of D at $t = 1$, i.e.,

$$V_\theta^{(0)} = \sum_{j=0}^{m} \theta_j \cdot S_j^{(0)} = \sum_{j=0}^{m} \theta_j \cdot \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot S_j^{(i)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)}$$
$$= \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_D^{(i)} \quad (7.10)$$

Now consider an *arbitrary portfolio* $\beta = (\beta_0, \beta_1, \ldots, \beta_m)$. Define a position $\gamma_D = (\alpha, \beta_0, \beta_1, \ldots, \beta_m)$. Assuming the derivative price $V_D^{(0)}$ is equal to $V_\theta^{(0)}$, the Spot Value (at $t = 0$) of position γ_D, denoted $V_{\gamma_D}^{(0)}$, is:

$$V_{\gamma_D}^{(0)} = \alpha \cdot V_\theta^{(0)} + \sum_{j=0}^{m} \beta_j \cdot S_j^{(0)} \quad (7.11)$$

Value of position γ_D in random outcome ω_i (at $t = 1$), denoted $V_{\gamma_D}^{(i)}$, is:

$$V_{\gamma_D}^{(i)} = \alpha \cdot V_D^{(i)} + \sum_{j=0}^{m} \beta_j \cdot S_j^{(i)} \text{ for all } i = 1, \ldots, n \quad (7.12)$$

Combining the linearity in Equations (7.3), (7.10), (7.11) and (7.12), we get:

$$V_{\gamma_D}^{(0)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_{\gamma_D}^{(i)} \quad (7.13)$$

So the position spot value (at $t = 0$) is the riskless rate-discounted expectation (under π) of the position value at $t = 1$. For any γ_D (containing any arbitrary portfolio β), with derivative price $V_D^{(0)}$ equal to $V_\theta^{(0)}$, if the following two conditions are satisfied:

- $V_{\gamma_D}^{(i)} \geq 0$ for all $i = 1, \ldots, n$
- There exists an $i \in \{1, \ldots, n\}$ such that $\mu(\omega_i) > 0$ ($\Rightarrow \pi(\omega_i) > 0$) and $V_{\gamma_D}^{(i)} > 0$

then:

$$V_{\gamma_D}^{(0)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_{\gamma_D}^{(i)} > 0$$

This eliminates any arbitrage possibility if D is priced at $V_\theta^{(0)}$.

To summarize, we have eliminated all candidate derivative prices other than $V_\theta^{(0)}$, and we have established the price $V_\theta^{(0)}$ as the correct price of D in the sense that we eliminate the possibility of an arbitrage position involving D if the price of D is $V_\theta^{(0)}$.

Finally, we note that with the derivative price $V_D^{(0)} = V_\theta^{(0)}$, from Equation (7.10), we have:

$$V_D^{(0)} = \frac{1}{1+r} \cdot \sum_{i=1}^{n} \pi(\omega_i) \cdot V_D^{(i)}$$

□

Now let us consider the special case of 1 risky asset ($m = 1$) and 2 random outcomes ($n = 2$), which we will show is a Complete Market. To lighten notation, we drop the subscript 1 on the risky asset price. Without loss of generality, we assume $S^{(1)} < S^{(2)}$. No-arbitrage requires:

$$S^{(1)} \leq (1+r) \cdot S^{(0)} \leq S^{(2)}$$

Assuming absence of arbitrage and invoking 1st FTAP, there exists a risk-neutral probability measure π such that:

$$S^{(0)} = \frac{1}{1+r} \cdot (\pi(\omega_1) \cdot S^{(1)} + \pi(\omega_2) \cdot S^{(2)})$$

$$\pi(\omega_1) + \pi(\omega_2) = 1$$

With 2 linear equations and 2 variables, this has a straightforward solution, as follows: This implies:

$$\pi(\omega_1) = \frac{S^{(2)} - (1+r) \cdot S^{(0)}}{S^{(2)} - S^{(1)}}$$

$$\pi(\omega_2) = \frac{(1+r) \cdot S^{(0)} - S^{(1)}}{S^{(2)} - S^{(1)}}$$

Conditions $S^{(1)} < S^{(2)}$ and $S^{(1)} \leq (1+r) \cdot S^{(0)} \leq S^{(2)}$ ensure that $0 \leq \pi(\omega_1), \pi(\omega_2) \leq 1$. Also note that this is a unique solution for $\pi(\omega_1), \pi(\omega_2)$, which means that the risk-neutral probability measure is unique, implying that this is a complete market.

We can use these probabilities to price a derivative D as:

$$V_D^{(0)} = \frac{1}{1+r} \cdot (\pi(\omega_1) \cdot V_D^{(1)} + \pi(\omega_2) \cdot V_D^{(2)})$$

Now let us try to form a replicating portfolio (θ_0, θ_1) for D

$$V_D^{(1)} = \theta_0 \cdot (1+r) + \theta_1 \cdot S^{(1)}$$

$$V_D^{(2)} = \theta_0 \cdot (1+r) + \theta_1 \cdot S^{(2)}$$

Solving this yields Replicating Portfolio (θ_0, θ_1) as follows:

$$\theta_0 = \frac{1}{1+r} \cdot \frac{V_D^{(1)} \cdot S^{(2)} - V_D^{(2)} \cdot S^{(1)}}{S^{(2)} - S^{(1)}} \text{ and } \theta_1 = \frac{V_D^{(2)} - V_D^{(1)}}{S^{(2)} - S^{(1)}} \tag{7.14}$$

Note that the derivative price can also be expressed as:

$$V_D^{(0)} = \theta_0 + \theta_1 \cdot S^{(0)}$$

Derivatives Pricing when Market is Incomplete

Theorem (7.0.4) assumed a complete market, but what about an incomplete market? Recall that an incomplete market means some derivatives can't be replicated. Absence of a replicating portfolio for a derivative precludes usual no-arbitrage arguments. The 2nd FTAP says that in an incomplete market, there are multiple risk-neutral probability measures which means there are multiple derivative prices (each consistent with no-arbitrage).

To develop intuition for derivatives pricing when the market is incomplete, let us consider the special case of 1 risky asset ($m = 1$) and 3 random outcomes ($n = 3$), which we will show is an Incomplete Market. To lighten notation, we drop the subscript 1 on the risky asset price. Without loss of generality, we assume $S^{(1)} < S^{(2)} < S^{(3)}$. No-arbitrage requires:

$$S^{(1)} \leq S^{(0)} \cdot (1+r) \leq S^{(3)}$$

Assuming absence of arbitrage and invoking the 1st FTAP, there exists a risk-neutral probability measure π such that:

$$S^{(0)} = \frac{1}{1+r} \cdot (\pi(\omega_1) \cdot S^{(1)} + \pi(\omega_2) \cdot S^{(2)} + \pi(\omega_3) \cdot S^{(3)})$$

$$\pi(\omega_1) + \pi(\omega_2) + \pi(\omega_3) = 1$$

So we have 2 equations and 3 variables, which implies there are multiple solutions for π. Each of these solutions for π provides a valid price for a derivative D.

$$V_D^{(0)} = \frac{1}{1+r} \cdot (\pi(\omega_1) \cdot V_D^{(1)} + \pi(\omega_2) \cdot V_D^{(2)} + \pi(\omega_3) \cdot V_D^{(3)})$$

Now let us try to form a replicating portfolio (θ_0, θ_1) for D

$$V_D^{(1)} = \theta_0 \cdot (1+r) + \theta_1 \cdot S^{(1)}$$

$$V_D^{(2)} = \theta_0 \cdot (1+r) + \theta_1 \cdot S^{(2)}$$

$$V_D^{(3)} = \theta_0 \cdot (1+r) + \theta_1 \cdot S^{(3)}$$

3 equations & 2 variables implies there is no replicating portfolio for *some* D. This means this is an Incomplete Market.

So with multiple risk-neutral probability measures (and consequent, multiple derivative prices), how do we go about determining how much to buy/sell derivatives for? One approach to handle derivative pricing in an incomplete market is the technique called *Superhedging*, which provides upper and lower bounds for the derivative price. The idea of Superhedging is to create a portfolio of fundamental assets whose Value *dominates* the derivative payoff in *all* random outcomes at $t = 1$. Superhedging Price is the smallest possible Portfolio Spot ($t = 0$) Value among all such Derivative-Payoff-Dominating portfolios. Without getting into too many details of the Superhedging technique (out of scope for this book), we shall simply sketch the outline of this technique for our simple setting.

We note that for our simple setting of discrete-time with a single-period, this is a constrained linear optimization problem:

$$\min_{\theta} \sum_{j=0}^{m} \theta_j \cdot S_j^{(0)} \text{ such that } \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)} \geq V_D^{(i)} \text{ for all } i = 1, \ldots, n \tag{7.15}$$

Let $\theta^* = (\theta_0^*, \theta_1^*, \ldots, \theta_m^*)$ be the solution to Equation (7.15). Let SP be the Superhedging Price $\sum_{j=0}^{m} \theta_j^* \cdot S_j^{(0)}$.

After establishing feasibility, we define the Lagrangian $J(\theta, \lambda)$ as follows:

$$J(\theta, \lambda) = \sum_{j=0}^{m} \theta_j \cdot S_j^{(0)} + \sum_{i=1}^{n} \lambda_i \cdot (V_D^{(i)} - \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)})$$

So there exists $\lambda = (\lambda_1, \ldots, \lambda_n)$ that satisfy the following KKT conditions:

$$\lambda_i \geq 0 \text{ for all } i = 1, \ldots, n$$

$$\lambda_i \cdot (V_D^{(i)} - \sum_{j=0}^{m} \theta_j^* \cdot S_j^{(i)}) = 0 \text{ for all } i = 1, \ldots, n \text{ (Complementary Slackness)}$$

$$\nabla_\theta J(\theta^*, \lambda) = 0 \Rightarrow S_j^{(0)} = \sum_{i=1}^{n} \lambda_i \cdot S_j^{(i)} \text{ for all } j = 0, 1, \ldots, m$$

This implies $\lambda_i = \frac{\pi(\omega_i)}{1+r}$ for all $i = 1, \ldots, n$ for a risk-neutral probability measure $\pi : \Omega \to [0, 1]$ (λ can be thought of as "discounted probabilities").

Define Lagrangian Dual

$$L(\lambda) = \inf_{\theta} J(\theta, \lambda)$$

Then, Superhedging Price

$$SP = \sum_{j=0}^{m} \theta_j^* \cdot S_j^{(0)} = \sup_{\lambda} L(\lambda) = \sup_{\lambda} \inf_{\theta} J(\theta, \lambda)$$

Complementary Slackness and some linear algebra over the space of risk-neutral probability measures $\pi : \Omega \to [0, 1]$ enables us to argue that:

$$SP = \sup_{\pi} \sum_{i=1}^{n} \frac{\pi(\omega_i)}{1+r} \cdot V_D^{(i)}$$

This means the Superhedging Price is the least upper-bound of the riskless rate-discounted expectation of derivative payoff across each of the risk-neutral probability measures in the incomplete market, which is quite an intuitive thing to do amidst multiple risk-neutral probability measures."

Likewise, the *Subhedging* price SB is defined as:

$$\max_{\theta} \sum_{j=0}^{m} \theta_j \cdot S_j^{(0)} \text{ such that } \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)} \leq V_D^{(i)} \text{ for all } i = 1, \ldots, n$$

Likewise arguments enable us to establish:

$$SB = \inf_{\pi} \sum_{i=1}^{n} \frac{\pi(\omega_i)}{1+r} \cdot V_D^{(i)}$$

This means the Subhedging Price is the highest lower-bound of the riskless rate-discounted expectation of derivative payoff across each of the risk-neutral probability measures in the incomplete market, which is quite an intuitive thing to do amidst multiple risk-neutral probability measures

So this technique provides an lower bound (SB) and an upper bound (SP) for the derivative price, meaning:

- A price outside these bounds leads to an arbitrage
- Valid prices must be established within these bounds

But often these bounds are not tight and so, not useful in practice.

The alternative approach is to identify hedges that maximize Expected Utility of the combination of the derivative along with it's hedges, for an appropriately chosen market/trader Utility Function (as covered in Chapter 5). The Utility function is a specification of reward-versus-risk preference that effectively chooses the risk-neutral probability measure and (hence, Price).

Consider a concave Utility function $U : \mathbb{R} \rightarrow \mathbb{R}$ applied to the Value in each random outcome $\omega_i, i = 1, \ldots n$, at $t = 1$ (eg: $U(x) = \frac{1-e^{-ax}}{a}$ where $a \in \mathbb{R}$ is the degree of risk-aversion). Let the real-world probabilities be given by $\mu : \Omega \rightarrow [0, 1]$. Denote $V_D = (V_D^{(1)}, \ldots, V_D^{(n)})$ as the payoff of Derivative D at $t = 1$. Let us say that you buy the derivative D at $t = 0$ and will receive the random outcome-contingent payoff V_D at $t = 1$. Let x be the candidate derivative price for D, which means you will pay a cash quantity of x at $t = 0$ for the privilege of receiving the payoff V_D at $t = 1$. We refer to the candidate hedge as Portfolio $\theta = (\theta_0, \theta_1, \ldots, \theta_m)$, representing the units held in the fundamental assets.

Note that at $t = 0$, the cash quantity x you'd be paying to buy the derivative and the cash quantity you'd be paying to buy the Portfolio θ should sum to 0 (note: either of these cash quantities can be positive or negative, but they need to sum to 0 since "money can't just appear or disappear"). Formally,

$$x + \sum_{j=0}^{m} \theta_j \cdot S_j^{(0)} = 0 \tag{7.16}$$

Our goal is to solve for the appropriate values of x and θ based on an *Expected Utility* consideration (that we are about to explain). Consider the Utility of the position consisting of derivative D together with portfolio θ in random outcome ω_i at $t = 1$:

$$U(V_D^{(i)} + \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)})$$

So, the Expected Utility of this position at $t = 1$ is given by:

$$\sum_{i=1}^{n} \mu(\omega_i) \cdot U(V_D^{(i)} + \sum_{j=0}^{m} \theta_j \cdot S_j^{(i)}) \tag{7.17}$$

Noting that $S_0^{(0)} = 1, S_0^{(i)} = 1 + r$ for all $i = 1, \ldots, n$, we can substitute for the value of $\theta_0 = -(x + \sum_{j=1}^{m} \theta_j \cdot S_j^{(0)})$ (obtained from Equation (7.16)) in the above Expected Utility expression (7.17), so as to rewrite this Expected Utility expression in terms of just $(\theta_1, \ldots, \theta_m)$ (call it $\theta_{1:n}$) as:

$$g(V_D, x, \theta_{1:n}) = \sum_{i=1}^{n} \mu(\omega_i) \cdot U(V_D^{(i)} - (1+r) \cdot x + \sum_{j=1}^{m} \theta_j \cdot (S_j^{(i)} - (1+r) \cdot S_j^{(0)}))$$

We define the *Price* of D as the "breakeven value" x^* such that:

$$\max_{\theta_{1:n}} g(V_D, x^*, \theta_{1:n}) = \max_{\theta_{1:n}} g(0, 0, \theta_{1:n})$$

The core principle here (known as *Expected-Utility-Indifference Pricing*) is that introducing a $t = 1$ payoff of V_D together with a derivative price payment of x^* at $t = 0$ keeps the Maximum Expected Utility unchanged.

The $(\theta_1^*, \ldots, \theta_m^*)$ that achieve $\max_{\theta_{1:n}} g(V_D, x^*, \theta_{1:n})$ and $\theta_0^* = -(x^* + \sum_{j=1}^{m} \theta_j^* \cdot S_j^{(0)})$ are the requisite hedges associated the derivative price x^*. Note that the Price of V_D will NOT be the negative of the Price of $-V_D$, hence these prices simply serve as bid prices or ask prices, depending on whether one pays or receives the random outcomes-contingent payoff V_D.

To develop some intuition for what this solution looks like, let us now write some code for the case of 1 risky asset (i.e., $m = 1$). To make things interesting, we will write code for the case where the risky asset price at $t = 1$ (denoted S) follows a normal distribution $S \sim \mathcal{N}(\mu, \sigma^2)$. This means we have a continuous (rather than discrete) set of values for the risky asset price at $t = 1$. Since there are more than 2 random outcomes at time $t = 1$, this is the case of an Incomplete Market. Moreover, we assume the CARA utility function:

$$U(y) = \frac{1 - e^{-a \cdot y}}{a}$$

where a is the CARA coefficient of risk-aversion.

We refer to the units of investment in the risky asset as α and the units of investment in the riskless asset as β. Let S_0 be the spot ($t = 0$) value of the risky asset (riskless asset value at $t = 0$ is 1). Let $f(S)$ be the payoff of the derivative D at $t = 1$. So, the price of derivative D is the breakeven value x^* such that:

$$\max_{\alpha} \mathbb{E}_{S \sim \mathcal{N}(\mu, \sigma^2)}[\frac{1 - e^{-a \cdot (f(S) - (1+r) \cdot x^* + \alpha \cdot (S - (1+r) \cdot S_0))}}{a}]$$
$$= \max_{\alpha} \mathbb{E}_{S \sim \mathcal{N}(\mu, \sigma^2)}[\frac{1 - e^{-a \cdot (\alpha \cdot (S - (1+r) \cdot S_0))}}{a}] \tag{7.18}$$

The maximizing value of α (call it α^*) on the left-hand-side of Equation (7.18) along with $\beta^* = -(x^* + \alpha^* \cdot S_0)$ are the requisite hedges associated with the derivative price x^*.

We set up a `@dataclass MaxExpUtility` with attributes to represent the risky asset spot price S_0 (`risky_spot`), the riskless rate r (`riskless_rate`), mean μ of S (`risky_mean`), standard deviation σ of S (`risky_stdev`), and the payoff function $f(\cdot)$ of the derivative (`payoff_func`).

```
@dataclass(frozen=True)
class MaxExpUtility:
    risky_spot: float  # risky asset price at t=0
    riskless_rate: float  # riskless asset price grows from 1 to 1+r
    risky_mean: float  # mean of risky asset price at t=1
    risky_stdev: float  # std dev of risky asset price at t=1
    payoff_func: Callable[[float], float]  # derivative payoff at t=1
```

Before we write code to solve the derivatives pricing and hedging problem for an incomplete market, let us write code to solve the problem for a complete market (as this will serve as a good comparison against the incomplete market solution). For a complete market, the risky asset has two random prices at $t = 1$: prices $\mu + \sigma$ and $\mu - \sigma$, with probabilities of 0.5 each. As we've seen in Section 7, we can perfectly replicate a derivative payoff in this complete market situation as it amounts to solving 2 linear equations in 2 unknowns (solution shown in Equation (7.14)). The requisite hedges units are simply the negatives of the replicating portfolio units. The method `complete_mkt_price_and_hedges` (of the `MaxExpUtility` class) shown below implements this solution, producing a dictionary comprising of the derivative price (`price`) and the hedge units α (`alpha`) and β (`beta`).

```
def complete_mkt_price_and_hedges(self) -> Mapping[str, float]:
    x = self.risky_mean + self.risky_stdev
    z = self.risky_mean - self.risky_stdev
    v1 = self.payoff_func(x)
    v2 = self.payoff_func(z)
    alpha = (v1 - v2) / (z - x)
    beta = - 1 / (1 + self.riskless_rate) * (v1 + alpha * x)
    price = - (beta + alpha * self.risky_spot)
    return {"price": price, "alpha": alpha, "beta": beta}
```

Next we write a helper method `max_exp_util_for_zero` (to handle the right-hand-side of Equation (7.18)) that calculates the maximum expected utility for the special case of a derivative with payoff equal to 0 in all random outcomes at $t = 1$, i.e., it calculates:

$$\max_{\alpha} \mathbb{E}_{S \sim \mathcal{N}(\mu, \sigma^2)}[\frac{1 - e^{-a \cdot (-(1+r) \cdot c + \alpha \cdot (S - (1+r) \cdot S_0))}}{a}]$$

where c is cash paid at $t = 0$ (so, $c = -(\alpha * S_0 + \beta)$).

The method max_exp_util_for_zero accepts as input c: float (representing the cash c paid at $t = 0$) and risk_aversion_param: float (representing the CARA coefficient of risk aversion a). Refering to Section A in Appendix A, we have a closed-form solution to this maximization problem:

$$\alpha^* = \frac{\mu - (1+r) \cdot S_0}{a \cdot \sigma^2}$$
$$\beta^* = -(c + \alpha \cdot S_0)$$

Substituting α^* in the Expected Utility expression above gives the following maximum value for the Expected Utility for this special case:

$$\frac{1 - e^{-a \cdot (-(1+r) \cdot c + \alpha^* \cdot (\mu - (1+r) \cdot S_0)) + \frac{(a \cdot \alpha^* \cdot \sigma)^2}{2}}}{a} = \frac{1 - e^{a \cdot (1+r) \cdot c - \frac{(\mu - (1+r) \cdot S_0)^2}{2\sigma^2}}}{a}$$

```
def max_exp_util_for_zero(
    self,
    c: float,
    risk_aversion_param: float
) -> Mapping[str, float]:
    ra = risk_aversion_param
    er = 1 + self.riskless_rate
    mu = self.risky_mean
    sigma = self.risky_stdev
    s0 = self.risky_spot
    alpha = (mu - s0 * er) / (ra * sigma * sigma)
    beta = - (c + alpha * self.risky_spot)
    max_val = (1 - np.exp(-ra * (-er * c + alpha * (mu - s0 * er))
                          + (ra * alpha * sigma) ** 2 / 2)) / ra
    return {"alpha": alpha, "beta": beta, "max_val": max_val}
```

Next we write a method max_exp_util that calculates the maximum expected utility for the general case of a derivative with an arbitrary payoff $f(\cdot)$ at $t = 1$ (provided as input pf: Callable[[float, float]] below), i.e., it calculates:

$$\max_{\alpha} \mathbb{E}_{S \sim \mathcal{N}(\mu, \sigma^2)}[\frac{1 - e^{-a \cdot (f(S) - (1+r) \cdot c + \alpha \cdot (S - (1+r) \cdot S_0))}}{a}]$$

Clearly, this has no closed-form solution since $f(\cdot)$ is an arbitary payoff. The method max_exp_util uses the scipy.integrate.quad function to calculate the expectation as an integral of the CARA utility function of $f(S) - (1+r) \cdot c + \alpha \cdot (S - (1+r) \cdot S_0)$ multiplied by the probability density of $\mathcal{N}(\mu, \sigma^2)$, and then uses the scipy.optimize.minimize_scalar function to perform the maximization over values of α.

```
from scipy.integrate import quad
from scipy.optimize import minimize_scalar
```

```python
def max_exp_util(
    self,
    c: float,
    pf: Callable[[float], float],
    risk_aversion_param: float
) -> Mapping[str, float]:
    sigma2 = self.risky_stdev * self.risky_stdev
    mu = self.risky_mean
    s0 = self.risky_spot
    er = 1 + self.riskless_rate
    factor = 1 / np.sqrt(2 * np.pi * sigma2)

    integral_lb = self.risky_mean - self.risky_stdev * 6
    integral_ub = self.risky_mean + self.risky_stdev * 6

    def eval_expectation(alpha: float, c=c) -> float:

        def integrand(rand: float, alpha=alpha, c=c) -> float:
            payoff = pf(rand) - er * c\
                     + alpha * (rand - er * s0)
            exponent = -(0.5 * (rand - mu) * (rand - mu) / sigma2
                         + risk_aversion_param * payoff)
            return (1 - factor * np.exp(exponent)) / risk_aversion_param

        return -quad(integrand, integral_lb, integral_ub)[0]

    res = minimize_scalar(eval_expectation)
    alpha_star = res["x"]
    max_val = - res["fun"]
    beta_star = - (c + alpha_star * s0)
    return {"alpha": alpha_star, "beta": beta_star, "max_val": max_val}
```

Finally, it's time to put it all together - the method `max_exp_util_price_and_hedge` below calculates the maximizing x^* in Equation (7.18). First, we call `max_exp_util_for_zero` (with c set to 0) to calculate the right-hand-side of Equation (7.18). Next, we create a wrapper function `prep_func` around `max_exp_util`, which is provided as input to `scipt.optimize.root_scalar` to solve for x^* in the right-hand-side of Equation (7.18). Plugging x^* (`opt_price` in the code below) in `max_exp_util` provides the hedges α^* and β^* (`alpha` and `beta` in the code below).

```python
from scipy.optimize import root_scalar

def max_exp_util_price_and_hedge(
    self,
    risk_aversion_param: float
) -> Mapping[str, float]:
```

```python
    meu_for_zero = self.max_exp_util_for_zero(
        0.,
        risk_aversion_param
    )["max_val"]

    def prep_func(pr: float) -> float:
        return self.max_exp_util(
            pr,
            self.payoff_func,
            risk_aversion_param
        )["max_val"] - meu_for_zero

    lb = self.risky_mean - self.risky_stdev * 10
    ub = self.risky_mean + self.risky_stdev * 10
    payoff_vals = [self.payoff_func(x) for x in np.linspace(lb, ub, 1001)]
    lb_payoff = min(payoff_vals)
    ub_payoff = max(payoff_vals)

    opt_price = root_scalar(
        prep_func,
        bracket=[lb_payoff, ub_payoff],
        method="brentq"
    ).root

    hedges = self.max_exp_util(
        opt_price,
        self.payoff_func,
        risk_aversion_param
    )
    alpha = hedges["alpha"]
    beta = hedges["beta"]
    return {"price": opt_price, "alpha": alpha, "beta": beta}
```

The above code for the class `MaxExpUtility` is in the file rl/chapter8/max_exp_utility.py. As ever, we encourge you to play with various choices of S_0, r, μ, σ, f in creating instances of `MaxExpUtility`, analyze the obtained prices/hedges and plot some graphs to develop intuition on how the results change as a function of the various inputs.

Running this code for $S_0 = 100, r = 5\%, \mu = 110, \sigma = 25$ when buying a call option (European since we have only one time period) with strike $= 105$, the method `complete_mkt_price_and hedges` gives an option price of 11.43, risky asset hedge units of -0.6 (i.e., we hedge the risk of owning the call option by short-selling 60% of the risky asset) and riskless asset hedge units of 48.57 (i.e., we take the $60 proceeds of short-sale less the $11.43 option price payment = $48.57 of cash and invest in a risk-free bank account earning 5% interest). As mentioned earlier, this is the perfect hedge if we had a complete market (i.e., two random outcomes). Running this code for the same inputs for an incomplete market (calling the method `max_exp_util_price_and_hedge` for risk-aversion parameter values of $a = 0.3, 0.6, 0.9$ gives us the following results:

```
--- Risk Aversion Param = 0.30 ---
{'price': 23.279, 'alpha': -0.473, 'beta': 24.055}
--- Risk Aversion Param = 0.60 ---
{'price': 12.669, 'alpha': -0.487, 'beta': 35.998}
--- Risk Aversion Param = 0.90 ---
{'price': 8.865, 'alpha': -0.491, 'beta': 40.246}
```

We note that the call option price is quite high (23.28) when the risk-aversion is low at $a = 0.3$ (relative to the complete market price of 11.43) but the call option price drops to 12.67 and 8.87 for $a = 0.6$ and $a = 0.9$ respectively. This makes sense since if you are more risk-averse (high a), then you'd be more unwilling to take the risk of buying a call option and hence, would want to pay less to buy the call option. Note how the risky asset short-sale is significantly less (~47% - 49%) compared the to the risky asset short-sale of 60% in the case of a complete market. The varying investments in the riskless asset (as a function of the risk-aversion a) essentially account for the variation in option prices (as a function of a). Figure 7.1 provides tremendous intuition on how the hedges work for the case of a complete market and for the cases of an incomplete market with the 3 choices of risk-aversion parameters. Note that we have plotted the negatives of the hedge portfolio values at $t = 1$ so as to visualize them appropriately relative to the payoff of the call option. Note that the hedge portfolio value is a linear function of the risky asset price at $t = 1$. Notice how the slope and intercept of the hedge portfolio value changes for the 3 risk-aversion scenarios and how they compare against the complete market hedge portfolio value.

Now let us consider the case of selling the same call option. In our code, the only change we make is to make the payoff function `lambda x: - max(x - 105.0, 0)` instead of `lambda x: max(x - 105.0, 0)` to reflect the fact that we are now selling the call option and so, our payoff will be the negative of that of an owner of the call option.

With the same inputs of $S_0 = 100, r = 5\%, \mu = 110, \sigma = 25$, and for the same risk-aversion parameter values of $a = 0.3, 0.6, 0.9$, we get the following results:

```
--- Risk Aversion Param = 0.30 ---
{'price': -6.307, 'alpha': 0.527, 'beta': -46.395}
--- Risk Aversion Param = 0.60 ---
{'price': -32.317, 'alpha': 0.518, 'beta': -19.516}
--- Risk Aversion Param = 0.90 ---
{'price': -44.236, 'alpha': 0.517, 'beta': -7.506}
```

We note that the sale price demand for the call option is quite low (6.31) when the risk-aversion is low at $a = 0.3$ (relative to the complete market price of 11.43) but the sale price demand for the call option rises sharply to 32.32 and 44.24 for

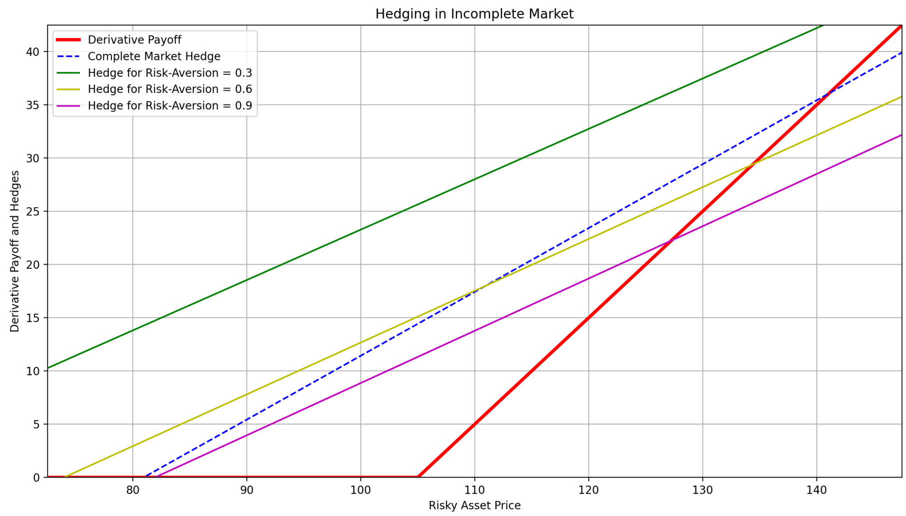

Figure 7.1.: Hedges when buying a Call Option

$a = 0.6$ and $a = 0.9$ respectively. This makes sense since if you are more risk-averse (high a), then you'd be more unwilling to take the risk of selling a call option and hence, would want to charge more for the sale of the call option. Note how the risky asset hedge units are less (~52% - 53%) compared the to the risky asset hedge units (60%) in the case of a complete market. The varying riskless borrowing amounts (as a function of the risk-aversion a) essentially account for the variation in option prices (as a function of a). Figure 7.2 provides the visual intuition on how the hedges work for the 3 choices of risk-aversion parameters (along with the hedges for the complete market, for reference).

Note that each buyer and each seller might have a different level of risk-aversion, meaning each of them would have a different buy price bid/different sale price ask. A transaction can occur between a buyer and a seller (with potentially different risk-aversion levels) if the buyer's bid matches the seller's ask.

Derivatives Pricing when Market has Arbitrage

Finally, we arrive at the case where the market has arbitrage. This is the case where there is no risk-neutral probability measure and there can be multiple replicating portfolios (which can lead to arbitrage). This lead to an inability to price derivatives. To provide intuition for the case of a market with arbitrage, we consider the special case of 2 risky assets ($m = 2$) and 2 random outcomes ($n = 2$), which we will show is a Market with Arbitrage. Without loss of generality, we assume $S_1^{(1)} < S_1^{(2)}$ and $S_2^{(1)} < S_2^{(2)}$. Let us try to determine a risk-neutral probability measure π:

$$S_1^{(0)} = e^{-r} \cdot (\pi(\omega_1) \cdot S_1^{(1)} + \pi(\omega_2) \cdot S_1^{(2)})$$

$$S_2^{(0)} = e^{-r} \cdot (\pi(\omega_1) \cdot S_2^{(1)} + \pi(\omega_2) \cdot S_2^{(2)})$$

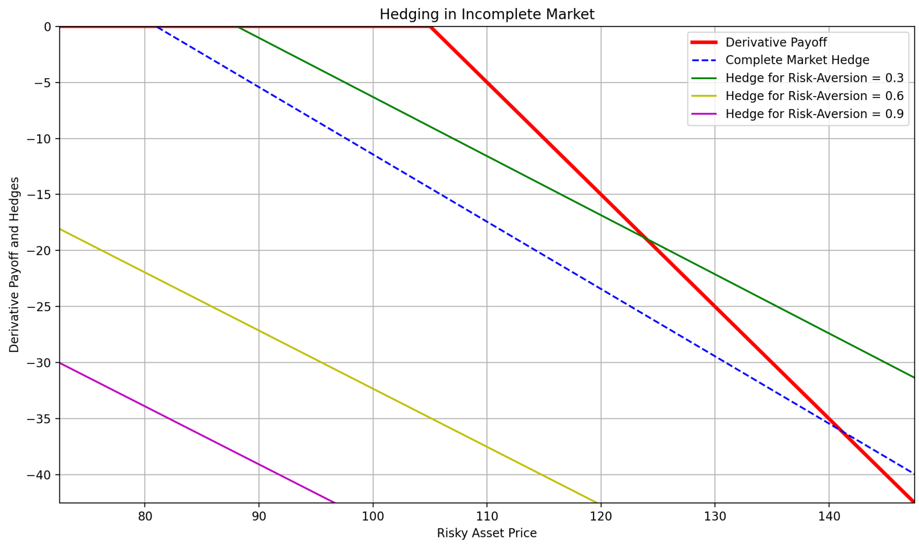

Figure 7.2.: Hedges when selling a Call Option

$$\pi(\omega_1) + \pi(\omega_2) = 1$$

3 equations and 2 variables implies that there is no risk-neutral probability measure π. Let's try to form a replicating portfolio $(\theta_0, \theta_1, \theta_2)$ for a derivative D:

$$V_D^{(1)} = \theta_0 \cdot e^r + \theta_1 \cdot S_1^{(1)} + \theta_2 \cdot S_2^{(1)}$$
$$V_D^{(2)} = \theta_0 \cdot e^r + \theta_1 \cdot S_1^{(2)} + \theta_2 \cdot S_2^{(2)}$$

2 equations and 3 variables implies that there are multiple replicating portfolios. Each such replicating portfolio yields a price for D as:

$$V_D^{(0)} = \theta_0 + \theta_1 \cdot S_1^{(0)} + \theta_2 \cdot S_2^{(0)}$$

Select two such replicating portfolios with different $V_D^{(0)}$. The combination of one of these replicating portfolios with the negative of the other replicating portfolio is an Arbitrage Portfolio because:

- They cancel off each other's portfolio value in each $t = 1$ states
- The combined portfolio value can be made to be negative at $t = 0$ (by choosing which replicating portfolio we negate)

So this is a market that admits arbitrage (no risk-neutral probability measure).

Derivatives Pricing in Multi-Period/Continuous-Time Settings

Now that we have understood the key concepts of derivatives pricing/hedging for the simple setting of discrete-time with a single-period, it's time to do an overview of derivatives pricing/hedging theory in the full-blown setting of multiple time-periods and in continuous-time. While an adequate coverage of this theory is beyond the scope of this book, we will sketch an overview in this section. Along the way, we will cover two derivatives pricing applications that can be modeled as MDPs (and hence, tackled with Dynamic Programming or Reinforcement Learning Algorithms).

The good news is that much of the concepts we learnt for the single-period setting carry over to multi-period and continuous-time settings. The key difference in going over from single-period to multi-period is that we need to adjust the replicating portfolio (i.e., adjust θ) at each time step. Other than this difference, the concepts of arbitrage, risk-neutral probability measures, complete market etc. carry over. In fact, the two fundamental theorems of asset pricing also carry over. It is indeed true that in the multi-period setting, no-arbitrage is equivalent to the existence of a risk-neutral probability measure and market completeness (i.e., replication of derivatives) is equivalent to having a unique risk-neutral probability measure.

Multi-Period Complete-Market Setting

We learnt in the single-period setting that if the market is complete, there are two equivalent ways to conceptualize derivatives pricing:

- Solve for the replicating portfolio (i.e., solve for the units in the fundamental assets that would replicate the derivative payoff), and then calculate the derivative price as the value of this replicating portfolio at $t = 0$.
- Calculate the probabilities of random-outcomes for the unique risk-neutral probability measure, and then calculate the derivative price as the riskless rate-discounted expectation (under this risk-neutral probability measure) of the derivative payoff.

It turns out that even in the multi-period setting, when the market is complete, we can calculate the derivative price (not just at $t = 0$, but at any random outcome at any future time) with either of the above two (equivalent) methods, as long as we appropriately adjust the fundamental asset's units in the replicating portfolio (depending on the random outcome) as we move from one time step to the next. It is important to note that when we alter the fundamental asset's units in the replicating portfolio at each time step, we need to respect the constraint that money cannot enter or leave the replicating portfolio (i.e., it is a *self-financing replicating portfolio* with the replicating portfolio value remaining unchanged in the process of altering the units in the fundamental assets). It is also important to note that the alteration in units in the fundamental assets is dependent on the prices of the fundamental assets (which are random outcomes as we move forward from one time step to the next). Hence, the fundamental asset's units in the replicating portfolio evolve as random variables, while respecting the self-financing constraint. Therefore, the replicating portfolio in a multi-period setting in often refered to as a *Dynamic Self-Financing Replicating*

Portfolio to reflect the fact that the replicating portfolio is adapting to the changing prices of the fundamental assets. The negatives of the fundamental asset's units in the replicating portfolio form the hedges for the derivative.

To ensure that the market is complete in a multi-period setting, we need to assume that the market is "frictionless" - that we can trade in real-number quantities in any asset and that there are no transaction costs for any trades at any time step. From a computational perspective, we walk back in time from the final time step (call it $t = T$) to $t = 0$, and calculate the fundamental asset's units in the replicating portfolio in a "backward recursive manner". As in the case of the single-period setting, each backward-recursive step from outcomes at time $t + 1$ to a specific outcome at time t simply involves solving a linear system of equations where each unknown is the replicating portfolio units in a specific fundamental asset and each equation corresponds to the value of the replicating portfolio at a specific outcome at time $t + 1$ (which is established recursively). The market is complete if there is a unique solution to each linear system of equations (for each time t and for each outcome at time t) in this backward-recursive computation. This gives us not just the replicating portfolio (and consequently, hedges) at each outcome at each time step, but also the price at each outcome at each time step (the price is equal to the value of the calculated replicating portfolio at that outcome at that time step).

Equivalently, we can do a backward-recursive calculation in terms of the risk-neutral probability measures, with each risk-neutral probability measure giving us the transition probabilities from an outcome at time step t to outcomes at time step $t + 1$. Again, in a complete market, it amounts to a unique solution of each of these linear system of equations. For each of these linear system of equations, an unknown is a transition probability to a time $t + 1$ outcome and an equation corresponds to a specific fundamental asset's prices at the time $t + 1$ outcomes. This calculation is popularized (and easily understood) in the simple context of a Binomial Options Pricing Model. We devote Section 7 to coverage of the original Binomial Options Pricing Model and model it as a Finite-State Finite-Horizon MDP (and utilize the ADP code developed in 3 to solve the MDP).

Continuous-Time Complete-Market Setting

To move on from multi-period to continuous-time, we simply make the time-periods smaller and smaller, and take the limit of the time-period tending to zero. We need to preserve the complete-market property as we do this, which means that we can trade in real-number units without transaction costs in continuous-time. As we've seen before, operating in continuous-time allows us to tap into stochastic calculus, which forms the foundation of much of the rich theory of continuous-time derivatives pricing/hedging. With this very rough and high-level overview, we refer you to Tomas Bjork's book on Arbitrage Theory in Continuous Time for a thorough understanding of this theory.

To provide a sneak-peek into this rich continuous-time theory, we've sketched in Appendix F the derivation of the famous Black-Scholes equation and it's solution for the case of European Call and Put Options.

So to summarize, we are in good shape to price/hedge in a multi-period and continuous-time setting if the market is complete. But what if the market is incomplete (which is typical in a real-world situation)? Founded on the Fundamental Theorems of Asset Pricing (which applies to multi-period and continuous-time settings as well), there is indeed considerable literature on how to price in incomplete markets for multi-period/continuous-time, which includes the superhedging approach as well as the *Expected-Utility-Indifference* approach, that we had covered in Subsection 7 for the simple setting of discrete-time with single-period. However, in practice, these approaches are not adopted as they fail to capture real-world nuances adequately. Besides, most of these approaches lead to fairly wide price bounds that are not particularly useful in practice. In Section 7, we extend the *Expected-Utility-Indifference* approach that we had covered for the single-period setting to the multi-period setting. It turns out that this approach can be modeled as an MDP, with the adjustments to the hedge quantities at each time step as the actions of the MDP - solving the optimal policy gives us the optimal derivative hedging strategy and the associated optimal value function gives us the derivative price. This approach is applicable to real-world situations and one can even incorporate all the real-world frictions in one's MDP to build a practical solution for derivatives trading (covered in Section 7).

Optimal Exercise of American Options cast as a Finite MDP

The original Binomial Options Pricing Model was developed to price (and hedge) options (including American) on an underlying whose price evolves according to a lognormal stochastic process, with the stochastic process approximated in the form of a simple discrete-time, discrete-states process that enables enormous computational tractability. The lognormal stochastic process is basically of the same form as the stochastic process of the underlying price in the Black-Scholes model (covered in Appendix F). However, the underlying price process in the Black-Scholes model is specified in the real-world probability measure whereas here we specify the underlying price process in the risk-neutral probability measure. This is because here we will employ the pricing method of riskless rate-discounted expectation (under the risk-neutral probability measure) of the option payoff. Recall that in the single-period setting, the underlying asset price's expected rate of growth is calibrated to be equal to the riskless rate r, under the risk-probability probability measure. This calibration applies even in the multi-period and continuous-time setting. For a continuous-time lognormal stochastic process, the lognormal drift will hence be equal to r in the risk-neutral probability measure (rather than μ in the real-world probability measure, as per the Black-Scholes model). Precisely, the stochastic process S for the underlying price in the risk-neutral probability measure is:

$$dS_t = r \cdot S_t \cdot dt + \sigma \cdot S_t \cdot dz_t$$

where σ is the lognormal dispersion (often refered to as "lognormal volatility" - we will simply call it volatility for the rest of this section). If you want to develop a thorough understanding of the broader topic of change of probability measures and how it affects the drift term (beyond the scope of this book, but an important topic in continuous-time financial pricing theory), we refer you to the technical material on Radon-Nikodym Derivative and Girsanov Theorem.

The Binomial Options Pricing Model serves as a discrete-time, finite-states approximation to this continuous-time process, and is essentially an extension to the single-period model we had covered earlier for the case of a single fundamental risky asset. We've learnt previously that in the single-period case for a single fundamental risky asset, in order to be a complete market, we need to have exactly two random outcomes. We basically extend this "two random outcomes" pattern to each outcome at each time step, by essentially growing out a "binary tree". But there is a caveat - with a binary tree, we end up with an exponential (2^i) number of outcomes after i time steps. To contain the exponential growth, we construct a "recombining tree", meaning an "up move" followed by a "down move" ends up in the same underlying price outcome as a "down move" followed by an "up move" (as illustrated in Figure 7.3). Thus, we have $i+1$ price outcomes after i time steps in this "recombining tree". We conceptualize the ascending-sorted sequence of $i+1$ price outcomes as the (time step $= i$) states $\mathcal{S}_i = \{0, 1, \ldots, i\}$ (since the price movements form a discrete-time, finite-states Markov Chain). Since we are modeling a lognormal process, we model the discrete-time price moves as multiplicative to the price. We denote $S_{i,j}$ as the price after i time steps in state j (for any $i \in \mathbb{Z}_{\geq 0}$ and for any $0 \leq j \leq i$). So the two random prices resulting from $S_{i,j}$ are $S_{i+1,j+1} = S_{i,j} \cdot u$ and $S_{i+1,j} = \frac{S_{i,j}}{u}$ for some constant u (that we will calibrate). The important point is that u remains a constant across time steps i and across states j at each time step i. Since the "up move" is a multiplicative factor of u and the "down move" is a multiplicative factor of $\frac{1}{u}$, we ensure the "recombining tree" feature.

Let q be the probability of the "up move" (typically, we use p to denote real-world probability and q to denote the risk-neutral probability) so that $1-q$ is the probability of the "down move", Our goal is to calibrate q and u so that the probability distribution of log-price-ratios $\{\log(\frac{S_{n,0}}{S_{0,0}}), \log(\frac{S_{n,1}}{S_{0,0}}), \ldots, \log(\frac{S_{n,n}}{S_{0,0}})\}$ after $n \in \mathbb{Z}_{\geq 0}$ time steps (with each time step of interval $\frac{T}{n}$ for a given $T \in \mathbb{R}^+$) serves as a good approximation to $\mathcal{N}(rT, \sigma^2 T)$ (that we know to be the distribution of $\log(\frac{S_T}{S_0})$, as derived in Section D in Appendix D). Note that the starting price $S_{0,0}$ of this discrete-time approximation process is equal to the starting price S_0 of the continuous-time process. We shall calibrate q and u in two steps:

- In the first step, we pretend that $q = 0.5$ and calibrate u such that the variance of the two random outcomes $\log(\frac{S_{i+1,j+1}}{S_{i,j}}) = \log(u)$ and $\log(\frac{S_{i+1,j}}{S_{i,j}}) =$

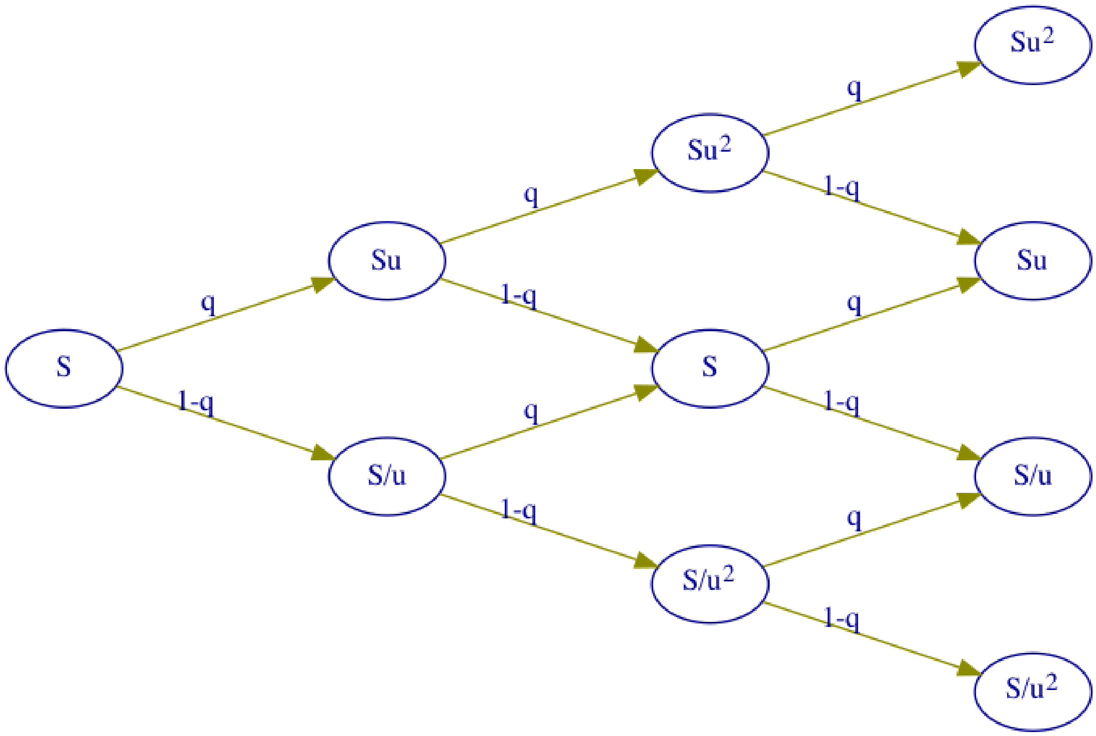

Figure 7.3.: Binomial Option Pricing Model (Binomial Tree)

$-\log(u)$ is equal to the variance $\frac{\sigma^2 T}{n}$ of the normally-distributed (stationary) process $\log\left(\frac{S_{t+\frac{T}{n}}}{S_t}\right)$ for any $i \in \mathbb{Z}_{\geq 0}$ for all $0 \leq j \leq i$. This yields:

$$\log^2(u) = \frac{\sigma^2 T}{n} \Rightarrow u = e^{\frac{\sigma\sqrt{T}}{n}}$$

This ensures that the variance of the symmetric binomial distribution after n time steps matches the variance $\sigma^2 T$ of the normal distribution of $\log\left(\frac{S_T}{S_0}\right)$.

- In the second step, we adjust the probability q so that the mean of the two random outcomes $\frac{S_{i+1,j+1}}{S_{i,j}} = u$ and $\frac{S_{i+1,j}}{S_{i,j}} = \frac{1}{u}$ is equal to the mean $e^{\frac{rT}{n}}$ of the lognormally-distributed (stationary) process $\frac{S_{t+\frac{T}{n}}}{S_t}$ for any $i \in \mathbb{Z}_{\geq 0}$ for all $0 \leq j \leq i$. This yields:

$$qu + \frac{1-q}{u} = e^{\frac{rT}{n}} \Rightarrow q = \frac{u \cdot e^{\frac{rT}{n}} - 1}{u^2 - 1} = \frac{e^{\frac{rT+\sigma\sqrt{T}}{n}} - 1}{e^{\frac{2\sigma\sqrt{T}}{n}} - 1}$$

Thus, we have the parameters u and q that fully specify the Binomial Options Pricing Model. Now we get to the application of this model. We are interested in using this model for optimal exercise (and hence, pricing) of American Options. This is in contrast to the Black-Scholes Partial Differential Equation which only enabled us to price options with a fixed payoff at a fixed point in time (eg:

European Call and Put Options). Of course, a special case of American Options is indeed European Options. It's important to note that here we are tackling the much harder problem of the ideal timing of exercise of an American Option - the Binomial Options Pricing Model is well suited for this.

As mentioned earlier, we want to model the problem of Optimal Exercise of American Options as a discrete-time finite-horizon MDP. To fit into our framework for discrete-time finite-horizon MDPs, we need to set the terminal time to be $t = T + 1$, meaning all the states at time $T + 1$ are terminal states. Here we will utilize the states and state transitions (probabilistic price movements of the underlying) given by the Binomial Options Pricing Model as the states and state transitions in the MDP. The MDP actions in each state will be binary - either exercise the option (and immediately move to a terminal state) or don't exercise the option (i.e., continue on to the next time step's random state, as given by the Binomial Options Pricing Model). If the exercise action is chosen, the MDP reward is the option payoff. If the continue action is chosen, the reward is 0. The discount factor γ is $e^{-\frac{rT}{n}}$ since (as we've learnt in the single-period case), the price (which translates here to the Optimal Value Function) is defined as the riskless rate-discounted expectation (under the risk-neutral probability measure) of the option payoff. In the multi-period setting, the overall discounting amounts to composition (multiplication) of each time step's discounting (which is equal to γ) and the overall risk-neutral probability measure amounts to the composition of each time step's risk-neutral probability measure (which is specified by the calibrated value q).

Now let's write some code to determine the Optimal Exercise of American Options (and hence, the price of American Options) by modeling this problem as a discrete-time finite-horizon MDP. We create a `dataclass OptimalExerciseBinTree` whose attributes are `spot_price` (specifying the current, i.e., time=0 price of the underlying), `payoff` (specifying the option payoff, when exercised), `expiry` (specifying the time T to expiration of the American Option), `rate` (specifying the riskless rate r), `vol` (specifying the lognormal volatility σ), and `num_steps` (specifying the number n of time steps in the binomial tree). Note that each time step is of interval $\frac{T}{n}$ (which is implemented below in the method `dt`). Note also that the `payoff` function is fairly generic taking two arguments - the first argument is the time at which the option is exercised, and the second argument is the underlying price at the time the option is exercised. Note that for a typical American Call or Put Option, the payoff does not depend on time and the dependency on the underlying price is the standard "hockey-stick" payoff that we are now fairly familiar with (however, we designed the interface to allow for more general option payoff functions).

The set of states $\mathcal{S}_i$ at time step i (for all $0 \leq i \leq T+1$) is: $\{0, 1, \ldots, i\}$ and the method `state_price` below calculates the price in state j at time step i as:

$$S_{i,j} = S_{0,0} \cdot e^{\frac{(2j-i)\sigma T}{n}}$$

Finally, the method `get_opt_vf_and_policy` calculates u (`up_factor`) and q (`up_prob`), prepares the requisite state-reward transitions (conditional on current state and action) to move from one time step to the next, and passes along the constructed time-sequenced transitions (`Sequence[StateActionMapping[int, bool]]`) to `rl.finite_horizon.get_opt_vf_and_policy` (which we had written in Chapter 3) to perform the requisite backward induction and return an `Iterator` on pairs of `V[int]` and `FinitePolicy[int, bool]`. Note that the states at any time-step i are the integers from 0 to i and hence, represented as `int`, and the actions are represented as `bool` (`True` for exercise and `False` for continue). We need to point out a couple of small details in the code. Firstly, we represent an early terminal state (in case of option exercise before expiration of the option) as -1. Secondly, note that there is no action map from an early terminal state and so, the value associated with a key (state) of -1 is `None` (according to the representation protocol in `StateActionMapping` which specifies a `FiniteMarkovDecisionProcess`).

```
from rl.distribution import Constant, Categorical
from rl.finite_horizon import optimal_vf_and_policy
from rl.dynamic_programming import V
from rl.markov_decision_process import FinitePolicy

@dataclass(frozen=True)
class OptimalExerciseBinTree:

    spot_price: float
    payoff: Callable[[float, float], float]
    expiry: float
    rate: float
    vol: float
    num_steps: int

    def dt(self) -> float:
        return self.expiry / self.num_steps

    def state_price(self, i: int, j: int) -> float:
        return self.spot_price * np.exp((2 * j - i) * self.vol *
                                        np.sqrt(self.dt()))

    def get_opt_vf_and_policy(self) -> \
            Iterator[Tuple[V[int], FinitePolicy[int, bool]]]:
        dt: float = self.dt()
        up_factor: float = np.exp(self.vol * np.sqrt(dt))
        up_prob: float = (np.exp(self.rate * dt) * up_factor - 1) / \
            (up_factor * up_factor - 1)
        return optimal_vf_and_policy(
            steps=[
                {j: None if j == -1 else {
                    True: Constant(
```

```python
                    (
                        -1,
                        self.payoff(i * dt, self.state_price(i, j))
                    )
                ),
                False: Categorical(
                    {
                        (j + 1, 0.): up_prob,
                        (j, 0.): 1 - up_prob
                    }
                )
            } for j in range(i + 1)}
            for i in range(self.num_steps + 1)
        ],
        gamma=np.exp(-self.rate * dt)
    )
```

Now we want to try out this code on an American Call Option and American Put Option. We know that it is never optimal to exercise an American Call Option before the option expiration. The reason for this is as follows: Upon early exercise (say at time $\tau < T$), we borrow cash K (to pay for the purchase of the underlying) and own the underlying (valued at S_τ). So, at option expiration T, we owe cash $K \cdot e^{r(T-\tau)}$ and own the underlying valued at S_T, which is an overall value at time T of $S_T - K \cdot e^{r(T-\tau)}$. We argue that this value is always less than the value $\max(S_T - K, 0)$ we'd obtain at option expiration T if we'd made the choice to not exercise early. If the call option ends up in-the-money at option expiration T (i.e., $S_T > K$), then $S_T - K \cdot e^{r(T-\tau)}$ is less than the value $S_T - K$ we'd get by exercising at option expiration T. If the call option ends up not being in-the-money at option expiration T (i.e., $S_T \leq K$), then $S_T - K \cdot e^{r(T-\tau)} < 0$ which is less than the 0 payoff we'd obtain at option expiration T. Hence, we are always better off waiting until option expiration (i.e. it is never optimal to exercise a call option early, no matter how much in-the-money we get before option expiration). Hence, the price of an American Call Option should be equal to the price of an European Call Option with the same strike and expiration time. However, for an American Put Option, it is indeed optimal to exercise early and hence, the price of an American Put Option is greater then the price of an European Put Option with the same strike and expiration time. Thus, it is interesting to ask the question: For each time $t < T$, what is the threshold of underlying price S_t below which it is optimal to exercise an American Put Option? It is interesting to view this threshold as a function of time (we call this function as the optimal exercise boundary of an American Put Option). One would expect that this optimal exercise boundary rises as one gets closer to the option expiration T. But exactly what shape does this optimal exercise boundary have? We can answer this question by analyzing the optimal policy at each time step - we just need to find the state k at each time step i such that the Optimal Policy $\pi_i^*(\cdot)$ evaluates to True for all states $j \leq k$ (and evaluates to False for all states $j > k$).

We write the following method to calculate the Optimal Exercise Boundary:

```
    def option_exercise_boundary(
        self,
        policy_seq: Sequence[FinitePolicy[int, bool]],
        is_call: bool
    ) -> Sequence[Tuple[float, float]]:
        dt: float = self.dt()
        ex_boundary: List[Tuple[float, float]] = []
        for i in range(self.num_steps + 1):
            ex_points = [j for j in range(i + 1)
                         if policy_seq[i].act(j).value and
                         self.payoff(i * dt, self.state_price(i, j)) > 0]
            if len(ex_points) > 0:
                boundary_pt = min(ex_points) if is_call else max(ex_points)
                ex_boundary.append(
                    (i * dt, opt_ex_bin_tree.state_price(i, boundary_pt))
                )
        return ex_boundary
```

option_exercise_boundary takes as input policy_seq which represents the sequence of optimal policies π_i^* for each time step $0 \leq i \leq T$, and produces as output the sequence of pairs $(\frac{iT}{n}, B_i)$ where

$$B_i = \max_{j:\pi_i^*(j)=True} S_{i,j}$$

with the little detail that we only consider those states j for which the option payoff is positive. For some time steps i, none of the states j qualify as $\pi_i^*(j) = True$, in which case we don't include that time step i in the output sequence.

To compare the results of American Call and Put Option Pricing on this Binomial Options Pricing Model against the corresponding European Options prices, we write the following method to implement the Black-Scholes closed-form solution (derived as Equations F.7 and F.8 in Appendix F):

```
from scipy.stats import norm

    def european_price(self, is_call: bool, strike: float) -> float:
        sigma_sqrt: float = self.vol * np.sqrt(self.expiry)
        d1: float = (np.log(self.spot_price / strike) +
                     (self.rate + self.vol ** 2 / 2.) * self.expiry) \
            / sigma_sqrt
        d2: float = d1 - sigma_sqrt
        if is_call:
            ret = self.spot_price * norm.cdf(d1) - \
                strike * np.exp(-self.rate * self.expiry) * norm.cdf(d2)
        else:
            ret = strike * np.exp(-self.rate * self.expiry) * norm.cdf(-d2) - \
```

```
                self.spot_price * norm.cdf(-d1)
        return ret
```

Here's some code to price an American Put Option (changing `is_call` to `True` will price American Call Options):

```
from rl.gen_utils.plot_funcs import plot_list_of_curves

spot_price_val: float = 100.0
strike: float = 100.0
is_call: bool = False
expiry_val: float = 1.0
rate_val: float = 0.05
vol_val: float = 0.25
num_steps_val: int = 300

if is_call:
    opt_payoff = lambda _, x: max(x - strike, 0)
else:
    opt_payoff = lambda _, x: max(strike - x, 0)

opt_ex_bin_tree: OptimalExerciseBinTree = OptimalExerciseBinTree(
    spot_price=spot_price_val,
    payoff=opt_payoff,
    expiry=expiry_val,
    rate=rate_val,
    vol=vol_val,
    num_steps=num_steps_val
)

vf_seq, policy_seq = zip(*opt_ex_bin_tree.get_opt_vf_and_policy())
ex_boundary: Sequence[Tuple[float, float]] = \
    opt_ex_bin_tree.option_exercise_boundary(policy_seq, is_call)
time_pts, ex_bound_pts = zip(*ex_boundary)
label = ("Call" if is_call else "Put") + " Option Exercise Boundary"
plot_list_of_curves(
    list_of_x_vals=[time_pts],
    list_of_y_vals=[ex_bound_pts],
    list_of_colors=["b"],
    list_of_curve_labels=[label],
    x_label="Time",
    y_label="Underlying Price",
    title=label
)

european: float = opt_ex_bin_tree.european_price(is_call, strike)
```

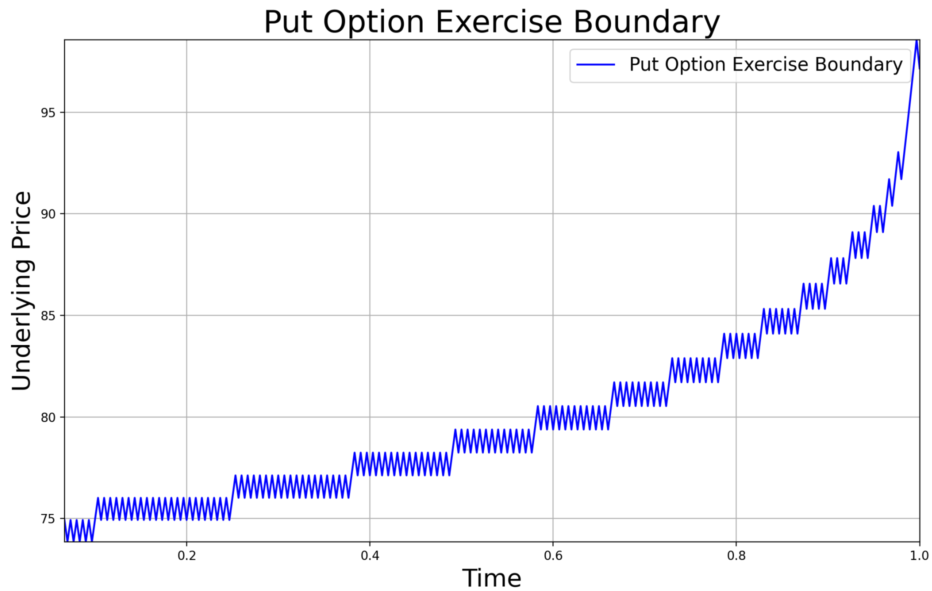

Figure 7.4.: Put Option Exercise Boundary

```
print(f"European Price = {european:.3f}")

am_price: float = vf_seq[0][0]
print(f"American Price = {am_price:.3f}")
```

This prints as output:

```
European Price = 7.459
American Price = 7.971
```

So we can see that the price of this American Put Option is significantly higher than the price of the corresponding European Put Option. The exercise boundary produced by this code is shown in Figure 7.4. The locally-jagged nature of the exercise boundary curve is because of the "diamond-like" local-structure of the underlying prices at the nodes in the binomial tree. We can see that when the time to expiry is large, it is not optimal to exercise unless the underlying price drops significantly. It is only when the time to expiry becomes quite small that the optimal exercise boundary rises sharply towards the strike value.

Changing `is_call` to `True` (and not changing any of the other inputs) prints as output:

```
European Price = 12.336
American Price = 12.328
```

This is a numerical validation of our proof above that it is never optimal to exercise an American Call Option before option expiration.

The above code is in the file rl/chapter8/optimal_exercise_bin_tree.py. As ever, we encourage you to play with various choices of inputs to develop intuition for how American Option Pricing changes as a function of the inputs (and how American Put Option Exercise Boundary changes). Note that you can specify the option payoff as any arbitrary function of time and the underlying price.

Generalizing to Optimal-Stopping Problems

In this section, we generalize the problem of Optimal Exercise of American Options to the problem of Optimal Stopping in Stochastic Calculus, which has several applications in Mathematical Finance, including pricing of exotic derivatives. After defining the Optimal Stopping problem, we show how this problem can be modeled as an MDP (generalizing the MDP modeling of Optimal Exercise of American Options), which affords us the ability to solve them with Dynamic Programming or Reinforcement Learning algorithms.

First we define the concept of *Stopping Time*. Informally, Stopping Time τ is a random time (time as a random variable) at which a given stochastic process exhibits certain behavior. Stopping time is defined by a *stopping policy* to decide whether to continue or stop a stochastic process based on the stochastic process' current and past values. Formally, it is a random variable τ such that the event $\{\tau \leq t\}$ is in the σ-algebra $\mathcal{F}_t$ of the stochastic process, for all t. This means the stopping decision (i.e., *stopping policy*) of whether $\tau \leq t$ only depends on information up to time t, i.e., we have all the information required to make the stopping decision at any time t.

A simple example of Stopping Time is *Hitting Time* of a set A for a process X. Informally, it is the first time when X takes a value within the set A. Formally, Hitting Time T_{X_A} is defined as:

$$T_{X,A} = \min\{t \in \mathbb{R} | X_t \in A\}$$

A simple and common example of Hitting Time is the first time a process exceeds a certain fixed threshold level. As an example, we might say we want to sell a stock when the stock price exceeds \$100. This \$100 threshold constitutes our stopping policy, which determines the stopping time (hitting time) in terms of when we want to sell the stock (i.e., exit from owning the stock). Different people may have different criterion for exiting owing the stock (your friend's threshold might be \$90), and each person's criterion defines their own stopping policy and hence, their own stopping time random variable.

Now that we have defined Stopping Time, we are ready to define the Optimal Stopping problem. *Optimal Stopping* for a stochastic process X is a function $W(\cdot)$ whose domain is the set of potential initial values of the stochastic process and co-domain is the range of time for which the stochastic process runs, defined as:

$$W(x) = \max_{\tau} \mathbb{E}[H(X_\tau)|X_0 = x]$$

where τ is a set of stopping times of X and $H(\cdot)$ is a function from the domain of the stochastic process values to the set of real numbers.

Intuitively, you should think of Optimal Stopping as searching through many Stopping Times (i.e., many Stopping Policies), and picking out the best Stopping Policy - the one that maximizes the expected value of a function $H(\cdot)$ applied on the stochastic process at the stopping time.

Unsurprisingly (noting the connection to Optimal Control in an MDP), $W(\cdot)$ is called the Value function, and H is called the Reward function. Note that sometimes we can have several stopping times that maximize $\mathbb{E}[H(X_\tau)]$ and we say that the optimal stopping time is the smallest stopping time achieving the maximum value. We mentioned above that Optimal Exercise of American Options is a special case of Optimal Stopping. Let's understand this specialization better:

- X is the stochastic process for the underlying's price in the risk-neutral probability measure.
- x is the underlying security's current price.
- τ is a set of exercise times, each exercise time corresponding to a specific policy of option exercise (i.e., specific stopping policy).
- $W(\cdot)$ is the American Option price as a function of the underlying's current price x.
- $H(\cdot)$ is the option payoff function (with riskless-rate discounting built into $H(\cdot)$).

Now let us define Optimal Stopping problems as control problems in Markov Decision Processes (MDPs).

- The MDP *State* at time t is X_t.
- The MDP *Action* is Boolean: Stop the Process or Continue the Process.
- The MDP *Reward* is always 0, except upon Stopping, when it is equal to $H(X_\tau)$.
- The MDP *Discount Factor* γ is equal to 1.
- The MDP probabilistic-transitions are governed by the Stochastic Process X.

A specific policy corresponds to a specific stopping-time random variable τ, the Optimal Policy π^* corresponds to the stopping-time τ^* that yields the maximum (over τ) of $\mathbb{E}[H(X_\tau)|X_0 = x]$, and the Optimal Value Function V^* corresponds to the maximum value of $\mathbb{E}[H(X_\tau)|X_0 = x]$.

For discrete time steps, the Bellman Optimality Equation is:

$$V^*(X_t) = \max(H(X_t), \mathbb{E}[V^*(X_{t+1})|X_t])$$

Thus, we see that Optimal Stopping is the solution to the above Bellman Optimality Equation (solving the Control problem of the MDP described above). For a finite number of time steps, we can run a backward induction algorithm from the final time step back to time step 0 (essentially a generalization of the backward induction we did with the Binomial Options Pricing Model to determine Optimal Exercise of American Options).

Many derivatives pricing problems (and indeed many problems in the broader space of Mathematical Finance) can be cast as Optimal Stopping and hence can be modeled as MDPs (as described above). The important point here is that this enables us to employ Dynamic Programming or Reinforcement Learning algorithms to identify optimal stopping policy for exotic derivatives (which typically yields a pricing algorithm for exotic derivatives). When the state space is large (eg: when the payoff depends on several underlying assets or when the payoff depends on the history of underlying's prices, such as Asian Options-payoff with American exercise feature), the classical algorithms used in the finance industry for exotic derivatives pricing are not computationally tractable. This points to the use of Reinforcement Learning algorithms which tend to be good at handling large state spaces by effectively leveraging sampling and function approximation methodologies in the context of solving the Bellman Optimality Equation. Hence, we propose Reinforcement Learning as a promising alternative technique to pricing of certain exotic derivatives that can be cast as Optimal Stopping problems. We will discuss this more after having covered Reinforcement Learning algorithms.

Pricing/Hedging in an Incomplete Market cast as an MDP

In Subsection 7, we developed a pricing/hedging approach based on *Expected-Utility-Indifference* for the simple setting of discrete-time with single-period, when the market is incomplete. In this section, we extend this approach to the case of discrete-time with multi-period. In the single-period setting, the solution is rather straightforward as it amounts to an unconstrained multi-variate optimization together with a single-variable root-solver. Now when we extend this solution approach to the multi-period setting, it amounts to a sequential/dynamic optimal control problem. Although this is far more complex than the single-period setting, the good news is that we can model this solution approach for the multi-period setting as a Markov Decision Process. This section will be dedicated to modeling this solution approach as an MDP, which gives us enormous flexibility in capturing the real-world nuances. Besides, modeling this approach as an MDP permits us to tap into some of the recent advances in Deep Learning and Reinforcement Learning (i.e. Deep Reinforcement Learning). Since we haven't yet learnt about Reinforcement Learning algorithms, this section won't cover the algorithmic aspects (i.e., how to solve the MDP) - it will simply cover how to model the MDP for the *Expected-Utility-Indifference* approach to pricing/hedging derivatives in an incomplete market.

Before we get into the MDP modeling details, it pays to remind that in an incomplete market, we have multiple risk-neutral probability measures and hence, multiple valid derivative prices (each consistent with no-arbitrage). This means the market/traders need to "choose" a suitable risk-neutral probability measure (which amount to choosing one out of the many valid derivative prices). In practice, this "choice" is typically made in ad-hoc and inconsistent ways. Hence, our proposal of making this "choice" in a mathematically-disciplined manner by noting that ultimately a trader is interested in maximizing the "risk-adjusted return" of a derivative together with it's hedges (by sequential/dynamic adjustment of the hedge quantities). Once we take this view, it is reminiscent of the *Asset Allocation* problem we covered in Chapter 6 and the maximization objective is based on the specification of preference for trading risk versus return (which in turn, amounts to specification of a Utility function). Therefore, similar to the Asset Allocation problem, the decision at each time step is the set of adjustments one needs to make to the hedge quantities. With this rough overview, we are now ready to formalize the MDP model for this approach to multi-period pricing/hedging in an incomplete market. For ease of exposition, we simplify the problem setup a bit, although the approach and model we describe below essentially applies to more complex, more frictionful markets as well.

Assume we have a portfolio of m derivatives and we refer to our collective position across the portfolio of m derivatives as D. Assume each of these m derivatives expires by time T (i.e., all of their contingent cashflows will transpire by time T). We model the problem as a discrete-time finite-horizon MDP with the terminal time at $t = T + 1$ (i.e., all states at time $t = T + 1$ are terminal states). We require the following notation to model the MDP:

- Denote the derivatives portfolio-aggregated *Contingent Cashflows* at time t as $X_t \in \mathbb{R}$.
- Assume we have n assets trading in the market that would serve as potential hedges for our derivatives position D.
- Denote the number of units held in the hedge positions at time t as $\alpha_t \in \mathbb{R}^n$.
- Denote the cashflows per unit of hedges at time t as $Y_t \in \mathbb{R}^n$.
- Denote the prices per unit of hedges at time t as $P_t \in \mathbb{R}^n$.
- Denote the PnL position at time t as $\beta_t \in \mathbb{R}$.

We will use the notation that we have previously used for discrete-time finite-horizon MDPs, i.e., we will use time-subscripts in our notation.

We denote the State Space at time t (for all $0 \leq t \leq T + 1$) as $\mathcal{S}_t$ and a specific state at time t as $s_t \in \mathcal{S}_t$. Among other things, the key ingredients of s_t includes: $t, \alpha_t, P_t, \beta_t, D$. In practice, s_t will include many other components (in general, any market information relevant to hedge trading decisions). However, for simplicity (motivated by ease of articulation), we assume s_t is simply the 5-tuple:

$$s_t := (t, \alpha_t, P_t, \beta_t, D)$$

We denote the Action Space at time t (for all $0 \leq t \leq T$) as $\mathcal{A}_t$ and a specific action at time t as $a_t \in \mathcal{A}_t$. a_t represents the number of units of hedges traded at time t (i.e., adjustments to be made to the hedges at each time step). Since there are n hedge positions (n assets to be traded), $a_t \in \mathbb{R}^n$, i.e., $\mathcal{A}_t \subseteq \mathbb{R}^n$. Note that for each of the n assets, it's corresponding component in a_t is positive if we buy the asset at time t and negative if we sell the asset at time t. Any trading restrictions (eg: constraints on short-selling) will essentially manifest themselves in terms of the exact definition of $\mathcal{A}_t$ as a function of s_t.

State transitions are essentially defined by the random movements of prices of the assets that make up the potential hedges, i.e., $\mathbb{P}[P_{t+1}|P_t]$. In practice, this is available either as an explicit transition-probabilities model, or more likely available in the form of a *simulator*, that produces an on-demand sample of the next time step's prices, given the current time step's prices. Either way, the internals of $\mathbb{P}[P_{t+1}|P_t]$ are estimated from actual market data and realistic trading/market assumptions. The practical details of how to estimate these internals are beyond the scope of this book - it suffices to say here that this estimation is a form of supervised learning, albeit fairly nuanced due to the requirement of capturing the complexities of market-price behavior. For the following description of the MDP, simply assume that we have access to $\mathbb{P}[P_{t+1}|P_t]$ in *some form*.

It is important to pay careful attention to the sequence of events at each time step $t = 0, \ldots, T$, described below:

1. Observe the state $s_t := (t, \alpha_t, P_t, \beta_t, D)$.
2. Perform action (trades) a_t, which produces trading PnL $= -a_t \cdot P_t$ (note: this is a dot-product in $\mathbb{R}^n$).
3. These trades incur transaction costs, for example equal to $\gamma P_t \cdot |a_t|$ for some $\gamma \in \mathbb{R}^+$ (note: the absolute value applies point-wise on $a_t \in \mathbb{R}^n$, and then we take it's dot-product with $P_t \in \mathbb{R}^n$).
4. Update α_t as:
$$\alpha_{t+1} = \alpha_t + a_t$$
At termination, we need to force-liquidate, which establishes the constraint: $a_T = -\alpha_T$.
5. Realize end-of-time-step cashflows from the derivatives position D as well as from the (updated) hedge positions. This is equal to $X_{t+1} + \alpha_{t+1} \cdot Y_{t+1}$ (note: $\alpha_{t+1} \cdot Y_{t+1}$ is a dot-product in $\mathbb{R}^n$).
6. Update PnL β_t as:
$$\beta_{t+1} = \beta_t - a_t \cdot P_t - \gamma P_t \cdot |a_t| + X_{t+1} + \alpha_{t+1} \cdot Y_{t+1}$$
7. MDP Reward $r_{t+1} = 0$ for all $t = 0, \ldots, T-1$ and $r_{T+1} = U(\beta_{T+1})$ for an appropriate concave Utility function (based on the extent of risk-aversion).
8. Hedge prices evolve from P_t to P_{t+1}, based on price-transition model of $\mathbb{P}[P_{t+1}|P_t]$.

Assume we now want to enter into an incremental position of derivatives-portfolio D' in m' derivatives. We denote the combined position as $D \cup D'$. We want to determine the *Price* of the incremental position D', as well as the hedging strategy for D'.

Denote the Optimal Value Function at time t (for all $0 \leq t \leq T$) as $V_t^* : \mathcal{S}_t \rightarrow \mathbb{R}$. Pricing of D' is based on the principle that introducing the incremental position of D' together with a calibrated cash payment/receipt (Price of D') at $t = 0$ should leave the Optimal Value (at $t = 0$) unchanged. Precisely, the Price of D' is the value x^* such that

$$V_0^*((0, \alpha_0, P_0, \beta_0 - x^*, D \cup D')) = V_0^*((0, \alpha_0, P_0, \beta_0, D))$$

This Pricing principle is known as the principle of *Indifference Pricing*. The hedging strategy at time t (for all $0 \leq t < T$) is given by the Optimal Policy $\pi_t^* : \mathcal{S}_t \rightarrow \mathcal{A}_t$

Key Takeaways from this Chapter

- The concepts of Arbitrage, Completeness and Risk-Neutral Probability Measure.
- The two fundamental theorems of Asset Pricing.
- Pricing of derivatives in a complete market in two equivalent ways: A) Based on construction of a replicating portfolio, and B) Based on riskless rate-discounted expectation in the risk-neutral probability measure.
- Optimal Exercise of American Options (and it's generalization to Optimal Stopping problems) cast as an MDP Control problem.
- Pricing and Hedging of Derivatives in an Incomplete (real-world) Market cast as an MDP Control problem.

8. Order-Book Trading Algorithms

In this chapter, we venture into the world of Algorithmic Trading and specifically, we cover a couple of problems involving a trading *Order Book* that can be cast as Markov Decision Processes, and hence tackled with Dynamic Programming or Reinforcement Learning. We start the chapter by covering the basics of how trade orders are submitted and executed on an *Order Book*, a structure that allows for efficient transactions between buyers and sellers of a financial asset. Without loss of generality, we refer to the financial asset being traded on the Order Book as a "stock" and the number of units of the asset as "shares". Next we will explain how a large trade can significantly shift the Order Book, a phenomenon known as *Price Impact*. Finally, we will cover the two algorithmic problems that can be cast as MDPs. The first problem is Optimal Execution of the sales of a large number of shares of a stock so as to yield the maximum utility of proceeds from the sale over a finite horizon. This involves breaking up the sale of the shares into appropriate pieces and selling those pieces at the right times so as to achieve the goal of maximizing the utility of sales proceeds. Hence, it is an MDP Control problem where the actions are the number of shares sold at each time step. The second problem is Optimal Market-Making, i.e., the optimal "bid"s (willingness to buy a certain number of shares at a certain price) and "ask"s (willingness to sell a certain number of shares at a certain price) to be submitted on the Order Book. Again, by optimal, we mean maximization of the utility of revenues generated by the market-maker over a finite-horizon (market-makers generate revenue through the spread, i.e. gap, between the bid and ask prices they offer). This is also an MDP Control problem where the actions are the bid and ask prices along with the bid and ask shares at each time step.

Basics of Order Book and Price Impact

Some of the financial literature refers to the Order Book as Limit Order Book (abbreviated as LOB) but we will stick with the lighter language - Order Book, abbreviated as OB. The Order Book is essentially a data structure that facilitates matching stock buyers with stock sellers (i.e., an electronic marketplace). Figure 8.1 depicts a simplified view of an order book. In this order book market, buyers and sellers express their intent to trade by submitting bids (intent to buy) and "ask"s (intent to sell). These expressions of intent to buy or sell are known as Limit Orders (abbreviated as LO). The word "limit" in Limit Order refers to the fact that one is interested in buying only below a certain price level (and likewise, one is interested in selling only above a certain price level). Each

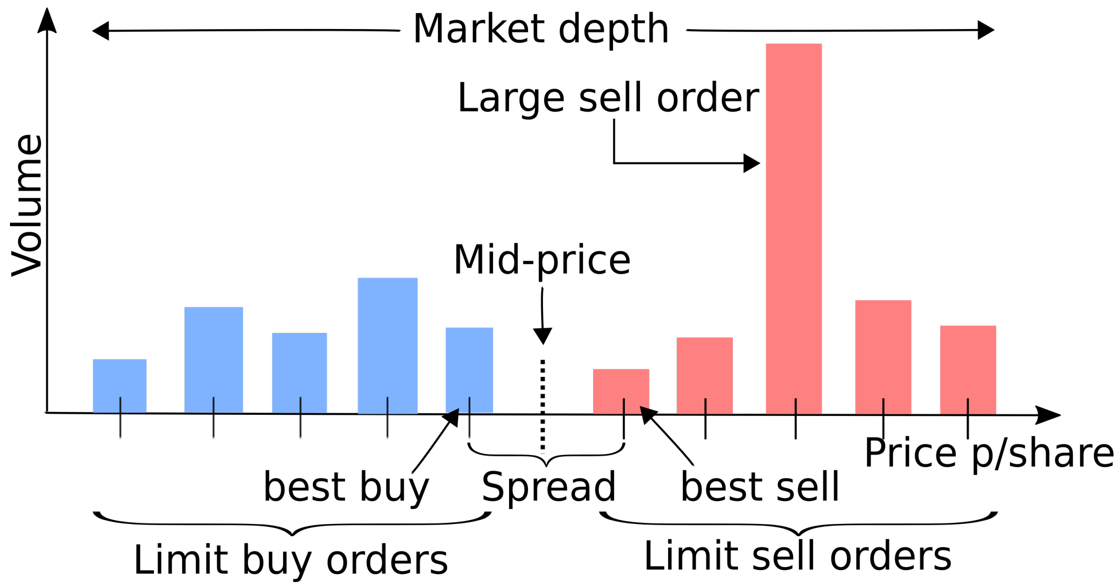

Figure 8.1.: Trading Order Book

LO is comprised of a price P and number of shares N. A bid, i.e., Buy LO (P, N) states willingness to buy N shares at a price less than or equal to P. Likewise, an ask, i.e., a Sell LO (P, N) states willingness to sell N shares at a price greater than or equal to P.

Note that multiple traders might submit LOs with the same price. The order book aggregates the number of shares for each unique price, and the OB data structure is typically presented for trading in the form of this aggregated view. Thus, the OB data structure can be represented as two sorted lists of (Price, Size) pairs:

$$\text{Buy LOs (Bids): } [(P_i^{(b)}, N_i^{(b)}) \mid 0 \leq i < m], P_i^{(b)} > P_j^{(b)} \text{ for } i < j$$

$$\text{Sell LOs (Asks): } [(P_i^{(a)}, N_i^{(a)}) \mid 0 \leq i < n], P_i^{(a)} < P_j^{(a)} \text{ for } i < j$$

Note that the Buy LOs are arranged in descending order and the Sell LOs are arranged in ascending order to signify the fact that the beginning of each list consists of the most important (best-price) LOs.

Now let's learn about some of the standard terminology:

- We refer to $P_0^{(b)}$ as *The Bid Price* (often lightened to the single word *Bid*) to signify that it is the highest offer to buy and hence, the *best* price for a seller to transact with.
- Likewise, we refer to $P_0^{(a)}$ as *The Ask Price* (often lightened to the single word *Ask*) to signify that it is the lowest offer to sell and hence, the *best* price for a buyer to transact with.
- $\frac{P_0^{(a)}+P_0^{(b)}}{2}$ is refered to as the *The Mid Price* (often lightened to the single word *Mid*).
- $P_0^{(a)} - P_0^{(b)}$ is refered to as *The Bid-Ask Spread* (often lightened to the single word *Spread*).

- $P^{(a)}_{n-1} - P^{(b)}_{m-1}$ is refered to as *The Market Depth* (often lightened to the single word *Depth*).

Although an actual real-world trading order book has many other details, we believe this simplified coverage is adequate for the purposes of core understanding of order book trading and to navigate the problems of optimal order execution and optimal market-making. Apart from Limit Orders, traders can express their interest to buy/sell with another type of order - a *Market Order* (abbreviated as MO). A Market Order (MO) states one's intent to buy/sell N shares at the *best possible price(s)* available on the OB at the time of MO submission. So, an LO is keen on price and not so keen on time (willing to wait to get the price one wants) while an MO is keen on time (desire to trade right away) and not so keen on price (will take whatever the best LO price is on the OB). So now let us understand the actual transactions that happens between LOs and MOs (buy and sell interactions, and how the OB changes as a result of these interactions). Firstly, we note that in normal trading activity, a newly submitted sell LO's price is typically above the price of the best buy LO on the OB. But if a new sell LO's price is less than or equal to the price of the best buy LO's price, we say that the *market has crossed* (to mean that the range of bid prices and the range of ask prices have intersected), which results in an immediate transaction that eats into the OB's Buy LOs.

Precisely, a new Sell LO (P, N) potentially transacts with (and hence, removes) the best Buy LOs on the OB.

$$\text{Removal: } [(P^{(b)}_i, \min(N^{(b)}_i, \max(0, N - \sum_{j=0}^{i-1} N^{(b)}_j))) \mid (i : P^{(b)}_i \geq P)] \tag{8.1}$$

After this removal, it potentially adds the following LO to the asks side of the OB:

$$(P, \max(0, N - \sum_{i:P^{(b)}_i \geq P} N^{(b)}_i)) \tag{8.2}$$

Likewise, a new Buy MO (P, N) potentially transacts with (and hence, removes) the best Sell LOs on the OB

$$\text{Removal: } [(P^{(a)}_i, \min(N^{(a)}_i, \max(0, N - \sum_{j=0}^{i-1} N^{(a)}_j))) \mid (i : P^{(a)}_i \leq P)] \tag{8.3}$$

After this removal, it potentially adds the following to the bids side of the OB:

$$(P, \max(0, N - \sum_{i:P^{(a)}_i \leq P} N^{(a)}_i)) \tag{8.4}$$

When a Market Order (MO) is submitted, things are simpler. A Sell Market Order of N shares will remove the best Buy LOs on the OB.

$$\text{Removal: } [(P_i^{(b)}, \min(N_i^{(b)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(b)}))) \mid 0 \leq i < m] \tag{8.5}$$

The sales proceeds for this MO is:

$$\sum_{i=0}^{m-1} P_i^{(b)} \cdot (\min(N_i^{(b)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(b)}))) \tag{8.6}$$

We note that if N is large, the sales proceeds for this MO can be significantly lower than the best possible sales proceeds ($= N \cdot P_0^{(b)}$), which happens if $N \leq N_0^{(b)}$. Note also that if N is large, the new Bid Price (new value of $P_0^{(b)}$) can be significantly lower than the Bid Price before the MO was submitted (because the MO "eats into" a significant volume of Buy LOs on the OB). This "eating into" the Buy LOs on the OB and consequent lowering of the Bid Price (and hence, Mid Price) is known as *Price Impact* of an MO (more specifically, as the *Temporary Price Impact* of an MO). We use the word "temporary" because subsequent to this "eating into" the Buy LOs of the OB (and consequent, "hole", ie., large Bid-Ask Spread), market participants will submit "replenishment LOs" (both Buy LOs and Sell LOs) on the OB. These replenishments LOs would typically mitigate the Bid-Ask Spread and the eventual settlement of the Bid/Mid/Ask Prices constitutes what we call *Permanent Price Impact* - which refers to the changes in OB Bid/Mid/Ask prices relative to the corresponding prices before submission of the MO.

Likewise, a Buy Market Order of N shares will remove the best Sell LOs on the OB

$$\text{Removal: } [(P_i^{(a)}, \min(N_i^{(a)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(a)}))) \mid 0 \leq i < n] \tag{8.7}$$

The purchase bill for this MO is:

$$\sum_{i=0}^{n-1} P_i^{(a)} \cdot (\min(N_i^{(a)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(a)}))) \tag{8.8}$$

If N is large, the purchase bill for this MO can be significantly lower than the best possible purchase bill ($= N \cdot P_0^{(a)}$), which happens if if $N \leq N_0^{(a)}$. All that we wrote above in terms of Temporary and Permanent Price Impact naturally apply in the opposite direction for a Buy MO.

We refer to all of the above-described OB movements, including both temporary and permanent Price Impacts broadly as *Order Book Dynamics*. There is considerable literature on modeling Order Book Dynamics and some of these models can get fairly complex in order to capture various real-world nuances. Much of this literature is beyond the scope of this book. In this chapter, we will cover a few simple models for how a sell MO will move the OB's *Bid Price* (rather than a model for how it will move the entire OB). The model for how a buy MO will move the OB's *Ask Price* is naturally identical.

Now let's write some code that models how LOs and MOs interact with the OB. We write a class `OrderBook` that represents the Buy and Sell Limit Orders on the Order Book, which are each represented as a sorted sequence of the type `DollarsAndShares`, which is a `dataclass` we created to represent any pair of a dollar amount (`dollar: float`) and number of shares (`shares: int`). Sometimes, we use `DollarsAndShares` to represent an LO (pair of price and shares) as in the case of the sorted lists of Buy and Sell LOs. At other times, we use `DollarsAndShares` to represent the pair of total dollars transacted and total shares transacted when an MO is executed on the OB. The `OrderBook` maintains a price-descending sequence of `PriceSizePairs` for Buy LOs (`descending_bids`) and a price-ascending sequence of `PriceSizePairs` for Sell LOs (`ascending_asks`). We write the basic methods to get the `OrderBook`'s highest bid price (method `bid_price`), lowest ask price (method `ask_price`), mid price (method `mid_price`), spread between the highest bid price and lowest ask price (method `bid_ask_spread`), and market depth (method `market_depth`).

```
@dataclass(frozen=True)
class DollarsAndShares:

    dollars: float
    shares: int

PriceSizePairs = Sequence[DollarsAndShares]

@dataclass(frozen=True)
class OrderBook:

    descending_bids: PriceSizePairs
    ascending_asks: PriceSizePairs

    def bid_price(self) -> float:
        return self.descending_bids[0].dollars

    def ask_price(self) -> float:
        return self.ascending_asks[0].dollars

    def mid_price(self) -> float:
        return (self.bid_price() + self.ask_price()) / 2

    def bid_ask_spread(self) -> float:
        return self.ask_price() - self.bid_price()

    def market_depth(self) -> float:
```

```
        return self.ascending_asks[-1].dollars - \
            self.descending_bids[-1].dollars
```

Next we want to write methods for LOs and MOs to interact with the OrderBook. Notice that each of Equation (8.1) (new Sell LO potentially removing some of the beginning of the Buy LOs on the OB), Equation (8.3) (new Buy LO potentially removing some of the beginning of the Sell LOs on the OB), Equation (8.5) (Sell MO removing some of the beginning of the Buy LOs on the OB) and Equation (8.7) (Buy MO removing some of the beginning of the Sell LOs on the OB) all perform a common core function - they "eat into" the most significant LOs (on the opposite side) on the OB. So we first write a `@staticmethod eat_book` for this common function.

`eat_book` takes as input a `ps_pairs: PriceSizePairs` (representing one side of the OB) and the number of `shares: int` to buy/sell. Notice `eat_book`'s return type: `Tuple[DollarsAndShares, PriceSizePairs]`. The returned `DollarsAndShares` represents the pair of dollars transacted and the number of shares transacted (with number of shares transacted being less than or equal to the input `shares`). The returned `PriceSizePairs` represents the remainder of `ps_pairs` after the transacted number of shares have eaten into the input `book`. `eat_book` first deletes (i.e. "eats up") as much of the *beginning* of the `ps_pairs: PriceSizePairs` data structure as it can (basically matching the input number of shares with an appropriate number of shares at the beginning of the `ps_pairs: PriceSizePairs` input). Note that the returned `PriceSizePairs` is a separate data structure, ensuring the immutability of the input `ps_pairs: PriceSizePairs`.

```
    @staticmethod
    def eat_book(
        ps_pairs: PriceSizePairs,
        shares: int
    ) -> Tuple[DollarsAndShares, PriceSizePairs]:
        rem_shares: int = shares
        dollars: float = 0.
        for i, d_s in enumerate(ps_pairs):
            this_price: float = d_s.dollars
            this_shares: int = d_s.shares
            dollars += this_price * min(rem_shares, this_shares)
            if rem_shares < this_shares:
                return (
                    DollarsAndShares(dollars=dollars, shares=shares),
                    [DollarsAndShares(
                        dollars=this_price,
                        shares=this_shares - rem_shares
                    )] + list(ps_pairs[i+1:])
                )
            else:
                rem_shares -= this_shares
```

```
        return (
            DollarsAndShares(dollars=dollars, shares=shares - rem_shares),
            []
        )
```

Now we are ready to write the method `sell_limit_order` which takes Sell MO Price and Sell MO shares as input. As you can see in the code below, it removes (if it "crosses") an appropriate number of shares on the Buy LO side of the OB (using the `@staticmethod eat_book`), and then potentially adds an appropriate number of shares at the Sell MO Price on the Sell MO side of the OB. `sell_limit_order` returns a new instance of `OrderBook` representing the altered OB after this transaction (note that we ensure the immutability of `self` by returning a newly-created `OrderBook`). We urge you to read the code below carefully as there are many subtle details that are handled in the code.

```
from dataclasses import replace

    def sell_limit_order(self, price: float, shares: int) -> OrderBook:
        index: Optional[int] = next((i for i, d_s
                                     in enumerate(self.descending_bids)
                                     if d_s.dollars < price), None)
        eligible_bids: PriceSizePairs = self.descending_bids \
            if index is None else self.descending_bids[:index]
        ineligible_bids: PriceSizePairs = [] if index is None else \
            self.descending_bids[index:]

        d_s, rem_bids = OrderBook.eat_book(eligible_bids, shares)
        new_bids: PriceSizePairs = list(rem_bids) + list(ineligible_bids)
        rem_shares: int = shares - d_s.shares

        if rem_shares > 0:
            new_asks: List[DollarsAndShares] = list(self.ascending_asks)
            index1: Optional[int] = next((i for i, d_s
                                          in enumerate(new_asks)
                                          if d_s.dollars >= price), None)
            if index1 is None:
                new_asks.append(DollarsAndShares(
                    dollars=price,
                    shares=rem_shares
                ))
            elif new_asks[index1].dollars != price:
                new_asks.insert(index1, DollarsAndShares(
                    dollars=price,
                    shares=rem_shares
                ))
```

```
            else:
                new_asks[index1] = DollarsAndShares(
                    dollars=price,
                    shares=new_asks[index1].shares + rem_shares
                )
            return OrderBook(
                ascending_asks=new_asks,
                descending_bids=new_bids
            )
        else:
            return replace(
                self,
                descending_bids=new_bids
            )
```

Next, we write the easier method `sell_market_order` which takes as input the number of shares to be sold (as a market order). `sell_market_order` transacts with the appropriate number of shares on the Buy LOs side of the OB (removing those many shares from the Buy LOs side). It returns a pair of `DollarsAndShares` type and `OrderBook` type. The returned `DollarsAndShares` represents the pair of dollars transacted and the number of shares transacted (with number of shares transacted being less than or equal to the input `shares`). The returned `OrderBook` represents the remainder of the OB after the transacted number of shares have eaten into the Buy LOs side of the OB. Note that the returned `OrderBook` is a newly-created data structure, ensuring the immutability of `self`.

```
    def sell_market_order(
        self,
        shares: int
    ) -> Tuple[DollarsAndShares, OrderBook]:
        d_s, rem_bids = OrderBook.eat_book(
            self.descending_bids,
            shares
        )
        return (d_s, replace(self, descending_bids=rem_bids))
```

We won't list the methods `buy_limit_order` and `buy_market_order` here as they are completely analogous (you can find the entire code for `OrderBook` in the file rl/chapter9/order_book.py). Now let us test out this code by creating a sample `OrderBook` and submitting some LOs and MOs to transact with the `OrderBook`.

```
bids: PriceSizePairs = [DollarsAndShares(
    dollars=x,
    shares=poisson(100. - (100 - x) * 10)
) for x in range(100, 90, -1)]
```

Figure 8.2.: Starting Order Book

```
asks: PriceSizePairs = [DollarsAndShares(
    dollars=x,
    shares=poisson(100. - (x - 105) * 10)
) for x in range(105, 115, 1)]

ob0: OrderBook = OrderBook(descending_bids=bids, ascending_asks=asks)
```

The above code creates an `OrderBook` in the price range [91, 114] with a bid-ask spread of 5. Figure 8.2 depicts this `OrderBook` visually.

Let's submit a Sell LO that says we'd like to sell 40 shares as long as the transacted price is greater than or equal to 107. Our Sell LO should simply get added to the Sell LO side of the OB.

```
ob1: OrderBook = ob0.sell_limit_order(107, 40)
```

The new `OrderBook` `ob1` has 40 more shares at the price level of 107, as depicted in Figure 8.3.

Now let's submit a Sell MO that says we'd like to sell 120 shares at the "best price". Our Sell MO should transact with 120 shares at "best prices" of 100 and 99 as well (since the OB does not have enough Buy LO shares at the price of 100).

```
d_s, ob2 = ob1.sell_market_order(120)
```

The new `OrderBook` `ob2` has 120 less shares on the Buy LO side of the OB, as depicted in Figure 8.4.

Now let's submit a Buy LO that says we'd like to buy 80 shares as long as the transacted price is less than or equal to 100. Our Buy LO should get added to the Buy LO side of the OB.

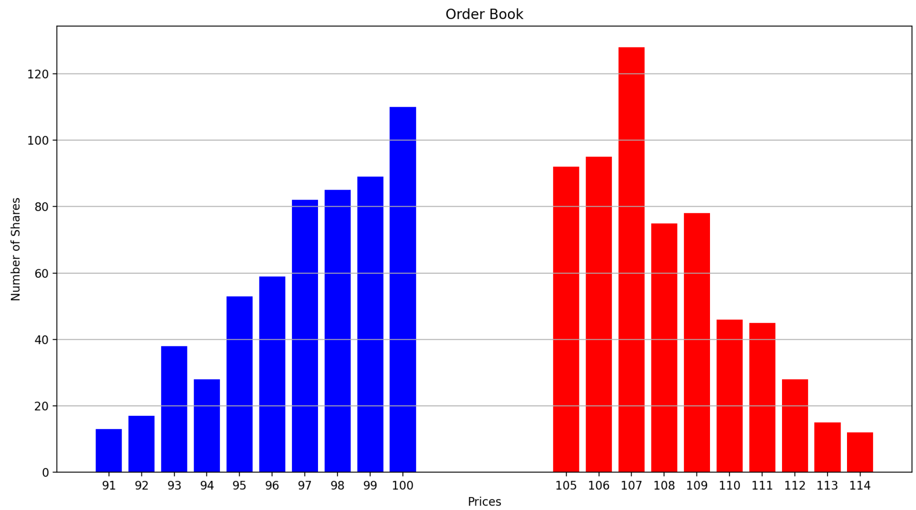

Figure 8.3.: Order Book after Sell LO

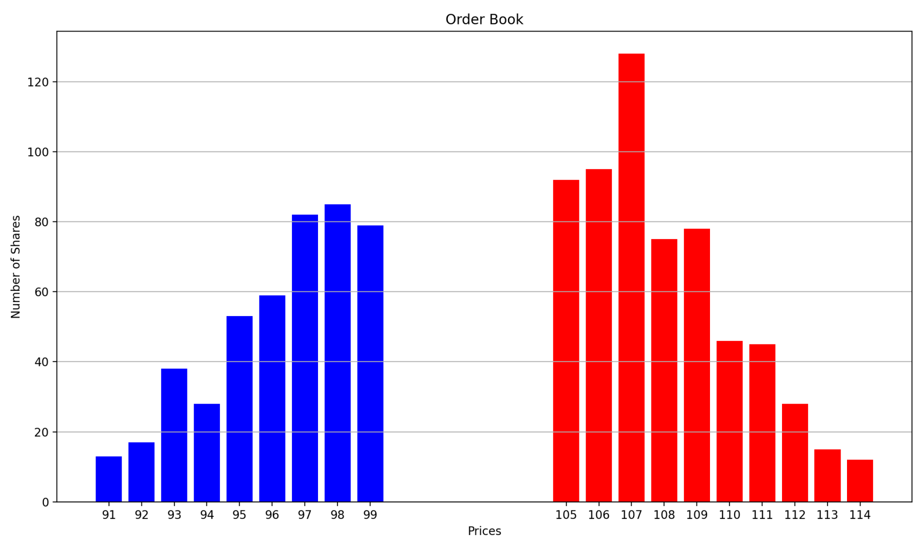

Figure 8.4.: Order Book after Sell MO

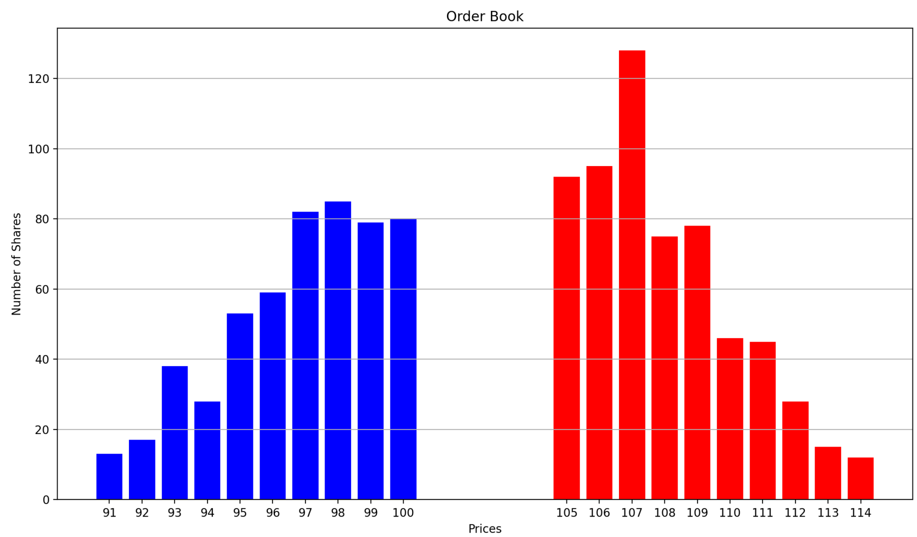

Figure 8.5.: Order Book after Buy LO

```
ob3: OrderBook = ob2.buy_limit_order(100, 80)
```

The new `OrderBook` `ob3` has re-introduced a Buy LO at the price level of 100 (now with 80 shares), as depicted in Figure 8.5.

Now let's submit a Sell LO that says we'd like to sell 60 shares as long as the transacted price is greater than or equal to 104. Our Sell LO should get added to the Sell LO side of the OB.

```
ob4: OrderBook = ob3.sell_limit_order(104, 60)
```

The new `OrderBook` `ob4` has introduced a Sell LO at a price of 104 with 60 shares, as depicted in Figure 8.6.

Now let's submit a Buy MO that says we'd like to buy 150 shares at the "best price". Our Buy MO should transact with 150 shares at "best prices" on the Sell LO side of the OB.

```
d_s, ob5 = ob4.buy_market_order(150)
```

The new `OrderBook` `ob5` has 150 less shares on the Sell LO side of the OB, wiping out all the shares at the price level of 104 and almost wiping out all the shares at the price level of 105, as depicted in Figure 8.7.

This has served as a good test of our code (transactions working as we'd like) and we encourage you to write more code of this sort to interact with the `OrderBook`, and to produce graphs of evolution of the `OrderBook` as this will help develop stronger intuition and internalize the concepts we've learnt above. All of the above code is in the file rl/chapter9/order_book.py.

Now we are ready to get started with the problem of Optimal Execution of a large-sized Market Order.

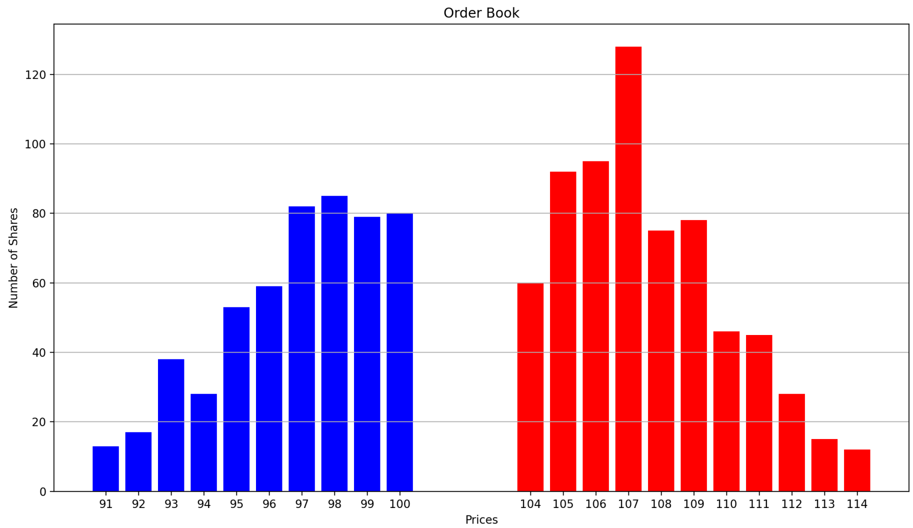

Figure 8.6.: Order Book after 2nd Sell LO

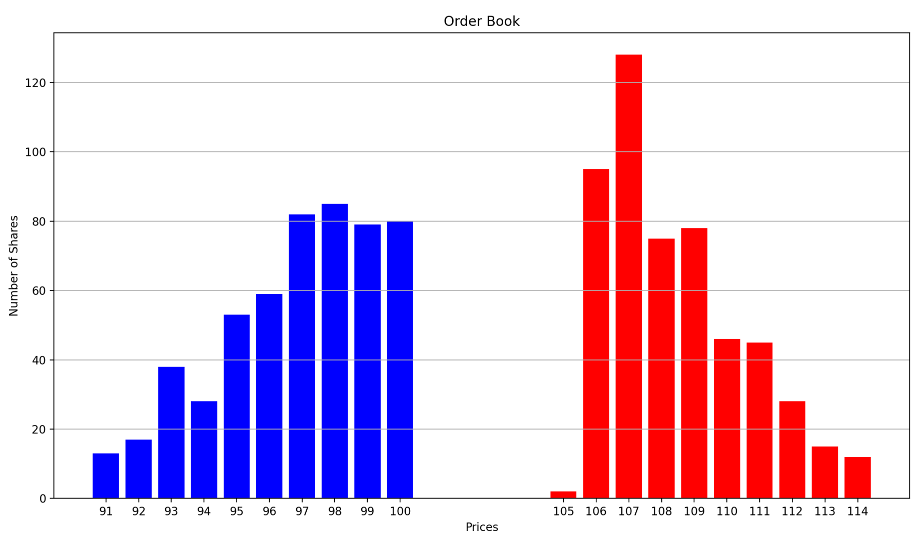

Figure 8.7.: Order Book after Buy MO

Optimal Execution of a Market Order

Imagine the following problem: You are a trader in a stock and your boss has instructed that you exit from trading in this stock because this stock doesn't meet your company's new investment requirements. You have to sell all of the N shares you own in this stock in the next T hours. You are allowed to submit sell market orders (of any size) at the start of each hour - so you have T opportunities to submit market orders of any size. Your goal is to maximize the Expected Total Utility of proceeds from sales of all N shares over the T hours. Your task is to break up N into T appropriate chunks to maximize the Expected Total Utility objective. If you attempt to sell the N shares too fast (i.e., too many in the first few hours), as we've learnt above, each (MO) sale will eat a lot into the Buy LOs on the OB (Temporary Price Impact) which would result in transacting at prices below the best price (Bid Price). Moreover, you risk moving the *Bid Price* on the OB significantly lower (Permanent Price Impact) that would affect the proceeds on the next few sales you'd make. On the other hand, if you sell the N shares too slow (i.e., too few in the first few hours), you might transact at good prices but then you risk running out of time, which means you will have to dump a lot of shares with time running out which in turn would mean transacting at prices below the best price. Moreover, selling too slow exposes you to more uncertainty in market price movements over a longer time period, and more uncertainty in sales proceeds means the Expected Utility objective gets hurt. Thus, the precise timing and sizes in the breakup of shares is vital. You will need to have an estimate of the Temporary and Permanent Price Impact of your Market Orders, which can help you identify the appropriate number of shares to sell at the start of each hour.

Unsurprisingly, we can model this problem as a Market Decision Process control problem where the actions at each time step (each hour, in this case) are the number of shares sold at the time step and the rewards are the Utility of sales proceeds at each time step. To keep things simple and intuitive, we shall model *Price Impact* of Market Orders in terms of their effect on the *Bid Price* (rather than in terms of their effect on the entire OB). In other words, we won't be modeling the entire OB Price Dynamics, just the Bid Price Dynamics. We shall refer to the OB activity of an MO immediately "eating into the Buy LOs" (and hence, potentially transacting at prices lower than the best price) as the *Temporary* Price Impact. As mentioned earlier, this is followed by subsequent replenishment of both Buy and Sell LOs on the OB (stabilizing the OB) - we refer to any eventual (end of the hour) lowering of the Bid Price (relative to the Bid Price before the MO was submitted) as the *Permanent* Price Impact. Modeling the temporary and permanent Price Impacts separately helps us in deciding on the optimal actions (optimal shares to be sold at the start of each hour).

Now we develop some formalism to describe this problem precisely. As mentioned earlier, we make a number of simplifying assumptions in modeling the OB Dynamics for ease of articulation (without diluting the most important concepts). We index discrete time by $t = 0, 1, \ldots, T$. We denote P_t as the Bid Price at the start of time step t (for all $t = 0, 1, \ldots, T$) and N_t as the number of shares sold at time step t for all $t = 0, 1, \ldots, T-1$. We denote the number of shares remaining to be sold at the start of time step t as R_t for all $t = 0, 1, \ldots, T$. Therefore,

$$R_t = N - \sum_{i=0}^{t-1} N_i \text{ for all } t = 0, 1, \ldots, T$$

Note that:

$$R_0 = N$$

$$R_{t+1} = R_t - N_t \text{ for all } t = 0, 1, \ldots, T-1$$

Also note that we need to sell everything by time $t = T$ and so:

$$N_{T-1} = R_{T-1} \Rightarrow R_T = 0$$

The model of Bid Price Dynamics from one time step to the next is given by:

$$P_{t+1} = f_t(P_t, N_t, \epsilon_t) \text{ for all } t = 0, 1, \ldots, T-1$$

where f_t is an arbitrary function incorporating:

- The Permanent Price Impact of selling N_t shares.
- The Price-Impact-independent market-movement of the Bid Price from time t to time $t+1$.
- Noise ϵ_t, a source of randomness in Bid Price movements.

The Proceeds from the sale at time step t, for all $t = 0, 1, \ldots, T-1$, is defined as:

$$N_t \cdot Q_t = N_t \cdot (P_t - g_t(P_t, N_t))$$

where g_t is a function modeling the Temporary Price Impact (i.e., the N_t MO "eating into" the Buy LOs on the OB). Q_t should be interpreted as the average Buy LO price transacted against by the N_t MO at time t.

Lastly, we denote the Utility (of Sales Proceeds) function as $U(\cdot)$.

As mentioned previously, solving for the optimal number of shares to be sold at each time step can be modeled as a discrete-time finite-horizon Markov Decision Process, which we describe below in terms of the order of MDP activity at each time step $t = 0, 1, \ldots, T-1$ (the MDP horizon is time T meaning all states at time T are terminal states). We follow the notational style of finite-horizon MDPs that should now be familiar from previous chapters.

Order of Events at time step t for all $t = 0, 1, \ldots, T-1$:

- Observe *State* $s_t := (P_t, R_t) \in \mathcal{S}_t$
- Perform *Action* $a_t := N_t \in \mathcal{A}_t$
- Receive *Reward* $r_{t+1} := U(N_t \cdot Q_t) = U(N_t \cdot (P_t - g_t(P_t, N_t)))$
- Experience Price Dynamics $P_{t+1} = f_t(P_t, N_t, \epsilon_t)$ and set $R_{t+1} = R_t - N_t$ so as to obtain the next state $s_{t+1} = (P_{t+1}, R_{t+1}) \in \mathcal{S}_{t+1}$.

Note that we have intentionally not specified the types of $\mathcal{S}_t$ and $\mathcal{A}_t$ as the types will be customized to the nuances/constraints of the specific Optimal Order Execution problem we'd be solving. Be default, we shall assume that $P_t \in \mathbb{R}^+$ and $N_t, R_t \in \mathbb{Z}_{\geq 0}$ (as these represent realistic trading situations), although we do consider special cases later in the chapter where $P_t, R_t \in \mathbb{R}$ (for analytical tractability).

The goal is to find the Optimal Policy $\pi^* = (\pi_0^*, \pi_1^*, \ldots, \pi_{T-1}^*)$ (defined as $\pi_t^*((P_t, R_t)) = N_t^*$ that maximizes:

$$\mathbb{E}[\sum_{t=0}^{T-1} \gamma^t \cdot U(N_t \cdot Q_t)]$$

where γ is the discount factor to account for the fact that future utility of sales proceeds can be modeled to be less valuable than today's.

Now let us write some code to solve this MDP. We write a class `OptimalOrderExecution` which models a fairly generic MDP for Optimal Order Execution as described above, and solves the Control problem with Approximate Value Iteration using the backward induction algorithm that we implemented in Chapter 4. Let us start by taking a look at the attributes (inputs) to `OptimalOrderExecution`:

- `shares` refers to the total number of shares N to be sold over T time steps.
- `time_steps` refers to the number of time steps T.
- `avg_exec_price_diff` refers to the time-sequenced functions g_t that return the reduction in the average price obtained by the Market Order at time t due to eating into the Buy LOs. g_t takes as input the type `PriceAndShares` that represents a pair of `price: float` and `shares: int` (in this case, the `price` is P_t and the `shares` is the MO size N_t at time t). As explained earlier, the sales proceeds at time t is: $N_t \cdot (P_t - g_t(P_t, N_t))$.
- `price_dynamics` refers to the time-sequenced functions f_t that represent the price dynamics: $P_{t+1} \sim f_t(P_t, N_t)$. f_t outputs a probability distribution of prices for P_{t+1}.
- `utility_func` refers to the Utility of Sales Proceeds function, incorporating any risk-aversion.
- `discount_factor` refers to the discount factor γ.
- `func_approx` refers to the `FunctionApprox` type to be used to approximate the Value Function for each time step.
- `initial_price_distribution` refers to the probability distribution of prices P_0 at time 0, which is used to generate the samples of states at each of the time steps (needed in the approximate backward induction algorithm).

```
@dataclass(frozen=True)
class PriceAndShares:
    price: float
    shares: int

@dataclass(frozen=True)
```

```
class OptimalOrderExecution:
    shares: int
    time_steps: int
    avg_exec_price_diff: Sequence[Callable[[PriceAndShares], float]]
    price_dynamics: Sequence[Callable[[PriceAndShares], Distribution[float]]]
    utility_func: Callable[[float], float]
    discount_factor: float
    func_approx: FunctionApprox[PriceAndShares]
    initial_price_distribution: Distribution[float]
```

The two key things we need to perform the backward induction are:

- A method `get_mdp` that given a time step t, produces the `MarkovDecisionProcess` object representing the transitions from time t to time $t+1$. The class `OptimalExecutionMDP` within `get_mdp` implements the abstract methods `step` and `action` of the abstract class `MarkovDecisionProcess`. The code should be fairly self-explanatory - just a couple of things to point out here. Firstly, the input `p_r: PriceAndShares` to the `step` method represents the state (P_t, R_t) at time t, and the variable `p_s: PriceAndDShares` represents the pair of (P_t, N_t), which serves as input to `avg_exec_price_diff` and `price_dynamics` (function attributes of `OptimalOrderExecution`). Secondly, note that the `actions` method returns an `Iterator` on a single `int` at time $t = T-1$ because of the constraint $N_{T-1} = R_{T-1}$.
- A method `get_states_distribution` that given a time step t, produces the probability distribution of states (P_t, R_t) at time t (of type `SampledDistribution[PriceA`
The code here is fairly similar to the `get_states_distribiution` method of `AssetAllocDiscrete` in Chapter 6 (essentially, walking forward from time 0 to time t by sampling from the state-transition probability distribution and also sampling from uniform choices over all actions at each time step).

```
    def get_mdp(self, t: int) -> MarkovDecisionProcess[PriceAndShares, int]:
        utility_f: Callable[[float], float] = self.utility_func
        price_diff: Sequence[Callable[[PriceAndShares], float]] = \
            self.avg_exec_price_diff
        dynamics: Sequence[Callable[[PriceAndShares], Distribution[float]]] = \
            self.price_dynamics
        steps: int = self.time_steps

        class OptimalExecutionMDP(MarkovDecisionProcess[PriceAndShares, int]):

            def step(
                self,
                p_r: PriceAndShares,
                sell: int
            ) -> SampledDistribution[Tuple[PriceAndShares, float]]:
```

```
            def sr_sampler_func(
                p_r=p_r,
                sell=sell
            ) -> Tuple[PriceAndShares, float]:
                p_s: PriceAndShares = PriceAndShares(
                    price=p_r.price,
                    shares=sell
                )
                next_price: float = dynamics[t](p_s).sample()
                next_rem: int = p_r.shares - sell
                next_state: PriceAndShares = PriceAndShares(
                    price=next_price,
                    shares=next_rem
                )
                reward: float = utility_f(
                    sell * (p_r.price - price_diff[t](p_s))
                )
                return (next_state, reward)

            return SampledDistribution(
                sampler=sr_sampler_func,
                expectation_samples=100
            )

        def actions(self, p_s: PriceAndShares) -> Iterator[int]:
            if t == steps - 1:
                return iter([p_s.shares])
            else:
                return iter(range(p_s.shares + 1))

    return OptimalExecutionMDP()

def get_states_distribution(self, t: int) -> \
        SampledDistribution[PriceAndShares]:

    def states_sampler_func() -> PriceAndShares:
        price: float = self.initial_price_distribution.sample()
        rem: int = self.shares
        for i in range(t):
            sell: int = Choose(set(range(rem + 1))).sample()
            price = self.price_dynamics[i](PriceAndShares(
                price=price,
                shares=rem
            )).sample()
            rem -= sell
        return PriceAndShares(
```

```python
                price=price,
                shares=rem
            )

        return SampledDistribution(states_sampler_func)
```

Finally, we produce the Optimal Value Function and Optimal Policy for each time step with the following method `backward_induction_vf_and_pi`:

```python
from rl.approximate_dynamic_programming import back_opt_vf_and_policy

    def backward_induction_vf_and_pi(
        self
    ) -> Iterator[Tuple[FunctionApprox[PriceAndShares],
                        Policy[PriceAndShares, int]]]:

        mdp_f0_mu_triples: Sequence[Tuple[
            MarkovDecisionProcess[PriceAndShares, int],
            FunctionApprox[PriceAndShares],
            SampledDistribution[PriceAndShares]
        ]] = [(
            self.get_mdp(i),
            self.func_approx,
            self.get_states_distribution(i)
        ) for i in range(self.time_steps)]

        num_state_samples: int = 10000
        error_tolerance: float = 1e-6

        return back_opt_vf_and_policy(
            mdp_f0_mu_triples=mdp_f0_mu_triples,
            gamma=self.discount_factor,
            num_state_samples=num_state_samples,
            error_tolerance=error_tolerance
        )
```

The above code is in the file rl/chapter9/optimal_order_execution.py. We encourage you to create a few different instances of `OptimalOrderExecution` by varying it's inputs (try different temporary and permanent price impact function, different utility functions, impose a few constraints etc.). Note the above code has been written with an educational motivation rather than an efficient-computation motivation, the convergence of the backward induction ADP algorithm is going to be slow. How do we know the above code is correct? Well, we need to create a simple special case that yields a closed-form solution that we can compare the Optimal Value Function and Optimal Policy produced by `OptimalOrderExecution` against. This will be the subject of the following subsection.

Simple Linear Price Impact Model with no Risk-Aversion

Now we consider a special case of the above-described MDP - a simple linear Price Impact model with no risk-aversion. Furthermore, for analytical tractability, we assume N, N_t, P_t are all continuous-valued (i.e., taking values $\in \mathbb{R}$).

We assume simple linear price dynamics as follows:

$$P_{t+1} = f_t(P_t, N_t, \epsilon) = P_t - \alpha \cdot N_t + \epsilon_t$$

where $\alpha \in \mathbb{R}$ and ϵ_t for all $t = 0, 1, \ldots, T-1$ are independent and identically distributed (i.i.d.) with $\mathbb{E}[\epsilon_t | N_t, P_t] = 0$. Therefore, the Permanent Price Impact is $\alpha \cdot N_t$.

As for the temporary price impact, we know that g_t needs to be a monotonically non-decreasing function of N_t. We assume a simple linear form for g_t as follows:

$$g_t(P_t, N_t) = \beta \cdot N_t \text{ for all } t = 0, 1, \ldots, T-1$$

for some constant $\beta \in \mathbb{R}_{\geq 0}$. So, $Q_t = P_t - \beta N_t$. As mentioned above, we assume no risk-aversion, i.e., the Utility function $U(\cdot)$ is assumed to be the identity function. Also, we assume that the MDP discount factor $\gamma = 1$.

Note that all of these assumptions are far too simplistic and hence, an unrealistic model of the real-world, but starting with this simple model helps build good intuition and enables us to develop more realistic models by incrementally adding complexity/nuances from this simple base model.

As ever, in order to solve the Control problem, we define the Optimal Value Function and invoke the Bellman Optimality Equation. We shall use the standard notation for discrete-time finite-horizon MDPs that we are now very familiar with.

Denote the Value Function for policy π at time t (for all $t = 0, 1, \ldots T-1$) as:

$$V_t^{\pi}((P_t, R_t)) = \mathbb{E}_{\pi}[\sum_{i=0}^{T-1} N_i \cdot (P_i - \beta \cdot N_i) | (t, P_t, R_t)]$$

Denote the Optimal Value Function at time t (for all $t = 0, 1, \ldots, T-1$) as:

$$V_t^*((P_t, R_t)) = \max_{\pi} V_t^{\pi}((P_t, R_t))$$

The Optimal Value Function satisfies the finite-horizon Bellman Optimality Equation for all $t = 0, 1, \ldots, T-2$, as follows:

$$V_t^*((P_t, R_t)) = \max_{N_t} \{N_t \cdot (P_t - \beta \cdot N_t) + \mathbb{E}[V_{t+1}^*((P_{t+1}, R_{t+1}))]\}$$

and

$$V_{T-1}^*((P_{T-1}, R_{T-1})) = N_{T-1} \cdot (P_{T-1} - \beta \cdot N_{T-1})) = R_{T-1} \cdot (P_{T-1} - \beta \cdot R_{T-1})$$

From the above, we can infer:

$$V^*_{T-2}((P_{T-2}, R_{T-2})) = \max_{N_{T-2}} \{N_{T-2} \cdot (P_{T-2} - \beta \cdot N_{T-2}) + \mathbb{E}[R_{T-1} \cdot (P_{T-1} - \beta \cdot R_{T-1})]\}$$

$$= \max_{N_{T-2}} \{N_{T-2} \cdot (P_{T-2} - \beta \cdot N_{T-2}) + \mathbb{E}[(R_{T-2} - N_{T-2})(P_{T-1} - \beta \cdot (R_{T-2} - N_{T-2}))\}$$

$$= \max_{N_{T-2}} \{N_{T-2} \cdot (P_{T-2} - \beta \cdot N_{T-2}) + (R_{T-2} - N_{T-2}) \cdot (P_{T-2} - \alpha \cdot N_{T-2} - \beta \cdot (R_{T-2} - N_{T-2}))\}$$

This simplifies to:

$$V^*_{T-2}((P_{T-2}, R_{T-2})) = \max_{N_{T-2}} \{R_{T-2} \cdot P_{T-2} - \beta \cdot R^2_{T-2} + (\alpha - 2\beta)(N^2_{T-2} - N_{T-2} \cdot R_{T-2})\} \tag{8.9}$$

For the case $\alpha \geq 2\beta$, noting that $N_{T-2} \leq R_{T-2}$, we have the trivial solution:

$$N^*_{T-2} = 0 \text{ or } N^*_{T-2} = R_{T-2}$$

Substituting either of these two values for N^*_{T-2} in the right-hand-side of Equation (8.9) gives:

$$V^*_{T-2}((P_{T-2}, R_{T-2})) = R_{T-2} \cdot (P_{T-2} - \beta \cdot R_{T-2})$$

Continuing backwards in time in this manner (for the case $\alpha \geq 2\beta$) gives:

$$N^*_t = 0 \text{ or } N^*_t = R_t \text{ for all } t = 0, 1, \ldots, T-1$$

$$V^*_t((P_t, R_t)) = R_t \cdot (P_t - \beta \cdot R_t) \text{ for all } t = 0, 1, \ldots, T-1$$

So the solution for the case $\alpha \geq 2\beta$ is to sell all N shares at any one of the time steps $t = 0, 1, \ldots, T-1$ (and none in the other time steps), and the Optimal Expected Total Sale Proceeds is $N \cdot (P_0 - \beta \cdot N)$

For the case $\alpha < 2\beta$, differentiating the term inside the max in Equation (8.9) with respect to N_{T-2}, and setting it to 0 gives:

$$(\alpha - 2\beta) \cdot (2N^*_{T-2} - R_{T-2}) = 0 \Rightarrow N^*_{T-2} = \frac{R_{T-2}}{2}$$

Substituting this solution for N^*_{T-2} in Equation (8.9) gives:

$$V^*_{T-2}((P_{T-2}, R_{T-2})) = R_{T-2} \cdot P_{T-2} - R^2_{T-2} \cdot (\frac{\alpha + 2\beta}{4})$$

Continuing backwards in time in this manner gives:

$$N^*_t = \frac{R_t}{T-t} \text{ for all } t = 0, 1, \ldots, T-1$$

$$V^*_t((P_t, R_t)) = R_t \cdot P_t - \frac{R^2_t}{2} \cdot (\frac{2\beta + \alpha \cdot (T-t-1)}{T-t}) \text{ for all } t = 0, 1, \ldots, T-1$$

Rolling forward in time, we see that $N^*_t = \frac{N}{T}$, i.e., splitting the N shares uniformly across the T time steps. Hence, the Optimal Policy is a constant deterministic function (i.e., independent of the *State*). Note that a uniform split makes intuitive sense because Price Impact and Market Movement are both linear and additive, and don't interact. This optimization is essentially equivalent to minimizing $\sum_{t=1}^T N_t^2$ with the constraint: $\sum_{t=1}^T N_t = N$. The Optimal Expected Total Sales Proceeds is equal to:

$$N \cdot P_0 - \frac{N^2}{2} \cdot (\alpha + \frac{2\beta - \alpha}{T})$$

Implementation Shortfall is the technical term used to refer to the reduction in Total Sales Proceeds relative to the maximum possible sales proceeds ($= N \cdot P_0$). So, in this simple linear model, the Implementation Shortfall from Price Impact is $\frac{N^2}{2} \cdot (\alpha + \frac{2\beta-\alpha}{T})$. Note that the Implementation Shortfall is non-zero even if one had infinite time available ($T \rightarrow \infty$) for the case of $\alpha > 0$. If Price Impact were purely temporary ($\alpha = 0$, i.e., Price fully snapped back), then the Implementation Shortfall is zero if one had infinite time available.

So now let's customize the class `OptimalOrderExecution` to this simple linear price impact model, and compare the Optimal Value Function and Optimal Policy produced by `OptimalOrderExecution` against the above-derived closed-form solutions. We write code below to create an instance of `OptimalOrderExecution` with time steps $T = 5$, total number of shares to be sold $N = 100$, linear temporary price impact with $\alpha = 0.03$, linear permanent price impact with $\beta = 0.03$, utility function as the identity function (no risk-aversion), and discount factor $\gamma = 1$. We set the standard deviation for the price dynamics probability distribution to 0 to speed up the calculation. Since we know the closed-form solution for the Optimal Value Function, we provide some assistance to `OptimalOrderExecution` by setting up a linear function approximation with two features: $P_t \cdot R_t$ and R_t^2. The task of `OptimalOrderExecution` is to infer the correct coefficients of these features for each time step. If the coefficients match that of the closed-form solution, it provides a great degree of confidence that our code is working correctly.

```
num_shares: int = 100
num_time_steps: int = 5
alpha: float = 0.03
beta: float = 0.05
init_price_mean: float = 100.0
init_price_stdev: float = 10.0

price_diff = [lambda p_s: beta * p_s.shares for _ in range(num_time_steps)]
dynamics = [lambda p_s: Gaussian(
    mu=p_s.price - alpha * p_s.shares,
    sigma=0.
) for _ in range(num_time_steps)]
ffs = [
    lambda p_s: p_s.price * p_s.shares,
    lambda p_s: float(p_s.shares * p_s.shares)
]
```

```
fa: FunctionApprox = LinearFunctionApprox.create(feature_functions=ffs)
init_price_distrib: Gaussian = Gaussian(
    mu=init_price_mean,
    sigma=init_price_stdev
)

ooe: OptimalOrderExecution = OptimalOrderExecution(
    shares=num_shares,
    time_steps=num_time_steps,
    avg_exec_price_diff=price_diff,
    price_dynamics=dynamics,
    utility_func=lambda x: x,
    discount_factor=1,
    func_approx=fa,
    initial_price_distribution=init_price_distrib
)
it_vf: Iterator[Tuple[FunctionApprox[PriceAndShares],
                      Policy[PriceAndShares, int]]] = \
    ooe.backward_induction_vf_and_pi()
```

Next we evaluate this Optimal Value Function and Optimal Policy on a particular state for all time steps, and compare that against the closed-form solution. The state we use for evaluation is as follows:

```
state: PriceAndShares = PriceAndShares(
    price=init_price_mean,
    shares=num_shares
)
```

The code to evaluate the obtained Optimal Value Function and Optimal Policy on the above state is as follows:

```
for t, (v, p) in enumerate(it_vf):
    print(f"Time {t:d}")
    print()
    opt_sale: int = p.act(state).value
    val: float = v.evaluate([state])[0]
    print(f"Optimal Sales = {opt_sale:d}, Opt Val = {val:.3f}")
    print()
    print("Optimal Weights below:")
    print(v.weights.weights)
    print()
```

With 100,000 state samples for each time step and only 10 state transition samples (since the standard deviation of ϵ is set to be very small), this prints the following:

```
Time 0

Optimal Sales = 20, Opt Val = 9779.976

Optimal Weights below:
[ 0.99999476 -0.02199715]

Time 1

Optimal Sales = 20, Opt Val = 9762.479

Optimal Weights below:
[ 0.99999345 -0.02374559]

Time 2

Optimal Sales = 20, Opt Val = 9733.324

Optimal Weights below:
[ 0.99999333 -0.02666096]

Time 3

Optimal Sales = 20, Opt Val = 9675.014

Optimal Weights below:
[ 0.99999314 -0.03249176]

Time 4

Optimal Sales = 20, Opt Val = 9500.000

Optimal Weights below:
[ 1.   -0.05]
```

Now let's compare these results against the closed-form solution.

```
for t in range(num_time_steps):
    print(f"Time {t:d}")
    print()
    left: int = num_time_steps - t
    opt_sale: float = num_shares / num_time_steps
    wt1: float = 1
    wt2: float = -(2 * beta + alpha * (left - 1)) / (2 * left)
    val: float = wt1 * state.price * state.shares + \
        wt2 * state.shares * state.shares
```

```
    print(f"Optimal Sales = {opt_sale:.3f}, Opt Val = {val:.3f}")
    print(f"Weight1 = {wt1:.3f}")
    print(f"Weight2 = {wt2:.3f}")
    print()
```

This prints the following:

```
Time 0

Optimal Sales = 20.000, Opt Val = 9780.000
Weight1 = 1.000
Weight2 = -0.022

Time 1

Optimal Sales = 20.000, Opt Val = 9762.500
Weight1 = 1.000
Weight2 = -0.024

Time 2

Optimal Sales = 20.000, Opt Val = 9733.333
Weight1 = 1.000
Weight2 = -0.027

Time 3

Optimal Sales = 20.000, Opt Val = 9675.000
Weight1 = 1.000
Weight2 = -0.033

Time 4

Optimal Sales = 20.000, Opt Val = 9500.000
Weight1 = 1.000
Weight2 = -0.050
```

We need to point out here that the general case of optimal order execution involving modeling of the entire Order Book's dynamics will need to deal with a large state space. This means the ADP algorithm will suffer from the curse of dimensionality, which means we will need to employ RL algorithms.

Paper by Bertsimas and Lo on Optimal Order Execution

This paper by Bertsimas and Lo on Optimal Order Execution considered a special case of the simple Linear Impact model we sketched above. Specifically, they assumed no risk-aversion (Utility function is identity function) and assumed that the Permanent Price Impact parameter α is equal to the Temporary Price Impact Parameter β. In the same paper, Bertsimas and Lo then extended this Linear Impact Model to include dependence on a serially-correlated variable X_t as follows:

$$P_{t+1} = P_t - (\beta \cdot N_t + \theta \cdot X_t) + \epsilon_t$$

$$X_{t+1} = \rho \cdot X_t + \eta_t$$

$$Q_t = P_t - (\beta \cdot N_t + \theta \cdot X_t)$$

where ϵ_t and η_t are each independent and identically distributed for all $t = 0, 1, \ldots, T-1$, ϵ_t and η_t are also mutually independent, and each has mean zero. X_t can be thought of as a market factor affecting P_t linearly. Applying the finite-horizon Bellman Optimality Equation on the Optimal Value Function (and the same backward-recursive approach as before) yields:

$$N_t^* = \frac{R_t}{T-t} + h(t, \beta, \theta, \rho) \cdot X_t$$

$$V_t^*((P_t, R_t, X_t)) = R_t \cdot P_t - (\text{quadratic in } (R_t, X_t) + \text{ constant})$$

Essentially, the serial-correlation predictability ($\rho \neq 0$) alters the uniform-split strategy.

In the same paper, Bertsimas and Lo presented a more realistic model called *Linear-Percentage Temporary* (abbreviated as LPT) Price Impact model, whose salient features include:

- Geometric random walk: consistent with real data, and avoids non-positive prices.
- Fractional Price Impact $\frac{g_t(P_t, N_t)}{P_t}$ doesn't depend on P_t (this is validated by real data).
- Purely Temporary Price Impact, i.e., the price P_t snaps back after the Temporary Price Impact (no Permanent effect of Market Orders on future prices).

The specific model is:

$$P_{t+1} = P_t \cdot e^{Z_t}$$

$$X_{t+1} = \rho \cdot X_t + \eta_t$$

$$Q_t = P_t \cdot (1 - \beta \cdot N_t - \theta \cdot X_t)$$

where Z_t is a random variable with mean μ_Z and variance σ_Z^2. With the same derivation methodology as before, we get the solution:

$$N_t^* = c_t^{(1)} + c_t^{(2)} R_t + c_t^{(3)} X_t$$

$$V_t^*((P_t, R_t, X_t)) = e^{\mu_Z + \frac{\sigma_Z^2}{2}} \cdot P_t \cdot (c_t^{(4)} + c_t^{(5)} R_t + c_t^{(6)} X_t + c_t^{(7)} R_t^2 + c_t^{(8)} X_t^2 + c_t^{(9)} R_t X_t)$$

where $c_t^{(k)}, 1 \leq k \leq 9$, are independent of P_t, R_t, X_t

As an exercise, we recommend implementing the above (LPT) model by customizing `OptimalOrderExecution` to compare the obtained Optimal Value Function and Optimal Policy against the closed-form solution (you can find the exact expressions for the $c_t^{(k)}$ coefficients in the Bertsimas and Lo paper).

Incorporating Risk-Aversion and Real-World Considerations

Bertsimas and Lo ignored risk-aversion for the purpose of analytical tractability. Although there was value in obtaining closed-form solutions, ignoring risk-aversion makes their model unrealistic. We have discussed in detail in Chapter 5 about the fact that traders are wary of the risk of uncertain revenues and would be willing to trade some expected revenues for lower variance of revenues. This calls for incorporating risk-aversion in the maximization objective. Almgren and Chriss wrote a seminal paper where they work in this Risk-Aversion framework. They consider our simple linear price impact model and incorporate risk-aversion by maximizing $E[Y] - \lambda \cdot Var[Y]$ where Y is the total (uncertain) sales proceeds $\sum_{t=0}^{T-1} N_t \cdot Q_t$ and λ controls the degree of risk-aversion. The incorporation of risk-aversion affects the time-trajectory of N_t^*. Clearly, if $\lambda = 0$, we get the usual uniform-split strategy: $N_t^* = \frac{N}{T}$. The other extreme assumption is to minimize $Var[Y]$ which yields: $N_0^* = N$ (sell everything immediately because the only thing we want to avoid is uncertainty of sales proceeds). In their paper, Almgren and Chriss go on to derive the *Efficient Frontier* (analogous to the Efficient Frontier Portfolio theory we outline in Appendix C). They also derive solutions for specific utility functions.

To model a real-world trading situation, the first step is to start with the MDP we described earlier with an appropriate model for the price dynamics $f_t(\cdot)$ and the temporary price impact $g_t(\cdot)$ (incorporating potential non-stationarity, non-linear price dynamics and non-linear impact). The `OptimalOrderExecution` class we wrote above allows us to incorporate all of the above. We can also model various real-world "frictions" such as discrete prices, discrete number of shares, constraints on prices and number of shares, as well as trading fees. To make the model truer to reality and more sophisticated, we can introduce various market factors in the *State* which would invariably lead to bloating of the State Space. We would also need to capture *Cross-Asset Market Impact*. As a further step, we could represent the entire Order Book (or a compact summary of the size/shape of the Order book) as part of the state, which leads to further bloating of the state space. All of this makes ADP infeasible and one would need to employ Reinforcement Learning algorithms. More importantly, we'd need to write a realistic Order Book Dynamics simulator capturing all of the above real-world considerations that an RL algorithm can use. There are a lot of practical and technical details involved in writing a real-world simulator and we won't be covering those details in this book. It suffices for here to say that the simulator would essentially be a sampling model that has learnt the Order Book Dynamics from market data (supervised learning of the Order Book

Dynamics). Using such a simulator and with a deep learning-based function approximation of the Value Function, we can solve a practical Optimal Order Execution problem with Reinforcement Learning. We refer you to a couple of papers for further reading on this:

- Paper by Nevmyvaka, Feng, Kearns in 2006
- Paper by Vyetrenko and Xu in 2019

Designing real-world simulators for Order Book Dynamics and using Reinforcement Learning for Optimal Order Execution is an exciting area for future research as well as engineering design. We hope this section has provided sufficient foundations for you to dig into this topic further.

Optimal Market-Making

Now we move on to the second problem of this chapter involving trading on an Order Book - the problem of Optimal Market-Making. A market-maker is a company/individual which/who regularly quotes bid and ask prices in a financial asset (which, without loss of generality, we will refer to as a "stock"). The market-maker typically holds some inventory in the stock, always looking to buy at their quoted bid price and sell at their quoted ask price, thus looking to make money of their *bid-ask spread*. The business of a market-maker is similar to that of a car dealer who maintains an inventory of cars and who will offer purchase and sales prices, looking to make a profit of the price spread and ensuring that the inventory of cars doesn't get too big. In this section, we consider the business of a market-maker who quotes their bid prices by submitting Buy LOs on an OB and quotes their ask prices by submitting Sell LOs on the OB. Market-makers are known as *liquidity providers* in the market because they make shares of the stock available for trading on the OB (both on the buy side and sell side). In general, anyone who submits LOs can be thought of as a market liquidity provider. Likewise, anyone who submits MOs can be thought of as a market *liquidity taker* (because an MO takes shares out of the volume that was made available for trading on the OB).

There is typically fairly complex interplay between liquidity providers (including market-makers) and liquidity takers. Modeling OB dynamics is about modeling this complex interplay, predicting arrivals of MOs and LOs, in response to market events and in response to observed activity on the OB. In this section, we view the OB from the perspective of a single market-maker who aims to make money with Buy/Sell LOs of appropriate bid-ask spread and with appropriate volume of shares (specified in their submitted LOs). The market-maker is likely to be successful if she can do a good job of forecasting OB Dynamics and dynamically adjusting her Buy/Sell LOs on the OB. The goal of the market-maker is to maximize their *Utility of Gains* at the end of a suitable horizon of time.

The core intuition in the decision of how to set the price and shares in the market-maker's Buy and Sell LOs is as follows: If the market-maker's bid-ask spread is too narrow, they will have more frequent transactions but smaller gains per transaction (more likelihood of their LOs being transacted against by an MO or an opposite-side LO). On the other hand, if the market-maker's bid-ask spread is too wide, they will have less frequent transactions but larger gains per transaction (less likelihood of their LOs being transacted against by an MO or an opposite-side LO). Also of great importance is the fact that a market-maker needs to carefully manage potentially large inventory buildup (either on the long side or the short side) so as to avoid scenarios of consequent unfavorable forced liquidation upon reaching the horizon time. Inventory buildup can occur if the market participants consistently transact against mostly one side of the market-maker's submitted LOs. With this high-level intuition, let us make these concepts of market-making precise. We start by developing some notation to help articulate the problem of Optimal Market-Making clearly. We will re-use some of the notation and terminology we had developed for the problem of Optimal Order Execution. As ever, for ease of exposition, we will simplify the setting for the Optimal Market-Making problem.

Assume there are a finite number of time steps indexed by $t = 0, 1, \ldots, T$. Assume the market-maker always shows a bid price and ask price (at each time t) along with the associated bid shares and ask shares on the OB. Also assume, for ease of exposition, that the market-maker can add or remove bid/ask shares from the OB *costlessly*. We use the following notation:

- Denote $W_t \in \mathbb{R}$ as the market-maker's trading PnL at time t.
- Denote $I_t \in \mathbb{Z}$ as the market-maker's inventory of shares at time t (assume $I_0 = 0$). Note that the inventory can be positive or negative (negative means the market-maker is short a certain number of shares).
- Denote $S_t \in \mathbb{R}^+$ as the OB Mid Price at time t (assume a stochastic process for S_t).
- Denote $P_t^{(b)} \in \mathbb{R}^+$ as the market-maker's Bid Price as time t.
- Denote $N_t^{(b)} \in \mathbb{Z}^+$ as the market-maker's Bid Shares at time t.
- Denote $P_t^{(a)} \in \mathbb{R}^+$ as the market-maker's Ask Price at time t.
- Denote $N_t^{(a)} \in \mathbb{Z}^+$ as the market-maker's Ask Shares at time t.
- We refer to $\delta_t^{(b)} = S_t - P_t^{(b)}$ as the market-maker's Bid Spread (relative to OB Mid).
- We refer to $\delta_t^{(a)} = P_t^{(a)} - S_t$ as the market-maker's Ask Spread (relative to OB Mid).
- We refer to $\delta_t^{(b)} + \delta_t^{(a)} = P_t^{(a)} - P_t^{(b)}$ as the market-maker's Bid-Ask Spread.
- Random variable $X_t^{(b)} \in \mathbb{Z}_{\geq 0}$ refers to the total number of market-maker's Bid Shares that have been transacted against (by MOs or by Sell LOs) up to time t ($X_t^{(b)}$ is often refered to as the cumulative "hits" up to time t, as in "the market-maker's buy offer has been *hit*").
- Random variable $X_t^{(a)} \in \mathbb{Z}_{\geq 0}$ refers to the total number of market-maker's Ask Shares that have been transacted against (by MOs or by Buy LOs) up to time t ($X_t^{(a)}$ is often refered to as the cumulative "lifts" up to time t, as in "the market-maker's sell offer has been *lifted*").

With this notation in place, we can write the PnL balance equation for all $t = 0, 1, \ldots, T-1$ as follows:

$$W_{t+1} = W_t + P_t^{(a)} \cdot (X_{t+1}^{(a)} - X_t^{(a)}) - P_t^{(b)} \cdot (X_{t+1}^{(b)} - X_t^{(b)}) \tag{8.10}$$

Note that since the inventory I_0 at time 0 is equal to 0, the inventory I_t at time t is given by the equation:

$$I_t = X_t^{(b)} - X_t^{(a)}$$

The market-maker's goal is to maximize (for an appropriately shaped concave utility function $U(\cdot)$) the sum of the PnL at time T and the value of the inventory of shares held at time T, i.e., we maximize:

$$\mathbb{E}[U(W_T + I_T \cdot S_T)]$$

As we alluded to earlier, this problem can be cast as a discrete-time finite-horizon Markov Decision Process (with discount factor $\gamma = 1$). Following the usual notation of discrete-time finite-horizon MDPs, the order of activity for the MDP at each time step $t = 0, 1, \ldots, T-1$ is as follows:

- Observe *State* $(S_t, W_t, I_t) \in \mathcal{S}_t$.
- Perform *Action* $(P_t^{(b)}, N_t^{(b)}, P_t^{(a)}, N_t^{(a)}) \in \mathcal{A}_t$.
- Random number of bid shares hit at time step t (this is equal to $X_{t+1}^{(b)} - X_t^{(b)}$).
- Random number of ask shares lifted at time step t (this is equal to $X_{t+1}^{(a)} - X_t^{(a)}$),
- Update of W_t to W_{t+1}.
- Update of I_t to I_{t+1}.
- Stochastic evolution of S_t to S_{t+1}.
- Receive *Reward* R_{t+1}, where

$$R_{t+1} := \begin{cases} 0 & \text{for } 1 \leq t+1 \leq T-1 \\ U(W_T + I_T \cdot S_T) & \text{for } t+1 = T \end{cases}$$

The goal is to find an *Optimal Policy* $\pi^* = (\pi_0^*, \pi_1^*, \ldots, \pi_{T-1}^*)$, where

$$\pi_t^*((S_t, W_t, I_t)) = (P_t^{(b)}, N_t^{(b)}, P_t^{(a)}, N_t^{(a)})$$

that maximizes:

$$\mathbb{E}[\sum_{t=1}^{T} R_t] = \mathbb{E}[R_T] = U(W_T + I_T \cdot S_T)$$

Avellaneda-Stoikov Continuous-Time Formulation

The landmark paper by Avellaneda and Stoikov in 2006 formulated this optimal market-making problem in it's continuous-time version. Their formulation lent itself to analytical tractability and they came up with a simple, clean and intuitive solution. In the subsection, we go over their formulation and in the next subsection, we show the derivation of their solution. We adapt our discrete-time notation above to their continuous-time setting.

$[(X_t^{(b)}|0 \leq t < T]$ and $[X_t^{(a)}|0 \leq t < T]$ are assumed to be *Poisson processes* with the means of the hit and lift rates at time t equal to $\lambda_t^{(b)}$ and $\lambda_t^{(a)}$ respectively. Hence, we can write the following:

$$dX_t^{(b)} \sim Poisson(\lambda_t^{(b)} \cdot dt)$$
$$dX_t^{(a)} \sim Poisson(\lambda_t^{(a)} \cdot dt)$$
$$\lambda_t^{(b)} = f^{(b)}(\delta_t^{(b)})$$
$$\lambda_t^{(a)} = f^{(a)}(\delta_t^{(a)})$$

for decreasing functions $f^{(b)}(\cdot)$ and $f^{(a)}(\cdot)$.

$$dW_t = P_t^{(a)} \cdot dX_t^{(a)} - P_t^{(b)} \cdot dX_t^{(b)}$$
$$I_t = X_t^{(b)} - X_t^{(a)} \text{ (note: } I_0 = 0)$$

Since infinitesimal Poisson random variables $dX_t^{(b)}$ (shares hit in time interval from t to $t + dt$) and $dX_t^{(a)}$ (shares lifted in time interval from t to $t + dt$) are Bernoulli random variables (shares hit/lifted within time interval of duration dt will be 0 or 1), $N_t^{(b)}$ and $N_t^{(a)}$ (number of shares in the submitted LOs for the infinitesimal time interval from t to $t + dt$) can be assumed to be 1.

This simplifies the *Action* at time t to be just the pair:

$$(\delta_t^{(b)}, \delta_t^{(a)})$$

OB Mid Price Dynamics is assumed to be scaled brownian motion:

$$dS_t = \sigma \cdot dz_t$$

for some $\sigma \in \mathbb{R}^+$.

The Utility function is assumed to be: $U(x) = -e^{-\gamma x}$ where $\gamma > 0$ is the coefficient of risk-aversion (this Utility function is essentially the CARA Utility function devoid of associated constants).

Solving the Avellaneda-Stoikov Formulation

We can express this continuous-time formulation as a Hamilton-Jacobi-Bellman (HJB) formulation (note: for reference, the general HJB formulation is covered in Appendix E).

We denote the Optimal Value function as $V^*(t, S_t, W_t, I_t)$.

$$V^*(t, S_t, W_t, I_t) = \max_{\delta_t^{(b)}, \delta_t^{(a)}} \mathbb{E}[-e^{-\gamma \cdot (W_T + I_T \cdot S_T)}]$$

$V^*(t, S_t, W_t, I_t)$ satisfies a recursive formulation for $0 \leq t < t_1 < T$ as follows:

$$V^*(t, S_t, W_t, I_t) = \max_{\delta_t^{(b)}, \delta_t^{(a)}} \mathbb{E}[V^*(t_1, S_{t_1}, W_{t_1}, I_{t_1})]$$

Rewriting in stochastic differential form, we have the HJB Equation:

$$\max_{\delta_t^{(b)}, \delta_t^{(a)}} \mathbb{E}[dV^*(t, S_t, W_t, I_t)] = 0 \text{ for } t < T$$

$$V^*(T, S_T, W_T, I_T) = -e^{-\gamma \cdot (W_T + I_T \cdot S_T)}$$

An infinitesimal change dV^* to $V^*(t, S_t, W_t, I_t)$ is comprised of 3 components:

- Due to pure movement in time (dependence of V^* on t).
- Due to randomness in OB Mid-Price (dependence of V^* on S_t).
- Due to randomness in hitting/lifting the market-maker's Bid/Ask (dependence of V^* on $\lambda_t^{(b)}$ and $\lambda_t^{(a)}$). Note that the probability of being hit in interval from t to $t + dt$ is $\lambda_t^{(b)} \cdot dt$ and probability of being lifted in interval from t to $t + dt$ is $\lambda_t^{(a)} \cdot dt$, upon which the PnL W_t changes appropriately and the inventory I_t increments/decrements by 1.

With this, we can expand $dV^*(t, S_t, W_t, I_t)$ and rewrite HJB as:

$$\begin{aligned}
\max_{\delta_t^{(b)}, \delta_t^{(a)}} \{ & \frac{\partial V^*}{\partial t} \cdot dt + \mathbb{E}[\sigma \cdot \frac{\partial V^*}{\partial S_t} \cdot dz_t + \frac{\sigma^2}{2} \cdot \frac{\partial^2 V^*}{\partial S_t^2} \cdot (dz_t)^2] \\
& + \lambda_t^{(b)} \cdot dt \cdot V^*(t, S_t, W_t - S_t + \delta_t^{(b)}, I_t + 1) \\
& + \lambda_t^{(a)} \cdot dt \cdot V^*(t, S_t, W_t + S_t + \delta_t^{(a)}, I_t - 1) \\
& + (1 - \lambda_t^{(b)} \cdot dt - \lambda_t^{(a)} \cdot dt) \cdot V^*(t, S_t, W_t, I_t) \\
& - V^*(t, S_t, W_t, I_t)\} = 0
\end{aligned}$$

Next, we want to convert the HJB Equation to a Partial Differential Equation (PDE). We can simplify the above HJB equation with a few observations:

- $\mathbb{E}[dz_t] = 0$.
- $\mathbb{E}[(dz_t)^2] = dt$.
- Organize the terms involving $\lambda_t^{(b)}$ and $\lambda_t^{(a)}$ better with some algebra.
- Divide throughout by dt.

$$\begin{aligned}
\max_{\delta_t^{(b)}, \delta_t^{(a)}} \{ & \frac{\partial V^*}{\partial t} + \frac{\sigma^2}{2} \cdot \frac{\partial^2 V^*}{\partial S_t^2} \\
& + \lambda_t^{(b)} \cdot (V^*(t, S_t, W_t - S_t + \delta_t^{(b)}, I_t + 1) - V^*(t, S_t, W_t, I_t)) \\
& + \lambda_t^{(a)} \cdot (V^*(t, S_t, W_t + S_t + \delta_t^{(a)}, I_t - 1) - V^*(t, S_t, W_t, I_t))\} = 0
\end{aligned}$$

Next, note that $\lambda_t^{(b)} = f^{(b)}(\delta_t^{(b)})$ and $\lambda_t^{(a)} = f^{(a)}(\delta_t^{(a)})$, and apply the max only on the relevant terms:

$$\begin{aligned}
&\frac{\partial V^*}{\partial t} + \frac{\sigma^2}{2} \cdot \frac{\partial^2 V^*}{\partial S_t^2} \\
&+ \max_{\delta_t^{(b)}} \{ f^{(b)}(\delta_t^{(b)}) \cdot (V^*(t, S_t, W_t - S_t + \delta_t^{(b)}, I_t + 1) - V^*(t, S_t, W_t, I_t)) \} \\
&+ \max_{\delta_t^{(a)}} \{ f^{(a)}(\delta_t^{(a)}) \cdot (V^*(t, S_t, W_t + S_t + \delta_t^{(a)}, I_t - 1) - V^*(t, S_t, W_t, I_t)) \} = 0
\end{aligned}$$

This combines with the boundary condition:

$$V^*(T, S_T, W_T, I_T) = -e^{-\gamma \cdot (W_T + I_T \cdot S_T)}$$

Next, we make an "educated guess" for the functional form of $V^*(t, S_t, W_t, I_t)$:

$$V^*(t, S_t, W_t, I_t) = -e^{-\gamma \cdot (W_t + \theta(t, S_t, I_t))} \tag{8.11}$$

to reduce the problem to a Partial Differential Equation (PDE) in terms of $\theta(t, S_t, I_t)$. Substituting this guessed functional form into the above PDE for $V^*(t, S_t, W_t, I_t)$ gives:

$$\begin{aligned}
&\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) \\
&+ \max_{\delta_t^{(b)}} \{ \frac{f^{(b)}(\delta_t^{(b)})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(b)} - S_t + \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t))}) \} \\
&+ \max_{\delta_t^{(a)}} \{ \frac{f^{(a)}(\delta_t^{(a)})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(a)} + S_t + \theta(t, S_t, I_t - 1) - \theta(t, S_t, I_t))}) \} = 0
\end{aligned}$$

The boundary condition is:

$$\theta(T, S_T, I_T) = I_T \cdot S_T$$

It turns out that $\theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t)$ and $\theta(t, S_t, I_t) - \theta(t, S_t, I_t - 1)$ are equal to financially meaningful quantities known as *Indifference Bid and Ask Prices*.

Indifference Bid Price $Q^{(b)}(t, S_t, I_t)$ is defined as follows:

$$V^*(t, S_t, W_t - Q^{(b)}(t, S_t, I_t), I_t + 1) = V^*(t, S_t, W_t, I_t) \tag{8.12}$$

$Q^{(b)}(t, S_t, I_t)$ is the price to buy a single share with a *guarantee of immediate purchase* that results in the Optimum Expected Utility staying unchanged.

Likewise, Indifference Ask Price $Q^{(a)}(t, S_t, I_t)$ is defined as follows:

$$V^*(t, S_t, W_t + Q^{(a)}(t, S_t, I_t), I_t - 1) = V^*(t, S_t, W_t, I_t) \tag{8.13}$$

$Q^{(a)}(t, S_t, I_t)$ is the price to sell a single share with *guarantee of immediate sale* that results in the Optimum Expected Utility staying unchanged.

For convenience, we abbreviate $Q^{(b)}(t, S_t, I_t)$ as $Q_t^{(b)}$ and $Q^{(a)}(t, S_t, I_t)$ as $Q_t^{(a)}$. Next, we express $V^*(t, S_t, W_t - Q_t^{(b)}, I_t + 1) = V^*(t, S_t, W_t, I_t)$ in terms of θ:

$$-e^{-\gamma \cdot (W_t - Q_t^{(b)} + \theta(t, S_t, I_t + 1))} = -e^{-\gamma \cdot (W_t + \theta(t, S_t, I_t))}$$

$$\Rightarrow Q_t^{(b)} = \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t) \tag{8.14}$$

Likewise for $Q_t^{(a)}$, we get:

$$Q_t^{(a)} = \theta(t, S_t, I_t) - \theta(t, S_t, I_t - 1) \tag{8.15}$$

Using Equations (8.14) and (8.15), bring $Q_t^{(b)}$ and $Q_t^{(a)}$ in the PDE for θ:

$$\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) + \max_{\delta_t^{(b)}} g(\delta_t^{(b)}) + \max_{\delta_t^{(a)}} h(\delta_t^{(b)}) = 0$$

$$\text{where } g(\delta_t^{(b)}) = \frac{f^{(b)}(\delta_t^{(b)})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(b)} - S_t + Q_t^{(b)})})$$

$$\text{and } h(\delta_t^{(a)}) = \frac{f^{(a)}(\delta_t^{(a)})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(a)} + S_t - Q_t^{(a)})})$$

To maximize $g(\delta_t^{(b)})$, differentiate g with respect to $\delta_t^{(b)}$ and set to 0:

$$e^{-\gamma \cdot (\delta_t^{(b)^*} - S_t + Q_t^{(b)})} \cdot (\gamma \cdot f^{(b)}(\delta_t^{(b)^*}) - \frac{\partial f^{(b)}}{\partial \delta_t^{(b)}}(\delta_t^{(b)^*})) + \frac{\partial f^{(b)}}{\partial \delta_t^{(b)}}(\delta_t^{(b)^*}) = 0$$

$$\Rightarrow \delta_t^{(b)^*} = S_t - P_t^{(b)^*} = S_t - Q_t^{(b)} + \frac{1}{\gamma} \cdot \log{(1 - \gamma \cdot \frac{f^{(b)}(\delta_t^{(b)^*})}{\frac{\partial f^{(b)}}{\partial \delta_t^{(b)}}(\delta_t^{(b)^*})})} \tag{8.16}$$

To maximize $h(\delta_t^{(a)})$, differentiate h with respect to $\delta_t^{(a)}$ and set to 0:

$$e^{-\gamma \cdot (\delta_t^{(a)^*} + S_t - Q_t^{(a)})} \cdot (\gamma \cdot f^{(a)}(\delta_t^{(a)^*}) - \frac{\partial f^{(a)}}{\partial \delta_t^{(a)}}(\delta_t^{(a)^*})) + \frac{\partial f^{(a)}}{\partial \delta_t^{(a)}}(\delta_t^{(a)^*}) = 0$$

$$\Rightarrow \delta_t^{(a)^*} = P_t^{(a)^*} - S_t = Q_t^{(a)} - S_t + \frac{1}{\gamma} \cdot \log{(1 - \gamma \cdot \frac{f^{(a)}(\delta_t^{(a)^*})}{\frac{\partial f^{(a)}}{\partial \delta_t^{(a)}}(\delta_t^{(a)^*})})} \tag{8.17}$$

Equations (8.16) and (8.17) are implicit equations for $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$ respectively.

Now let us write the PDE in terms of the Optimal Bid and Ask Spreads:

$$
\begin{aligned}
&\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) \\
&+ \frac{f^{(b)}(\delta_t^{(b)*})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(b)*} - S_t + \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t))}) \\
&+ \frac{f^{(a)}(\delta_t^{(a)*})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(a)*} + S_t + \theta(t, S_t, I_t - 1) - \theta(t, S_t, I_t))}) = 0 \\
&\text{with boundary condition: } \theta(T, S_T, I_T) = I_T \cdot S_T
\end{aligned}
\tag{8.18}
$$

How do we go about solving this? Here are the steps:

- First we solve PDE (8.18) for θ in terms of $\delta_t^{(b)*}$ and $\delta_t^{(a)*}$. In general, this would be a numerical PDE solution.
- Using Equations (8.14) and (8.15), and using the above-obtained θ in terms of $\delta_t^{(b)*}$ and $\delta_t^{(a)*}$, we get $Q_t^{(b)}$ and $Q_t^{(a)}$ in terms of $\delta_t^{(b)*}$ and $\delta_t^{(a)*}$.
- Then we substitute the above-obtained $Q_t^{(b)}$ and $Q_t^{(a)}$ (in terms of $\delta_t^{(b)*}$ and $\delta_t^{(a)*}$) in Equations (8.16) and (8.17).
- Finally, we solve the implicit equations for $\delta_t^{(b)*}$ and $\delta_t^{(a)*}$ (in general, numerically).

This completes the (numerical) solution to the Avellandea-Stoikov continuous-time formulation for the Optimal Market-Making problem. Having been through all the heavy equations above, let's now spend some time on building intuition.

Define the *Indifference Mid Price* $Q_t^{(m)} = \frac{Q_t^{(b)} + Q_t^{(a)}}{2}$. To develop intuition for Indifference Prices, consider a simple case where the market-maker doesn't supply any bids or asks after time t. This means the trading PnL W_T at time T must be the same as the trading PnL at time t and the inventory I_T at time T must be the same as the inventory I_t at time t. This implies:

$$V^*(t, S_t, W_t, I_t) = \mathbb{E}[-e^{-\gamma \cdot (W_t + I_t \cdot S_T)}]$$

The diffusion $dS_t = \sigma \cdot dz_t$ implies that $S_T \sim \mathcal{N}(S_t, \sigma^2 \cdot (T - t))$, and hence:

$$V^*(t, S_t, W_t, I_t) = -e^{-\gamma \cdot (W_t + I_t \cdot S_t - \frac{\gamma \cdot I_t^2 \cdot \sigma^2 \cdot (T-t)}{2})}$$

Hence,

$$V^*(t, S_t, W_t - Q_t^{(b)}, I_t + 1) = -e^{-\gamma \cdot (W_t - Q_t^{(b)} + (I_t - 1) \cdot S_t - \frac{\gamma \cdot I_t^2 \cdot \sigma^2 \cdot (T-t)}{2})}$$

But from Equation (8.12), we know that:

$$V^*(t, S_t, W_t, I_t) = V^*(t, S_t, W_t - Q_t^{(b)}, I_t + 1)$$

Therefore,

$$-e^{-\gamma \cdot (W_t + I_t \cdot S_t - \frac{\gamma \cdot I_t^2 \cdot \sigma^2 \cdot (T-t)}{2})} = -e^{-\gamma \cdot (W_t - Q_t^{(b)} + (I_t - 1) \cdot S_t - \frac{\gamma \cdot (I_t - 1)^2 \cdot \sigma^2 \cdot (T-t)}{2})}$$

This implies:

$$Q_t^{(b)} = S_t - (2I_t + 1) \cdot \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2}$$

Likewise, we can derive:

$$Q_t^{(a)} = S_t - (2I_t - 1) \cdot \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2}$$

The formulas for the Indifference Mid Price and the Indifference Bid-Ask Price Spread are as follows:

$$Q_t^{(m)} = S_t - I_t \cdot \gamma \cdot \sigma^2 \cdot (T - t)$$
$$Q_t^{(a)} - Q_t^{(b)} = \gamma \cdot \sigma^2 \cdot (T - t)$$

These results for the simple case of no-market-making-after-time-t serve as approximations for our problem of optimal market-making. Think of $Q_t^{(m)}$ as a *pseudo mid price*, an adjustment to the OB mid price S_t that takes into account the magnitude and sign of I_t. If the market-maker is long inventory $(I_t > 0)$, then $Q_t^{(m)} < S_t$, which makes intuitive sense since the market-maker is interested in reducing her risk of inventory buildup and so, would be be more inclined to sell than buy, leading her to offer bid and ask prices whose average is lower than the OB mid price S_t. Likewise, if the market-maker is short inventory $(I_t < 0)$, then $Q_t^{(m)} > S_t$ indicating inclination to buy rather than sell.

Armed with this intuition, we come back to optimal market-making, observing from Equations (8.16) and (8.17):

$$P_t^{(b)^*} < Q_t^{(b)} < Q_t^{(m)} < Q_t^{(a)} < P_t^{(a)^*}$$

Visualize this ascending sequence of prices $[P_t^{(b)^*}, Q_t^{(b)}, Q_t^{(m)}, Q_t^{(a)}, P_t^{(a)^*}]$ as jointly sliding up/down (relative to OB mid price S_t) as a function of the inventory I_t's magnitude and sign, and perceive $P_t^{(b)^*}, P_t^{(a)^*}$ in terms of their spreads to the "pseudo mid price" $Q_t^{(m)}$:

$$Q_t^{(b)} - P_t^{(m)^*} = \frac{Q_t^{(b)} + Q_t^{(a)}}{2} + \frac{1}{\gamma} \cdot \log{(1 - \gamma \cdot \frac{f^{(b)}(\delta_t^{(b)^*})}{\frac{\partial f^{(b)}}{\partial \delta_t^{(b)}}(\delta_t^{(b)^*})})}$$

$$P_t^{(a)^*} - Q_t^{(m)} = \frac{Q_t^{(b)} + Q_t^{(a)}}{2} + \frac{1}{\gamma} \cdot \log{(1 - \gamma \cdot \frac{f^{(a)}(\delta_t^{(a)^*})}{\frac{\partial f^{(a)}}{\partial \delta_t^{(a)}}(\delta_t^{(a)^*})})}$$

Analytical Approximation to the Solution to Avellaneda-Stoikov Formulation

The PDE (8.18) we derived above for θ and the associated implicit Equations (8.16) and (8.17) for $\delta_t^{(b)^*}, \delta_t^{(a)^*}$ are messy. So we make some assumptions, simplify, and derive analytical approximations. We start by assuming a fairly standard functional form for $f^{(b)}$ and $f^{(a)}$:

$$f^{(b)}(\delta) = f^{(a)}(\delta) = c \cdot e^{-k \cdot \delta}$$

This reduces Equations (8.16) and (8.17) to:

$$\delta_t^{(b)^*} = S_t - Q_t^{(b)} + \frac{1}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} \tag{8.19}$$

$$\delta_t^{(a)^*} = Q_t^{(a)} - S_t + \frac{1}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} \tag{8.20}$$

which means $P_t^{(b)^*}$ and $P_t^{(a)^*}$ are equidistant from $Q_t^{(m)}$. Substituting these simplified $\delta_t^{(b)^*}, \delta_t^{(a)^*}$ in Equation (8.18) reduces the PDE to:

$$\begin{gathered}\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) + \frac{c}{k+\gamma} \cdot (e^{-k \cdot \delta_t^{(b)^*}} + e^{-k \cdot \delta_t^{(a)^*}}) = 0 \\ \text{with boundary condition } \theta(T, S_T, I_T) = I_T \cdot S_T\end{gathered} \tag{8.21}$$

Note that this PDE (8.21) involves $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$. However, Equations (8.19), (8.20), (8.14) and (8.15) enable expressing $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$ in terms of $\theta(t, S_t, I_t - 1), \theta(t, S_t, I_t)$ and $\theta(t, S_t, I_t + 1)$. This would give us a PDE just in terms of θ. Solving that PDE for θ would give us not only $V^*(t, S_t, W_t, I_t)$ but also $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$ (using Equations (8.19), (8.20), (8.14) and (8.15)). To solve the PDE, we need to make a couple of approximations.

First we make a linear approximation for $e^{-k \cdot \delta_t^{(b)^*}}$ and $e^{-k \cdot \delta_t^{(a)^*}}$ in PDE (8.21) as follows:

$$\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) + \frac{c}{k+\gamma} \cdot (1 - k \cdot \delta_t^{(b)^*} + 1 - k \cdot \delta_t^{(a)^*}) = 0 \tag{8.22}$$

Combining the Equations (8.19), (8.20), (8.14) and (8.15) gives us:

$$\delta_t^{(b)^*} + \delta_t^{(a)^*} = \frac{2}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} + 2\theta(t, S_t, I_t) - \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t - 1)$$

With this expression for $\delta_t^{(b)^*} + \delta_t^{(a)^*}$, PDE (8.22) takes the form:

$$\begin{aligned}\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) + \frac{c}{k+\gamma} \cdot (2 - \frac{2k}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} \\ - k \cdot (2\theta(t, S_t, I_t) - \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t - 1))) = 0\end{aligned} \tag{8.23}$$

To solve PDE (8.23), we consider the following asymptotic expansion of θ in I_t:

$$\theta(t, S_t, I_t) = \sum_{n=0}^{\infty} \frac{I_t^n}{n!} \cdot \theta^{(n)}(t, S_t)$$

So we need to determine the functions $\theta^{(n)}(t, S_t)$ for all $n = 0, 1, 2, \ldots$

For tractability, we approximate this expansion to the first 3 terms:

$$\theta(t, S_t, I_t) \approx \theta^{(0)}(t, S_t) + I_t \cdot \theta^{(1)}(t, S_t) + \frac{I_t^2}{2} \cdot \theta^{(2)}(t, S_t)$$

We note that the Optimal Value Function V^* can depend on S_t only through the current *Value of the Inventory* (i.e., through $I_t \cdot S_t$), i.e., it cannot depend on S_t in any other way. This means $V^*(t, S_t, W_t, 0) = -e^{-\gamma(W_t + \theta^{(0)}(t, S_t))}$ is independent of S_t. This means $\theta^{(0)}(t, S_t)$ is independent of S_t . So, we can write it as simply $\theta^{(0)}(t)$, meaning $\frac{\partial \theta^{(0)}}{\partial S_t}$ and $\frac{\partial^2 \theta^{(0)}}{\partial S_t^2}$ are equal to 0. Therefore, we can write the approximate expansion for $\theta(t, S_t, I_t)$ as:

$$\theta(t, S_t, I_t) = \theta^{(0)}(t) + I_t \cdot \theta^{(1)}(t, S_t) + \frac{I_t^2}{2} \cdot \theta^{(2)}(t, S_t) \tag{8.24}$$

Substituting this approximation Equation (8.24) for $\theta(t, S_t, I_t)$ in PDE (8.23), we get:

$$\frac{\partial \theta^{(0)}}{\partial t} + I_t \cdot \frac{\partial \theta^{(1)}}{\partial t} + \frac{I_t^2}{2} \cdot \frac{\partial \theta^{(2)}}{\partial t} + \frac{\sigma^2}{2} \cdot (I_t \cdot \frac{\partial^2 \theta^{(1)}}{\partial S_t^2} + \frac{I_t^2}{2} \cdot \frac{\partial^2 \theta^{(2)}}{\partial S_t^2})$$
$$- \frac{\gamma \sigma^2}{2} \cdot (I_t \cdot \frac{\partial \theta^{(1)}}{\partial S_t} + \frac{I_t^2}{2} \cdot \frac{\partial \theta^{(2)}}{\partial S_t})^2 + \frac{c}{k + \gamma} \cdot (2 - \frac{2k}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} + k \cdot \theta^{(2)}) = 0$$

with boundary condition:

$$\theta^{(0)}(T) + I_T \cdot \theta^{(1)}(T, S_T) + \frac{I_T^2}{2} \cdot \theta^{(2)}(T, S_T) = I_T \cdot S_T \tag{8.25}$$

We will separately collect terms involving specific powers of I_t, each yielding a separate PDE:

- Terms devoid of I_t (i.e., I_t^0)
- Terms involving I_t (i.e., I_t^1)
- Terms involving I_t^2

We start by collecting terms involving I_t

$$\frac{\partial \theta^{(1)}}{\partial t} + \frac{\sigma^2}{2} \cdot \frac{\partial^2 \theta^{(1)}}{\partial S_t^2} = 0 \text{ with boundary condition } \theta^{(1)}(T, S_T) = S_T$$

The solution to this PDE is:

$$\theta^{(1)}(t, S_t) = S_t \tag{8.26}$$

Next, we collect terms involving I_t^2

$$\frac{\partial \theta^{(2)}}{\partial t} + \frac{\sigma^2}{2} \cdot \frac{\partial^2 \theta^{(2)}}{\partial S_t^2} - \gamma \cdot \sigma^2 \cdot (\frac{\partial \theta^{(1)}}{\partial S_t})^2 = 0 \text{ with boundary } \theta^{(2)}(T, S_T) = 0$$

Noting that $\theta^{(1)}(t, S_t) = S_t$, we solve this PDE as:

$$\theta^{(2)}(t, S_t) = -\gamma \cdot \sigma^2 \cdot (T - t) \tag{8.27}$$

Finally, we collect the terms devoid of I_t

$$\frac{\partial \theta^{(0)}}{\partial t} + \frac{c}{k+\gamma} \cdot (2 - \frac{2k}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} + k \cdot \theta^{(2)}) = 0 \text{ with boundary } \theta^{(0)}(T) = 0$$

Noting that $\theta^{(2)}(t, S_t) = -\gamma\sigma^2 \cdot (T - t)$, we solve as:

$$\theta^{(0)}(t) = \frac{c}{k+\gamma} \cdot ((2 - \frac{2k}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})}) \cdot (T - t) - \frac{k\gamma\sigma^2}{2} \cdot (T - t)^2) \tag{8.28}$$

This completes the PDE solution for $\theta(t, S_t, I_t)$ and hence, for $V^*(t, S_t, W_t, I_t)$. Lastly, we derive formulas for $Q_t^{(b)}, Q_t^{(a)}, Q_t^{(m)}, \delta_t^{(b)^*}, \delta_t^{(a)^*}$.

Using Equations (8.14) and (8.15), we get:

$$Q_t^{(b)} = \theta^{(1)}(t, S_t) + (2I_t + 1) \cdot \theta^{(2)}(t, S_t) = S_t - (2I_t + 1) \cdot \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2} \tag{8.29}$$

$$Q_t^{(a)} = \theta^{(1)}(t, S_t) + (2I_t - 1) \cdot \theta^{(2)}(t, S_t) = S_t - (2I_t - 1) \cdot \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2} \tag{8.30}$$

Using equations (8.19) and (8.20), we get:

$$\delta_t^{(b)^*} = \frac{(2I_t + 1) \cdot \gamma \cdot \sigma^2 \cdot (T - t)}{2} + \frac{1}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} \tag{8.31}$$

$$\delta_t^{(a)^*} = \frac{(1 - 2I_t) \cdot \gamma \cdot \sigma^2 \cdot (T - t)}{2} + \frac{1}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} \tag{8.32}$$

$$\text{Optimal Bid-Ask Spread } \delta_t^{(b)^*} + \delta_t^{(a)^*} = \gamma \cdot \sigma^2 \cdot (T - t) + \frac{2}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})} \tag{8.33}$$

$$\text{Optimal Pseudo-Mid } Q_t^{(m)} = \frac{Q_t^{(b)} + Q_t^{(a)}}{2} = \frac{P_t^{(b)^*} + P_t^{(a)^*}}{2} = S_t - I_t \cdot \gamma \cdot \sigma^2 \cdot (T - t) \tag{8.34}$$

Now let's get back to developing intuition. Think of $Q_t^{(m)}$ as *inventory-risk-adjusted* mid-price (adjustment to S_t). If the market-maker is long inventory ($I_t > 0$), $Q_t^{(m)} < S_t$ indicating inclination to sell rather than buy, and if market-maker is short inventory, $Q_t^{(m)} > S_t$ indicating inclination to buy rather than sell. Think of $[P_t^{(b)^*}, P_t^{(a)^*}]$ as "centered" at $Q_t^{(m)}$ (rather than at S_t), i.e., the interval

$[P_t^{(b)^*}, P_t^{(a)^*}]$ will move up/down in tandem with $Q_t^{(m)}$ moving up/down (as a function of inventory I_t). Note from Equation (8.33) that the Optimal Bid-Ask Spread $P_t^{(a)^*} - P_t^{(b)^*}$ is independent of inventory I_t.

A useful view is:

$$P_t^{(b)^*} < Q_t^{(b)} < Q_t^{(m)} < Q_t^{(a)} < P_t^{(a)^*}$$

with the spreads as follows:

$$\text{Outer Spreads } P_t^{(a)^*} - Q_t^{(a)} = Q_t^{(b)} - P_t^{(b)^*} = \frac{1}{\gamma} \cdot \log{(1 + \frac{\gamma}{k})}$$

$$\text{Inner Spreads } Q_t^{(a)} - Q_t^{(m)} = Q_t^{(m)} - Q_t^{(b)} = \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2}$$

This completes the analytical approximation to the solution of the Avellaneda-Stoikov continuous-time formulation of the Optimal Market-Making problem.

Real-World Market-Making

Note that while the Avellaneda-Stoikov continuous-time formulation and solution is elegant and intuitive, it is far from a real-world model. Real-world OB dynamics are non-stationary, non-linear and far more complex. Furthermore, there are all kinds of real-world frictions we need to capture, such as discrete time, discrete prices/number of shares in a bid/ask submitted by the market-maker, various constraints on prices and number of shares in the bid/ask, and fees to be paid by the market-maker. Moreover, we need to capture various market factors in the *State* and in the OB Dynamics. This invariably leads to the *Curse of Dimensionality* and *Curse of Modeling*. This takes us down the same path that we've now got all too familiar with - Reinforcement Learning algorithms. This means we need a simulator that captures all of the above factors, features and frictions. Such a simulator is basically a *Market-Data-learnt Sampling Model* of OB Dynamics. We won't be covering the details of how to build such a simulator as that is outside the scope of this book (a topic under the umbrella of supervised learning of market patterns and behaviors). Using this simulator and neural-networks-based function approximation of the Value Function (and/or of the Policy function), we can leverage the power of RL algorithms (to be covered in the following chapters) to solve the problem of optimal market-making in practice. There are a number of papers written on how to build practical and useful market simulators and using Reinforcement Learning for Optimal Market-Making. We refer you to two such papers here:

- 2018 paper from University of Liverpool
- 2019 paper from J.P.Morgan Research

This topic of development of models for OB Dynamics and RL algorithms for practical market-making is an exciting area for future research as well as engineering design. We hope this section has provided sufficient foundations for you to dig into this topic further.

Key Takeaways from this Chapter

- Foundations of Order Book, Limit Orders, Market Orders, Price Impact of large Market Orders, and complexity of Order Book Dynamics.
- Casting the Optimal Order Execution problem as a Markov Decision Process, developing intuition by deriving closed-form solutions for highly simplified assumptions (eg: Bertsimas-Lo and Almgren-Chriss formulations), developing a deeper understanding by implementing a backward-induction ADP algorithm, and then moving on to develop RL algorithms (and associated market simulator) to solve this problem in a real-world setting to overcome the Curse of Dimensionality and Curse of Modeling.
- Casting the Optimal Market-Making problem as a Markov Decision Process, developing intuition by deriving closed-form solutions for highly simplified assumptions (eg: Avellaneda-Stoikov formulation), developing a deeper understanding by implementing a backward-induction ADP algorithm, and then moving on to develop RL algorithms (and associated market simulator) to solve this problem in a real-world setting to overcome the Curse of Dimensionality and Curse of Modeling.

Part III.

Reinforcement Learning Algorithms

9. Monte-Carlo (MC) and Temporal-Difference (TD) for Prediction

Overview of the Reinforcement Learning approach

In Module I, we covered Dynamic Programming (DP) and Approximate Dynamic Programming (ADP) algorithms to solve the problems of Prediction and Control. DP and ADP algorithms assume that we have access to a *model* of the MDP environment (by *model*, we mean the transitions defined by $\mathcal{P}_R$ - notation from Chapter 2 - refering to probabilities of next state and reward, given current state and action). However, in real-world situations, we often do not have access to a model of the MDP environment and so, we'd need to access the actual MDP environment directly. As an example, a robotics application might not have access to a model of a certain type of terrain to learn to walk on, and so we'd need to access the actual (physical) terrain. This means we'd need to *interact* with the actual MDP environment. Note that the actual MDP environment doesn't give us transition probabilities - it simple serves up a new state and reward when we take an action in a certain state. In other words, it gives us individual instances of next state and reward, rather than the actual probabilities of occurrence of next states and rewards. So, the natural question to ask is whether we can infer the Optimal Value Function/Optimal Policy without access to a model (in the case of Prediction - the question is whether we can infer the Value Function for a given policy). The answer to this question is *Yes* and the algorithms that achieve this are known as Reinforcement Learning algorithms.

It's also important to recognize that even if we had access to a model, a typical real-world environment is non-stationary (meaning the probabilities $\mathcal{P}_R$ change over time) and so, the model would need to be re-estimated periodically. Moreover, real-world models typically have large state spaces and complex transitions structure, and so transition probabilities are either hard to compute or impossible to store/compute (within current storage/compute constraints). This means even if we could *theoretically* estimate a model from interactions with the actual environment and then run a DP/ADP algorithm, it's typically intractable/infeasible in a practical real-world situation. However, sometimes it's possible to construct a *sampling model* (a model that serves up samples of next state and reward) even when it's hard/impossible to construct a model of explicit transition probabilities. This means practically there are only two options:

1. The Agent interacts with the actual environment and doesn't bother with either a model of explicit transition probabilities or a model of transition samples.
2. We create a model (from interaction with the actual environment) of transition samples, treating this model as a simulated environment, and hence, the agent interacts with this simulated environment.

From the perspective of the agent, either way there is an environment interface that will serve up (at each time step) a single instance of (next state, reward) pair when the agent performs a certain action in a given state. So essentially, either way, our access is simply to instances of next state and reward rather than their explicit probabilities. So, then the question is - at a conceptual level, how does RL go about solving Prediction and Control problems with just this limited access (access to only instances and not explicit probabilities)? This will become clearer and clearer as we make our way through Module III, but it would be a good idea now for us to briefly sketch an intuitive overview of the RL approach (before we dive into the actual RL algorithms).

To understand the core idea of how RL works, we take you back to the start of the book where we went over how a baby learns to walk. Specifically, we'd like you to develop intuition for how humans and other animals learn to perform requisite tasks or behave in appropriate ways, so as to get trained to make suitable decisions. Humans/animals don't build a model of explicit probabilities in their minds in a way that a DP/ADP algorithm would require. Rather, their learning is essentially a sort of "trial and error" method - they try an action, receive an experience (i.e., next state and reward) from their environment, then take a new action, receive another experience, and so on, and over a period of time, they figure out which actions might be leading to good outcomes (producing good rewards) and which actions might be leading to poor outcomes (poor rewards). This learning process involves raising the priority of actions perceived as good, and lowering the priority of actions perceived as bad. Humans/animals don't quite link their actions to the immediate reward - they link their actions to the cumulative rewards (*Return*s) obtained after performing an action. Linking actions to cumulative rewards is challenging because multiple actions have significantly overlapping rewards sequences, and often rewards show up in a delayed manner. Indeed, learning by attributing good versus bad outcomes to specific past actions is the powerful part of human/animal learning. Humans/animals are essentially estimating a Q-Value Function and are updating their Q-Value function each time they receive a new experience (of essentially a pair of next state and reward). Exactly how humans/animals manage to estimate Q-Value functions efficiently is unclear (a big area of ongoing research), but RL algorithms have specific techniques to estimate the Q-Value function in an incremental manner by updating the Q-Value function in subtle ways after each experience (i.e., after every received instance of next state and reward received from either the actual environment or simulated environment).

We should also point out another important feature of human/animal learning - it is the fact that humans/animals are good at generalizing their inferences from experiences, i.e., they can interpolate and extrapolate the linkages between their actions and the outcomes received from their environment. Technically, this translates to a suitable function approximation of the Q-Value function. So before we embark on studying the details of various RL algorithms, it's important to recognize that RL overcomes complexity (specifically, the Curse of Dimensionality and Curse of Modeling, as we have alluded to in previous chapters) with a combination of:

1. Learning incrementally by updating the Q-Value function from received instances of next state and reward received after performing actions in specific states.
2. Good generalization ability of the Q-Value function with a suitable function approximation (indeed, recent progress in capabilities of deep neural networks have helped considerably).

Lastly, as mentioned in previous chapters, most RL algorithms are founded on the Bellman Equations and all RL Control algorithms are based on the fundamental idea of *Generalized Policy Iteration* that we have explained in Chapter 2. But the exact ways in which the Bellman Equations and Generalized Policy Iteration idea are utilized in RL algorithms differ from one algorithm to another, and they differ significantly from how the Bellman Equations/Generalized Policy Iteration idea is utilized in DP algorithms.

As has been our practice, we start with the Prediction problem (this chapter) and then move to the Control problem (next chapter).

RL for Prediction

We re-use a lot of the notation we had developed in Module I. As a reminder, Prediction is the problem of estimating the Value Function of an MDP for a given policy π. We know from Chapter 2 that this is equivalent to estimating the Value Function of the π-implied MRP. So in this chapter, we assume that we are working with an MRP (rather than an MDP) and we assume that the MRP is available in the form of an interface that serves up an instance of (next state, reward) pair, given current state. The interface might be a real environment or a simulated environment. We refer to the agent's receipt of an instance of (next state, reward), given current state, as an *atomic experience*. Interacting with this interface in succession (starting from a state S_0) gives us a *trace experience* consisting of alternating states and rewards as follows:

$$S_0, R_1, S_1, R_2, S_2, \ldots$$

Given a stream of atomic experiences or a stream of trace experiences, the RL Prediction problem is to estimate the *Value Function* $V : \mathcal{N} \rightarrow \mathbb{R}$ of the MRP defined as:

$$V(s) = \mathbb{E}[G_t|S_t = s] \text{ for all } s \in \mathcal{N}, \text{ for all } t = 0, 1, 2, \ldots$$

where the *Return* G_t for each $t = 0, 1, 2, \ldots$ is defined as:

$$G_t = \sum_{i=t+1}^{\infty} \gamma^{i-t-1} \cdot R_i = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots = R_{t+1} + \gamma \cdot G_{t+1}$$

We use the above definition of *Return* even for a terminating trace experience (say terminating at $t = T$, i.e., $S_T \in \mathcal{T}$), by treating $R_i = 0$ for all $i > T$.

The RL prediction algorithms we will soon develop consume a stream of atomic experiences or a stream of trace experiences to learn the requisite Value Function. So we want the input to an RL Prediction algorithm to be either an `Iterable` of atomic experiences or an `Iterable` of trace experiences. Now let's talk about the representation (in code) of a single atomic experience and the representation of a single trace experience. We take you back to the code in Chapter 1 where we had set up a `@dataclass TransitionStep` that served as a building block in the method `simulate_reward` in the abstract class `MarkovRewardProcess`. `TransitionStep[S]` will be our representation for a single atomic experience. `simulate_reward` produces an `Iterator[TransitionStep[S]]` (i.e., a stream of atomic experiences in the form of a sampling trace) but in general, we can represent a single trace experience as an `Iterable[TransitionStep[S]]` (i.e., a sequence *or* stream of atomic experiences). Therefore, we want the input to an RL prediction problem to be either an `Iterable[TransitionStep[S]]` (representing an `Iterable` of atomic experiences) or an `Iterable[Iterable[TransitionStep[S]]]` (representing an `Iterable` of trace experiences).

Let's add a method `reward_traces` to `MarkovRewardProcess` that produces an `Iterator` (stream) of the sampling traces produced by `simulate_reward`. So then we'd be able to use the output of `reward_traces` as the `Iterable[Iterable[TransitionStep` input to an RL Prediction algorithm. Note that the input `start_state_distribution` is the specification of the probability distribution of start states (state from which we start a sampling trace that can be used as a trace experience).

```
def reward_traces(
        self,
        start_state_distribution: Distribution[S]
) -> Iterable[Iterable[TransitionStep[S]]]:
    while True:
        yield self.simulate_reward(start_state_distribution)
```

The code above is in the file rl/markov_process.py.

Monte-Carlo (MC) Prediction

Monte-Carlo (MC) Prediction is a very simple RL algorithm that performs supervised learning to predict the expected return from any state of an MRP (i.e., it estimates the Value Function of an MRP), given a stream of trace experiences. Note that we wrote the abstract class `FunctionApprox` in Chapter 4 for supervised learning that takes data in the form of (x, y) pairs where x is the predictor variable and $y \in \mathbb{R}$ is the response variable. For the Monte-Carlo prediction problem, the x-values are the encountered states across the stream of input trace experiences and the y-values are the associated returns on the trace experience (starting from the corresponding encountered state). The following function (in the file rl/monte_carlo.py) `evaluate_mrp` takes as input an `Iterable` of trace experiences, with each trace experience represented as an `Iterable` of `TransitionSteps`. `evaluate_mrp` performs the requisite supervised learning in an incremental manner, by calling the method `update` of `approx_0: FunctionApprox[S]` on an `Iterator` of (state, return) pairs that are extracted from the input trace experiences. `evaluate_mrp` produces an `Iterator` of `FunctionApprox[S]`, i.e., an updated function approximation of the Value Function at the end of each trace experience in the input trace experiences (note that function approximation updates can be done only at the end of trace experiences because the trace experience returns are available only at the end of trace experiences).

```
import MarkovRewardProcess as mp

def evaluate_mrp(
        traces: Iterable[Iterable[mp.TransitionStep[S]]],
        approx_0: FunctionApprox[S],
        gamma: float,
        tolerance: float = 1e-6
) -> Iterator[FunctionApprox[S]]:
    episodes = (returns(trace, gamma, tolerance) for trace in traces)

    return approx_0.iterate_updates(
        ((step.state, step.return_) for step in episode)
        for episode in episodes
    )
```

The core of the `evaluate_mrp` function above is the call to the `returns` function (detailed below and available in the file rl/returns.py). `returns` takes as input `trace` representing a trace experience (`Iterable` of `TransitionStep`), the discount factor `gamma`, and a `tolerance` that determines how many time steps to cover in each trace experience when $\gamma < 1$ (as many steps as until $\gamma^{steps} \leq tolerance$ or until the trace experience ends in a terminal state, whichever happens first). If $\gamma = 1$, each trace experience needs to end in a terminal state (else the `returns` function will loop forever).

The `returns` function calculates the returns G_t (accumulated discounted rewards) starting from each state S_t in the trace experience. The key is to walk backwards from the end of the trace experience to the start (so as to reuse the calculated returns while walking backwards: $G_t = R_{t+1} + \gamma \cdot G_{t+1}$). Note the use of `itertools.accumulate` to perform this backwards-walk calculation, which in turn uses the `add_return` method in `TransitionStep` to create an instance of `ReturnStep`. The `ReturnStep` (as seen in the code below) class is derived from the `TransitionStep` class and includes the additional attribute named `return_`. We add a method called `add_return` in `TransitionStep` so we can augment the attributes `state`, `reward`, `next_state` with the additional attribute `return_` that is comprised of the reward plus gamma times the `return_` from the next state.

```
@dataclass(frozen=True)
class TransitionStep(Generic[S]):
    state: S
    next_state: S
    reward: float

    def add_return(self, gamma: float, return_: float) -> ReturnStep[S]:
        return ReturnStep(
            self.state,
            self.next_state,
            self.reward,
            return_=self.reward + gamma * return_
        )

@dataclass(frozen=True)
class ReturnStep(TransitionStep[S]):
    return_: float
```

The above code is in the file rl/markov_process.py. The code below is in the file rl/returns.py.

```
import itertools
import rl.markov_process as mp

def returns(
        trace: Iterable[mp.TransitionStep[S]],
        gamma: float,
        tolerance: float
) -> Iterator[mp.ReturnStep[S]]:
    trace = iter(trace)

    max_steps = round(math.log(tolerance) / math.log(gamma)) \
        if gamma < 1 else None
    if max_steps is not None:
        trace = itertools.islice(trace, max_steps * 2)

    *transitions, last_transition = list(trace)

    return_steps = itertools.accumulate(
        reversed(transitions),
```

```
        func=lambda next, curr: curr.add_return(gamma, next.return_),
        initial=last_transition.add_return(gamma, 0)
    )
    return_steps = reversed(list(return_steps))

    if max_steps is not None:
        return_steps = itertools.islice(return_steps, max_steps)

    return return_steps
```

We say that the trace experiences are *episodic traces* if each trace experience ends in a terminal state to signify that each trace experience is an episode, after whose termination we move on to the next episode. Trace experiences that do not terminate are known as *continuing traces*. We say that an RL problem is *episodic* if the input trace experiences are *episodic* (likewise, we say that an RL problem is *continuing* if the input trace experiences as *continuing*).

It is common to assume that the probability distribution of returns conditional on a state is a normal distribution with mean given by a function approximation for the Value Function that we denote as $V(s;\boldsymbol{w})$ where s is a state for which the function approximation is being evaluated and $\boldsymbol{w}$ is the set of parameters in the function approximation (eg: the weights in a neural network). Then, the loss function for supervised learning of the Value Function is the sum of squares of differences between observed returns and the Value Function estimate from the function approximation. For a state S_t visited at time t in a trace experience and the associated return G_t on the trace experience, the contribution to the loss function is:

$$\mathcal{L}_{(S_t,G_t)}(\boldsymbol{w}) = \frac{1}{2} \cdot (V(S_t;\boldsymbol{w}) - G_t)^2 \tag{9.1}$$

It's gradient with respect to $\boldsymbol{w}$ is:

$$\nabla_{\boldsymbol{w}} \mathcal{L}_{(S_t,G_t)}(\boldsymbol{w}) = (V(S_t;\boldsymbol{w}) - G_t) \cdot \nabla_{\boldsymbol{w}} V(S_t;\boldsymbol{w})$$

We know that the change in the parameters (adjustment to the parameters) is equal to the negative of the gradient of the loss function, scaled by the learning rate (let's denote the learning rate as α). Then the change in parameters is:

$$\Delta \boldsymbol{w} = \alpha \cdot (G_t - V(S_t;\boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t;\boldsymbol{w}) \tag{9.2}$$

This is a standard formula for change in parameters in response to incoming data for supervised learning when the response variable has a conditional normal distribution. But it's useful to see this formula in an intuitive manner for this specialization of supervised learning to Reinforcement Learning parameter updates. We should interpret the change in parameters $\Delta \boldsymbol{w}$ as the product of three conceptual entities:

- *Learning Rate* α

- *Return Residual* of the observed return G_t relative to the estimated conditional expected return $V(S_t; \boldsymbol{w})$
- *Estimate Gradient* of the conditional expected return $V(S_t; \boldsymbol{w})$ with respect to the parameters $\boldsymbol{w}$

This interpretation of the change in parameters as the product of these three conceptual entities: (Learning rate, Return Residual, Estimate Gradient) is important as this will be a repeated pattern in many of the RL algorithms we will cover.

Now we consider a simple case of Monte-Carlo Prediction where the MRP consists of a finite state space with the non-terminal states $\mathcal{N} = \{s_1, s_2, \ldots, s_m\}$. In this case, we represent the Value Function of the MRP in a data structure (dictionary) of (state, expected return) pairs. This is known as "Tabular" Monte-Carlo (more generally as Tabular RL to reflect the fact that we represent the calculated Value Function in a "table", i.e., dictionary). Note that in this case, Monte-Carlo Prediction reduces to a very simple calculation wherein for each state, we simply maintain the average of the trace experience returns from that state onwards (averaged over state visitations across trace experiences), and the average is updated in an incremental manner. Recall from Section 4 of Chapter 4 that this is exactly what's done in the `Tabular` class (in file rl/func_approx.py). We also recall from Section4 of Chapter 4 that `Tabular` implements the interface of the abstract class `FunctionApprox` and so, we can perform Tabular Monte-Carlo Prediction by passing a `Tabular` instance as the `approx0: FunctionApprox` argument to the `evaluate_mrp` function above. The implementation of the `update` method in Tabular is exactly as we desire: it performs an incremental averaging of the trace experience returns obtained from each state onwards (over a stream of trace experiences).

Let us denote $V_n(s_i)$ as the estimate of the Value Function for a state s_i after the n-th occurrence of the state s_i (when doing Tabular Monte-Carlo Prediction) and let $Y_i^{(1)}, Y_i^{(2)}, \ldots, Y_i^{(n)}$ be the trace experience returns associated with the n occurrences of state s_i. Let us denote the `count_to_weight_func` attribute of `Tabular` as f, Then, the `Tabular` update at the n-th occurrence of state s_i (with it's associated return $Y_i^{(n)}$) is as follows:

$$V_n(s_i) = (1-f(n)) \cdot V_{n-1}(s_i) + f(n) \cdot Y_i^{(n)} = V_{n-1}(s_i) + f(n) \cdot (Y_i^{(n)} - V_{n-1}(s_i)) \quad (9.3)$$

Thus, we see that the update (change) to the Value Function for a state s_i is equal to $f(n)$ (weight for the latest trace experience return $Y_i^{(n)}$ from state s_i) times the difference between the latest trace experience return $Y_i^{(n)}$ and the current Value Function estimate $V_{n-1}(s_i)$. This is a good perspective as it tells us how to adjust the Value Function estimate in an intuitive manner. In the case of the default setting of `count_to_weight_func` as $f(n) = \frac{1}{n}$, we get:

$$V_n(s_i) = \frac{n-1}{n} \cdot V_{n-1}(s_i) + \frac{1}{n} \cdot Y_i^{(n)} = V_{n-1}(s_i) + \frac{1}{n} \cdot (Y_i^{(n)} - V_{n-1}(s_i)) \quad (9.4)$$

So if we have 9 occurrences of a state with an average trace experience return of 50 and if the 10th occurrence of the state gives a trace experience return of 60, then we consider $\frac{1}{10}$ of $60 - 50$ (equal to 1) and increase the Value Function estimate for the state from 50 to 50+1 = 51. This illustrates how we move the Value Function in the direction of of the gap between the latest return and the current estimated expected return, but by a magnitude of only $\frac{1}{n}$ of the gap.

Expanding the incremental updates across values of n in Equation (9.3), we get:

$$\begin{aligned} V_n(s_i) =& f(n) \cdot Y_i^{(n)} + (1 - f(n)) \cdot f(n-1) \cdot Y_i^{(n-1)} + \ldots \\ & + (1 - f(n)) \cdot (1 - f(n-1)) \cdots (1 - f(2)) \cdot f(1) \cdot Y_i^{(1)} \end{aligned} \tag{9.5}$$

In the case of the default setting of `count_to_weight_func` as $f(n) = \frac{1}{n}$, we get:

$$V_n(s_i) = \frac{1}{n} \cdot Y_i^{(n)} + \frac{n-1}{n} \cdot \frac{1}{n-1} \cdot Y_i^{(n-1)} + \ldots + \frac{n-1}{n} \cdot \frac{n-2}{n-1} \cdots \frac{1}{2} \cdot \frac{1}{1} \cdot Y_i^{(1)} = \frac{\sum_{k=1}^{n} Y_i^{(k)}}{n} \tag{9.6}$$

which is an equally-weighted average of the trace experience returns from the state. From the Law of Large Numbers, we know that the sample average converges to the expected value, which is the core idea behind the Monte-Carlo method.

Note that the `Tabular` class as an implementation of the abstract class `FunctionApprox` is not just a software design happenstance - there is a formal mathematical specialization here that is vital to recognize. This tabular representation is actually a special case of linear function approximation by setting a feature function $\phi_i(\cdot)$ for each x_i as: $\phi_i(x) = 1$ for $x = x_i$ and $\phi(x) = 0$ for each $x \neq x_i$ (i.e., $\phi_i(x)$ is the indicator function for x_i, and the key $\boldsymbol{\Phi}$ matrix of Chapter 4 reduces to the identity matrix). In using `Tabular` for Monte-Carlo Prediction, the feature functions are the indicator functions for each of the non-terminal states and the linear-approximation parameters w_i are the Value Function estimates for the corresponding non-terminal states.

With this understanding, we can view Tabular RL as a special case of RL with Linear Function Approximation of the Value Function. Moreover, the `count_to_weight_func` attribute of `Tabular` plays the role of the learning rate (as a function of the number of iterations in stochastic gradient descent). This becomes clear if we write Equation (9.3) in terms of parameter updates: write $V_n(s_i)$ as parameter value $w_i^{(n)}$ to denote the n-th update to parameter w_i corresponding to state s_i, and write $f(n)$ as learning rate α_n for the n-th update to w_i.

$$w_i^{(n)} = w_i^{(n-1)} + \alpha_n \cdot (Y_i^{(n)} - w_i^{(n-1)})$$

So, the change in parameter w_i for state s_i is α_n times $Y_i^{(n)} - w_i^{(n-1)}$. We observe that $Y_i^{(n)} - w_i^{(n-1)}$ represents the gradient of the loss function for the data point

$(s_i, Y_i^{(n)})$ in the case of linear function approximation with features as indicator variables (for each state). This is because the loss function for the data point $(s_i, Y_i^{(n)})$ is $\frac{1}{2} \cdot (Y_i^{(n)} - \sum_{j=1}^{m} \phi_j(s_i) \cdot w_j)^2$ which reduces to $\frac{1}{2} \cdot (Y_i^{(n)} - w_i^{(n-1)})^2$, whose gradient in the direction of w_i is $Y_i^{(n)} - w_i^{(n-1)}$ and 0 in the other directions (for $j \neq i$). So we see that `Tabular` updates are basically a special case of `LinearFunctionApprox` updates if we set the features to be indicator functions for each of the states (with `count_to_weight_func` playing the role of the learning rate).

Now that you recognize that `count_to_weight_func` essentially plays the role of the learning rate and governs the importance given to the latest trace experience return relative to past trace experience returns, we want to point out that real-world situations are non-stationary in the sense that the environment typically evolves over a period of time and so, RL algorithms have to appropriately adapt to the changing environment. The way to adapt effectively is to have an element of "forgetfulness" of the past because if one learns about the distant past far too strongly in a changing environment, our predictions (and eventually control) would not be effective. So, how does an RL algorithm "forget"? Well, one can "forget" through an appropriate time-decay of the weights when averaging trace experience returns. If we set a constant learning rate α (in `Tabular`, this would correspond to `count_to_weight_func=lambda _: alpha`), we'd obtain "forgetfulness" with lower weights for old data points and higher weights for recent data points. This is because with a constant learning rate α, Equation (9.5) reduces to:

$$
\begin{aligned}
V_n(s_i) &= \alpha \cdot Y_i^{(n)} + (1-\alpha) \cdot \alpha \cdot Y_i^{(n-1)} + \ldots + (1-\alpha)^{n-1} \cdot \alpha \cdot Y_i^{(1)} \\
&= \sum_{j=1}^{n} \alpha \cdot (1-\alpha)^{n-j} \cdot Y_i^{(j)}
\end{aligned}
$$

which means we have exponentially-decaying weights in the weighted average of the trace experience returns for any given state.

Note that for $0 < \alpha \leq 1$, the weights sum up to 1 as n tends to infinity, i.e.,

$$
\lim_{n \to \infty} \sum_{j=1}^{n} \alpha \cdot (1-\alpha)^{n-j} = \lim_{n \to \infty} 1 - (1-\alpha)^n = 1
$$

It's worthwhile pointing out that the Monte-Carlo algorithm we've implemented above is known as Each-Visit Monte-Carlo to refer to the fact that we include each occurrence of a state in a trace experience. So if a particular state appears 10 times in a given trace experience, we have 10 (state, return) pairs that are used to make the update (for just that state) at the end of that trace experience. This is in contrast to First-Visit Monte-Carlo in which only the first occurrence of a state in a trace experience is included in the set of (state, return) pairs used to make an update at the end of the trace experience. So First-Visit Monte-Carlo needs to keep track of whether a state has already been visited in a trace experience (repeat occurrences of states in a trace experience are ignored).

We won't implement First-Visit Monte-Carlo in this book, and leave it to you as an exercise.

Now let's write some code to test our implementation of Monte-Carlo Prediction. To do so, we go back to a simple finite MRP example from Chapter 1 - `SimpleInventoryMRPFinite`. The following code creates an instance of the MRP and computes it's Value Function with an exact calculation.

```
from rl.chapter2.simple_inventory_mrp import SimpleInventoryMRPFinite

user_capacity = 2
user_poisson_lambda = 1.0
user_holding_cost = 1.0
user_stockout_cost = 10.0
user_gamma = 0.9

si_mrp = SimpleInventoryMRPFinite(
    capacity=user_capacity,
    poisson_lambda=user_poisson_lambda,
    holding_cost=user_holding_cost,
    stockout_cost=user_stockout_cost
)
si_mrp.display_value_function(gamma=user_gamma)
```

This prints the following:

```
{InventoryState(on_hand=0, on_order=0): -35.511,
 InventoryState(on_hand=1, on_order=0): -28.932,
 InventoryState(on_hand=0, on_order=1): -27.932,
 InventoryState(on_hand=0, on_order=2): -28.345,
 InventoryState(on_hand=2, on_order=0): -30.345,
 InventoryState(on_hand=1, on_order=1): -29.345}
```

Next, we run Monte-Carlo Prediction by first generating a stream of trace experiences (in the form of sampling traces) from the MRP, and then calling `evaluate_mrp` using `Tabular` with equal-weights-learning-rate (i.e., default `count_to_weight_func` of `lambda n: 1.0 / n`).

```
from rl.chapter2.simple_inventory_mrp import InventoryState
from rl.function_approx import Tabular, FunctionApprox
from rl.distribution import Choose
from rl.iterate import last
from rl.monte_carlo import evaluate_mrp
from itertools import islice
from pprint import pprint

traces: Iterable[Iterable[TransitionStep[S]]] = \
        mrp.reward_traces(Choose(set(si_mrp.non_terminal_states)))
```

```python
it: Iterator[FunctionApprox[InventoryState]] = evaluate_mrp(
    traces=traces,
    approx_0=Tabular(),
    gamma=user_gamma,
    tolerance=1e-6
)

num_traces = 100000

last_func: FunctionApprox[InventoryState] = last(islice(it, num_traces))
pprint({s: round(last_func.evaluate([s])[0], 3)
        for s in si_mrp.non_terminal_states})
```

This prints the following:

```
{InventoryState(on_hand=0, on_order=0): -35.506,
 InventoryState(on_hand=1, on_order=0): -28.933,
 InventoryState(on_hand=0, on_order=1): -27.931,
 InventoryState(on_hand=0, on_order=2): -28.340,
 InventoryState(on_hand=2, on_order=0): -30.343,
 InventoryState(on_hand=1, on_order=1): -29.343}
```

We see that the Value Function computed by Tabular Monte-Carlo Prediction with 100000 trace experiences is within 0.005 of the exact Value Function.

This completes the coverage of our first RL Prediction algorithm: Monte-Carlo Prediction. This has the advantage of being a very simple, easy-to-understand algorithm with an unbiased estimate of the Value Function. But Monte-Carlo can be slow to converge to the correct Value Function and another disadvantage of Monte-Carlo is that it requires entire trace experiences (or long-enough trace experiences when $\gamma < 1$). The next RL Prediction algorithm we cover (Temporal-Difference) overcomes these weaknesses.

Temporal-Difference (TD) Prediction

To understand Temporal-Difference (TD) Prediction, we start with it's Tabular version as it is simple to understand (and then we can generalize to TD Prediction with Function Approximation). To understand Tabular TD prediction, we begin by taking another look at the Value Function update in Tabular Monte-Carlo (MC) Prediction.

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (G_t - V(S_t))$$

where S_t is the state visited at time step t in the current trace experience, G_t is the trace experience return obtained from time step t onwards, and α denotes the learning rate (based on `count_to_weight_func` attribute in the `Tabular` class). The key in moving from MC to TD is to take advantage of the recursive structure of the Value Function as given by the MRP Bellman Equation (Equation

(1.1)). Although we only have access to instances of next state S_{t+1} and reward R_{t+1}, and not the transition probabilities of next state and reeward, we can approximate G_t as instance reward R_{t+1} plus γ times $V(S_{t+1})$ (where S_{t+1} is the instance of next state). The idea is to re-use (the technical term we use is *bootstrap*) the Value Function that is currently estimated. Clearly, this is a biased estimate of the Value Function meaning the update to the Value Function for S_t will be biased. But the bias disadvantage is outweighed by the reduction in variance (which we will discuss more about later), by speedup in convergence (bootstrapping is our friend here), and by the fact that we don't actually need entire/long-enough trace experiences (again, bootstrapping is our friend here). So, the update for Tabular TD Prediction is:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t))$$

We refer to $R_{t+1} + \gamma \cdot V(S_{t+1})$ as the TD target and we refer to $\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)$ as the TD error. The TD error is the crucial quantity since it represents the "sample Bellman Error" and hence, the TD error can be used to move $V(S_t)$ appropriately (as shown in the above adjustment to $V(S_t)$), which in turn has the effect of bridging the TD error (on an expected basis).

An important practical advantage of TD is that (unlike MC) we can use it in situations where we have incomplete trace experiences (happens often in real-world situations where experiments gets curtailed/disrupted) and also, we can use it in situations where there are no terminal states (*continuing* traces). The other appealing thing about TD is that it is learning (updating Value Function) after each atomic experience (we call it *continuous learning*) versus MC's learning at the end of trace experiences.

Now that we understand how TD Prediction works for the Tabular case, let's consider TD Prediction with Function Approximation. Here, each time we transition from a state S_t to state S_{t+1} with reward R_{t+1}, we make an update to the parameters of the function approximation. To understand how the parameters of the function approximation will update, let's consider the loss function for TD. We start with the single-state loss function for MC (Equation (9.1)) and simply replace G_t with $R_{t+1} + \gamma \cdot V(S_{t+1}, \boldsymbol{w})$ as follows:

$$\mathcal{L}_{(S_t, S_{t+1}, R_{t+1})}(\boldsymbol{w}) = \frac{1}{2} \cdot (V(S_t; \boldsymbol{w}) - (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w})))^2 \tag{9.7}$$

Unlike MC, in the case of TD, we don't take the gradient of this loss function. Instead we "cheat" in the gradient calculation by ignoring the dependency of $V(S_{t+1}; \boldsymbol{w})$ on $\boldsymbol{w}$. This "gradient with cheating" calculation is known as *semi-gradient*. Specifically, we pretend that the only dependency of the loss function on $\boldsymbol{w}$ is through $V(S_t; \boldsymbol{w})$. Hence, the semi-gradient calculation results in the following formula for change in parameters $\boldsymbol{w}$:

$$\Delta \boldsymbol{w} = \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w}) - V(S_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w}) \tag{9.8}$$

This looks similar to the formula for parameters update in the case of MC (with G_t replaced by $R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w})$). Hence, this has the same structure as MC in terms of conceptualizing the change in parameters as the product of the following 3 entities:

- *Learning Rate* α
- *TD Error* $R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w}) - V(S_t; \boldsymbol{w})$
- *Estimate Gradient* of the conditional expected return $V(S_t; \boldsymbol{w})$ with respect to the parameters $\boldsymbol{w}$

Now let's write some code to implement TD Prediction (with Function Approximation). Unlike MC which takes as input a stream of trace experiences, TD works with a more granular stream: a stream of *atomic experiences*. Note that a stream of trace experiences can be broken up into a stream of atomic experiences, but we could also obtain a stream of atomic experiences in other ways (not necessarily from a stream of trace experiences). Thus, the TD prediction algorithm we write below (`evaluate_mrp`) takes as input an `Iterable[TransitionStep[S]]`. `evaluate_mrp` produces an `Iterator` of `FunctionApprox[S]`, i.e., an updated function approximation of the Value Function after each atomic experience in the input atomic experiences stream. Similar to our implementation of MC, our implementation of TD is based on supervised learning on a stream of (x, y) pairs, but there are two key differences:

1. The (x, y) pairs will be provided as one pair at a time (corresponding to a single atomic experience) for a single `update`, and not as a set of (x, y) pairs corresponding to a single trace experience.
2. The y-value depends on the Value Function, as seen from the update Equation (9.8) above. This means we cannot use the `iterate_updates` method of `FunctionApprox` that MC Prediction uses. Rather, we need to directly use the `itertool.accumulate` function. As seen in the code below, the accumulation is performed on the input `transitions: Iterable[TransitionStep[S]]` and the function governing the accumulation is the `step` function in the code below that calls the `update` method of `FunctionApprox` (note that the y-values passed to `update` involve a call to the estimated Value Function `v` for the `next_state` of each `transition`).

```
import rl.markov_process as mp
import itertools

def evaluate_mrp(
        transitions: Iterable[mp.TransitionStep[S]],
        approx_0: FunctionApprox[S],
        gamma: float,
) -> Iterator[FunctionApprox[S]]:

    def step(v, transition):
        return v.update([(transition.state,
                          transition.reward + gamma * v(transition.next_state))])

    return itertools.accumulate(transitions, step, initial=approx_0)
```

The above code is in the file rl/td.py.

Now let's write some code to test our implementation of TD Prediction. We test on the same `SimpleInventoryMRPFinite` that we had tested MC Prediction on. Let us see how close we can get to the true Value Function (that we had calculated above while testing MC Prediction). But first we need to write a function to construct a stream of atomic experiences (`Iterator[TransitionStep[S]]`) from a given `FiniteMarkovRewardProcess` (below code is in the file rl/chapter10/prediction_utils.py). Note the use of `itertools.chain.from.iterable` to chain together a stream of trace experiences (obtained by calling method `reward_traces`) into a stream of atomic experiences in the below function `unit_experiences_from_episodes`.

```
import itertools
from rl.distribution import Distribution, Choose

def mrp_episodes_stream(
    mrp: MarkovRewardProcess[S],
    start_state_distribution: Distribution[S]
) -> Iterable[Iterable[TransitionStep[S]]]:
    return mrp.reward_traces(start_state_distribution)

def fmrp_episodes_stream(
    fmrp: FiniteMarkovRewardProcess[S]
) -> Iterable[Iterable[TransitionStep[S]]]:
    return mrp_episodes_stream(fmrp, Choose(set(fmrp.non_terminal_states)))

def unit_experiences_from_episodes(
    episodes: Iterable[Iterable[TransitionStep[S]]],
    episode_length: int
) -> Iterable[TransitionStep[S]]:
    return itertools.chain.from_iterable(
        itertools.islice(episode, episode_length) for episode in episodes
    )
```

Effective use of Tabular TD Prediction requires us to create an appropriate learning rate schedule by appropriately lowering the learning rate as a function of the number of occurrences of a state in the atomic experiences stream (learning rate schedule specified by `count_to_weight_func` attribute of `Tabular` class). We write below (code in the file rl/function_approx.py) the following learning rate schedule:

$$\alpha_n = \frac{\alpha}{1 + (\frac{n-1}{H})^\beta} \tag{9.9}$$

where α_n is the learning rate to be used at the n-th Value Function update for a given state, α is the initial learning rate (i.e. $\alpha = \alpha_1$), H (we call it "half life") is the number of updates for the learning rate to decrease to half the initial learning rate (if β is 1), and β is the exponent controlling the curvature of the decrease in the learning rate. We shall often set $\beta = 0.5$.

```python
def learning_rate_schedule(
    initial_learning_rate: float,
    half_life: float,
    exponent: float
) -> Callable[[int], float]:
    def lr_func(n: int) -> float:
        return initial_learning_rate * (1 + (n - 1) / half_life) ** -exponent
    return lr_func
```

With these functions available, we can now write code to test our implementation of TD Prediction. We use the same instance `si_mrp: SimpleInventoryMRPFinite` that we had created above when testing MC Prediction. We use the same number of episodes (100000) we had used when testing MC Prediction. We set initial learning rate $\alpha = 0.03$, half life $H = 1000$ and exponent $\beta = 0.5$. We set the episode length (number of atomic experiences in a single trace experience) to be 100 (about the same as with the settings we had for testing MC Prediction). We use the same discount factor $\gamma = 0.9$.

```python
import rl.iterate as iterate
import rl.td as td
import itertools
from pprint import pprint
from rl.chapter10.prediction_utils import fmrp_episodes_stream
from rl.chapter10.prediction_utils import unit_experiences_from_episodes
from rl.function_approx import learning_rate_schedule

episode_length: int = 100
initial_learning_rate: float = 0.03
half_life: float = 1000.0
exponent: float = 0.5
gamma: float = 0.9

episodes: Iterable[Iterable[TransitionStep[S]]] = \
    fmrp_episodes_stream(si_mrp)
td_experiences: Iterable[TransitionStep[S]] = \
    unit_experiences_from_episodes(
        episodes,
        episode_length
    )
learning_rate_func: Callable[[int], float] = learning_rate_schedule(
    initial_learning_rate=initial_learning_rate,
    half_life=half_life,
    exponent=exponent
)
td_vfs: Iterator[FunctionApprox[S]] = td.evaluate_mrp(
    transitions=td_experiences,
```

```
    approx_0=Tabular(count_to_weight_func=learning_rate_func),
    gamma=gamma
)

num_episodes = 100000

final_td_vf: FunctionApprox[S] = \
    iterate.last(itertools.islice(td_vfs, episode_length * num_episodes))
pprint({s: round(final_td_vf(s), 3) for s in si_mrp.non_terminal_states})
```

This prints the following:

```
{InventoryState(on_hand=0, on_order=0): -35.529,
 InventoryState(on_hand=1, on_order=0): -28.888,
 InventoryState(on_hand=0, on_order=1): -27.899,
 InventoryState(on_hand=0, on_order=2): -28.354,
 InventoryState(on_hand=2, on_order=0): -30.363,
 InventoryState(on_hand=1, on_order=1): -29.361}
```

Thus, we see that our implementation of TD prediction with the above settings fetches us an estimated Value Function within 0.05 of the true Value Function. As ever, we encourage you to play with various settings for MC Prediction and TD prediction to develop an intuition for how the results change as you change the settings. You can play with the code in the file rl/chapter10/simple_inventory_mrp.py.

TD versus MC

It is often claimed that TD is the most significant and innovative idea in the development of the field of Reinforcement Learning. The key to TD is that it blends the advantages of Dynamic Programming (DP) and Monte-Carlo (MC). Like DP, TD updates the Value Function estimate by bootstrapping from the Value Function estimate of the next state experienced (essentially, drawing from Bellman Equation). Like MC, TD learns from experiences without requiring access to transition probabilities (MC and TD updates are *experience updates* while DP updates are *transition-probabilities-averaged-updates*). So TD overcomes curse of dimensionality and curse of modeling (computational limitation of DP), and also has the advantage of not requiring entire trace experiences (practical limitation of MC).

TD learning akin to human learning

Perhaps the most attractive thing about TD is that it is akin to how humans learn (versus MC). Let us illustrate this point with how a soccer player learns to improve her game in the process of playing many soccer games. Let's simplify the soccer game to a "golden-goal" soccer game, i.e., the game ends when a team scores a goal. The reward in such a soccer game is +1 for scoring (and winning), 0 if the opponent scores, and also 0 for the entire duration of the game. The soccer player who is learning has a state comprising of her position/velocity/posture etc., the other players' positions/velocity etc., the soccer ball's position/velocity etc. The actions of the soccer player are her physical movements, including the ways to dribble/kick the ball. If the soccer player learns in an MC style (a single episode is a single soccer game), then the soccer player analyzes (at the end of the game) all possible states and actions that occurred during the game and assesses how the actions in each state might have affected the final outcome of the game. You can see how laborious and difficult this actions-reward linkage would be, and you might even argue that it's impossible to disentangle the various actions during the game, leading up to the game's outcome. In any case, you should recognize that this is absolutely not how a soccer player would analyze and learn. Rather, a soccer player analyzes *during the game* - she is continuously evaluating how her actions change the probability of scoring the goal (which is essentially the Value Function). If a pass to her teammate did not result in a goal but greatly increased the chances of scoring a goal, then the action of passing the ball to one's teammate in that state is a good action, boosting the action's Q-value immediately, and she will likely try that action (or a similar action) again, meaning actions with better Q-values are prioritized, which drives towards better and quicker goal-scoring opportunities, and hopefully eventually results in a goal. Such goal-scoring (based on active learning during the game, cutting out poor actions and promoting good actions) would be hailed by commentators as "success from continuous and eager learning" on the part of the soccer player. This is essentially TD learning.

If you think about career decisions and relationship decisions in our lives, MC-style learning is quite infeasible because we simply don't have sufficient "episodes" (for certain decisions, our entire life might be a single episode), and waiting to analyze and adjust until the end of an episode might be far too late in our lives. Rather, we learn and adjust our evaluations of situations constantly in very much a TD-style. Think about various important decisions we make in our lives and you will see that we learn by perpetual adjustment of estimates and efficiency in the use of limited experiences we obtain in our lives.

Bias, Variance and Convergence

Now let's talk about bias and variance of the MC and TD prediction estimates, and their convergence properties.

Say we are at state S_t at time step t on a trace experience, and G_t is the return from that state S_t onwards on this trace experience. G_t is an unbiased estimate of the true value function for state S_t, which is a big advantage for MC when it comes to convergence, even with function approximation of the Value Function. On the other hand, the TD Target $R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w})$ is a biased estimate of the true value function for state S_t. There is considerable literature on formal proofs of TD convergence and we won't cover it in detail here, but here's a qualitative summary. Tabular TD converges to the true value function in the mean for constant learning rate, and converges to the true value function if the following stochastic approximation conditions are satisfied for the learning rate schedule $\alpha_n, n = 1, 2, \ldots$, where the index n refers to the n-th occurrence of a particular state whose Value Function is being updated:

$$\sum_{n=1}^{\infty} \alpha_n = \infty \text{ and } \sum_{n=1}^{\infty} \alpha_n^2 < \infty$$

The stochastic approximation conditions above are known as the Robbins-Monro schedule and apply to a general class of iterative methods used for root-finding or optimization when data is noisy. The intuition here is that the steps should be large enough (first condition) to eventually overcome any unfavorable initial values or noisy data and yet the steps should eventually become small enough (second condition) to ensure convergence. Note that in Equation (9.9), exponent $\beta = 1$ satisfies the Robbins-Monro conditions. In particular, our default choice of `count_to_weight_func=lambda n: 1.0 / n` in `Tabular` satisfies the Robbins-Monro conditions, but our other common choice of constant learning rate does not satisfy the Robbins-Monro conditions. However, we want to emphasize that the Robbins-Monro conditions are typically not that useful in practice because it is not a statement of speed of convergence and it is not a statement on closeness to the true optima (in practice, the goal is typically simply to get fairly close to the true answer reasonably quickly).

The bad news with TD (due to the bias in it's update) is that TD with function approximation does not always converge to the true value function. Most TD convergence proofs are for the Tabular case, however some proofs are for the case of linear function approximation of the Value Function.

The flip side of MC's bias advantage over TD is that the TD Target $R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w})$ has much lower variance that G_t because G_t depends on many random state transitions and random rewards (on the remainder of the trace experience) whose variances accumulate, whereas the TD Target depends on only the next random state transition S_{t+1} and the next random reward R_{t+1}.

As for speed of convergence and efficiency in use of limited set of experiences data, we still don't have formal proofs on whether MC is better or TD. More importantly, because MC and TD have significant differences in their usage of data, nature of updates, and frequency of updates, it is not even clear how to create a level-playing field when comparing MC and TD for speed of convergence or for efficiency in usage of limited experiences data. The typical comparisons between MC and TD are done with constant learning rates, and it's been determined that practically TD learns faster than MC with constant learning rates.

A popular simple problem in the literature (when comparing RL prediction algorithms) is a random walk MRP with states $\{0, 1, 2, \ldots, B\}$ with 0 and B as the terminal states (think of these as terminating barriers of a random walk) and the remaining states as the non-terminal states. From any non-terminal state i, we transition to state $i + 1$ with probability p and to state $i - 1$ with probability $1 - p$. The reward is 0 upon each transition, except if we transition from state $B - 1$ to terminal state B which results in a reward of 1. It's quite obvious that the Value Function is given by: $V(i) = \frac{i}{B}$ for all $0 < i < B$. We'd like to analyze how MC and TD converge, if at all, to this Value Function, starting from a neutral initial Value Function of $V(i) = 0.5$ for all $0 < i < B$. The following code sets up this random walk MRP.

```
from rl.distribution import Categorical

class RandomWalkMRP(FiniteMarkovRewardProcess[int]):
    barrier: int
    p: float

    def __init__(
        self,
        barrier: int,
        p: float
    ):
        self.barrier = barrier
        self.p = p
        super().__init__(self.get_transition_map())

    def get_transition_map(self) -> \
            Mapping[int, Optional[Categorical[Tuple[int, float]]]]:
        d: Dict[int, Optional[Categorical[Tuple[int, float]]]] = {
            i: Categorical({
                (i + 1, 0. if i < self.barrier - 1 else 1.): self.p,
                (i - 1, 0.): 1 - self.p
            }) for i in range(1, self.barrier)
        }
        d[0] = None
        d[self.barrier] = None
        return d
```

The above code is in the file rl/random_walk_mrp.py. Next, we generate a stream of trace experiences from the MRP, use the trace experiences stream to perform MC Prediction, split the trace experiences stream into a stream of atomic experiences so as to perform TD Prediction, run MC and TD Prediction with a variety of learning rate choices, and plot the root-mean-squared-errors (RMSE) of the Value Function averaged across the non-terminal states as a function of batches of episodes (i.e., visualize how the RMSE of the Value Function evolves as the MC/TD algorithm progresses). This is done by calling the function `compare_mc_and_td` which is in the file rl/prediction_utils.py.

Figure 9.1 depicts the convergence for our implementations of MC and TD Prediction for constant learnings rates of $\alpha = 0.01$ (blue curves) and $\alpha = 0.05$ (green curves). We produced this Figure by using data from 700 episodes generated from the random walk MRP with barrier $B = 10$, $p = 0.5$ and discount factor $\gamma = 1$ (a single episode refers to a single trace experience that terminates either at state 0 or state B). We plotted the RMSE after each batch of 7

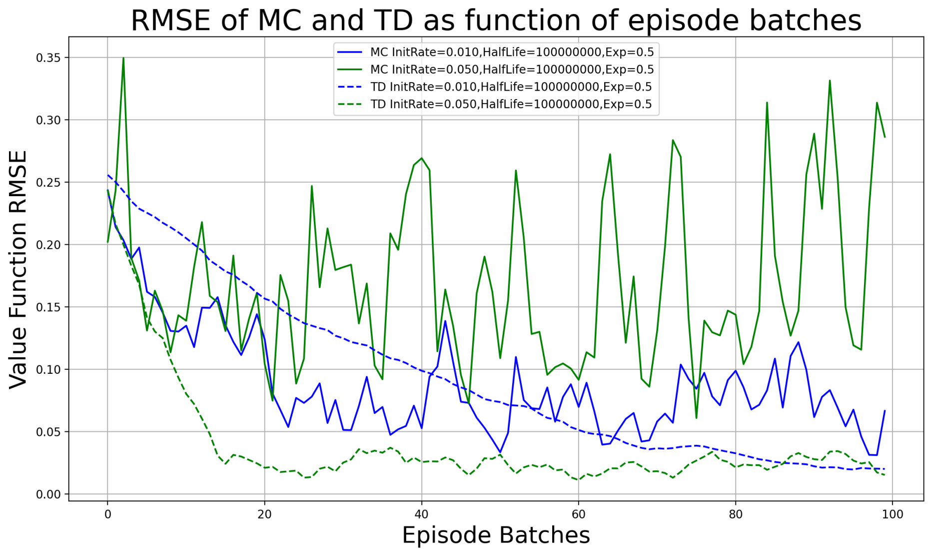

Figure 9.1.: MC and TD Convergence for Random Walk MRP

episodes, hence each of the 4 curves shown in the Figure has 100 RMSE data points plotted. Firstly, we clearly see that MC has significantly more variance as evidenced by the choppy MC RMSE progression curves. Secondly, we note that $\alpha = 0.01$ is a fairly small learning rate and so, the progression of RMSE is quite slow on the blue curves. On the other hand, notice the quick learning for $\alpha = 0.05$ (green curves). MC RMSE is not just choppy, it's evident that it progresses quite quickly in the first few episode batches (relative to the corresponding TD) but the MC RMSE progression is slow after the first few episode batches (relative to the corresponding TD). This results in TD reaching fairly small RMSE quicker than the corresponding MC (this is especially stark for TD with $\alpha = 0.005$, i.e. the dashed green curve in the Figure). This behavior of TD outperforming the comparable MC (with constant learning rate) is typical in most MRP problems.

Lastly, it's important to recognize that MC is not very sensitive to the initial Value Function while TD is more sensitive to the initial Value Function. We encourage you to play with the initial Value Function for this random walk example and evaluate how it affects MC and TD convergence speed.

More generally, we encourage you to play with the `compare_mc_and_td` function on other choices of MRP (ones we have created earlier in this book such as the inventory examples, or make up your own MRPs) so you can develop good intuition for how MC and TD Prediction algorithms converge for a variety of choices of learning rate schedules, initial Value Function choices, choices of discount factor etc.

Fixed-Data Experience Replay on TD versus MC

We have talked a lot about *how* TD learns versus *how* MC learns. In this subsection, we turn our focus to *what* TD learns and *what* MC learns, which is a profound conceptual difference between TD and MC. We illuminate this difference with a special setting - we are given a fixed finite set of trace experiences (versus usual settings considered in this chapter so far where we had an "endless" stream of trace experiences). The agent is allowed to tap into this fixed finite set of traces experiences endlessly, i.e., the MC or TD prediction RL agent can indeed consume an endless stream of data, but all of that stream of data must ultimately be sourced from the given fixed finite set of trace experiences. This means we'd end up tapping into trace experiences (or it's component atomic experiences) repeatedly. We call this technique of repeatedly tapping into the same data as *Experience Replay*. We will uncover the key conceptual difference of *what* MC and TD learn by running the algorithms on an *Experience Replay* of a fixed finite set of trace experiences.

So let us start by setting up this experience replay with some code. Firstly, we represent the given input data of the fixed finite set of trace experience as the type:

`Sequence[Sequence[Tuple[str, float]]]`

The outer `Sequence` refers to the sequence of trace experiences, and the inner `Sequence` refers to the sequence of (state, reward) pairs in a trace experience (to represent the alternating sequence of states and rewards in a trace experience). The first function we write is to convert this data set into a:

`Sequence[Sequence[TransitionStep[S]]]`

which is consumable by MC and TD Prediction algorithms (since their interfaces work with the `TransitionStep[S]` data type). The following function does this job:

```
def get_fixed_episodes_from_sr_pairs_seq(
    sr_pairs_seq: Sequence[Sequence[Tuple[S, float]]],
    terminal_state: S
) -> Sequence[Sequence[TransitionStep[S]]]:
    return [[TransitionStep(
        state=s,
        reward=r,
        next_state=trace[i+1][0] if i < len(trace) - 1 else terminal_state
    ) for i, (s, r) in enumerate(trace)] for trace in sr_pairs_seq]
```

We'd like MC Prediction to run on an endless stream of `Sequence[TransitionStep[S]` sourced from the fixed finite data set produced by `get_fixed_episodes_from_sr_pairs_seq`. So we write the following function to generate an endless stream by repeatedly randomly (uniformly) sampling from the fixed finite set of trace experiences, as follows:

```
import numpy as np
```

```
def get_episodes_stream(
    fixed_episodes: Sequence[Sequence[TransitionStep[S]]]
) -> Iterator[Sequence[TransitionStep[S]]]:
    num_episodes: int = len(fixed_episodes)
    while True:
        yield fixed_episodes[np.random.randint(num_episodes)]
```

As we know, TD works with atomic experiences rather than trace experiences. So we need the following function to split the fixed finite set of trace experiences into a fixed finite set of atomic experiences:

```
import itertools

def fixed_experiences_from_fixed_episodes(
    fixed_episodes: Sequence[Sequence[TransitionStep[S]]]
) -> Sequence[TransitionStep[S]]:
    return list(itertools.chain.from_iterable(fixed_episodes))
```

We'd like TD Prediction to run on an endless stream of `TransitionStep[S]` sourced from the fixed finite set of atomic experiences produced by `fixed_experiences_from_fixed_` So we write the following function to generate an endless stream by repeatedly randomly (uniformly) sampling from the fixed finite set of atomic experiences, as follows:

```
def get_experiences_stream(
    fixed_experiences: Sequence[TransitionStep[S]]
) -> Iterator[TransitionStep[S]]:
    num_experiences: int = len(fixed_experiences)
    while True:
        yield fixed_experiences[np.random.randint(num_experiences)]
```

Ok - now we are ready to run MC and TD Prediction algorithms on an experience replay of the given input of a fixed finite set of trace experiences. It is quite obvious what MC Prediction algorithm would learn. MC Prediction is simply supervised learning of a data set of states and their associated returns, and here we have a fixed finite set of states (across the trace experiences) and the corresponding trace experience returns associated with each of those states. Hence, MC Prediction should return a Value Function that is equal to the average returns seen in the fixed finite data set for each of the states in the data set. So let us first write a function to explicitly calculate the average returns, and then we can confirm that MC Prediction will give the same answer.

```
from rl.returns import returns
from rl.markov_process import ReturnStep

def get_return_steps_from_fixed_episodes(
    fixed_episodes: Sequence[Sequence[TransitionStep[S]]],
```

```
    gamma: float
) -> Sequence[ReturnStep[S]]:
    return list(itertools.chain.from_iterable(returns(episode, gamma, 1e-8)
                                              for episode in fixed_episodes))


def get_mean_returns_from_return_steps(
    returns_seq: Sequence[ReturnStep[S]]
) -> Mapping[S, float]:
    def by_state(ret: ReturnStep[S]) -> S:
        return ret.state

    sorted_returns_seq: Sequence[ReturnStep[S]] = sorted(
        returns_seq,
        key=by_state
    )
    return {s: np.mean([r.return_ for r in l])
            for s, l in itertools.groupby(
                sorted_returns_seq,
                key=by_state
            )}
```

To facilitate comparisons, we will do all calculations on the following simple hand-entered input data set:

```
given_data: Sequence[Sequence[Tuple[str, float]]] = [
    [('A', 2.), ('A', 6.), ('B', 1.), ('B', 2.)],
    [('A', 3.), ('B', 2.), ('A', 4.), ('B', 2.), ('B', 0.)],
    [('B', 3.), ('B', 6.), ('A', 1.), ('B', 1.)],
    [('A', 0.), ('B', 2.), ('A', 4.), ('B', 4.), ('B', 2.), ('B', 3.)],
    [('B', 8.), ('B', 2.)]
]
```

The following code runs get_mean_returns_from_return_steps on this simple input data set.

```
from pprint import pprint
gamma: float = 0.9

fixed_episodes: Sequence[Sequence[TransitionStep[str]]] = \
    get_fixed_episodes_from_sr_pairs_seq(
        sr_pairs_seq=given_data,
        terminal_state='T'
    )

returns_seq: Sequence[ReturnStep[str]] = \
    get_return_steps_from_fixed_episodes(
        fixed_episodes=fixed_episodes,
```

```python
        gamma=gamma
    )

mean_returns: Mapping[str, float] = get_mean_returns_from_return_steps(
    returns_seq
)

pprint(mean_returns)
```

This prints:

```
{'A': 8.261809999999999, 'B': 5.190378571428572}
```

Now let's run MC Prediction with experience-replayed 100,000 trace experiences with equal weighting for each of the (state, return) pairs, i.e., with `count_to_weights_func` attribute of `Tabular` as the function `lambda n: 1.0 / n`:

```python
import rl.monte_carlo as mc

def mc_prediction(
    episodes_stream: Iterator[Sequence[TransitionStep[S]]],
    gamma: float,
    num_episodes: int
) -> Mapping[S, float]:
    return iterate.last(itertools.islice(
        mc.evaluate_mrp(
            traces=episodes_stream,
            approx_0=Tabular(),
            gamma=gamma,
            tolerance=1e-10
        ),
        num_episodes
    )).values_map

num_mc_episodes: int = 100000

episodes: Iterator[Sequence[TransitionStep[str]]] = \
    get_episodes_stream(fixed_episodes)

mc_pred: Mapping[str, float] = mc_prediction(
    episodes_stream=episodes,
    gamma=gamma,
    num_episodes=num_mc_episodes
)

pprint(mc_pred)
```

This prints:

```
{'A': 8.259354513588503, 'B': 5.18847638381789}
```

So, as expected, it ties out within the standard error for 100,000 trace experiences. Now let's move on to TD Prediction. Let's run TD Prediction on experience-replayed 1,000,000 atomic experiences with a learning rate schedule having an initial learning rate of 0.01, decaying with a half life of 10000, and with an exponent of 0.5.

```
import rl.td as td
from rl.function_approx import learning_rate_schedule, Tabular

def td_prediction(
    experiences_stream: Iterator[TransitionStep[S]],
    gamma: float,
    num_experiences: int
) -> Mapping[S, float]:
    return iterate.last(itertools.islice(
        td.evaluate_mrp(
            transitions=experiences_stream,
            approx_0=Tabular(count_to_weight_func=learning_rate_schedule(
                initial_learning_rate=0.01,
                half_life=10000,
                exponent=0.5
            )),
            gamma=gamma
        ),
        num_experiences
    )).values_map

num_td_experiences: int = 1000000

fixed_experiences: Sequence[TransitionStep[str]] = \
    fixed_experiences_from_fixed_episodes(fixed_episodes)

experiences: Iterator[TransitionStep[str]] = \
    get_experiences_stream(fixed_experiences)


td_pred: Mapping[str, float] = td_prediction(
    experiences_stream=experiences,
    gamma=gamma,
    num_experiences=num_td_experiences
)

pprint(td_pred)
```

This prints:

```
{'A': 9.733383341548377, 'B': 7.483985631705235}
```

We note that this Value Function is vastly different from the Value Function produced by MC Prediction. Is there a bug in our code, or perhaps a more serious conceptual problem? It turns out there is no bug or a more serious problem. This is exactly what TD Prediction on Experience Replay on a fixed finite data set is meant to produce. So, what Value Function does this correspond to? It turns out that TD Prediction drives towards a Value Function of an MRP that is *implied* by the fixed finite set of given experiences. By the term *implied*, we mean the maximum likelihood estimate for the transition probabilities $\mathcal{P}_R$ estimated from the given fixed finite data, i.e.,

$$\mathcal{P}_R(s, r, s') = \frac{\sum_{i=1}^{N} \mathbb{I}_{S_i=s, R_{i+1}=r, S_{i+1}=s'}}{\sum_{i=1}^{N} \mathbb{I}_{S_i=s}} \tag{9.10}$$

where the fixed finite set of transitions are $[(S_i, R_{i+1} = r, S_{i+1} = s')|1 \leq i \leq N]$, and $\mathbb{I}$ denotes the indicator function.

So let's write some code to construct this MRP based on the above formula.

```
from rl.distribution import Categorical
from rl.markov_process import FiniteMarkovRewardProcess

def finite_mrp(
    fixed_experiences: Sequence[TransitionStep[S]]
) -> FiniteMarkovRewardProcess[S]:
    def by_state(tr: TransitionStep[S]) -> S:
        return tr.state

    terminal_state: S = fixed_experiences[-1].next_state

    d: Mapping[S, Sequence[Tuple[S, float]]] = \
        {s: [(t.next_state, t.reward) for t in l] for s, l in
         itertools.groupby(
             sorted(fixed_experiences, key=by_state),
             key=by_state
         )}
    mrp: Dict[S, Optional[Categorical[Tuple[S, float]]]] = \
        {s: Categorical({x: y / len(l) for x, y in
                         collections.Counter(l).items()})
         for s, l in d.items()}
    mrp[terminal_state] = None
    return FiniteMarkovRewardProcess(mrp)
```

Now let's print it's Value Function.

```
fmrp: FiniteMarkovRewardProcess[str] = finite_mrp(fixed_experiences)
fmrp.display_value_function(gamma)
```

This prints:

```
{'A': 9.958, 'B': 7.545}
```

This Value Function is quite close to the Value Function produced by TD Prediction. The TD Prediction algorithm doesn't exactly match the Value Function of the data-implied MRP, but is quite close. It turns out that a variation of our TD Prediction algorithm produces the Value Function of the data-implied MRP. We won't implement this variation in this chapter, but will describe it briefly here. The variation is as follows:

- The Value Function is not updated after each atomic experience, rather the Value Function is updated at the end of each *batch of atomic experiences*.
- Each batch of atomic experiences consists of a single occurrence of each atomic experience in the given fixed finite data set.
- The updates to the Value Function to be performed at the end of each batch are accumulated in a buffer after each atomic experience and the buffer's contents are used to update the Value Function only at the end of the batch. Specifically, this means that the right-hand-side of Equation (9.8) is calculated at the end of each atomic experience and these calculated values are accumulated in the buffer until the end of the batch, at which point the buffer's contents are used to update the Value Function.

This variant of the TD Prediction algorithm is known as *Batch Updating* and more broadly, RL algorithms that update the Value Function at the end of a batch are refered to as *Batch Methods*. This contrasts with *Incremental Methods*, which are RL algorithms that update the Value Functions after each atomic experience (in the case of TD) or at the end of each trace experience (in the case of MC). The MC and TD Prediction algorithms we implemented earlier in this chapter are Incremental Methods. We will cover Batch Methods in detail in Chapter 11.

Although our TD Prediction algorithm is an Incremental Method, it did get fairly close to the Value Function of the data-implied MRP (and the Value Function that would be obtained by a TD Prediction algorithm with Batch Method). So let us ignore the nuance that our TD Prediction algorithm didn't exactly match the Value Function of the data-implied MDP and instead focus on the fact that our MC Prediction algorithm and TD Prediction algorithm drove towards two very different Value Functions. The MC Prediction algorithm learns a "fairly naive" Value Function - one that is based on the mean of the observed returns (for each state) in the given fixed finite data. The TD Prediction algorithm is learning something "deeper" - it is (implicitly) constructing an MRP based on the given fixed finite data (Equation (9.10)), and then (implicitly) calculating the Value Function of the constructed MRP. The mechanics of the TD Prediction algorithms don't actually construct the MRP and calculate the Value

Function of the MRP - rather, the TD Prediction algorithm directly drives towards the Value Function of the data-implied MRP. However, the fact that it gets to this Value Function means it is (implictly) trying to infer a transitions structure from the given data, and hence, we say that it is learning something "deeper" than what MC is learning. This has practical implications. Firstly, this learning facet of TD means that it exploits any Markov property in the environment and so, TD algorithms are more efficient (learn faster than MC) in Markov environments. On the other hand, the naive nature of MC (not exploiting any Markov property in the environment) is advantageous (more effective than TD) in non-Markov environments.

We encourage you to play with the above code (in the file rl/chapter10/mc_td_experience_replay.p by trying Experience Replay on larger input data sets. We also encourage you to code up Batch Method variants of MC and TD Prediction algorithms.

Bootstrapping and Experiencing

We summarize MC, TD and DP in terms of whether they bootstrap (or not) and in terms of whether they use experiences/samples (or use a model of transitions).

- Bootstrapping: By "bootstrapping"", we mean that an update to the Value Function utilizes a current or prior estimate of the Value Function. MC *does not bootstrap* since it's Value Function updates use actual trace experience returns and not any current or prior estimates of the Value Function. On the other hand, TD and DP *do bootstrap*.
- Experiencing: By "experiencing", we mean that the algorithm uses experiences or samples rather than performing expectation calculations with a model of transition probabilities. MD and TD *do experience*, while DP *does not experience*.

We illustrate this perspective of bootstrapping (or not) and experiencing (or not) with some very popular diagrams that we are borrowing from teaching content prepared by Richard Sutton Richard Sutton, who has the written the most influential book on Reinforcement Learning (along with Andrew Barto.

The first diagram is Figure 9.2, known as the MC *backup* diagram for an MDP (although we are covering Prediction in this chapter, these concepts also apply to MDP Control). The root of the tree is the state whose Value Function we want to update. The remaining nodes of the tree are the future states/actions that might be visited. The branching on the tree is due to the probabilistic transitions of the MDP and the choices of actions that might be taken. The green nodes (marked as "T") are the terminal states. The red-colored path on the tree from the root node (current state) to a terminal state indicates the trace experience used by MC. The red-colored path is the set of future states/actions used in updating the Value Function of the current state (root node). We say that the Value Function is "backed up" along this red-colored path (to mean that the Value Function propagates from the bottom of the red-colored path to the

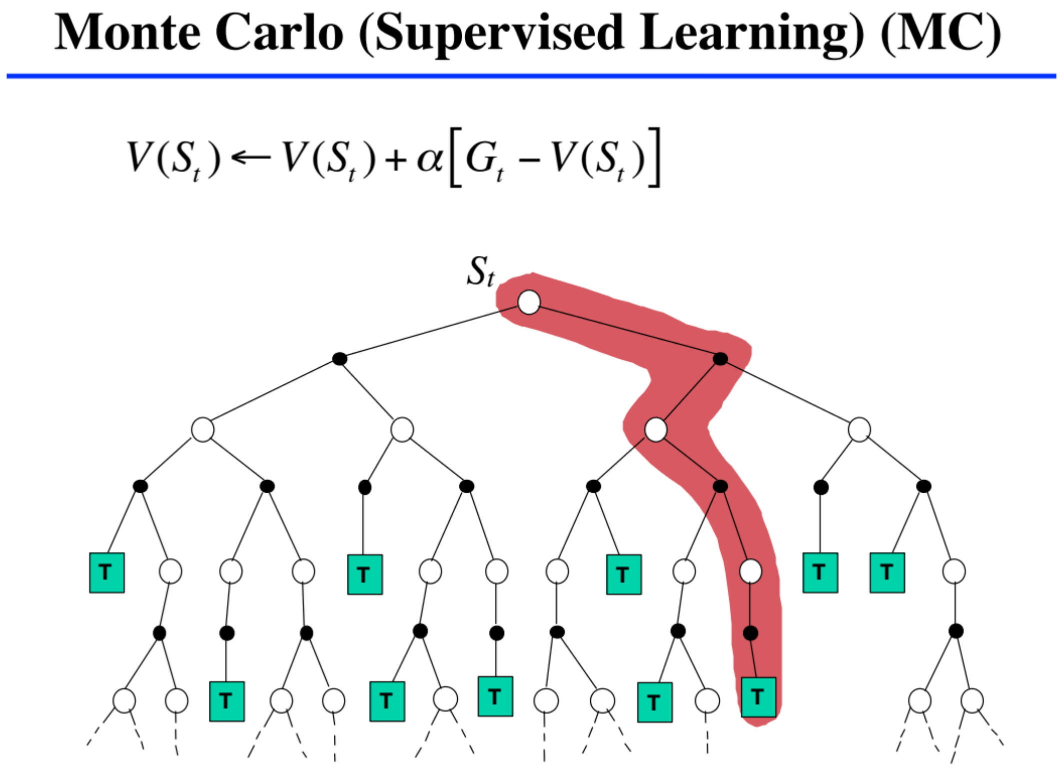

Figure 9.2.: MC Backup Diagram

top, since the returns are calculated accumulated rewards from the bottom to the top, i.e., from the end of a trace experience to the beginning of the trace experience). This is why we refer to such diagrams as *backup* diagrams. Since MC "experiences", it only considers a single child node from any node (rather than all the child nodes, which would be the case if we considered all probabilistic transitions or considered all action choices). Since MC does not "bootstrap", it doesn't just use the Value Function estimate from it's child/grandchild node (next state/action) - it considers all future states/actions along the entire trace experience. Hence, the backup (colored red) goes deep into the tree and is narrow (doesn't go wide across the tree).

The next diagram is Figure 9.3, known as the TD *backup* diagram for an MDP. Again, the red-coloring applies to the future states/actions used in updating the Value Function of the current state (root node). The Value Function is "backed up" along this red-colored section of the tree. Since TD "experiences", it only considers a single child node from any node (rather than all the child nodes, which would be the case if we considered all probabilistic transitions or considered all actions choices). Since TD "bootstraps", it uses the Value Function estimate from it's child/grandchild node (next state/action) and doesn't utilize any information from nodes beyond the child/grandchild node. Hence, the backup (colored red) is shallow (doesn't go deep into the tree) and is narrow (doesn't go wide across the tree).

The next diagram is Figure 9.4, known as the DP *backup* diagram for an MDP.

Simplest TD Method

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

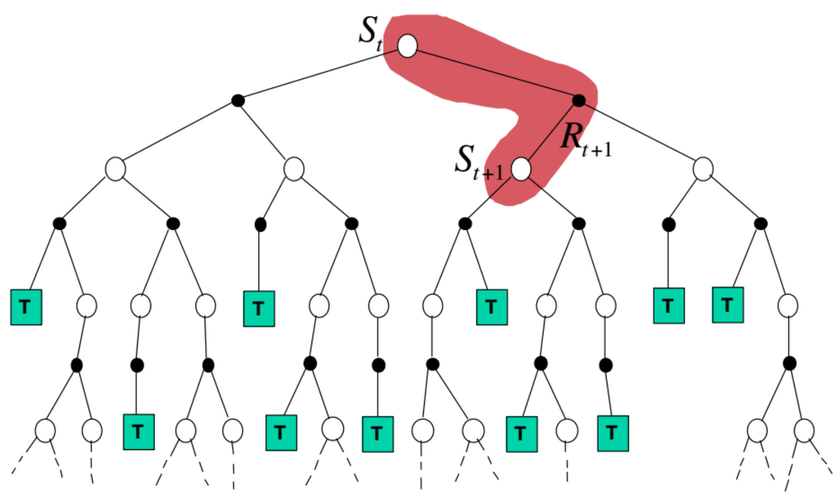

Figure 9.3.: TD Backup Diagram

cf. Dynamic Programming

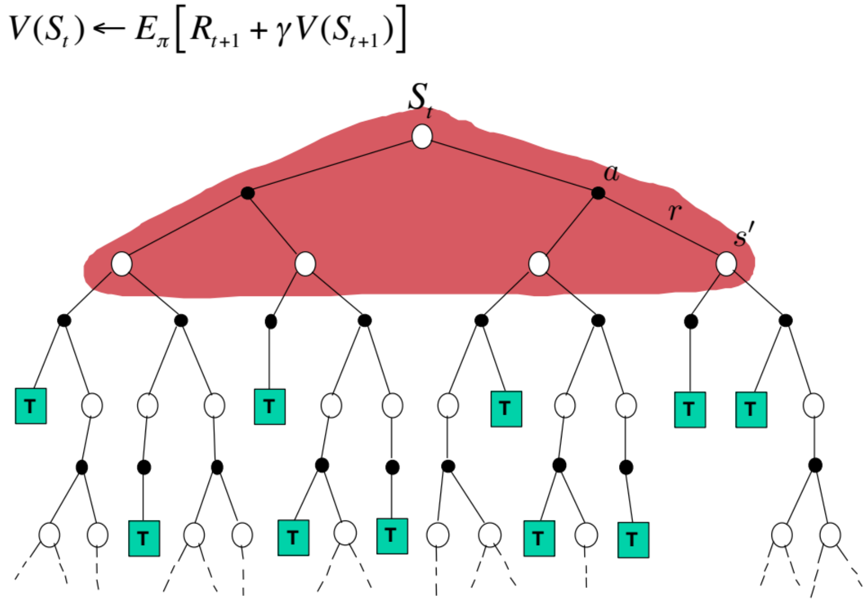

Figure 9.4.: DP Backup Diagram

Again, the red-coloring applies to the future states/actions used in updating the Value Function of the current state (root node). The Value Function is "backed up" along this red-colored section of the tree. Since DP does not "experience" and utilizes the knowledge of probabilities of all next states and considers all choices of actions (in the case of Control), it considers all child nodes (all choices of actions) and all grandchild nodes ((all probabilistic transitions to next states) from the root node (current state). Since DP "bootstraps", it uses the Value Function estimate from it's children/grandchildren nodes (next states/actions) and doesn't utilize any information from nodes beyond the children/grandchildren nodes. Hence, the backup (colored red) is shallow (doesn't go deep into the tree) and goes wide across the tree.

This perspective of shallow versus deep (for "bootstrapping" or not) and of narrow versus wide (for "experiencing" or not) is a great way to visualize and recollect MC, TD and DP and it helps us compare and contrast these methods in a simple and intuitive manner. We thank Rich Sutton for this excellent pedagogical contribution. This brings us to the next image (Figure 9.5) which provides a unified view of RL in a single picture. The top of this Figure shows methods that "bootstrap" (including TD and DP) and the bottom of this Figure shows methods that do not "bootstrap" (including MC and methods known as "Exhaustive Search" that go both deep into the tree and wide across the tree - we shall cover some of these methods in a later chapter). Therefore the vertical di-

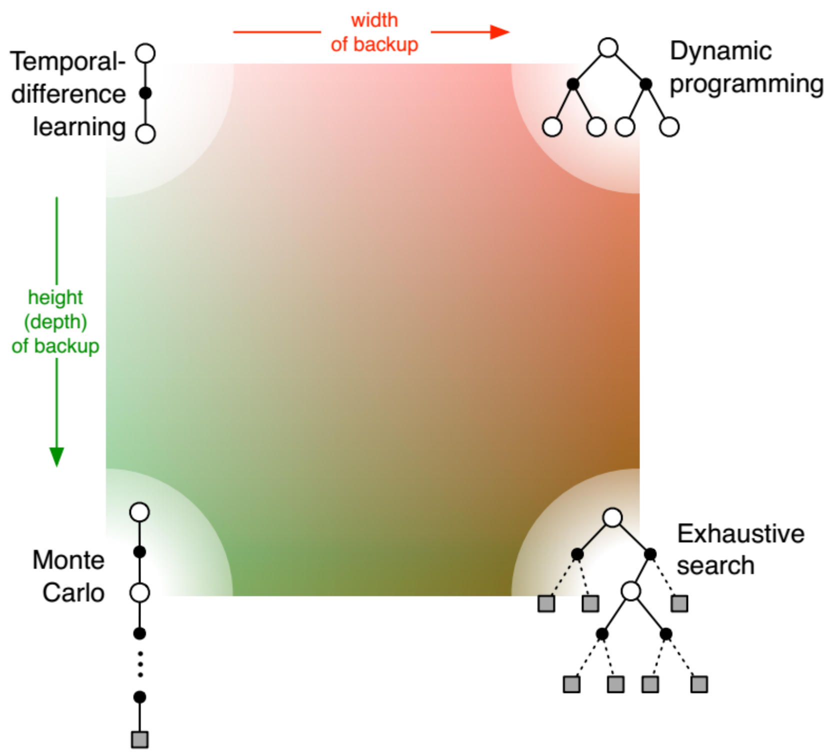

Figure 9.5.: Unified View of RL

mension of this Figure refers to the depth of the backup. The left of this Figure shows methods that "experience" (including TD and MC) and the right of this Figure shows methods than do not "experience" (including DP and "Exhaustive Search"). Therefore, the horizontal dimension of this Figure refers to the width of the backup.

TD(λ)

- Write MC Error as sum of TD Errors
- Write TD(λ) Prediction algorithm

10. Monte-Carlo (MC) and Temporal-Difference (TD) for Control

11. Experience Replay, Least-Squares Policy Iteration, and Gradient TD

12. Policy Gradient Algorithms

13. Learning versus Planning

14. Multi-Armed Bandits: Exploration versus Exploitation

15. RL in Real-World Finance: Reality versus Hype, Present versus Future

Part IV.

Appendix

A. Moment Generating Function and its Applications

The purpose of this Appendix is to introduce the *Moment Generating Function* (*MGF*) and demonstrate it's utility in several applications in Applied Mathematics.

The Moment Generating Function (MGF)

The Moment Generating Function (MGF) of a random variable x (discrete or continuous) is defined as a function $f_x : \mathbb{R} \rightarrow \mathbb{R}^+$ such that:

$$f_x(t) = \mathbb{E}_x[e^{tx}] \text{ for all } t \in \mathbb{R} \tag{A.1}$$

Let us denote the n^{th}-derivative of f_x as $f_x^{(n)} : \mathbb{R} \rightarrow \mathbb{R}$ for all $n \in \mathbb{Z}_{\geq 0}$ ($f_x^{(0)}$ is defined to be simply the MGF f_x).

$$f_x^{(n)}(t) = \mathbb{E}_x[x^n \cdot e^{tx}] \text{ for all } n \in \mathbb{Z}_{\geq 0} \text{ for all } t \in \mathbb{R} \tag{A.2}$$

$$f_x^{(n)}(0) = \mathbb{E}_x[x^n] \tag{A.3}$$

$$f_x^{(n)}(1) = \mathbb{E}_x[x^n \cdot e^x] \tag{A.4}$$

Equation (A.3) tells us that $f^{(n)}(0)$ gives us the n^{th} moment of x. In particular, $f_x^{(1)}(0) = f'(0)$ gives us the mean and $f_x^{(2)}(0) - (f_x^{(1)}(0))^2 = f_x''(0) - (f_x'(0))^2$ gives us the variance. Note that this holds true for any distribution for x. This is rather convenient since all we need is the functional form for the distribution of x. This would lead us to the expression for the MGF (in terms of t). Then, we take derivatives of this MGF and evaluate those derivatives at 0 to obtain the moments of x.

Equation (A.4) helps us calculate the often-appearing expectation $\mathbb{E}_x[x^n \cdot e^x]$. In fact, $\mathbb{E}_x[e^x]$ and $\mathbb{E}_x[x \cdot e^x]$ are very common in several areas of Applied Mathematics. Again, note that this holds true for any distribution for x.

MGF should be thought of as an alternative specification of a random variable (alternative to specifying it's Probability Distribution). This alternative specification is very valuable because it can sometimes provide better analytical tractability than working with the Probability Density Function or Cumulative Distribution Function (as an example, see the below section on MGF for linear functions of independent random variables).

MGF for Linear Functions of Random Variables

Consider m independent random variables $x_1, x_2, \ldots, x_m$. Let $\alpha_0, \alpha_1, \ldots, \alpha_m \in \mathbb{R}$. Now consider the random variable

$$x = x_0 + \sum_{i=1}^{m} \alpha_i x_i$$

The Probability Density Function of x is complicated to calculate as it involves convolutions. However, observe that the MGF f_x of x is given by:

$$f_x(t) = \mathbb{E}[e^{t(\alpha_0 + \sum_{i=1}^{m} \alpha_i x_i)}] = e^{\alpha_0 t} \cdot \prod_{i=1}^{m} \mathbb{E}[e^{t\alpha_i x_i}] = e^{\alpha_0 t} \cdot \prod_{i=1}^{m} f_{\alpha_i x_i}(t) = e^{\alpha_0 t} \cdot \prod_{i=1}^{m} f_{x_i}(\alpha_i t)$$

This means the MGF of x can be calculated as $e^{\alpha_0 t}$ times the product of the MGFs of $\alpha_i x_i$ (or of α_i-scaled MGFs of x_i) for all $i = 1, 2, \ldots, m$. This gives us a much better way to analytically tract the probability distribution of x (compared to the convolution approach).

MGF for the Normal Distribution

Here we assume that the random variables x follows a normal distribution. Let $x \sim \mathcal{N}(\mu, \sigma^2)$.

$$\begin{aligned}
f_{x \sim \mathcal{N}(\mu,\sigma^2)}(t) &= \mathbb{E}_{x \sim \mathcal{N}(\mu,\sigma^2)}[e^{tx}] \\
&= \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}} \cdot e^{tx} \cdot dx \\
&= \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\frac{(x-(\mu+t\sigma^2))^2}{2\sigma^2}} \cdot e^{\mu t + \frac{\sigma^2 t^2}{2}} \cdot dx \\
&= e^{\mu t + \frac{\sigma^2 t^2}{2}} \cdot \mathbb{E}_{x \sim \mathcal{N}(\mu+t\sigma^2,\sigma^2)}[1] \\
&= e^{\mu t + \frac{\sigma^2 t^2}{2}}
\end{aligned} \tag{A.5}$$

$$f'_{x \sim \mathcal{N}(\mu,\sigma^2)}(t) = \mathbb{E}_{x \sim \mathcal{N}(\mu,\sigma^2)}[x \cdot e^{tx}] = (\mu + \sigma^2 t) \cdot e^{\mu t + \frac{\sigma^2 t^2}{2}} \tag{A.6}$$

$$f''_{x \sim \mathcal{N}(\mu,\sigma^2)}(t) = \mathbb{E}_{x \sim \mathcal{N}(\mu,\sigma^2)}[x^2 \cdot e^{tx}] = ((\mu + \sigma^2 t)^2 + \sigma^2) \cdot e^{\mu t + \frac{\sigma^2 t^2}{2}} \tag{A.7}$$

$$f'_{x \sim \mathcal{N}(\mu,\sigma^2)}(0) = \mathbb{E}_{x \sim \mathcal{N}(\mu,\sigma^2)}[x] = \mu$$

$$f''_{x \sim \mathcal{N}(\mu,\sigma^2)}(0) = \mathbb{E}_{x \sim \mathcal{N}(\mu,\sigma^2)}[x^2] = \mu^2 + \sigma^2$$

$$f'_{x \sim \mathcal{N}(\mu,\sigma^2)}(1) = \mathbb{E}_{x \sim \mathcal{N}(\mu,\sigma^2)}[x \cdot e^x] = (\mu + \sigma^2) e^{\mu + \frac{\sigma^2}{2}}$$

$$f''_{x \sim \mathcal{N}(\mu,\sigma^2)}(1) = \mathbb{E}_{x \sim \mathcal{N}(\mu,\sigma^2)}[x^2 \cdot e^x] = ((\mu + \sigma^2)^2 + \sigma^2) e^{\mu + \frac{\sigma^2}{2}}$$

Minimizing the MGF

Now let us consider the problem of minimizing the MGF. The problem is to:

$$\min_{t \in \mathbb{R}} f_x(t) = \min_{t \in \mathbb{R}} \mathbb{E}_x[e^{tx}]$$

This problem of minimizing $\mathbb{E}_x[e^{tx}]$ shows up a lot in various places in Applied Mathematics when dealing with exponential functions (eg: when optimizing the Expectation of a Constant Absolute Risk-Aversion (CARA) Utility function $U(y) = \frac{1-e^{-\gamma y}}{\gamma}$ where γ is the coefficient of risk-aversion and where y is a parameterized function of a random variable x).

Let us denote t^* as the value of t that minimizes the MGF. Specifically,

$$t^* = \arg\min_{t \in \mathbb{R}} f_x(t) = \arg\min_{t \in \mathbb{R}} \mathbb{E}_x[e^{tx}]$$

Minimizing the MGF when x follows a normal distribution

Here we consider the fairly typical case where x follows a normal distribution. Let $x \sim \mathcal{N}(\mu, \sigma^2)$. Then we have to solve the problem:

$$\min_{t \in \mathbb{R}} f_{x \sim \mathcal{N}(\mu,\sigma^2)}(t) = \min_{t \in \mathbb{R}} \mathbb{E}_{x \sim \mathcal{N}(\mu,\sigma^2)}[e^{tx}] = \min_{t \in \mathbb{R}} e^{\mu t + \frac{\sigma^2 t^2}{2}}$$

From Equation (A.6) above, we have:

$$f'_{x \sim \mathcal{N}(\mu,\sigma^2)}(t) = (\mu + \sigma^2 t) \cdot e^{\mu t + \frac{\sigma^2 t^2}{2}}$$

Setting this to 0 yields:

$$(\mu + \sigma^2 t^*) \cdot e^{\mu t^* + \frac{\sigma^2 t^{*2}}{2}} = 0$$

which leads to:

$$t^* = \frac{-\mu}{\sigma^2} \tag{A.8}$$

From Equation (A.7) above, we have:

$$f''_{x \sim \mathcal{N}(\mu,\sigma^2)}(t) = ((\mu + \sigma^2 t)^2 + \sigma^2) \cdot e^{\mu t + \frac{\sigma^2 t^2}{2}} > 0 \text{ for all } t \in \mathbb{R}$$

which confirms that t^* is a minima.

Substituting $t = t^*$ in $f_{x \sim \mathcal{N}(\mu,\sigma^2)}(t) = e^{\mu t + \frac{\sigma^2 t^2}{2}}$ yields:

$$\min_{t \in \mathbb{R}} f_{x \sim \mathcal{N}(\mu,\sigma^2)}(t) = e^{\mu t^* + \frac{\sigma^2 t^{*2}}{2}} = e^{\frac{-\mu^2}{2\sigma^2}} \tag{A.9}$$

Minimizing the MGF when x is a symmetric binary distribution

Here we consider the case where x follows a binary distribution: x takes values $\mu + \sigma$ and $\mu - \sigma$ with probability 0.5 each. Let us refer to this distribution as $x \sim \mathcal{B}(\mu+\sigma, \mu-\sigma)$. Note that the mean and variance of x under $\mathcal{B}(\mu+\sigma, \mu-\sigma)$ are μ and σ^2 respectively. So we have to solve the problem:

$$\min_{t \in \mathbb{R}} f_{x \sim \mathcal{B}(\mu+\sigma,\mu-\sigma)}(t) = \min_{t \in \mathbb{R}} \mathbb{E}_{x \sim \mathcal{B}(\mu+\sigma,\mu-\sigma)}[e^{tx}] = \min_{t \in \mathbb{R}} 0.5(e^{(\mu+\sigma)t} + e^{(\mu-\sigma)t})$$

$$f'_{x \sim \mathcal{B}(\mu+\sigma,\mu-\sigma)}(t) = 0.5((\mu+\sigma) \cdot e^{(\mu+\sigma)t} + (\mu-\sigma) \cdot e^{(\mu-\sigma)t})$$

Note that unless $\mu \in$ open interval $(-\sigma, \sigma)$ (i.e., absolute value of mean is less than standard deviation), $f'_{x \sim \mathcal{B}(\mu+\sigma,\mu-\sigma)}(t)$ will not be 0 for any value of t. Therefore, for this minimization to be non-trivial, we will henceforth assume $\mu \in (-\sigma, \sigma)$. With this assumption in place, setting $f'_{x \sim \mathcal{B}(\mu+\sigma,\mu-\sigma)}(t)$ to 0 yields:

$$(\mu+\sigma) \cdot e^{(\mu+\sigma)t^*} + (\mu-\sigma) \cdot e^{(\mu-\sigma)t^*} = 0$$

which leads to:

$$t^* = \frac{1}{2\sigma} \ln\left(\frac{\sigma-\mu}{\mu+\sigma}\right)$$

Note that

$$f''_{x \sim \mathcal{B}(\mu+\sigma,\mu-\sigma)}(t) = 0.5((\mu+\sigma)^2 \cdot e^{(\mu+\sigma)t} + (\mu-\sigma)^2 \cdot e^{(\mu-\sigma)t}) > 0 \text{ for all } t \in \mathbb{R}$$

which confirms that t^* is a minima.

Substituting $t = t^*$ in $f_{x \sim \mathcal{B}(\mu+\sigma,\mu-\sigma)}(t) = 0.5(e^{(\mu+\sigma)t} + e^{(\mu-\sigma)t})$ yields:

$$\min_{t \in \mathbb{R}} f_{x \sim \mathcal{B}(\mu+\sigma,\mu-\sigma)}(t) = 0.5(e^{(\mu+\sigma)t^*} + e^{(\mu-\sigma)t^*}) = 0.5\left(\left(\frac{\sigma-\mu}{\mu+\sigma}\right)^{\frac{\mu+\sigma}{2\sigma}} + \left(\frac{\sigma-\mu}{\mu+\sigma}\right)^{\frac{\mu-\sigma}{2\sigma}}\right)$$

B. Function Approximations as Vector Spaces

Definition of a Vector Space

A Vector space is defined as a commutative group $\mathcal{V}$ under an addition operation (written as $+$), together with multiplication of elements of $\mathcal{V}$ with elements of a field $\mathcal{K}$ (known as scalars), expressed as a binary in-fix operation $\cdot : \mathcal{K} \times \mathcal{V} \rightarrow \mathcal{V}$, with the following properties:

- $a \cdot (b \cdot \boldsymbol{v}) = (a \cdot b) \cdot \boldsymbol{v}$, for all $a, b \in \mathcal{K}$, for all $\boldsymbol{v} \in \mathcal{V}$.
- $1 \cdot \boldsymbol{v} = \boldsymbol{v}$ for all $\boldsymbol{v} \in \mathcal{V}$ where 1 denotes the multiplicative identity of $\mathcal{K}$.
- $a \cdot (\boldsymbol{v_1} + \boldsymbol{v_2}) = a \cdot \boldsymbol{v_1} + a \cdot \boldsymbol{v_2}$ for all $a \in \mathcal{K}$, for all $\boldsymbol{v_1}, \boldsymbol{v_2} \in \mathcal{V}$.
- $(a + b) \cdot \boldsymbol{v} = a \cdot \boldsymbol{v} + b \cdot \boldsymbol{v}$ for all $a, b \in \mathcal{K}$, for all $\boldsymbol{v} \in \mathcal{V}$.

Definition of a Function Space

The set $\mathcal{F}$ of all functions from an arbitrary generic domain $\mathcal{X}$ to a vector space co-domain $\mathcal{V}$ (over scalars field $\mathcal{K}$) constitutes a vector space (known as function space) over the scalars field $\mathcal{K}$ with addition operation $(+)$ defined as:

$$(f + g)(x) = f(x) + g(x) \text{ for all } f, g \in \mathcal{F}, \text{ for all } x \in \mathcal{X}$$

and scalar multiplication operation $(\cdot)$ defined as:

$$(a \cdot f)(x) = a \cdot f(x) \text{ for all } f \in \mathcal{F}, \text{ for all } a \in \mathcal{K}, \text{ for all } x \in \mathcal{X}$$

Function Space of Linear Maps

A linear map is a function $f : \mathcal{V} \rightarrow \mathcal{W}$ where $\mathcal{V}$ is a vector space over a scalars field $\mathcal{K}$ and $\mathcal{W}$ is a vector space over the same scalars field $\mathcal{K}$, having the following two properties:

- $f(\boldsymbol{v_1} + \boldsymbol{v_2}) = f(\boldsymbol{v_1}) + f(\boldsymbol{v_2})$ for all $\boldsymbol{v_1}, \boldsymbol{v_2} \in \mathcal{V}$ (i.e., application of f commutes with the addition operation).
- $f(a \cdot \boldsymbol{v}) = a \cdot f(\boldsymbol{v})$ for all $\boldsymbol{v} \in \mathcal{V}$, for all $a \in \mathcal{K}$ (i.e., applications of f commutes with the scalar multiplication operation).

Then the set of all linear maps with domain $\mathcal{V}$ and co-domain $\mathcal{W}$ constitute a function space (restricted to just this subspace of all linear maps, rather than the space of all $\mathcal{V} \to \mathcal{W}$ functions). This function space (restricted to the subspace of all $\mathcal{V} \to \mathcal{W}$ linear maps) is denoted as the vector space $\mathcal{L}(\mathcal{V}, \mathcal{W})$.

The specialization of the function space of linear maps to the space $\mathcal{L}(\mathcal{V}, \mathcal{K})$ (i.e., specializing the vector space $\mathcal{W}$ to the scalars field $\mathcal{K}$) is known as the dual vector space and is denoted as $\mathcal{V}^*$.

Vector Spaces in Function Approximations

We represent function approximations by parameterized functions $f : \mathcal{X} \times D[\mathbb{R}] \to \mathbb{R}$ where $\mathcal{X}$ is the input domain and $D[\mathbb{R}]$ is the parameters domain. The notation $D[Y]$ refers to a generic container data type D over a component generic data type Y. The data type D is specified as a generic container type because we consider generic function approximations here. A specific family of function approximations will customize to a specific container data type for D (eg: linear function approximations will customize D to a Sequence data type, a feed-forward deep neural network will customize D to a Sequence of 2-dimensional arrays). We consider 2 different Vector Spaces relevant to Function Approximations:

Parameters Space $\mathcal{P}$

$D[\mathbb{R}]$ forms a vector space $\mathcal{P}$ over the scalars field $\mathbb{R}$ with addition operation defined as element-wise real-numbered addition and scalar multiplication operation defined as element-wise multiplication with real-numbered scalars. We refer to this vector space $\mathcal{P}$ as the *Parameters Space*.

Representational Space $\mathcal{G}$

We consider a function $I : \mathcal{P} \to (\mathcal{X} \to \mathbb{R})$ defined as $I(\boldsymbol{w}) = g : \mathcal{X} \to \mathbb{R}$ for all $\boldsymbol{w} \in \mathcal{P}$ such that $g(x) = f(x, \boldsymbol{w})$ for all $x \in \mathcal{X}$. The *Range* of this function I forms a vector space $\mathcal{G}$ over the scalars field $\mathbb{R}$ with addition operation defined as:

$$I(\boldsymbol{w_1}) + I(\boldsymbol{w_2}) = I(\boldsymbol{w_1} + \boldsymbol{w_2}) \text{ for all } \boldsymbol{w_1}, \boldsymbol{w_2} \in \mathcal{P}$$

and multiplication operation defined as:

$$a \cdot I(\boldsymbol{w}) = I(a \cdot \boldsymbol{w}) \text{ for all } \boldsymbol{w} \in \mathcal{P}, \text{ for all } a \in \mathbb{R}$$

We refer to this vector space $\mathcal{G}$ as the *Representational Space* (to signify the fact that addition and multiplication operations in $\mathcal{G}$ essentially "delegate" to addition and multiplication operations in the Parameters Space $\mathcal{P}$, with any parameters $\boldsymbol{w} \in \mathcal{P}$ serving as the internal representation of a function approximation $I(\boldsymbol{w}) : \mathcal{X} \to \mathbb{R}$). This "delegation" from $\mathcal{G}$ to $\mathcal{P}$ implies that I is a linear map from Parameters Space $\mathcal{P}$ to Representational Space $\mathcal{G}$.

The Gradient Function

The gradient of a function approximation $f : \mathcal{X} \times D[\mathbb{R}] \to \mathbb{R}$ with respect to parameters $\boldsymbol{w} \in D[\mathbb{R}]$ (denoted as $\nabla_{\boldsymbol{w}} f(x, \boldsymbol{w})$) is an element of the parameters domain $D[\mathbb{R}]$. By treating both $\boldsymbol{w}$ and $\nabla_{\boldsymbol{w}} f(x, \boldsymbol{w})$ as vectors in the Parameters Space $\mathcal{P}$, we define the gradient function

$$G : \mathcal{X} \to (\mathcal{P} \to \mathcal{P})$$

as:

$$G(x)(\boldsymbol{w}) = \nabla_{\boldsymbol{w}} f(x, \boldsymbol{w})$$

for all $x \in \mathcal{X}$, for all $\boldsymbol{w} \in \mathcal{P}$.

Linear Function Approximations

If we restrict to linear function approximations, for all $x \in \mathcal{X}$,

$$f(x, \boldsymbol{w}) = h(\boldsymbol{w}) = \boldsymbol{\Phi}(x) \circ \boldsymbol{w}$$

where $\boldsymbol{w} \in \mathbb{R}^m = \mathcal{P}$ and $\boldsymbol{\Phi} : \mathcal{X} \to \mathbb{R}^m$ represents the feature functions (with $\circ$ denoting inner-product in the vector space $\mathbb{R}^m$).

Then the gradient function $G : \mathcal{X} \to (\mathbb{R}^m \to \mathbb{R}^m)$ can be written as:

$$G(x)(\boldsymbol{w}) = \nabla_{\boldsymbol{w}}(\boldsymbol{\Phi}(x) \circ \boldsymbol{w}) = \boldsymbol{\Phi}(x)$$

for all $x \in \mathcal{X}$, for all $\boldsymbol{w} \in \mathbb{R}^m$.

Also note that in the case of linear function approximations, the function $I : \mathcal{R}^m \to (\mathcal{X} \to \mathbb{R})$ is a linear map from $\mathcal{R}^m = \mathcal{P}$ to a vector subspace of the function space $\mathcal{F}$ of all $\mathcal{X} \to \mathbb{R}$ functions over scalars field $\mathbb{R}$ (with pointwise operations). This is because for all $x \in \mathcal{X}$:

$$\boldsymbol{\Phi}(x) \circ (\boldsymbol{w}_1 + \boldsymbol{w}_2) = \boldsymbol{\Phi}(x) \circ \boldsymbol{w}_1 + \boldsymbol{\Phi}(x) \circ \boldsymbol{w}_2 \text{ for all } \boldsymbol{w}_1, \boldsymbol{w}_2 \in \mathbb{R}^m$$
$$\boldsymbol{\Phi}(x) \circ (a \cdot \boldsymbol{w}) = a \cdot (\boldsymbol{\Phi}(x) \circ \boldsymbol{w}) \text{ for all } \boldsymbol{w} \in \mathbb{R}^m, \text{ for all } a \in \mathbb{R}$$

The key concept here is that for the case of linear function approximations, addition and multiplication "delegating" operations in $\mathcal{G}$ coincide with addition and multiplication "pointwise" operations in $\mathcal{F}$, which implies that $\mathcal{G}$ is isomorphic to a vector subspace of $\mathcal{F}$.

Stochastic Gradient Descent

Stochastic Gradient Descent is a function

$$SGD : \mathcal{X} \times \mathbb{R} \to (\mathcal{P} \to \mathcal{P})$$

representing a mapping from (predictor, response) data to a "parameters-update" function (in order to improve the function approximation), defined as:

$$SGD(x,y)(\boldsymbol{w}) = \boldsymbol{w} - \alpha \cdot (f(x,\boldsymbol{w}) - y) \cdot G(x)(\boldsymbol{w})$$

for all $x \in \mathcal{X}, y \in \mathbb{R}, \boldsymbol{w} \in \mathcal{P}$, where $\alpha \in \mathbb{R}^+$ represents the learning rate (step size of SGD).

For a fixed data pair $(x,y) \in \mathcal{X} \times \mathbb{R}$, with prediction error function $e : \mathcal{P} \to \mathbb{R}$ defined as $e(\boldsymbol{w}) = y - f(x,\boldsymbol{w})$, the (SGD-based) parameters change function (function from parameters to change in parameters)

$$U : \mathcal{P} \to \mathcal{P}$$

is defined as:

$$U(\boldsymbol{w}) = SGD(x,y)(\boldsymbol{w}) - \boldsymbol{w} = \alpha \cdot e(\boldsymbol{w}) \cdot G(x)(\boldsymbol{w})$$

for all $\boldsymbol{w} \in \mathcal{P}$.

So, we can conceptualize the parameters change function U as the product of:

- Learning rate $\alpha \in \mathbb{R}^+$
- Prediction error function $e : \mathcal{P} \to \mathbb{R}$
- Gradient operator $G(x) : \mathcal{P} \to \mathcal{P}$

Note that the product of functions e and $G(x)$ above is element-wise in their common domain $\mathcal{P} = D[\mathbb{R}]$, resulting in the scalar ($\mathbb{R}$) multiplication of vectors in $\mathcal{P}$.

Updating vector $\boldsymbol{w}$ to vector $\boldsymbol{w} + U(\boldsymbol{w})$ in the Parameter Space $\mathcal{P}$ results in updating function $I(\boldsymbol{w}) : \mathcal{X} \to \mathbb{R}$ to function $I(\boldsymbol{w} + U(\boldsymbol{w})) : \mathcal{X} \to \mathbb{R}$ in the Representational Space $\mathcal{G}$. This is rather convenient since we can view all of the above addition/multiplication operations in the Parameter Space $\mathcal{P}$ as addition/multiplication operations in the Representational Space $\mathcal{G}$.

SGD Update for Linear Function Approximations

As a reminder, for the case of linear function approximations, $\mathcal{G}$ is isomorphic to a vector subspace of the function space $\mathcal{F}$ of all $\mathcal{X} \to \mathbb{R}$ functions over the scalars field $\mathbb{R}$ (addition and multiplication "delegating" operations in $\mathcal{G}$ coincide with addition and multiplication "pointwise" operations in $\mathcal{F}$).

So in the case of linear function approximations, when updating vector $\boldsymbol{w}$ to vector $\boldsymbol{w} + \alpha \cdot (y - \boldsymbol{\Phi}(x) \circ \boldsymbol{w}) \cdot \boldsymbol{\Phi}(x)$ in the Parameter Space $\mathcal{P} = \mathbb{R}^m$, applying the linear map $I : \mathbb{R}^m \to \mathcal{G}$ updates functions in $\mathcal{G}$ with corresponding pointwise addition and multiplication operations.

Concretely, a linear function approximation $g : \mathcal{X} \to \mathbb{R}$ defined as $g(z) = \boldsymbol{\Phi}(z) \circ \boldsymbol{w}$ for all $z \in \mathcal{X}$ updates correspondingly to the function $g^{(x,y)} : \mathcal{X} \to \mathbb{R}$ defined as $g^{(x,y)}(z) = \boldsymbol{\Phi}(z) \circ \boldsymbol{w} + \alpha \cdot (y - \boldsymbol{\Phi}(x) \circ \boldsymbol{w}) \cdot (\boldsymbol{\Phi}(z) \circ \boldsymbol{\Phi}(x))$ for all $z \in \mathcal{X}$.

It's useful to note that the change in the evaluation at $z \in \mathcal{X}$ is simply the product of:

- Learning rate $\alpha \in \mathbb{R}^+$

- Prediction Error $y - \mathbf{\Phi}(x) \circ \boldsymbol{w} \in \mathbb{R}$ for the updating data $(x, y) \in \mathcal{X} \times \mathbb{R}$
- Inner-product of the feature vector $\mathbf{\Phi}(x) \in \mathbb{R}^m$ of the updating input value $x \in \mathcal{X}$ and the feature vector $\mathbf{\Phi}(z) \in \mathbb{R}^m$ of the evaluation input value $z \in \mathcal{X}$.

C. Portfolio Theory

In this Appendix, we provide a quick and terse introduction to *Portfolio Theory*. While this topic is not a direct pre-requisite for the topics we cover in the chapters, we believe one should have some familiarity with the risk versus reward considerations when constructing portfolios of financial assets, and know of the important results. To keep this Appendix brief, we will provide the minimal content required to understand the *essence* of the key concepts. We won't be doing rigorous proofs. We will also ignore details pertaining to edge-case/irregular-case conditions so as to focus on the core concepts.

Setting and Notation

In this section, we go over the core setting of Portfolio Theory, along with the requisite notation.

Assume there are n assets in the economy and that their mean returns are represented in a column vector $R \in \mathbb{R}^n$. We denote the covariance of returns of the n assets by an $n \times n$ non-singular matrix V.

We consider arbitrary portfolios p comprised of investment quantities in these n assets that are normalized to sum up to 1. Denoting column vector $X_p \in \mathbb{R}^n$ as the investment quantities in the n assets for portfolio p, we can write the normality of the investment quantities in vector notation as:

$$X_p^T \cdot 1_n = 1$$

where $1_n \in \mathbb{R}^n$ is a column vector comprising of all 1's.

We shall drop the subscript p in X_p whenever the reference to portfolio p is clear.

Portfolio Returns

- A single portfolio's mean return is $X^T \cdot R \in \mathbb{R}$
- A single portfolio's variance of return is the quadratic form $X^T \cdot V \cdot X \in \mathbb{R}$
- Covariance between portfolios p and q is the bilinear form $X_p^T \cdot V \cdot X_q \in \mathbb{R}$
- Covariance of the n assets with a single portfolio is the vector $V \cdot X \in \mathbb{R}^n$

Derivation of Efficient Frontier Curve

An asset which has no variance in terms of how it's value evolves in time is known as a risk-free asset. The Efficient Frontier is defined for a world with no risk-free assets. The Efficient Frontier is the set of portfolios with minimum variance of return for each level of portfolio mean return (we refer to a portfolio in the Efficient Frontier as an *Efficient Portfolio*). Hence, to determine the Efficient Frontier, we solve for X so as to minimize portfolio variance $X^T \cdot V \cdot X$ subject to constraints:

$$X^T \cdot 1_n = 1$$

$$X^T \cdot R = r_p$$

where r_p is the mean return for Efficient Portfolio p. We set up the Lagrangian and solve to express X in terms of R, V, r_p. Substituting for X gives us the efficient frontier parabola of Efficient Portfolio Variance σ_p^2 as a function of its mean r_p:

$$\sigma_p^2 = \frac{a - 2br_p + cr_p^2}{ac - b^2}$$

where

- $a = R^T \cdot V^{-1} \cdot R$
- $b = R^T \cdot V^{-1} \cdot 1_n$
- $c = 1_n^T \cdot V^{-1} \cdot 1_n$

Global Minimum Variance Portfolio (GMVP)

The global minimum variance portfolio (GMVP) is the portfolio at the tip of the efficient frontier parabola, i.e., the portfolio with the lowest possible variance among all portfolios on the Efficient Frontier. Here are the relevant characteristics for the GMVP:

- It has mean $r_0 = \frac{b}{c}$
- It has variance $\sigma_0^2 = \frac{1}{c}$
- It has investment proportions $X_0 = \frac{V^{-1} \cdot 1_n}{c}$

GMVP is positively correlated with all portfolios and with all assets. GMVP's covariance with all portfolios and with all assets is a constant value equal to $\sigma_0^2 = \frac{1}{c}$ (which is also equal to its own variance).

Orthogonal Efficient Portfolios

For every efficient portfolio p (other than GMVP), there exists a unique orthogonal efficient portfolio z (i.e. $Covariance(p, z) = 0$) with finite mean

$$r_z = \frac{a - br_p}{b - cr_p}$$

z always lies on the opposite side of p on the (efficient frontier) parabola. If we treat the Efficient Frontier as a curve of mean (y-axis) versus variance (x-axis), the straight line from p to GMVP intersects the mean axis (y-axis) at r_z. If we treat the Efficient Frontier as a curve of mean (y-axis) versus standard deviation (x-axis), the tangent to the efficient frontier at p intersects the mean axis (y-axis) at r_z. Moreover, all portfolios on one side of the efficient frontier are positively correlated with each other.

Two-fund Theorem

The X vector (normalized investment quantities in assets) of any efficient portfolio is a linear combination of the X vectors of two other efficient portfolios. Notationally,

$$X_p = \alpha X_{p_1} + (1 - \alpha) X_{p_2} \text{ for some scalar } \alpha$$

Varying α from $-\infty$ to $+\infty$ basically traces the entire efficient frontier. So to construct all efficient portfolios, we just need to identify two canonical efficient portfolios. One of them is GMVP. The other is a portfolio we call Special Efficient Portfolio (SEP) with:

- Mean $r_1 = \frac{a}{b}$
- Variance $\sigma_1^2 = \frac{a}{b^2}$
- Investment proportions $X_1 = \frac{V^{-1} \cdot R}{b}$

The orthogonal portfolio to SEP has mean $r_z = \frac{a - b\frac{a}{b}}{b - c\frac{a}{b}} = 0$

An example of the Efficient Frontier for 16 assets

Figure C.1 shows a plot of the mean daily returns versus the standard deviation of daily returns collected over a 3-year period for 16 assets. The blue curve is the Efficient Frontier for these 16 assets. Note the special portfolios GMVP and SEP on the Efficient Frontier. This curve was generated from the code at rl/appendix2/efficient_frontier.py. We encourage you to play with different choices (and count) of assets, and to also experiment with different time ranges as well as to try weekly and monthly returns.

CAPM: Linearity of Covariance Vector w.r.t. Mean Returns

Important Theorem: The covariance vector of individual assets with a portfolio (note: covariance vector $= V \cdot X \in \mathbb{R}^n$) can be expressed as an exact linear function of the individual assets' mean returns vector if and only if the portfolio is efficient. If the efficient portfolio is p (and its orthogonal portfolio z), then:

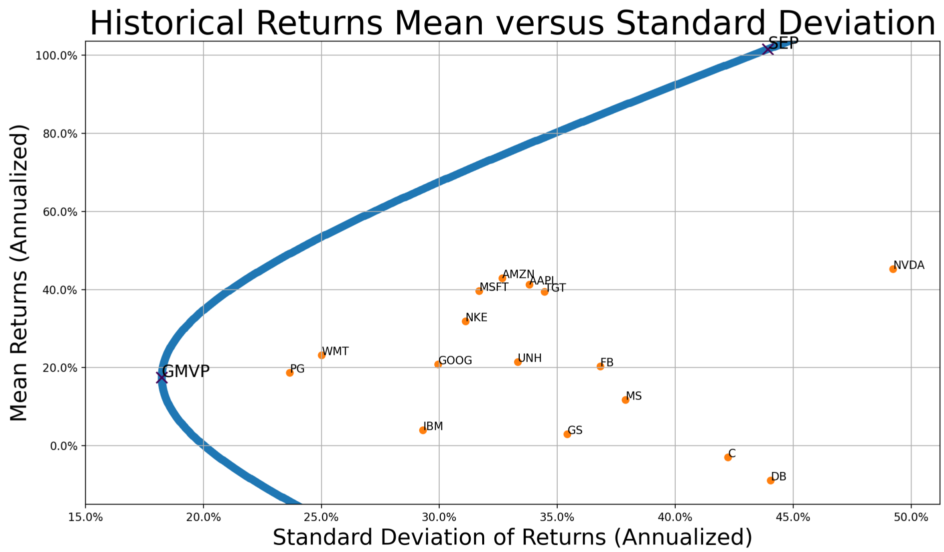

Figure C.1.: Efficient Frontier for 16 Assets

$$R = r_z 1_n + \frac{r_p - r_z}{\sigma_p^2}(V \cdot X_p) = r_z 1_n + (r_p - r_z)\beta_p$$

where $\beta_p = \frac{V \cdot X_p}{\sigma_p^2} \in \mathbb{R}^n$ is the vector of slope coefficients of regressions where the explanatory variable is the portfolio mean return $r_p \in \mathbb{R}$ and the n dependent variables are the asset mean returns $R \in \mathbb{R}^n$.

The linearity of β_p w.r.t. mean returns R is famously known as the Capital Asset Pricing Model (CAPM).

Useful Corollaries of CAPM

- If p is SEP, $r_z = 0$ which would mean:

$$R = r_p \beta_p = \frac{r_p}{\sigma_p^2} \cdot V \cdot X_p$$

- So, in this case, covariance vector $V \cdot X_p$ and β_p are just scalar multiples of asset mean vector.
- The investment proportion X in a given individual asset changes monotonically along the efficient frontier.
- Covariance $V \cdot X$ is also monotonic along the efficient frontier.
- But β is not monotonic, which means that for every individual asset, there is a unique pair of efficient portfolios that result in maximum and minimum βs for that asset.

Cross-Sectional Variance

- The cross-sectional variance in βs (variance in βs across assets for a fixed efficient portfolio) is zero when the efficient portfolio is GMVP and is also zero when the efficient portfolio has infinite mean.
- The cross-sectional variance in βs is maximum for the two efficient portfolios with means: $r_0 + \sigma_0^2\sqrt{|A|}$ and $r_0 - \sigma_0^2\sqrt{|A|}$ where A is the 2 $\times$ 2 matrix consisting of a, b, b, c
- These two portfolios lie symmetrically on opposite sides of the efficient frontier (their βs are equal and of opposite signs), and are the only two orthogonal efficient portfolios with the same variance ($= 2\sigma_0^2$)

Efficient Set with a Risk-Free Asset

If we have a risk-free asset with return r_F, then V is singular. So we first form the Efficient Frontier without the risk-free asset. The Efficient Set (including the risk-free asset) is defined as the tangent to this Efficient Frontier (without the risk-free asset) from the point $(0, r_F)$ when the Efficient Frontier is considered to be a curve of mean returns (y-axis) against standard deviation of returns (x-axis).

Let's say the tangent touches the Efficient Portfolio at the point (Portfolio) T and let it's return be r_T. Then:

- If $r_F < r_0, r_T > r_F$
- If $r_F > r_0, r_T < r_F$
- All portfolios on this efficient set are perfectly correlated

D. Introduction to and Overview of Stochastic Calculus Basics

In this Appendix, we provide a quick introduction to the *Basics of Stochastic Calculus*. To be clear, Stochastic Calculus is a vast topic requiring an entire graduate-level course to develop a good understanding. We shall only be scratching the surface of Stochastic Calculus and even with the very basics of this subject, we will focus more on intuition than rigor, and familiarize you with just the most important results relevant to this book. For an adequate treatment of Stochastic Calculus relevant to Finance, we recommend Steven Shreve's two-volume discourse Stochastic Calculus for Finance I and Stochastic Calculus for Finance II. For a broader treatment of Stochastic Calculus, we recommend Bernt Oksendal's book on Stochastic Differential Equations.

Simple Random Walk

The best way to get started with Stochastic Calculus is to first get familiar with key properties of a *simple random walk* viewed as a discrete-time, countable state-space, stationary Markov Process. The state space is the set of integers $\mathbb{Z}$. Denoting the random state at time t as Z_t, the state transitions are defined in terms of the independent and identically distributed (i.i.d.) random variables Y_t for all $t = 0, 1, \ldots$

$$Z_{t+1} = Z_t + Y_t \text{ and } \mathbb{P}[Y_t = 1] = \mathbb{P}[Y_t = -1] = 0.5 \text{ for all } t = 0, 1, \ldots$$

A quick point on notation: We refer to the random state at time t as Z_t (i.e., as a random variable at time t), whereas we refer to the Markov Process for this simple random walk as Z (i.e., without any subscript).

Since the random variables $\{Y_t | t = 0, 1, \ldots\}$ are i.i.d, the *increments* $Z_{t_{i+1}} - Z_{t_i}, i = 0, 1, \ldots n-1$ in the random walk states for any set of time steps $t_0 < t_1 < \ldots < t_n$ have the following properties:

- **Independent Increments**: Increments $Z_{t_1} - Z_{t_0}, Z_{t_2} - Z_{t_1}, \ldots, Z_{t_n} - Z_{t_{n-1}}$ are independent of each other
- **Martingale (i.e., Zero-Drift) Property**: Expected Value of Increment $\mathbb{E}[(Z_{t_{i+1}} - Z_{t_i})] = 0$ for all $i = 0, 1, \ldots, n-1$
- **Variance of Increment equals Time Steps**: Variance of Increment

$$\mathbb{E}[(Z_{t_{i+1}} - Z_{t_i})^2] = \sum_{j=t_i}^{t_{i+1}-1} \mathbb{E}[(Z_{j+1} - Z_j)^2] = t_{i+1} - t_i \text{ for all } i = 0, 1, \ldots, n-1$$

Moreover, we have an important property that **Quadratic Variation equals Time Steps**. Quadratic Variation over the time interval $[t_i, t_{i+1}]$ for all $i = 0, 1, \ldots, n-1$ is defined as:

$$\sum_{j=t_i}^{t_{i+1}-1} (Z_{j+1} - Z_j)^2$$

Since $(Z_{j+1} - Z_j)^2 = Y_j^2 = 1$ for all $j = t_i, t_i+1, \ldots, t_{i+1}-1$, Quadratic Variation

$$\sum_{j=t_i}^{t_{i+1}-1} (Z_{j+1} - Z_j)^2 = t_{i+1} - t_i \text{ for all } i = 0, 1, \ldots n-1$$

It pays to emphasize the important conceptual difference between the Variance of Increment property and Quadratic Variation property: Variance of Increment property is a statement about *expectation* of the square of the $Z_{t_{i+1}} - Z_{t_i}$ increment whereas Quadratic Variation property is a statement of certainty (note: there is no $\mathbb{E}[\cdots]$ in this statement) about the sum of squares of *atomic* increments Y_j over the discrete-steps time-interval $[t_i, t_{i+1}]$. The Quadratic Variation property owes to the fact that $\mathbb{P}[Y_t^2 = 1] = 1$ for all $t = 0, 1, \ldots$.

We can view the Quadratic Variations of a Process X over all discrete-step time intervals $[0, t]$ as a Process denoted $[X]$, defined as:

$$[X]_t = \sum_{j=0}^{t} (X_{j+1} - X_j)^2$$

Thus, for the simple random walk Markov Process Z, we have the succinct formula: $[Z]_t = t$ for all t (i.e., this Quadratic Variation process is a deterministic process).

Brownian Motion as Scaled Random Walk

Now let us take our simple random walk process Z, and simultaneously A) speed up time and B) scale down the size of the atomic increments Y_t. Specifically, define for any fixed positive integer n:

$$z_t^{(n)} = \frac{1}{\sqrt{n}} \cdot Z_{nt} \text{ for all } t \in \frac{\mathbb{Z}_{\geq 0}}{n}$$

It's easy to show that the above properties of the simple random walk process holds for the $z^{(n)}$ process as well. Now consider the continuous-time process z defined as:

$$z_t = \lim_{n \to \infty} z_t^{(n)} \text{ for all } t \in \mathbb{R}_{\geq 0}$$

This continuous-time process z with $z_0 = 0$ is known as standard Brownian Motion. z retains the same properties as those of the simple random walk process that we have listed above (independent increments, martingale, increment variance equal to time interval, and quadratic variation equal to the time interval). Also, by Central Limit Theorem,

$$z_t | z_s \sim \mathcal{N}(z_s, t - s) \text{ for any } 0 \leq s < t$$

We denote dz_t as the increment in z over the infinitesimal time interval $[t, t + dt]$.

$$dz_t \sim \mathcal{N}(0, dt)$$

Continuous-Time Stochastic Processes

Brownian motion z was our first example of a continuous-time stochastic process. Now let us define a general continuous-time stochastic process, although for the sake of simplicity, we shall restrict ourselves to one-dimensional real-valued continuous-time stochastic processes.

Definition D.0.1. A *One-dimensional Real-Valued Continuous-Time Stochastic Process* denoted X is defined as a collection of real-valued random variables $\{X_t | t \in [0, T]\}$ (for some fixed $T \in \mathbb{R}$, with index t interpreted as continuous-time) defined on a common probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is a sample space, $\mathcal{F}$ is a σ-algebra and $\mathbb{P}$ is a probability measure (so, $X_t : \Omega \to \mathbb{R}$ for each $t \in [0, T]$).

We can view a stochastic process X as an $\mathbb{R}$-valued function of two variables:

- $t \in [0, T]$
- $\omega \in \Omega$

As a two-variable function, if we fix t, then we get the random variable $X_t : \Omega \to \mathbb{R}$ for time t and if we fix ω, then we get a single $\mathbb{R}$-valued outcome for each random variable across time (giving us a *sample path* in time, denoted $X(\omega)$).

Now let us come back to Brownian motion, viewed as a Stochastic Process.

Properties of Brownian Motion sample paths

- Sample paths $z(\omega)$ of Brownian motion z are continuous
- Sample paths $z(\omega)$ are almost always non-differentiable, meaning:

$$\text{Random variable } \lim_{h \to 0} \frac{z_{t+h} - z_t}{h} \text{ is almost always infinite}$$

  The intuition is that $\frac{dz_t}{dt}$ has standard deviation of $\frac{1}{\sqrt{dt}}$, which goes to ∞ as dt goes to 0
- Sample paths $z(\omega)$ have infinite total variation, meaning:

$$\text{Random variable } \int_S^T |dz_t| = \infty \text{ (almost always)}$$

The quadratic variation property can be expressed as:

$$\int_S^T (dz_t)^2 = T - S$$

This means each sample random path of brownian motion has quadratic variation equal to the time interval of the path. The quadratic variation of z expressed as a process $[z]$ has the deterministic value of t at time t. Expressed in infinitesimal terms, we say that:

$$(dz_t)^2 = dt$$

This formula generalizes to:

$$(dz_t^{(1)}) \cdot (dz_t^{(2)}) = \rho \cdot dt$$

where $z^{(1)}$ and $z^{(2)}$ are two different brownian motions with correlation between the random variables $z_t^{(1)}$ and $z_t^{(2)}$ equal to ρ for all $t > 0$.

You should intuitively interpret the formula $(dz_t)^2 = dt$ (and it's generalization) as a deterministic statement, and in fact this statement is used as an algebraic convenience in Brownian motion-based stochastic calculus, forming the core of *Ito Isometry* and *Ito's Lemma* (which we cover shortly, but first we need to define the Ito Integral).

Ito Integral

We want to define a stochastic process Y from a stochastic process X as follows:

$$Y_t = \int_0^t X_s \cdot dz_s$$

In the interest of focusing on intuition rather than rigor, we skip the technical details of filtrations and adaptive processes that make the above integral sensible. Instead, we simply say that this integral makes sense only if random variable X_s for any time s is disallowed from depending on $z_{s'}$ for any $s' > s$ (i.e., the stochastic process X cannot peek into the future) and that the time-integral $\int_0^t X_s^2 \cdot ds$ is finite for all $t \geq 0$. So we shall roll forward with the assumption that the stochastic process Y is defined as the above-specified integral (known as the *Ito Integral*) of a stochastic process X with respect to Brownian motion. The equivalent notation is:

$$dY_t = X_t \cdot dz_t$$

We state without proof the following properties of the Ito Integral stochastic process Y:

- Y is a martingale, i.e., $\mathbb{E}[Y_t|Y_s] = 0$ for all $0 \leq s < t$
- **Ito Isometry**: $\mathbb{E}[Y_t^2] = \int_0^t \mathbb{E}[X_s^2] \cdot ds$.

- Quadratic Variance formula: $[Y]_t = \int_0^t X_s^2 \cdot ds$

Ito Isometry generalizes to:

$$\mathbb{E}[(\int_S^T X_t^{(1)} \cdot dz_t^{(1)})(\int_S^T X_t^{(2)} \cdot dz_t^{(2)})] = \int_S^T \mathbb{E}[X_t^{(1)} \cdot X_t^{(2)} \cdot \rho \cdot dt]$$

where $X^{(1)}$ and $X^{(2)}$ are two different stochastic processes, and $z^{(1)}$ and $z^{(2)}$ are two different brownian motions with correlation between the random variables $z_t^{(1)}$ and $z_t^{(2)}$ equal to ρ for all $t > 0$.

Likewise, the Quadratic Variance formula generalizes to:

$$\int_S^T (X_t^{(1)} \cdot dz_t^{(1)})(X_t^{(2)} \cdot dz_t^{(2)}) = \int_S^T X_t^{(1)} \cdot X_t^{(2)} \cdot \rho \cdot dt$$

Ito's Lemma

We can extend the above Ito Integral to an Ito process Y as defined below:

$$dY_t = \mu_t \cdot dt + \sigma_t \cdot dz_t$$

We require the same conditions for the stochastic process σ as we required above for X in the definition of the Ito Integral. Moreover, we require that: $\int_0^t |\mu_s| \cdot ds$ is finite for all $t \geq 0$.

In the context of this Ito process Y described above, we refer to μ as the *drift* process and we refer to σ as the *dispersion* process.

Now, consider a twice-differentiable function $f : [0, T] \times \mathbb{R} \to \mathbb{R}$. We define a stochastic process whose (random) value at time t is $f(t, Y_t)$. Let's write it's Taylor series with respect to the variables t and Y_t.

$$df(t, Y_t) = \frac{\partial f(t, Y_t)}{\partial t} \cdot dt + \frac{\partial f(t, Y_t)}{\partial Y_t} \cdot dY_t + \frac{1}{2} \cdot \frac{\partial^2 f(t, Y_t)}{\partial Y_t^2} \cdot (dY_t)^2 + \ldots$$

Substituting for dY_t and lightening notation, we get:

$$df(t, Y_t) = \frac{\partial f}{\partial t} \cdot dt + \frac{\partial f}{\partial Y_t} \cdot (\mu_t \cdot dt + \sigma_t \cdot dz_t) + \frac{1}{2} \cdot \frac{\partial^2 f}{\partial Y_t^2} \cdot (\mu_t \cdot dt + \sigma_t \cdot dz_t)^2 + \ldots$$

Next, we use the rules: $(dt)^2 = 0, dt \cdot dz_t = 0, (dz_t)^2 = dt$ to get **Ito's Lemma**:

$$df(t, Y_t) = (\frac{\partial f}{\partial t} + \mu_t \cdot \frac{\partial f}{\partial Y_t} + \frac{\sigma_t^2}{2} \cdot \frac{\partial^2 f}{\partial Y_t^2}) \cdot dt + \sigma_t \cdot \frac{\partial f}{\partial Y_t} \cdot dz_t \tag{D.1}$$

Ito's Lemma describes the stochastic process of a function (f) of an Ito Process (Y) in terms of the partial derivatives of f, and in terms of the drift (μ) and dispersion (σ) processes that define Y.

If we generalize $\boldsymbol{Y}$ to be an n-dimensional stochastic process (as a column vector) with $\boldsymbol{\mu}_t$ as an n-dimensional (stochastic) column vector, $\boldsymbol{\sigma}_t$ as an $n \times$

m (stochastic) matrix, and $\boldsymbol{z}_t$ as an m-dimensional vector of m independent standard brownian motions (as follows)

$$d\boldsymbol{Y}_t = \boldsymbol{\mu}_t \cdot dt + \boldsymbol{\sigma}_t \cdot d\boldsymbol{z}_t$$

then we get the multi-variate version of Ito's Lemma, as follows:

$$df(t, \boldsymbol{Y}_t) = (\frac{\partial f}{\partial t} + (\nabla_{\boldsymbol{Y}} f)^T \cdot \boldsymbol{\mu}_t + \frac{1}{2} Tr[\boldsymbol{\sigma}_t^T \cdot (\Delta_{\boldsymbol{Y}} f) \cdot \boldsymbol{\sigma}_t]) \cdot dt + (\nabla_{\boldsymbol{Y}} f)^T \cdot \boldsymbol{\sigma}_t \cdot d\boldsymbol{z}_t \quad \text{(D.2)}$$

where the symbol ∇ represents the gradient of a function, the symbol Δ represents the Hessian of a function, and the symbol Tr represents the Trace of a matrix.

Next, we cover two common Ito processes, and use Ito's Lemma to solve the Stochastic Differential Equation represented by these Ito Processes:

A Lognormal Process

Consider a stochastic process x described in the form of the following Ito process:

$$dx_t = \mu(t) \cdot x_t \cdot dt + \sigma(t) \cdot x_t \cdot dz_t$$

Note that here z is standard (one-dimensional) Brownian motion, and μ, σ are deterministic functions of time t. This is solved easily by defining an appropriate function of x_t and applying Ito's Lemma, as follows:

$$y_t = \log(x_t)$$

Applying Ito's Lemma on y_t with respect to x_t, we get:

$$dy_t = \frac{dx_t}{x_t} - \frac{\sigma^2(t)}{2} \cdot dt = (\mu(t) - \frac{\sigma^2(t)}{2}) \cdot dt + \sigma(t) \cdot dz_t$$

$$y_T = y_S + \int_S^T (\mu(t) - \frac{\sigma^2(t)}{2}) \cdot dt + \int_S^T \sigma(t) \cdot dz_t$$

$$x_T = x_S \cdot e^{\int_S^T (\mu(t) - \frac{\sigma^2(t)}{2}) \cdot dt + \int_S^T \sigma(t) \cdot dz_t}$$

$x_T | x_S$ follows a lognormal distribution, i.e.,

$$y_T = \log(x_T) \sim \mathcal{N}(\log(x_S) + \int_S^T (\mu(t) - \frac{\sigma^2(t)}{2}) \cdot dt, \int_S^T \sigma^2(t) \cdot dt)$$

$$E[x_T | x_S] = x_S \cdot e^{\int_S^T \mu(t) \cdot dt}$$

$$E[x_T^2 | x_S] = x_S^2 \cdot e^{\int_S^T (2\mu(t) + \sigma^2(t)) \cdot dt}$$

$$Variance[x_T | x_S] = E[x_T^2 | x_S] - (E[x_T | x_S])^2 = x_S^2 \cdot e^{\int_S^T 2\mu(t) \cdot dt} \cdot (e^{\int_S^T \sigma^2(t) \cdot dt} - 1)$$

The special case of $\mu(t) = \mu$ (constant) and $\sigma(t) = \sigma$ (constant) is a very common Ito process used all over Finance/Economics (for its simplicity, tractability as well as practicality), and is known as Geometric Brownian Motion, to reflect the fact that the stochastic increment of the process ($\sigma \cdot x_t \cdot dz_t$) is multiplicative to the level of the process x_t. If we consider this special case, we get:

$$y_T = \log(x_T) \sim \mathcal{N}(\log(x_S) + (\mu - \frac{\sigma^2}{2})(T-S), \sigma^2(T-S))$$

$$E[x_T|x_S] = x_S \cdot e^{\mu(T-S)}$$

$$Variance[x_T|x_S] = x_S^2 \cdot e^{2\mu(T-S)} \cdot (e^{\sigma^2(T-S)} - 1)$$

A Mean-Reverting Process

Now we consider a stochastic process x described in the form of the following Ito process:

$$dx_t = \mu(t) \cdot x_t \cdot dt + \sigma(t) \cdot dz_t$$

As in the process of the previous section, z is standard (one-dimensional) Brownian motion, and μ, σ are deterministic functions of time t. This is solved easily by defining an appropriate function of x_t and applying Ito's Lemma, as follows:

$$y_t = x_t \cdot e^{-\int_0^t \mu(u) \cdot du}$$

Applying Ito's Lemma on y_t with respect to x_t, we get:

$$dy_t = e^{-\int_0^t \mu(u) \cdot du} \cdot dx_t - x_t \cdot e^{-\int_0^t \mu(u) \cdot du} \cdot \mu(t) \cdot dt = \sigma(t) \cdot e^{-\int_0^t \mu(u) \cdot du} \cdot dz_t$$

y_t is a martingale. Using Ito Isometry, we get:

$$y_T \sim \mathcal{N}(y_S, \int_S^T \sigma^2(t) \cdot e^{-\int_0^t 2\mu(u) \cdot du} \cdot dt)$$

Therefore,

$$x_T \sim \mathcal{N}(x_S \cdot e^{\int_S^T \mu(t) \cdot dt}, e^{\int_0^T 2\mu(t) \cdot dt} \cdot \int_S^T \sigma^2(t) \cdot e^{-\int_0^t 2\mu(u) \cdot du} \cdot dt)$$

We call this process "mean-reverting" because with negative $\mu(t)$, the process is "pulled" to a baseline level of 0, at a speed whose expectation is proportional to $-\mu(t)$ and proportional to the distance from the baseline (so we say the process reverts to a baseline of 0 and the strength of mean-reversion is greater if the distance from the baseline is greater). If $\mu(t)$ is positive, then we say that the process is "mean-diverting" to signify that it gets pulled away from the baseline level of 0.

The special case of $\mu(t) = \mu$ (constant) and $\sigma(t) = \sigma$ (constant) is a fairly common Ito process (again for it's simplicity, tractability as well as practicality), and is known as the Ornstein-Uhlenbeck Process with the mean (baseline) level set to 0. If we consider this special case, we get:

$$x_T \sim \mathcal{N}(x_S \cdot e^{\mu(T-S)}, \frac{\sigma^2}{2\mu} \cdot (e^{2\mu(T-S)} - 1))$$

E. The Hamilton-Jacobi-Bellman (HJB) Equation

In this Appendix, we provide a quick coverage of the Hamilton-Bellman-Jacobi (HJB) Equation, which is the continuous-time version of the Bellman Optimality Equation. Although much of this book covers Markov Decision Processes in a discrete-time setting, we do cover some classical Mathematical Finance Stochastic Control formulations in continuous-time. To understand these formulations, one must first understand the HJB Equation, which is the purpose of this Appendix. As is the norm in the Appendices in this book, we will compromise on some of the rigor and emphasize the intuition to develop basic familiarity with HJB.

HJB as a continuous-time version of Bellman Optimality Equation

In order to develop the continuous-time setting, we shall consider a non-stationary process where the set of states at time t are denoted as $\mathcal{S}_t$ and the set of allowable actions for each state at time t are denoted as $\mathcal{A}_t$. Since time is continuous, Rewards are represented as a *Reward Rate* function $\mathcal{R}$ such that for any state $s_t \in \mathcal{S}_t$ and for any action $a_t \in \mathcal{A}_t$, $\mathcal{R}(t, s_t, a_t) \cdot dt$ is the *Expected Reward* in the time interval $(t, t+dt]$, conditional on state s_t and action a_t (note the functional dependency of $\mathcal{R}$ on t since we will be integrating $\mathcal{R}$ over time). Instead of the discount factor γ as in the case of discrete-time MDPs, here we employ a *discount rate* (akin to interest-rate discounting) $\rho \in \mathbb{R}_{\geq 0}$ so that the discount factor over any time interval $(t, t+dt]$ is $e^{-\rho \cdot dt}$.

We denote the Optimal Value Function as V^* such that the Optimal Value for state $s_t \in \mathcal{S}_t$ at time t is $V^*(t, s_t)$. Note that unlike Section 3 in Chapter 3 where we denoted the Optimal Value Function as a time-indexed sequence $V_t^*(s_t)$, here we make t an explicit functional argument of V^*. This is because in the continuous-time setting, we are interested in the time-differential of the Optimal Value Function.

Now let us write the Bellman Optimality Equation in it's continuous-time version, i.e, let us consider the process V^* over the time interval $[t, t+dt]$ as follows:

$$V^*(t, s_t) = \max_{a_t \in \mathcal{A}_t} \{\mathcal{R}(t, s_t, a_t) \cdot dt + \mathbb{E}_{(t, s_t, a_t)}[e^{-\rho \cdot dt} \cdot V^*(t+dt, s_{t+dt})]\}$$

Multiplying throughout by $e^{-\rho t}$ and re-arranging, we get:

$$\max_{a_t \in \mathcal{A}_t} \{e^{-\rho t} \cdot \mathcal{R}(t, s_t, a_t) \cdot dt + \mathbb{E}_{(t,s_t,a_t)}[e^{-\rho(t+dt)} \cdot V^*(t+dt, s_{t+dt}) - e^{-\rho t} \cdot V^*(t, s_t)]\} = 0$$

$$\Rightarrow \max_{a_t \in \mathcal{A}_t} \{e^{-\rho t} \cdot \mathcal{R}(t, s_t, a_t) \cdot dt + \mathbb{E}_{(t,s_t,a_t)}[d\{e^{-\rho t} \cdot V^*(t, s_t)\}]\} = 0$$

$$\Rightarrow \max_{a_t \in \mathcal{A}_t} \{e^{-\rho t} \cdot \mathcal{R}(t, s_t, a_t) \cdot dt + \mathbb{E}_{(t,s_t,a_t)}[e^{-\rho t} \cdot (dV^*(t, s_t) - \rho \cdot V^*(t, s_t) \cdot dt)]\} = 0$$

Multiplying throughout by $e^{\rho t}$ and re-arranging, we get:

$$\rho \cdot V^*(t, s_t) \cdot dt = \max_{a_t \in \mathcal{A}_t} \{\mathbb{E}_{(t,s_t,a_t)}[dV^*(t, s_t)] + \mathcal{R}(t, s_t, a_t) \cdot dt\} \tag{E.1}$$

For a finite-horizon problem terminating at time T, the above equation is subject to terminal condition:

$$V^*(T, s_T) = \mathcal{T}(s_T)$$

for some terminal reward function $\mathcal{T}(\cdot)$.

Equation (E.1) is known as the Hamilton-Jacobi-Bellman Equation - the continuous-time analog of the Bellman Optimality Equation. In the literature, it is often written in a more compact form that essentially takes the above form and "divides throughout by dt". This requires a few technical details involving the stochastic differentiation operator. To keep things simple, we shall stick to the HJB formulation of Equation (E.1).

HJB with State Transitions as an Ito Process

Although we have expressed the HJB Equation for V^*, we cannot do anything useful with it unless we know the state transition probabilities (all of which are buried inside the calculation of $\mathbb{E}_{(t,s_t,a_t)}[\cdot]$ in the HJB Equation). In continuous-time, the state transition probabilities are modeled as a stochastic process for states (or of it's features). Let us assume that states are real-valued vectors, i.e, state $\boldsymbol{s}_t \in \mathbb{R}^n$ at any time $t \geq 0$ and that the transitions for $\boldsymbol{s}$ are given by an Ito process, as follows:

$$d\boldsymbol{s}_t = \boldsymbol{\mu}(t, \boldsymbol{s}_t, a_t) \cdot dt + \boldsymbol{\sigma}(t, \boldsymbol{s}_t, a_t) \cdot d\boldsymbol{z}_t$$

where the function $\boldsymbol{\mu}$ (drift function) gives an $\mathbb{R}^n$ valued process, the function $\boldsymbol{\sigma}$ (dispersion function) gives an $\mathbb{R}^{n \times m}$-valued process and $\boldsymbol{z}$ is an m-dimensional process consisting of m independent standard brownian motions.

Now we can apply multivariate Ito's Lemma (Equation (D.2) from Appendix D) for V^* as a function of t and s_t (we lighten notation by writing $\boldsymbol{\mu}_t$ and $\boldsymbol{\sigma}_t$ instead of $\boldsymbol{\mu}(t, \boldsymbol{s}_t, a_t)$ and $\boldsymbol{\sigma}(t, \boldsymbol{s}_t, a_t)$):

$$dV^*(t, \boldsymbol{s}_t) = (\frac{\partial V^*}{\partial t} + (\nabla_{\boldsymbol{s}} V^*)^T \cdot \boldsymbol{\mu}_t + \frac{1}{2} Tr[\boldsymbol{\sigma}_t^T \cdot (\Delta_{\boldsymbol{s}} V^*) \cdot \boldsymbol{\sigma}_t]) \cdot dt + (\nabla_{\boldsymbol{s}} V^*)^T \cdot \boldsymbol{\sigma}_t \cdot d\boldsymbol{z}_t$$

Substituting this expression for $dV^*(t, \boldsymbol{s}_t)$ in Equation (E.1), noting that

$$\mathbb{E}_{(t,\boldsymbol{s}_t,a_t)}[(\nabla_{\boldsymbol{s}} V^*)^T \cdot \boldsymbol{\sigma}_t \cdot d\boldsymbol{z}_t] = 0$$

and dividing throughout by dt, we get:

$$\rho \cdot V^*(t, \boldsymbol{s}_t) = \max_{a_t \in \mathcal{A}_t} \{ \frac{\partial V^*}{\partial t} + (\nabla_{\boldsymbol{s}} V^*)^T \cdot \boldsymbol{\mu}_t + \frac{1}{2} Tr[\boldsymbol{\sigma}_t^T \cdot (\Delta_{\boldsymbol{s}} V^*) \cdot \boldsymbol{\sigma}_t] + \mathcal{R}(t, \boldsymbol{s}_t, a_t) \} \quad \text{(E.2)}$$

For a finite-horizon problem terminating at time T, the above equation is subject to terminal condition:

$$V^*(T, \boldsymbol{s}_T) = \mathcal{T}(\boldsymbol{s}_T)$$

for some terminal reward function $\mathcal{T}(\cdot)$.

F. Black-Scholes Equation and it's Solution for Call/Put Options

In this Appendix, we sketch the derivation of the much-celebrated Black-Scholes equation and it's solution for Call and Put Options. As is the norm in the Appendices in this book, we will compromise on some of the rigor and emphasize the intuition to develop basic familiarity with concepts in continuous-time derivatives pricing and hedging.

Assumptions

The Black-Scholes Model is about pricing and hedging of a derivative on a single underlying asset (henceforth, simply known as "underlying"). The model makes several simplifying assumptions for analytical convenience. Here are the assumptions:

- The underlying (whose price we denote as S_t as time t) follows a special case of the lognormal process we covered in D of Appendix D, where the drift $\mu(t)$ is a constant (call it $\mu \in \mathbb{R}$) and the dispersion $\sigma(t)$ is also a constant (call it $\sigma \in \mathbb{R}^+$):

$$dS_t = \mu \cdot S_t \cdot dt + \sigma \cdot S_t \cdot dz_t \tag{F.1}$$

  This process is often refered to as *Geometric Brownian Motion* to reflect the fact that the stochastic increment of the process ($\sigma \cdot S_t \cdot dz_t$) is multiplicative to the level of the process S_t.
- The derivative has a known payoff at time $t = T$, as a function $f : \mathbb{R}^+ \rightarrow \mathbb{R}$ of the underlying price S_T at time T.
- Apart from the underlying, the market also includes a riskless asset (which should be thought of as lending/borrowing money at a constant infinitesimal rate of annual return equal to r). The riskless asset (denote it's price as R_t at time t) movements can thus be described as:

$$dR_t = r \cdot R_t \cdot dt$$

- Assume that we can trade in any real-number quantity in the underlying as well as in the riskless asset, in continuous-time, without any transaction costs (i.e., the typical "frictionless" market assumption).

Derivation of the Black-Scholes Equation

We denote the price of the derivative at any time t for any price S_t of the underlying as $V(t, S_t)$. Thus, $V(T, S_T)$ is equal to the payoff $f(S_T)$. Applying Ito's Lemma on $V(t, S_t)$ (see Equation (D.1) in Appendix **??**), we get:

$$dV(t, S_t) = (\frac{\partial V}{\partial t} + \mu \cdot S_t \cdot \frac{\partial V}{\partial S_t} + \frac{\sigma^2}{2} \cdot S_t^2 \cdot \frac{\partial^2 V}{\partial S_t^2}) \cdot dt + \sigma \cdot S_t \cdot \frac{\partial V}{\partial S_t} \cdot dz_t \tag{F.2}$$

Now here comes the key idea: create a portfolio comprising of the derivative and the underlying so as to eliminate the incremental uncertainty arising from the brownian motion increment dz_t. It's clear from the coefficients of dz_t in Equation (F.1) and (F.2) that this can be accomplished with a portfolio comprising of $\frac{\partial V}{\partial S_t}$ units of the underlying and -1 units of the derivative (i.e., by selling a derivative contract written on a single unit of the underlying). Let us refer to the value of this portfolio as Π_t at time t. Thus,

$$\Pi_t = -V(t, S_t) + \frac{\partial V}{\partial S_t} \cdot S_t \tag{F.3}$$

Over an infinitesimal time-period $[t, t + dt]$, the change in the portfolio value Π_t is given by:

$$d\Pi_t = -dV(t, S_t) + \frac{\partial V}{\partial S_t} \cdot dS_t$$

Substituting for dS_t and $dV(t, S_t)$ from Equations (F.1) and (F.2), we get:

$$d\Pi_t = (-\frac{\partial V}{\partial t} - \frac{\sigma^2}{2} \cdot S_t^2 \cdot \frac{\partial^2 V}{\partial S_t^2}) \cdot dt \tag{F.4}$$

Thus, we have eliminated the incremental uncertainty arising from dz_t and hence, this is a riskless portfolio. To ensure the market remains free of arbitrage, the infinitesimal rate of annual return for this riskless portfolio must be the same as that for the riskless asset, i.e., must be equal to r. Therefore,

$$d\Pi_t = r \cdot \Pi_t \cdot dt \tag{F.5}$$

From Equations (F.4) and (F.5), we infer that:

$$-\frac{\partial V}{\partial t} - \frac{\sigma^2}{2} \cdot S_t^2 \cdot \frac{\partial^2 V}{\partial S_t^2} = r \cdot \Pi_t$$

Substituting for Π_t from Equation (F.3), we get:

$$-\frac{\partial V}{\partial t} - \frac{\sigma^2}{2} \cdot S_t^2 \cdot \frac{\partial^2 V}{\partial S_t^2} = r \cdot (-V(t, S_t) + \frac{\partial V}{\partial S_t} \cdot S_t)$$

Re-arranging, we arrive at the famous Black-Scholes equation:

$$\frac{\partial V}{\partial t} + \frac{\sigma^2}{2} \cdot S_t^2 \cdot \frac{\partial^2 V}{\partial S_t^2} + r \cdot S_t \cdot \frac{\partial V}{\partial S_t} + r \cdot V(t, S_t) = 0 \tag{F.6}$$

A few key points to note here:

1. The Black-Scholes equation is a partial differential equation (PDE) in t and S_t, and it is valid for any derivative with arbitary payoff $f(S_T)$ at a fixed time $t = T$, and the derivative price function $V(t, S_t)$ needs to be twice differentiable with respect to S_t and once differentiable with respect to t.
2. The infinitesimal change in the portfolio value ($= d\Pi_t$) incorporates only the infinitesimal changes in the prices of the underlying and the derivative, and not the changes in the units held in the underlying and the derivative (meaning the portfolio is assumed to be self-financing). The portfolio composition does change continuously though since the units held in the underlying at time t needs to be $\frac{\partial V}{\partial S_t}$, which in general would change as time evolves and as the price S_t of the underlying changes. Note that $-\frac{\partial V}{\partial S_t}$ represents the hedge units in the underlying at any time t for any underlying price S_t, which nullifies the risk of changes to the derivative price $V(t, S_t)$.
3. The drift μ of the underlying price movement (interpreted as expected annual rate of return of the underlying) does not appear in the Black-Scholes Equation and hence, the price of any derivative will be independent of the expected rate of return of the underlying. Note though the prominent appearance of σ (refered to as the underlying volatility) and the riskless rate of return r in the Black-Scholes equation.

Solution of the Black-Scholes Equation for Call/Put Options

The Black–Scholes PDE can be solved numerically using standard methods such as finite-differences. It turns out we can solve this PDE as an exact formula (closed-form solution) for the case of European call and put options, whose payoff functions are $\max(S_T - K, 0)$ and $\max(K - S_T, 0)$ respectively, where K is the option strike. We shall denote the call and put option prices at time t for underlying price of S_t as $C(t, S_t)$ and $P(t, S_t)$ respectively (as specializations of $V(t, S_t)$). We derive the solution below for call option pricing, with put option pricing derived similarly. Note that we could simply use the put-call parity: $C(t, S_t) - P(t, S_t) = S_t - K$ to obtain the put option price from the call option price. The put-call parity holds because buying a call option and selling a put option is a combined payoff of $S_T - K$ - this means owning the underlying and borrowing $K \cdot e^{-rT}$, which at any time t would be valued at $S_t - K \cdot e^{-r \cdot (T-t)}$.

To derive the formula for $C(t, S_t)$, we perform the following change-of-variables transformation:

$$\tau = T - t$$

$$x = \log\left(\frac{S_t}{K}\right) + (r - \frac{\sigma^2}{2}) \cdot \tau$$

$$u(\tau, x) = C(t, S_t) \cdot e^{r\tau}$$

This reduces the Black-Scholes PDE into the *Heat Equation*:

$$\frac{\partial u}{\partial \tau} = \frac{\sigma^2}{2} \cdot \frac{\partial^2 u}{\partial x^2}$$

The terminal condition $C(T, S_T) = \max(S_T - K, 0)$ transforms into the Heat Equation's initial condition:

$$u(0, x) = K \cdot (e^{\max(x,0)} - 1)$$

Using the standard convolution method for solving this Heat Equation with initial condition $u(0, x)$, we obtain the Green's Function Solution:

$$u(\tau, x) = \frac{1}{\sigma\sqrt{2\pi\tau}} \cdot \int_{-\infty}^{+\infty} u(0, y) \cdot e^{-\frac{(x-y)^2}{2\sigma^2\tau}} \cdot dy$$

With some manipulations, this yields:

$$u(\tau, x) = K \cdot e^{x + \frac{\sigma^2\tau}{2}} \cdot N(d_1) - K \cdot N(d_2)$$

where $N(\cdot)$ is the standard normal cumulative distribution function:

$$N(z) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{z} e^{-\frac{y^2}{2}} \cdot dy$$

and d_1, d_2 are the quantities:

$$d_1 = \frac{x + \sigma^2\tau}{\sigma\sqrt{\tau}}$$

$$d_2 = d_1 - \sigma\sqrt{\tau}$$

Substituting for $\tau, x, u(\tau, x)$ with $t, S_t, C(t, S_t)$, we get:

$$C(t, S_t) = S \cdot N(d_1) - K \cdot e^{-r\cdot(T-t)} \cdot N(d_2) \tag{F.7}$$

where

$$d_1 = \frac{\log(\frac{S_t}{K}) + (r + \frac{\sigma^2}{2}) \cdot (T - t)}{\sigma \cdot \sqrt{T - t}}$$

$$d_2 = d_1 - \sigma\sqrt{T - t}$$

The put option price is:

$$P(t, S_t) = K \cdot e^{-r\cdot(T-t)} \cdot N(-d_2) - S_t \cdot N(-d_1) \tag{F.8}$$