Optimal Control and Reinforcement Learning

Heng Yang

2025-11-25

Contents

Preface

This is the textbook for Harvard ES/AM 158: Introduction to Optimal Control and Reinforcement Learning.

Feedback

I would like to invite you to provide feedback to the textbook via inline comments with Hypothesis:

- Go to Hypothesis and create an account
- Install the Chrome extension of Hypothesis
- Provide public comments to textbook contents and I will try to address them

Offerings

2025 Fall

Time: Mon/Wed 2:15 - 3:30pm

Location: SEC 1.413

Instructor: Heng Yang

Teaching Fellow: Haoyu Han, Han Qi

[Syllabus], [Problem Sets], [Canvas]

2023 Fall

The course was previously offered as Introduction to Optimal Control and Estimation.

Starting Fall 2025, contents about reinforcement learning have been added to the course.

Chapter 1

Markov Decision Process

Optimal control (OC) and reinforcement learning (RL) address the problem of making **optimal decisions** in the presence of a **dynamic environment**.

- In **optimal control**, this dynamic environment is often referred to as a *plant* or a *dynamical system*.
- In **reinforcement learning**, it is modeled as a *Markov decision process* (MDP).

The goal in both fields is to evaluate and design decision-making strategies that optimize long-term performance:

- **RL** typically frames this as maximizing a long-term *reward*.
- **OC** often formulates it as minimizing a long-term *cost*.

The emphasis on **long-term** evaluation is crucial. Because the environment evolves over time, decisions that appear beneficial in the short term may lead to poor long-term outcomes and thus be suboptimal.

With this motivation, we now formalize the framework of Markov Decision Processes (MDPs), which are discrete-time stochastic dynamical systems.

1.1 Finite-Horizon MDP

We begin with finite-horizon MDPs and introduce infinite-horizon MDPs in the following section. An abstract definition of the finite-horizon case will be presented first, followed by illustrative examples.

A finite-horizon MDP is given by the following tuple:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, T),$$

where

- $\mathcal{S}$: state space (set of all possible states)
- $\mathcal{A}$: action space (set of all possible actions)
- $P(s' \mid s, a)$: probability of transitioning to state s' from state s under action a (i.e., dynamics)
- $R(s, a)$: reward of taking action a in state s
- T: horizon, a positive integer

For now, let us assume both the state space and the action space are discrete and have a finite number of elements. In particular, denote the number of elements in $\mathcal{S}$ as $|\mathcal{S}|$, and the number of elements in $\mathcal{A}$ as $|\mathcal{A}|$. This is also referred to as a *tabular MDP*.

Policy. Decision-making in MDPs is represented by policies. A policy is a function that, given any state, outputs a distribution of actions: $\pi : \mathcal{S} \mapsto \Delta(\mathcal{A})$. That is, $\pi(a \mid s)$ returns the probability of taking action a in state s. In finite-horizon MDPs, we consider a tuple of policies:

$$\pi = (\pi_0, \dots, \pi_t, \dots, \pi_{T-1}), \tag{1.1}$$

where each π_t denotes the policy at step $t \in [0, T-1]$.

Trajectory and Return. Given an initial state $s_0 \in \mathcal{S}$ and a policy π, the MDP will evolve as

1. Start at state s_0
2. Take action $a_0 \sim \pi_0(a \mid s_0)$ following policy π_0
3. Collect reward $r_0 = R(s_0, a_0)$ (assume R is deterministic)
4. Transition to state $s_1 \sim P(s' \mid s_0, a_0)$ following the dynamics
5. Go to step 2 and continue until reaching state s_T

This evolution generates a trajectory of states, actions, and rewards:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T).$$

The cumulative reward of this trajectory is $g_0 = \sum_{t=0}^{T-1} r_t$, which is called the *return* of the trajectory. Clearly, g_0 is a random variable due to the stochasticity

of both the policy and the dynamics. Similarly, if the state at time t is s_t, we denote:

$$g_t = r_t + \cdots + r_{T-1}$$

as the return of the policy starting at s_t.

1.1.1 Value Functions

State-Value Function. Given a policy π as in (1.1), which states are preferable at time t? The (time-indexed) state-value function assigns to each $s \in \mathcal{S}$ the expected return from t onward when starting in s and following π thereafter. Formally, define

$$V_t^\pi(s) := \mathbb{E}\left[g_t \mid s_t = s\right] = \mathbb{E}\left[\sum_{i=t}^{T-1} R(s_i, a_i) \middle| s_t = s, a_i \sim \pi_i(\cdot \mid s_i), s_{i+1} \sim P(\cdot \mid s_i, a_i)\right]. \tag{1.2}$$

The expectation is over the randomness induced by both the policy and the dynamics. Thus, if $V_t^\pi(s_1) > V_t^\pi(s_2)$, then at time t under policy π it is better in expectation to be in s_1 than in s_2 because the former yields a larger expected return.

$V_t^\pi(s)$: given policy π, how good is it to start in state s at time t?

Action-Value Function. Similarly, the action-value function assigns to each state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ the expected return obtained by starting in state s, taking action a first, and then following policy π thereafter:

$$\begin{aligned} Q_t^\pi(s, a) :=& \mathbb{E}\left[R(s, a) + g_{t+1} \mid s_{t+1} \sim P(\cdot \mid s, a)\right] \\ =& \mathbb{E}\left[R(s, a) + \sum_{i=t+1}^{T-1} R(s_i, a_i) \middle| s_{t+1} \sim P(\cdot \mid s, a)\right]. \end{aligned} \tag{1.3}$$

The key distinction is that the action-value function evaluates the return when the first action may deviate from policy π, whereas the state-value function assumes strict adherence to π. This flexibility makes the action-value function central to improving π, since it reveals whether alternative actions can yield higher returns.

$Q_t^\pi(s, a)$: At time t, how good is it to take action a in state s, then follow the policy π?

It is easy to verify that the state-value function and the action-value function satisfy:

$$V_t^\pi(s) = \sum_{a \in \mathcal{A}} \pi_t(a \mid s) Q_t^\pi(s, a), \tag{1.4}$$

$$Q_t^\pi(s, a) = R(s, a) + \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V_{t+1}^\pi(s'). \tag{1.5}$$

From these two equations, we can derive the Bellman Consistency equations.

Proposition 1.1 (Bellman Consistency (Finite Horizon))**.** *The state-value function $V_t^\pi(\cdot)$ in* (1.2) *satisfies the following recursion:*

$$\begin{aligned} V_t^\pi(s) &= \sum_{a\in\mathcal{A}} \pi_t(a \mid s) \left(R(s,a) + \sum_{s'\in\mathcal{S}} P(s' \mid s,a) V_{t+1}^\pi(s') \right) \\ &=: \mathbb{E}_{a\sim\pi_t(\cdot|s)} \left[R(s,a) + \mathbb{E}_{s'\sim P(\cdot|s,a)}[V_{t+1}^\pi(s')] \right]. \end{aligned} \tag{1.6}$$

Similarly, the action-value function $Q_t^\pi(s,a)$ in (1.3) *satisfies the following recursion:*

$$\begin{aligned} Q_t^\pi(s,a) &= R(s,a) + \sum_{s'\in\mathcal{S}} P(s' \mid s,a) \left(\sum_{a'\in\mathcal{A}} \pi_{t+1}(a' \mid s') Q_{t+1}^\pi(s',a') \right) \\ &=: R(s,a) + \mathbb{E}_{s'\sim P(\cdot|s,a)} \left[\mathbb{E}_{a'\sim\pi_{t+1}(\cdot|s')}[Q_{t+1}^\pi(s',a')] \right]. \end{aligned} \tag{1.7}$$

1.1.2 Policy Evaluation

The Bellman consistency result in Proposition 1.1 is fundamental because it directly yields an algorithm for evaluating a given policy π—that is, for computing its state-value and action-value functions—provided the transition dynamics of the MDP are known.

Policy evaluation for the state-value function proceeds as follows:

- **Initialization:** set $V_T^\pi(s) = 0$ for all $s \in \mathcal{S}$.
- **Backward recursion:** for $t = T-1, T-2, \ldots, 0$, update each $s \in \mathcal{S}$ by

$$V_t^\pi(s) = \mathbb{E}_{a\sim\pi_t(\cdot|s)} \left[R(s,a) + \mathbb{E}_{s'\sim P(\cdot|s,a)}[V_{t+1}^\pi(s')] \right].$$

Similarly, policy evaluation for the action-value function is given by:

- **Initialization:** set $Q_T^\pi(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$.
- **Backward recursion:** for $t = T-1, T-2, \ldots, 0$, update each $(s,a) \in \mathcal{S}\times\mathcal{A}$ by

$$Q_t^\pi(s,a) = R(s,a) + \mathbb{E}_{s'\sim P(\cdot|s,a)} \left[\mathbb{E}_{a'\sim\pi_{t+1}(\cdot|s')}[Q_{t+1}^\pi(s',a')] \right].$$

The essential feature of this algorithm is its backward-in-time recursion: the value functions are first set at the terminal horizon T, and then propagated backward step by step through the Bellman consistency equations.

Example 1.1 (MDP, Transition Graph, and Policy Evaluation)**.** It is often useful to visualize small MDPs as transition graphs, where states are represented by nodes and actions are represented by directed edges connecting those nodes.

As a simple illustrative example, consider a robot navigating on a two-state grid. At each step, the robot can either Stay in its current state or Move to the other state. This finite-horizon MDP is fully specified by the tuple of states, actions, transition dynamics, rewards, and horizon:

- States: $\mathcal{S} = \{\alpha, \beta\}$
- Actions: $\mathcal{A} = \{\text{Move}, \text{Stay}\}$
- Transition dynamics: we can specify the transition dynamics in the following table

State s	Action a	Next State s'	Probability $P(s' \mid s, a)$
α	Stay	α	1
α	Move	β	1
β	Stay	β	1
β	Move	α	1

- Reward: $R(s, a) = 1$ if $a = \text{Move}$ and $R(s, a) = 0$ if $a = \text{Stay}$
- Horizon: $T = 2$.

This MDP can be represented by the transition graph in Fig. 1.1. Note that for this MDP, the transition dynamics is deterministic. We will see a stochastic MDP soon.

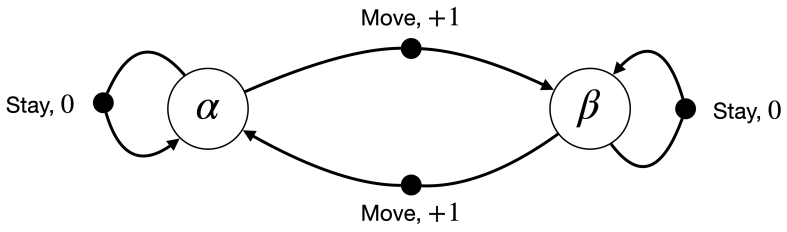

Figure 1.1: A Simple Transition Graph.

At time $t = 0$, if the robot starts at $s_0 = \alpha$, first chooses action $a_0 = \text{Move}$, and then chooses action $a_1 = \text{Stay}$, the resulting trajectory is

$$\tau = (\alpha, \text{Move}, +1, \beta, \text{Stay}, 0, \beta).$$

The return of this trajectory is:

$$g_0 = +1 + 0 = +1.$$

Policy Evaluation. Given a policy

$$\pi = (\pi_0, \pi_1), \quad \pi_0(a \mid s) = \begin{cases} 0.5 & a = \text{Move} \\ 0.5 & a = \text{Stay} \end{cases}, \quad \pi_1(a \mid s) = \begin{cases} 0.8 & a = \text{Move} \\ 0.2 & a = \text{Stay} \end{cases}. \tag{1.8}$$

We can use the Bellman consistency equations to compute the state-value function. We first initialize:

$$V_2^\pi = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

where the first row contains the value at $s = \alpha$ and the second row contains the value at $s = \beta$. We then perform the backward recursion for $t = 1$. For $s = \alpha$, we have

$$V_1^\pi(\alpha) = \begin{bmatrix} \pi_1(\text{Move} \mid \alpha) \\ \pi_1(\text{Stay} \mid \alpha) \end{bmatrix}^\top \begin{bmatrix} R(\alpha, \text{Move}) + V_2^\pi(\beta) \\ R(\alpha, \text{Stay}) + V_2^\pi(\alpha) \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}^\top \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0.8 \tag{1.9}$$

For $s = \beta$, we have

$$V_1^\pi(\beta) = \begin{bmatrix} \pi_1(\text{Move} \mid \beta) \\ \pi_1(\text{Stay} \mid \beta) \end{bmatrix}^\top \begin{bmatrix} R(\beta, \text{Move}) + V_2^\pi(\alpha) \\ R(\beta, \text{Stay}) + V_2^\pi(\beta) \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}^\top \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0.8. \tag{1.10}$$

Therefore, we have

$$V_1^\pi = \begin{bmatrix} 0.8 \\ 0.8 \end{bmatrix}.$$

We then proceed to the backward recursion for $t = 0$:

$$V_0^\pi(\alpha) = \begin{bmatrix} \pi_0(\text{Move} \mid \alpha) \\ \pi_0(\text{Stay} \mid \alpha) \end{bmatrix}^\top \begin{bmatrix} R(\alpha, \text{Move}) + V_1^\pi(\beta) \\ R(\alpha, \text{Stay}) + V_1^\pi(\alpha) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}^\top \begin{bmatrix} 1.8 \\ 0.8 \end{bmatrix} = 1.3. \tag{1.11}$$

$$V_0^\pi(\beta) = \begin{bmatrix} \pi_0(\text{Move} \mid \beta) \\ \pi_0(\text{Stay} \mid \beta) \end{bmatrix}^\top \begin{bmatrix} R(\beta, \text{Move}) + V_0^\pi(\alpha) \\ R(\beta, \text{Stay}) + V_0^\pi(\beta) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}^\top \begin{bmatrix} 1.8 \\ 0.8 \end{bmatrix} = 1.3. \tag{1.12}$$

Therefore, the state-value function at $t = 0$ is

$$V_0^\pi = \begin{bmatrix} 1.3 \\ 1.3 \end{bmatrix}.$$

You are encouraged to carry out the similar calculations for the action-value function.

The toy example was small enough to carry out policy evaluation by hand; in realistic MDPs, we will need the help from computers.

Consider now an MDP whose transition graph is shown in Fig. 1.2. This example is adapted from here.

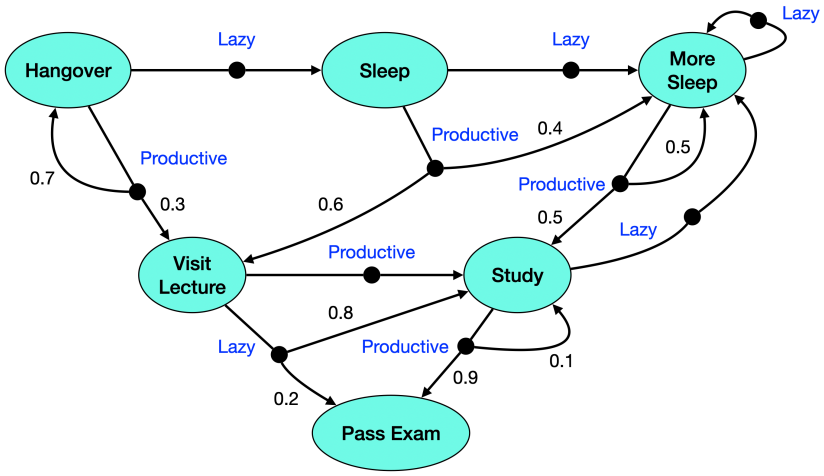

Figure 1.2: Hangover Transition Graph.

This MDP has six states:

$$\mathcal{S} = \{\text{Hangover}, \text{Sleep}, \text{More Sleep}, \text{Visit Lecture}, \text{Study}, \text{Pass Exam}\},$$

and two actions:

$$\mathcal{A} = \{\text{Lazy}, \text{Productive}\}.$$

The stochastic transition dynamics are labeled in the transition graph. For example, at state "Hangover", taking action "Productive" will lead to state "Visit Lecture" with probability 0.3 and state "Hangover" with probability 0.7. The rewards of the MDP are defined as:

$$R(s,a) = \begin{cases} +1 & s = \text{Pass Exam} \\ -1 & \text{otherwise.} \end{cases}.$$

Policy Evaluation. Consider a time-invariant random policy

$$\pi = \{\pi_0, \dots, \pi_{T-1}\}, \quad \pi_t(a \mid s) = \begin{cases} \alpha & a = \text{Lazy} \\ 1-\alpha & a = \text{Productive} \end{cases},$$

that takes "Lazy" with probability α and "Productive" with probability $1-\alpha$.

The following Python code performs policy evaluation for this MDP, with $T = 10$ and $\alpha = 0.4$.

```python
# Finite-horizon policy evaluation for the Hangover MDP

from collections import defaultdict
from typing import Dict, List, Tuple

State = str
Action = str

# --- MDP spec ---------------------------------------------------------------

S: List[State] = [
    "Hangover", "Sleep", "More Sleep", "Visit Lecture", "Study", "Pass Exam"
]
A: List[Action] = ["Lazy", "Productive"]

# P[s, a] -> list of (s_next, prob)
P: Dict[Tuple[State, Action], List[Tuple[State, float]]] = {
    # Hangover
    ("Hangover", "Lazy"):       [("Sleep", 1.0)],
    ("Hangover", "Productive"): [("Visit Lecture", 0.3), ("Hangover", 0.7)],

    # Sleep
    ("Sleep", "Lazy"):          [("More Sleep", 1.0)],
    ("Sleep", "Productive"):    [("Visit Lecture", 0.6), ("More Sleep", 0.4)],

    # More Sleep
    ("More Sleep", "Lazy"):       [("More Sleep", 1.0)],
    ("More Sleep", "Productive"): [("Study", 0.5), ("More Sleep", 0.5)],

    # Visit Lecture
    ("Visit Lecture", "Lazy"):       [("Study", 0.8), ("Pass Exam", 0.2)],
    ("Visit Lecture", "Productive"): [("Study", 1.0)],

    # Study
    ("Study", "Lazy"):         [("More Sleep", 1.0)],
    ("Study", "Productive"):   [("Pass Exam", 0.9), ("Study", 0.1)],

    # Pass Exam (absorbing)
    ("Pass Exam", "Lazy"):       [("Pass Exam", 1.0)],
    ("Pass Exam", "Productive"): [("Pass Exam", 1.0)],
}

def R(s: State, a: Action) -> float:
    """Reward: +1 in Pass Exam, -1 otherwise."""
```

```
    return 1.0 if s == "Pass Exam" else -1.0

# --- Policy: time-invariant, state-independent -----------------------------

def pi(a: Action, s: State, alpha: float) -> float:
    """pi(a|s): Lazy with prob alpha, Productive with prob 1-alpha."""
    return alpha if a == "Lazy" else (1.0 - alpha)

# --- Policy evaluation -----------------------------------------------------

def policy_evaluation(T: int, alpha: float):
    """
    Compute {V_t(s)} and {Q_t(s,a)} for t=0..T with terminal condition V_T = Q_T = 0.
    Returns:
        V: Dict[int, Dict[State, float]]
        Q: Dict[int, Dict[Tuple[State, Action], float]]
    """
    assert T >= 0
    # sanity: probabilities sum to 1 for each (s,a)
    for key, rows in P.items():
        total = sum(p for _, p in rows)
        if abs(total - 1.0) > 1e-9:
            raise ValueError(f"Probabilities for {key} sum to {total}, not 1.")

    V: Dict[int, Dict[State, float]] = defaultdict(dict)
    Q: Dict[int, Dict[Tuple[State, Action], float]] = defaultdict(dict)

    # Terminal boundary
    for s in S:
        V[T][s] = 0.0
        for a in A:
            Q[T][(s, a)] = 0.0

    # Backward recursion
    for t in range(T - 1, -1, -1):
        for s in S:
            # First compute Q_t(s,a)
            for a in A:
                exp_next = sum(p * V[t + 1][s_next] for s_next, p in P[(s, a)])
                Q[t][(s, a)] = R(s, a) + exp_next
            # Then V_t(s) = E_{a~pi}[Q_t(s,a)]
            V[t][s] = sum(pi(a, s, alpha) * Q[t][(s, a)] for a in A)

    return V, Q
```

```python
# --- Example run -----------------------------------------------------------

if __name__ == "__main__":
    T = 10         # horizon
    alpha = 0.4  # probability of choosing Lazy
    V, Q = policy_evaluation(T=T, alpha=alpha)

    # Print V_0
    print(f"V_0(s) with T={T}, alpha={alpha}:")
    for s in S:
        print(f"  {s:13s}: {V[0][s]: .3f}")
```

The code returns the following state values at $t = 0$:

$$V_0^\pi = \begin{bmatrix} -3.582 \\ -2.306 \\ -2.180 \\ 1.757 \\ 2.939 \\ 10 \end{bmatrix}, \tag{1.13}$$

where the ordering of the states follows that defined in $\mathcal{S}$.

You can find the code here.

1.1.3 Principle of Optimality

Every policy π induces a value function V_0^π that can be evaluated by policy evaluation (assuming the transition dynamics are known). The goal of reinforcement learning is to find an optimal policy that maximizes the value function with respect to a given initial state distribution:

$$V_0^\star = \max_\pi \ \mathbb{E}_{s_0 \sim \mu(\cdot)}[V_0^\pi(s_0)], \tag{1.14}$$

where we have used the superscript "$\star$" to denote the optimality of the value function. $V_0^\star$ is often known as the *optimal value function*.

At first glance, (1.14) appears daunting: a naive approach would enumerate all stochastic policies π, evaluate their value functions, and select the best. A central result in reinforcement learning and optimal control—rooted in the principle of optimality—is that the *optimal* value functions satisfy a Bellman-style recursion, analogous to Proposition 1.1. This Bellman optimality recursion enables backward computation of the optimal value functions without enumerating policies.

Theorem 1.1 (Bellman Optimality (Finite Horizon, State-Value))**.** *Consider a finite-horizon MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, T)$ with finite state and action sets and bounded rewards. Define the optimal value functions $\{V_t^\star\}_{t=0}^T$ by the following* Bellman optimality *recursion*

$$\begin{aligned} &V_T^\star(s) \equiv 0, \\ &V_t^\star(s) = \max_{a \in \mathcal{A}} \Big\{ R(s,a) \;+\; \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V_{t+1}^\star(s') \Big\},\ t = T-1, \dots, 0. \end{aligned} \tag{1.15}$$

Then, the optimal value functions are optimal in the sense of statewise dominance*:*

$$V_t^\star(s) \;\geq\; V_t^\pi(s) \quad \text{for all policies } \pi,\ s \in \mathcal{S},\ t = 0, \dots, T. \tag{1.16}$$

Moreover, the deterministic policy $\pi^\star = (\pi_0^\star, \dots, \pi_{T-1}^\star)$ with

$$\begin{aligned} \pi_t^\star(s) \;\in\; \arg\max_{a \in \mathcal{A}} \Big\{ R(s,a) \;+\; \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V_{t+1}^\star(s') \Big\}, \\ \text{for any } s \in \mathcal{S},\ t = 0, \dots, T-1 \end{aligned} \tag{1.17}$$

is optimal, where ties can be broken by any fixed rule.

Proof. We first show that the value functions defined by the Bellman optimality recursion (1.15) are *optimal* in the sense that they dominate the value functions of any other policy. The proof proceeds by backward induction.

Base case ($t = T$). For every $s \in \mathcal{S}$,

$$V_T^\star(s) \;=\; 0 \;=\; V_T^\pi(s),$$

so $V_T^\star(s) \geq V_T^\pi(s)$ holds trivially.

Inductive step. Assume $V_{t+1}^\star(s) \geq V_{t+1}^\pi(s)$ for all $s \in \mathcal{S}$. Then, for any $s \in \mathcal{S}$,

$$\begin{aligned} V_t^\pi(s) &= \sum_{a \in \mathcal{A}} \pi_t(a \mid s) \left(R(s,a) + \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V_{t+1}^\pi(s') \right) \\ &\leq \sum_{a \in \mathcal{A}} \pi_t(a \mid s) \left(R(s,a) + \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V_{t+1}^\star(s') \right) \\ &\leq \max_{a \in \mathcal{A}} \left(R(s,a) + \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V_{t+1}^\star(s') \right) \;=\; V_t^\star(s), \end{aligned}$$

where the first inequality uses the induction hypothesis and the second uses that an expectation is bounded above by a maximum. Hence $V_t^\star(s) \geq V_t^\pi(s)$ for all s, completing the induction. Therefore, $\{V_t^\star\}_{t=0}^T$ dominates the value functions attainable by any policy.

Next, we show that $\{V_t^\star\}$ is *attainable* by some policy. Since $\mathcal{A}$ is finite (tabular setting), the maximizer in the Bellman optimality operator exists for every (t, s); thus we can define a (deterministic) greedy policy

$$\pi_t^\star(s) \;\in\; \arg\max_{a\in\mathcal{A}} \Big\{ R(s,a) + \sum_{s'\in\mathcal{S}} P(s' \mid s,a)\, V_{t+1}^\star(s') \Big\}.$$

A simple backward induction then shows $V_t^{\pi^\star}(s) = V_t^\star(s)$ for all t and s: at $t = T$ both are 0, and if $V_{t+1}^{\pi^\star} = V_{t+1}^\star$, then by construction of $\pi_t^\star$ the Bellman equality yields $V_t^{\pi^\star} = V_t^\star$. Consequently, the optimal value functions are achieved by the greedy (deterministic) policy $\pi^\star$. □

Corollary 1.1 (Bellman Optimality (Finite Horizon, Action-Value))**.** *Given the optimal (state-)value functions $V_t^\star, t = 0, \dots, T$, define the optimal action-value function*

$$Q_t^\star(s,a) \;=\; R(s,a) + \sum_{s'\in\mathcal{S}} P(s' \mid s,a)\, V_{t+1}^\star(s'), \quad t = 0, \dots, T-1. \tag{1.18}$$

Then we have

$$V_t^\star(s) = \max_{a\in\mathcal{A}} Q_t^\star(s,a), \qquad \pi_t^\star(s) \in \arg\max_{a\in\mathcal{A}} Q_t^\star(s,a). \tag{1.19}$$

The optimal action-value functions satisfy:

$$\begin{aligned} &Q_T^\star(s,a) \equiv 0, \\ &Q_t^\star(s,a) = R(s,a) \;+\; \mathbb{E}_{s'\sim P(\cdot|s,a)}\Big[\max_{a'\in\mathcal{A}} Q_{t+1}^\star(s',a')\Big], \quad t = T-1, \dots, 0. \end{aligned} \tag{1.20}$$

1.1.4 Dynamic Programming

The principle of optimality in Theorem 1.1 yields a constructive procedure to compute the optimal value functions and an associated deterministic optimal policy. This backward-induction procedure is the *dynamic programming* (DP) algorithm.

Dynamic programming (finite horizon).

- **Initialization.** Set $V_T^\star(s) = 0$ for all $s \in \mathcal{S}$.
- **Backward recursion.** For $t = T-1, T-2, \dots, 0$:
  - *Optimal value:* for each $s \in \mathcal{S}$,

$$V_t^\star(s) = \max_{a\in\mathcal{A}} \Big\{ R(s,a) + \mathbb{E}_{s'\sim P(\cdot|s,a)}\big[V_{t+1}^\star(s')\big] \Big\}.$$

– *Greedy policy (deterministic):* for each $s \in \mathcal{S}$,

$$\pi_t^\star(s) \in \arg\max_{a \in \mathcal{A}} \left\{ R(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)}[V_{t+1}^\star(s')] \right\}.$$

Exercise 1.1. How does dynamic programming look like when applied to the action-value function?

Exercise 1.2. What is the computational complexity of dynamic programming?

Let us try dynamic programming for the Hangover MDP presented before.

Example 1.2 (Dynamic Programming for Hangover MDP)**.** Consider the Hangover MDP defined by the transition graph shown in Fig. 1.2. With slight modification to the policy evaluation code, we can find the optimal value functions and optimal policies.

```
# Dynamic programming (finite-horizon optimal control) for the Hangover MDP

from collections import defaultdict
from typing import Dict, List, Tuple

State = str
Action = str

# --- MDP spec ---------------------------------------------------------------

S: List[State] = [
    "Hangover", "Sleep", "More Sleep", "Visit Lecture", "Study", "Pass Exam"
]
A: List[Action] = ["Lazy", "Productive"]

# P[s, a] -> list of (s_next, prob)
P: Dict[Tuple[State, Action], List[Tuple[State, float]]] = {
    # Hangover
    ("Hangover", "Lazy"):       [("Sleep", 1.0)],
    ("Hangover", "Productive"): [("Visit Lecture", 0.3), ("Hangover", 0.7)],

    # Sleep
    ("Sleep", "Lazy"):          [("More Sleep", 1.0)],
    ("Sleep", "Productive"):    [("Visit Lecture", 0.6), ("More Sleep", 0.4)],

    # More Sleep
    ("More Sleep", "Lazy"):       [("More Sleep", 1.0)],
    ("More Sleep", "Productive"): [("Study", 0.5), ("More Sleep", 0.5)],
```

```
    # Visit Lecture
    ("Visit Lecture", "Lazy"):       [("Study", 0.8), ("Pass Exam", 0.2)],
    ("Visit Lecture", "Productive"): [("Study", 1.0)],

    # Study
    ("Study", "Lazy"):         [("More Sleep", 1.0)],
    ("Study", "Productive"):   [("Pass Exam", 0.9), ("Study", 0.1)],

    # Pass Exam (absorbing)
    ("Pass Exam", "Lazy"):       [("Pass Exam", 1.0)],
    ("Pass Exam", "Productive"): [("Pass Exam", 1.0)],
}

def R(s: State, a: Action) -> float:
    """Reward: +1 in Pass Exam, -1 otherwise."""
    return 1.0 if s == "Pass Exam" else -1.0

# --- Dynamic programming (Bellman optimality) -------------------------------

def dynamic_programming(T: int):
    """
    Compute optimal finite-horizon tables:
      - V[t][s] = V_t^*(s)
      - Q[t][(s,a)] = Q_t^*(s,a)
      - PI[t][s] = optimal action at (t,s)
    with terminal condition V_T^* = 0.
    """
    assert T >= 0

    # sanity: probabilities sum to 1 for each (s,a)
    for key, rows in P.items():
        total = sum(p for _, p in rows)
        if abs(total - 1.0) > 1e-9:
            raise ValueError(f"Probabilities for {key} sum to {total}, not 1.")

    V: Dict[int, Dict[State, float]] = defaultdict(dict)
    Q: Dict[int, Dict[Tuple[State, Action], float]] = defaultdict(dict)
    PI: Dict[int, Dict[State, Action]] = defaultdict(dict)

    # Terminal boundary
    for s in S:
        V[T][s] = 0.0
        for a in A:
            Q[T][(s, a)] = 0.0
```

```
    # Backward recursion (Bellman optimality)
    for t in range(T - 1, -1, -1):
        for s in S:
            # compute Q*_t(s,a)
            for a in A:
                exp_next = sum(p * V[t + 1][s_next] for s_next, p in P[(s, a)])
                Q[t][(s, a)] = R(s, a) + exp_next

            # greedy action and optimal value
            # tie-breaking is deterministic by the order in A
            best_a = max(A, key=lambda a: Q[t][(s, a)])
            PI[t][s] = best_a
            V[t][s] = Q[t][(s, best_a)]

    return V, Q, PI

# --- Example run -------------------------------------------------------------

if __name__ == "__main__":
    T = 10  # horizon
    V, Q, PI = dynamic_programming(T=T)

    print(f"Optimal V_0(s) with T={T}:")
    for s in S:
        print(f"  {s:13s}: {V[0][s]: .3f}")

    print("\nGreedy policy at t=0:")
    for s in S:
        print(f"  {s:13s}: {PI[0][s]}")

    print("\nAction value at t=0:")
    for s in S:
        print(f"  {s:13s}: {Q[0][s, A[0]]: .3f}, {Q[0][s, A[1]]: .3f}")
```

The optimal value function at $t = 0$ is:

$$V_0^\star = \begin{bmatrix} 1.259 \\ 3.251 \\ 3.787 \\ 6.222 \\ 7.778 \\ 10 \end{bmatrix}. \tag{1.21}$$

Clearly, the optimal value function dominates the value function shown in (1.13) of the random policy at every state.

The optimal actions at $t = 0$ are:

$$\begin{aligned} \text{Hangover} &: \text{Lazy} \\ \text{Sleep} &: \text{Productive} \\ \text{More Sleep} &: \text{Productive} \\ \text{Visit Lecture} &: \text{Lazy} \\ \text{Study} &: \text{Productive} \\ \text{Pass Exam} &: \text{Lazy} \end{aligned} . \tag{1.22}$$

You can play with the code here.

1.2 Infinite-Horizon MDP

In a finite-horizon MDP, the horizon T must be specified in advance in order to carry out policy evaluation and dynamic programming. The finite horizon naturally provides a terminal condition, which serves as the boundary condition that allows backward recursion to proceed.

In many practical applications, however, the horizon T is not well defined or is difficult to determine. In such cases, it is often more natural and convenient to adopt the infinite-horizon MDP formulation.

An infinite-horizon MDP is given by the following tuple:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma),$$

where $\mathcal{S}$, $\mathcal{A}$, P, and R are the same as defined before in a finite-horizon MDP. We still restrict ourselves to the tabular MDP setup where $\mathcal{S}$ and $\mathcal{A}$ both have a finite number of elements.

The key difference between the finite-horizon and infinite-horizon formulations is that the fixed horizon T is replaced by a **discount factor** $\gamma \in [0, 1)$. This discount factor weights future rewards less heavily than immediate rewards, as we will see shortly.

Stationary Policy. In an infinite-horizon MDP, we focus on *stationary* policies $\pi : \mathcal{S} \mapsto \Delta(\mathcal{A})$, where $\pi(a \mid s)$ denotes the probability of taking action a in state s.

In contrast, in a finite-horizon MDP we considered a tuple of T policies (see (1.1)), where each π_t could vary with time (i.e., policies were non-stationary).

Intuitively, in the infinite-horizon setting, it suffices to consider stationary policies because the decision-making problem at time t is equivalent to the problem at time $t + k$ for any $k \in \mathbb{N}$, as both face the same infinite horizon.

Trajectory and Return. Given an initial state $s_0 \in \mathcal{S}$ and a stationary policy π, the MDP will evolve as

1. Start at state s_0
2. Take action $a_0 \sim \pi(\cdot \mid s_0)$ following policy π
3. Collect reward $r_0 = R(s_0, a_0)$
4. Transition to state $s_1 \sim P(s' \mid s_0, a_0)$ following the dynamics
5. Go to step 2 and continue forever

This process generates a trajectory of states, actions, and rewards:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots).$$

The return of a trajectory is defined as

$$g_0 = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{t=0}^{\infty} \gamma^t r_t.$$

Here, the discount factor γ plays a key role: it progressively reduces the weight of rewards received further in the future, making them less influential as t increases.

1.2.1 Value Functions

Similar to the case of finite-horizon MDP, we can define the state-value function and the action-value function associated with a policy π.

State-Value Function. The value of a state $s \in \mathcal{S}$ under policy π is the expected discounted return obtained when starting from s at time 0:

$$V^\pi(s) := \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \,\middle|\, s_0 = s, a_t \sim \pi(\cdot \mid s_t), s_{t+1} \sim P(\cdot \mid s_t, a_t)\right]. \quad (1.23)$$

Action-Value Function. The value of a state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ under policy π is the expected discounted return obtained by first taking action a in state s, and then following policy π thereafter:

$$Q^\pi(s, a) := \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \,\middle|\, s_0 = s, a_0 = a, a_t \sim \pi(\cdot \mid s_t), s_{t+1} \sim P(\cdot \mid s_t, a_t)\right]. \quad (1.24)$$

Note that a nice feature of having a discount factor $\gamma \in [0, 1)$ is that both the state-value and the action-value functions are guaranteed to be bounded even if the horizon is unbounded (assuming the reward function is bounded).

We can verify the state-value function and the action value function satisfy the following relationship:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) Q^\pi(s, a) \quad (1.25)$$

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^\pi(s'). \quad (1.26)$$

Combining these two equations, we arrive at the Bellman consistency result for infinite-horizon MDP.

Proposition 1.2 (Bellman Consistency (Infinite Horizon))**.** *The state-value function V^π in* (1.23) *satisfies the following recursion:*

$$\begin{aligned} V^\pi(s) &= \sum_{a\in\mathcal{A}} \pi(a \mid s) \left(R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a) V^\pi(s') \right) \\ &=: \mathbb{E}_{a\sim\pi(\cdot|s)} \left[R(s,a) + \gamma \mathbb{E}_{s'\sim P(\cdot|s,a)} [V^\pi(s')] \right]. \end{aligned} \tag{1.27}$$

Similarly, the action-value function $Q^\pi(s,a)$ in (1.24) *satisfies the following recursion:*

$$\begin{aligned} Q^\pi(s,a) &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a) \left(\sum_{a'\in\mathcal{A}} \pi(a' \mid s') Q^\pi(s',a') \right) \\ &=: R(s,a) + \gamma \mathbb{E}_{s'\sim P(\cdot|s,a)} \left[\mathbb{E}_{a'\sim\pi(\cdot|s')} [Q^\pi(s',a')] \right]. \end{aligned} \tag{1.28}$$

1.2.2 Policy Evaluation

Given a policy π, how can we compute its associated state-value and action-value functions?

- **Finite-horizon case.** We initialize the terminal value function $V_T^\pi(s) = 0$ for every $s \in \mathcal{S}$, and then apply the Bellman Consistency result (Proposition 1.1) to perform backward recursion.
- **Infinite-horizon case.** The Bellman Consistency result (Proposition 1.2) takes a different form and does not provide the same simple recipe for backward recursion.

System of Linear Equations. A closer look at the Bellman Consistency equation (1.27) for the state-value function shows that it defines a square system of linear equations. Specifically, the value function V^π can be represented as a vector with $|\mathcal{S}|$ variables, and (1.27) provides $|\mathcal{S}|$ linear equations over these variables.
Thus, one way to compute the state-value function is to set up this linear system and solve it. However, doing so typically requires matrix inversion or factorization, which can be computationally expensive.

The same reasoning applies to the action-value function Q^π, which can be represented as a vector of $|\mathcal{S}||\mathcal{A}|$ variables constrained by $|\mathcal{S}||\mathcal{A}|$ linear equations.

The following proposition states that, instead of solving a linear system of equations, one can use a globally convergent iterative scheme, one that is very much like the policy evaluation algorithm for the finite-horizon MDP, to evaluate the state-value function associated with a policy π.

Proposition 1.3 (Policy Evaluation (Infinite Horizon, State-Value))**.** *Consider an infinite-horizon MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$. Fix a policy π and consider the iterative scheme for the state-value function:*

$$V_{k+1}(s) \leftarrow \sum_{a\in\mathcal{A}} \pi(a \mid s) \left[R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a) V_k(s') \right], \quad \forall s \in \mathcal{S}. \tag{1.29}$$

Then, starting from any initialization $V_0 \in \mathbb{R}^{|\mathcal{S}|}$, the sequence $\{V_k\}$ converges to the unique fixed point V^π, the state-value function associated with policy π.

Proof. To prove the convergence of the policy evaluation algorithm, we shall introduce the notion of a Bellman operator.

Bellman Operator. Any value function $V(s)$ can be interpreted as a vector in $\mathbb{R}^{|\mathcal{S}|}$ (recall we are in the tabular MDP case). Given any value function $V \in \mathbb{R}^{|\mathcal{S}|}$, and a policy π, define the Bellman operator associated with π as $T^\pi : \mathbb{R}^{|\mathcal{S}|} \mapsto \mathbb{R}^{|\mathcal{S}|}$:

$$(T^\pi V)(s) := \sum_{a\in\mathcal{A}} \pi(a \mid s) \left[R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a) V(s') \right]. \tag{1.30}$$

We claim that T^π has two important properties.

- **Monotonicity**. If $V \le W$ (i.e., $V(s) \le W(s)$ for any $s \in \mathcal{S}$), then $T^\pi V \le T^\pi W$. To see this, observe that

$$\begin{aligned}(T^\pi V)(s) - (T^\pi W)(s) &= \sum_a \pi(a \mid s) \left(\gamma \sum_{s'} P(s' \mid s,a)(V(s') - W(s')) \right) \\ &= \gamma \mathbb{E}_{a\sim\pi(\cdot|s), s'\sim P(\cdot|s,a)}[V(s') - W(s')].\end{aligned}$$

  Therefore, if $V(s') - W(s') \le 0$ for any $s' \in \mathcal{S}$, then $T^\pi V \le T^\pi W$.

- **γ-Contraction**. For any value function $V \in \mathbb{R}^{|\mathcal{S}|}$, define the ℓ_∞ norm (sup norm) as

$$\|V\|_\infty = \max_{s\in\mathcal{S}} |V(s)|.$$

We claim that the Bellman operator T^π is a γ-contraction in the sup norm, i.e.,

$$\|T^\pi V - T^\pi W\|_\infty \le \gamma \|V - W\|_\infty, \quad \forall V, W \in \mathbb{R}^{|\mathcal{S}|}. \tag{1.31}$$

To prove this, observe that for any $s \in \mathcal{S}$, we have:

$$\begin{aligned}
|(T^\pi V)(s) - (T^\pi W)(s)| &= \left| \sum_a \pi(a|s)\, \gamma \sum_{s'} P(s'|s,a)(V(s') - W(s')) \right| \\
&\leq \gamma \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)\, |V(s') - W(s')| \\
&\leq \gamma \|V - W\|_\infty \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \\
&= \gamma \|V - W\|_\infty.
\end{aligned}$$

Taking the maximum over s gives

$$\|T^\pi V - T^\pi W\|_\infty \leq \gamma \|V - W\|_\infty,$$

so T^π is a γ-contraction in the sup norm.

With the Bellman operator defined, we observe that the value function of π, denoted V^π in (1.27), is a **fixed point** of T^π. That is to say V^π satisfies:

$$T^\pi V^\pi = V^\pi.$$

In other words, V^π is fixed (remains unchanged) under the Bellman operator.

Since T^π is a γ-contraction, by the Banach Fixed-Point Theorem, we know that there exists a unique fixed point to T^π, which is V^π. Moreover, since

$$\|V_k - V^\pi\|_\infty = \|T^\pi V_{k-1} - T^\pi V^\pi\|_\infty \leq \gamma \|V_{k-1} - V^\pi\|_\infty,$$

we can deduce the rate of convergence

$$\|V_k - V^\pi\|_\infty \leq \gamma^k \|V_0 - V^\pi\|_\infty.$$

Therefore, policy evaluation globally converges from any initialization V_0 at a linear rate of γ. □

We have a similar policy evaluation algorithm for the action-value function.

Proposition 1.4 (Policy Evaluation (Infinite Horizon, Action-Value))**.** *Fix a policy π. Consider the iterative scheme on $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$:*

$$Q_{k+1}(s,a) \;\leftarrow\; R(s,a) \;+\; \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \left(\sum_{a' \in \mathcal{A}} \pi(a' \mid s')\, Q_k(s',a') \right), \quad \forall (s,a) \in \mathcal{S} \times \mathcal{A}. \tag{1.32}$$

Then, for any initialization Q_0, the sequence $\{Q_k\}$ converges to the unique fixed point Q^π, the action-value function associated with policy π.

Proof. Define the Bellman operator on action-values

$$(T^\pi Q)(s,a) := R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \Big(\sum_{a'} \pi(a' \mid s')\, Q(s',a') \Big).$$

T^π is a γ-contraction in the sup-norm on $\mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$; hence by the Banach fixed-point theorem, global convergence holds regardless of initialization. □

Let us apply policy evaluation to an infinite-horizon MDP.

Example 1.3 (Policy Evaluation for Inverted Pendulum)**.**

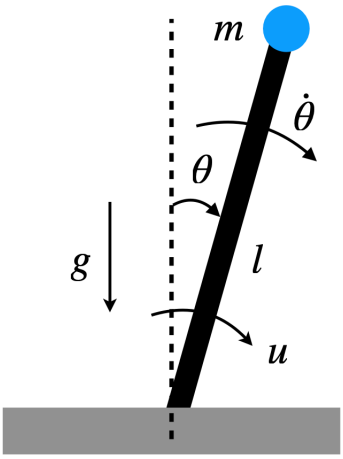

Figure 1.3: Inverted Pendulum.

We consider the inverted pendulum with state $s = (\theta, \dot\theta)$ and action (torque) $a = u$, as visualized in Fig. 1.3. Our goal is to swing up the pendulum from any initial state to the upright position $s = (0, 0)$.

Continuous-Time Dynamics. The continuous-time dynamics of the inverted pendulum is

$$\ddot\theta \;=\; \frac{g}{l}\sin(\theta) \;+\; \frac{1}{ml^2}u \;-\; c\,\dot\theta,$$

where $m > 0$ is the mass of the pendulum, $l > 0$ is the length of the pole, $c > 0$ is the damping coefficient, and g is the gravitational constant.

Discretization (Euler). With timestep Δt, we obtain the following discrete-time dynamics:

$$\begin{aligned} \theta_{k+1} &= \theta_k + \Delta t\, \dot\theta_k, \\ \dot\theta_{k+1} &= \dot\theta_k + \Delta t \Big(\tfrac{g}{l}\sin(\theta_k) + \tfrac{1}{ml^2} u_k - c\,\dot\theta_k \Big). \end{aligned} \tag{1.33}$$

We wrap angles to $[-\pi, \pi]$ via $\text{wrap}(\theta) = \text{atan2}(\sin\theta, \cos\theta)$.

Tabular MDP. We convert the discrete-time dynamics into a tabular MDP.

- **State grid.** $\theta \in [-\pi, \pi]$, $\dot{\theta} \in [-\pi, \pi]$ on uniform grids:

$$\mathcal{S} = \{\ (\theta_i, \dot{\theta}_j)\ :\ i = 1, \dots, N_\theta,\ j = 1, \dots, N_{\dot{\theta}}\ \}.$$

- **Action grid.** $u \in [-mgl/2, mgl/2]$ on N_u uniform points:

$$\mathcal{A} = \{u_\ell : \ell = 1, \dots, N_u\}.$$

- **Stochastic transition kernel (nearest-3 interpolation).** From a grid point $s = (\theta_i, \dot{\theta}_j)$ and an action u_ℓ, compute the next continuous state $s^+ = (\theta^+, \dot{\theta}^+)$ via the discrete-time dynamics in (1.33). If $s^+ \notin \mathcal{S}$, choose the three closest grid states $\{s^{(1)}, s^{(2)}, s^{(3)}\}$ by Euclidean distance in $(\theta, \dot{\theta})$ and assign probabilities

$$p_r \propto \frac{1}{\|s^+ - s^{(r)}\|_2 + \varepsilon}, \quad r = 1, 2, 3, \qquad \sum_r p_r = 1,$$

  so nearer grid points receive higher probability (use a small $\varepsilon > 0$ to avoid division by zero).
- **Reward.** A quadratic shaping penalty around the upright equilibrium:

$$R(s, a) = -\Big(\theta^2 + 0.1\,\dot{\theta}^2 + 0.01\,u^2\Big).$$

- **Discount.** $\gamma \in [0, 1)$. We obtain a discounted, infinite-horizon, **tabular** MDP.

Policy. For policy evaluation, consider $\pi(a \mid s)$ be uniform over the discretized actions, i.e., a random policy.

Policy Evaluation. The following python script performs policy evaluation.

```
import numpy as np
import matplotlib.pyplot as plt

# ----- Physical & MDP parameters -----
g, l, m, c = 9.81, 1.0, 1.0, 0.1
dt = 0.05
gamma = 0.97
eps = 1e-8

# Grids
N_theta = 41
```

```
N_thetadot = 41
N_u = 21

theta_grid = np.linspace(-np.pi, np.pi, N_theta)
thetadot_grid = np.linspace(-np.pi, np.pi, N_thetadot)
u_max = 0.5 * m * g * l
u_grid = np.linspace(-u_max, u_max, N_u)

# Helpers to index/unwrap
def wrap_angle(x):
    return np.arctan2(np.sin(x), np.cos(x))

def state_index(i, j):
    return i * N_thetadot + j

def index_to_state(idx):
    i = idx // N_thetadot
    j = idx % N_thetadot
    return theta_grid[i], thetadot_grid[j]

S = N_theta * N_thetadot
A = N_u

# ----- Dynamics step (continuous -> one Euler step) -----
def step_euler(theta, thetadot, u):
    theta_next = wrap_angle(theta + dt * thetadot)
    thetadot_next = thetadot + dt * ((g/l) * np.sin(theta) + (1/(m*l*l))*u - c*thetadot)
    # clip angular velocity to grid range (bounded MDP)
    thetadot_next = np.clip(thetadot_next, thetadot_grid[0], thetadot_grid[-1])
    return theta_next, thetadot_next

# ----- Find 3 nearest grid states and probability weights (inverse-distance) -----
# Pre-compute all grid points for fast nearest neighbor search
grid_pts = np.stack(np.meshgrid(theta_grid, thetadot_grid, indexing='ij'), axis=-1).reshape(-1, 2

def nearest3_probs(theta_next, thetadot_next):
    x = np.array([theta_next, thetadot_next])
    dists = np.linalg.norm(grid_pts - x[None, :], axis=1)
    nn_idx = np.argpartition(dists, 3)[:3]  # three smallest (unordered)
    # sort those 3 by distance for stability
    nn_idx = nn_idx[np.argsort(dists[nn_idx])]
    d = dists[nn_idx]
    w = 1.0 / (d + eps)
    p = w / w.sum()
```

```
    return nn_idx.astype(int), p

# ----- Reward -----
def reward(theta, thetadot, u):
    return -(theta**2 + 0.1*thetadot**2 + 0.01*u**2)

# ----- Build tabular MDP: R[s,a] and sparse P[s,a,3] -----
R = np.zeros((S, A))
NS_idx = np.zeros((S, A, 3), dtype=int)   # next-state indices (3 nearest)
NS_prob = np.zeros((S, A, 3))             # their probabilities

for i, th in enumerate(theta_grid):
    for j, thd in enumerate(thetadot_grid):
        s = state_index(i, j)
        for a, u in enumerate(u_grid):
            # reward at current (s,a)
            R[s, a] = reward(th, thd, u)
            # next continuous state
            th_n, thd_n = step_euler(th, thd, u)
            # map to 3 nearest grid states
            nn_idx, p = nearest3_probs(th_n, thd_n)
            NS_idx[s, a, :] = nn_idx
            NS_prob[s, a, :] = p

# ----- Fixed policy: uniform over actions -----
Pi = np.full((S, A), 1.0 / A)

# ----- Iterative policy evaluation -----
V = np.zeros(S)  # initialization (any vector works)
tol = 1e-6
max_iters = 10000

for k in range(max_iters):
    V_new = np.zeros_like(V)
    # Compute Bellman update: V_{k+1}(s) = sum_a Pi(s,a)[ R(s,a) + gamma * sum_j P(s,a
    # First, expected next V for each (s,a)
    EV_next = (NS_prob * V[NS_idx]).sum(axis=2)  # shape: (S, A)
    # Then expectation over actions under Pi
    V_new = (Pi * (R + gamma * EV_next)).sum(axis=1)  # shape: (S,)
    # Check convergence
    if np.max(np.abs(V_new - V)) < tol:
        V = V_new
        print(f"Converged in {k+1} iterations (sup-norm change < {tol}).")
        break
```

```
    V = V_new
else:
    print(f"Reached max_iters={max_iters} without meeting tolerance {tol}.")

V_grid = V.reshape(N_theta, N_thetadot)

# V_grid: shape (N_theta, N_thetadot)
# theta_grid, thetadot_grid already defined
fig, ax = plt.subplots(figsize=(7,5), dpi=120)
im = ax.imshow(
    V_grid,
    origin="lower",
    extent=[thetadot_grid.min(), thetadot_grid.max(),
            theta_grid.min(), theta_grid.max()],
    aspect="auto",
    cmap="viridis"  # any matplotlib colormap, e.g., "plasma", "inferno"
)
cbar = fig.colorbar(im, ax=ax)
cbar.set_label(r"$V^\pi(\theta,\dot{\theta})$")

ax.set_xlabel(r"$\dot{\theta}$")
ax.set_ylabel(r"$\theta$")
ax.set_title(r"State-value $V^\pi$ (tabular policy evaluation)")

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

Running the code, it shows that policy evaluation converges in 518 iterations under tolerance 10^{-6}.

Fig. 1.4 plots the value function over the state grid.

You can play with the code here.

1.2.3 Principle of Optimality

In an infinite-horizon MDP, our goal is to find the optimal policy that maximizes the expected long-term discounted return:

$$V^\star := \max_\pi \mathbb{E}_{s\sim\mu(\cdot)}[V^\pi(s)],$$

where μ is a given initial distribution. We call $V^\star$ the optimal value function.

Given a policy π and its associated value function V^π, how do we know if the policy is already optimal?

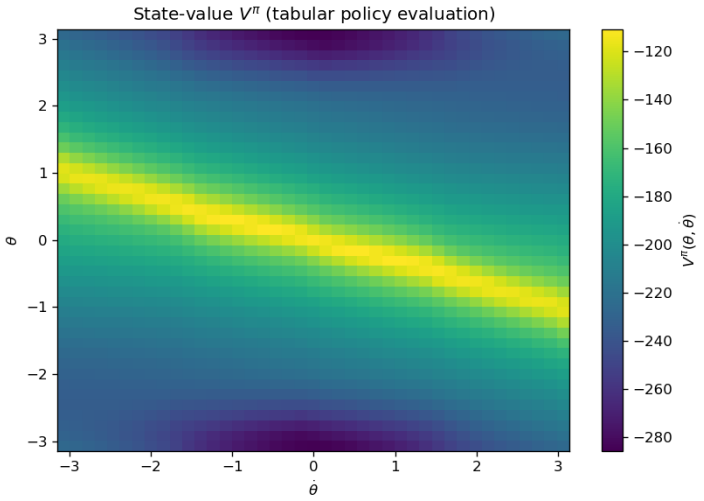

Figure 1.4: Value Function from Policy Evaluation.

Theorem 1.2 (Bellman Optimality (Infinite Horizon))**.** *For an infinite-horizon MDP with discount factor $\gamma \in [0, 1)$, the optimal state-value function $V^\star(s)$ satisfies the Bellman optimality equation*

$$V^\star(s) \;=\; \max_{a \in \mathcal{A}} \Big[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V^\star(s') \Big]. \tag{1.34}$$

Define the optimal action-value function as

$$Q^\star(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) V^\star(s'). \tag{1.35}$$

We have that $Q^\star(s,a)$ satisfies

$$Q^\star(s,a) \;=\; R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \;\Big[\max_{a' \in \mathcal{A}} Q^\star(s',a') \Big]. \tag{1.36}$$

Moreover, any greedy policy with respect to $V^\star$ (equivalently, to $Q^\star$) is optimal:

$$\begin{aligned} &\pi^\star(s) \in \arg\max_{a \in \mathcal{A}} \Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V^\star(s') \Big] \quad \Longleftrightarrow \\ &\pi^\star(s) \in \arg\max_{a \in \mathcal{A}} Q^\star(s,a). \end{aligned} \tag{1.37}$$

Proof. We will first show that $V^\star$ has statewise dominance over all other policies, and then show that $V^\star$ can be attained by the greedy policy.

Claim. For any discounted MDP with $\gamma \in [0, 1)$ and any policy π,

$$V^\star(s) \;\geq\; V^\pi(s) \qquad \forall s \in \mathcal{S},$$

where $V^\star$ is the unique solution of the Bellman **optimality** equation and V^π solves the Bellman **consistency** equation for π.

Proof via Bellman Operators. Define the Bellman operators

$$(T^\pi V)(s) := \sum_a \pi(a \mid s)\Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s')\Big],$$

$$(T^\star V)(s) := \max_a \Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s')\Big].$$

Key facts:

1. (**Monotonicity**) If $V \geq W$ componentwise, then $T^\pi V \geq T^\pi W$ and $T^\star V \geq T^\star W$.
2. (**Dominance of** T^*) For any V and any π,

   $$T^\star V \;\geq\; T^\pi V$$

   because the max over actions is at least the π-weighted average.
3. (**Fixed points**) $V^\pi = T^\pi V^\pi$ and $V^\star = T^\star V^\star$.
4. (**Contraction**) Each T^π and $T^\star$ is a γ-contraction in the sup-norm; hence their fixed points are unique.

Now start from V^π. Using (2),

$$V^\pi = T^\pi V^\pi \;\leq\; T^\star V^\pi.$$

Applying $T^\star$ repeatedly and using (1),

$$V^\pi \;\leq\; T^\star V^\pi \;\leq\; (T^\star)^2 V^\pi \;\leq\; \cdots$$

The sequence $(T^\star)^k V^\pi$ converges (by contraction) to the unique fixed point of $T^\star$, namely $V^\star$. Taking limits preserves the inequality, yielding $V^\pi \leq V^\star$ statewise. □

The Bellman optimality condition tells us, if a policy π is already greedy with respect to its value function V^π, then π is the optimal policy and V^π is the optimal value function.

In the next, we introduce two algorithms that can guarantee finding the optimal policy and the optimal value function.

The first algorithm, policy iteration (PI), iterates over the space of policies; while the second algorithm, value iteration (VI), iterates over the space of value functions.

1.2.4 Policy Improvement

The policy evaluation algorithm enables us to compute the value functions associated with a given policy π. The next result, known as the *Policy Improvement Lemma*, shows that once we have V^π, constructing a greedy policy with respect to V^π guarantees performance that is at least as good as π, and strictly better in some states unless π is already greedy with respect to V^π.

Lemma 1.1 (Policy Improvement)**.** *Let π be any policy and let V^π be its state-value function.*
Define a new policy π' such that for each state s,

$$\pi'(s) \in \arg\max_{a \in \mathcal{A}} \Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s') \Big].$$

Then for all states $s \in \mathcal{S}$,

$$V^{\pi'}(s) \;\geq\; V^\pi(s).$$

Moreover, the inequality is strict for some state s unless π is already greedy with respect to V^π (which implies optimality).

Proof. Let V^π be the value function of a policy π, and define a new (possibly stochastic) policy π' that is greedy w.r.t. V^π:

$$\pi'(\cdot \mid s) \in \arg\max_{\mu \in \Delta(\mathcal{A})} \sum_a \mu(a) \Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V^\pi(s') \Big].$$

Define the Bellman operators

$$(T^\pi V)(s) := \sum_a \pi(a \mid s) \Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \Big],$$
$$(T^{\pi'} V)(s) := \sum_a \pi'(a \mid s) \Big[\cdots \Big].$$

Step 1: One-step improvement at V^π. By greediness of π' w.r.t. V^π,

$$(T^{\pi'} V^\pi)(s) = \max_\mu \sum_a \mu(a) \Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s') \Big] \;\geq\; (T^\pi V^\pi)(s) = V^\pi(s),$$

for all s. Hence

$$T^{\pi'} V^\pi \;\geq\; V^\pi \quad \text{(componentwise)}. \tag{1.38}$$

Step 2: Monotonicity + contraction yield global improvement. The operator $T^{\pi'}$ is **monotone** (order-preserving) and a γ**-contraction** in the sup-norm.
Apply $T^{\pi'}$ repeatedly to both sides of (1.38):

$$(T^{\pi'})^k V^\pi \;\geq\; (T^{\pi'})^{k-1} V^\pi \;\geq\; \cdots \;\geq\; V^\pi, \qquad k = 1, 2, \ldots$$

By contraction, $(T^{\pi'})^k V^{\pi} \to V^{\pi'}$, the unique fixed point of $T^{\pi'}$.
Taking limits preserves the inequality, so

$$V^{\pi'} \geq V^{\pi} \quad \text{statewise.}$$

Strict improvement condition. If there exists a state s such that

$$(T^{\pi'} V^{\pi})(s) > V^{\pi}(s),$$

then by monotonicity we have a strict increase at that state after one iteration, and the limit remains strictly larger at that state (or at any state that can reach it with positive probability under π').
This happens precisely when π' selects, with positive probability, an action a for which

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V^{\pi}(s') > V^{\pi}(s),$$

i.e., when π was not already greedy (optimal) at s. □

1.2.5 Policy Iteration

The policy improvement lemma and the principle of optimality, combined together, leads to the first algorithm that guarantees convergence to an optimal policy. This algorithm is called policy iteration.

Theorem 1.3 (Convergence of Policy Iteration)**.** *Consider a discounted MDP with finite state and action sets and $\gamma \in [0,1)$. Let $\{\pi_k\}_{k \geq 0}$ be the sequence produced by* Policy Iteration (PI)*:*

1. ***Policy evaluation:*** *compute V^{π_k} such that $V^{\pi_k} = T^{\pi_k} V^{\pi_k}$.*
2. ***Policy improvement:*** *choose π_{k+1} greedy w.r.t. V^{π_k}:*

$$\pi_{k+1}(s) \in \arg\max_a \Big[R(s,a) + \gamma \sum_{s'} P(s'|s,a)\, V^{\pi_k}(s') \Big].$$

Then:

a. *$V^{\pi_{k+1}} \geq V^{\pi_k}$ componentwise, and the inequality is strict for some state unless $\pi_{k+1} = \pi_k$.*

b. *If $\pi_{k+1} = \pi_k$, then V^{π_k} satisfies the Bellman optimality equation; hence π_k is optimal and $V^{\pi_k} = V^*$.*

c. *Because the number of stationary policies is finite, PI terminates in finitely many iterations at an optimal policy π^* with value V^*.*

d. $\|V^{\pi_{k+1}} - V^\star\|_\infty \leq \gamma \|V^{\pi_k} - V^\star\|_\infty$, *for any* k *(i.e., contraction).*

Proof. By the policy improvement lemma, we have

$$V^{\pi_{k+1}} \geq V^{\pi_k}.$$

By monotonicity of the Bellman operator $T^{\pi_{k+1}}$, we have

$$V^{\pi_{k+1}} = T^{\pi_{k+1}} V^{\pi_{k+1}} \geq T^{\pi_{k+1}} V^{\pi_k}.$$

By definition of the Bellman optimality operator, we have

$$T^{\pi_{k+1}} V^{\pi_k} = T^\star V^{\pi_k}.$$

Therefore,

$$0 \geq V^{\pi_{k+1}} - V^\star \geq T^{\pi_{k+1}} V^{\pi_k} - V^\star = T^\star V^{\pi_k} - T^\star V^\star$$

As a result,

$$\|V^{\pi_{k+1}} - V^\star\|_\infty \leq \|T^\star V^{\pi_k} - T^\star V^\star\|_\infty \leq \gamma \|V^{\pi_k} - V^\star\|_\infty.$$

This proves the contraction result (d). □

Let us apply Policy Iteration to the inverted pendulum problem.

Example 1.4 (Policy Iteration for Inverted Pendulum)**.** The following code performs policy iteration for the inverted pendulum problem.

```
import numpy as np
import matplotlib.pyplot as plt

# ----- Physical & MDP parameters -----
g, l, m, c = 9.81, 1.0, 1.0, 0.1
dt = 0.05
gamma = 0.97
eps = 1e-8

# Grids
N_theta = 101
N_thetadot = 101
N_u = 51

theta_grid = np.linspace(-1.5*np.pi, 1.5*np.pi, N_theta)
thetadot_grid = np.linspace(-1.5*np.pi, 1.5*np.pi, N_thetadot)
u_max = 0.5 * m * g * l
u_grid = np.linspace(-u_max, u_max, N_u)
```

```python
# Helpers to index/unwrap
def wrap_angle(x):
    return np.arctan2(np.sin(x), np.cos(x))

def state_index(i, j):
    return i * N_thetadot + j

def index_to_state(idx):
    i = idx // N_thetadot
    j = idx % N_thetadot
    return theta_grid[i], thetadot_grid[j]

S = N_theta * N_thetadot
A = N_u

# ----- Dynamics step (continuous -> one Euler step) -----
def step_euler(theta, thetadot, u):
    theta_next = wrap_angle(theta + dt * thetadot)
    thetadot_next = thetadot + dt * ((g/l) * np.sin(theta) + (1/(m*l*l))*u - c*thetadot)
    # clip angular velocity to grid range (bounded MDP)
    thetadot_next = np.clip(thetadot_next, thetadot_grid[0], thetadot_grid[-1])
    return theta_next, thetadot_next

# ----- Find 3 nearest grid states and probability weights (inverse-distance) -----
grid_pts = np.stack(np.meshgrid(theta_grid, thetadot_grid, indexing='ij'), axis=-1).reshape(-1, 2

def nearest3_probs(theta_next, thetadot_next):
    x = np.array([theta_next, thetadot_next])
    dists = np.linalg.norm(grid_pts - x[None, :], axis=1)
    nn_idx = np.argpartition(dists, 3)[:3]       # three smallest (unordered)
    nn_idx = nn_idx[np.argsort(dists[nn_idx])]  # sort those 3 by distance
    d = dists[nn_idx]
    w = 1.0 / (d + eps)
    p = w / w.sum()
    return nn_idx.astype(int), p

# ----- Reward -----
def reward(theta, thetadot, u):
    return -(theta**2 + 0.1*thetadot**2 + 0.01*u**2)

# ----- Build tabular MDP: R[s,a] and sparse P[s,a,3] -----
R = np.zeros((S, A))
NS_idx = np.zeros((S, A, 3), dtype=int)    # next-state indices (3 nearest)
NS_prob = np.zeros((S, A, 3))              # their probabilities
```

```
for i, th in enumerate(theta_grid):
    for j, thd in enumerate(thetadot_grid):
        s = state_index(i, j)
        for a, u in enumerate(u_grid):
            # reward at current (s,a)
            R[s, a] = reward(th, thd, u)
            # next continuous state
            th_n, thd_n = step_euler(th, thd, u)
            # map to 3 nearest grid states
            nn_idx, p = nearest3_probs(th_n, thd_n)
            NS_idx[s, a, :] = nn_idx
            NS_prob[s, a, :] = p

# ======================
#     POLICY ITERATION
# ======================

# Represent policy as a deterministic action index per state: pi[s] in {0..A-1}
# Start from uniform-random policy (deterministic tie-breaker: middle action)
pi = np.full(S, A // 2, dtype=int)

def policy_evaluation(pi, V_init=None, tol=1e-6, max_iters=10000):
    """Iterative policy evaluation for deterministic pi (action index per state)."""
    V = np.zeros(S) if V_init is None else V_init.copy()
    for k in range(max_iters):
        # For each state s, use chosen action a = pi[s]
        a = pi  # shape (S,)
        # Expected next value under chosen action
        EV_next = (NS_prob[np.arange(S), a] * V[NS_idx[np.arange(S), a]]).sum(axis=1)
        V_new = R[np.arange(S), a] + gamma * EV_next
        if np.max(np.abs(V_new - V)) < tol:
            # print(f"Policy evaluation converged in {k+1} iterations.")
            return V_new
        V = V_new
    # print("Policy evaluation reached max_iters without meeting tolerance.")
    return V

def policy_improvement(V, pi_old=None):
    """Greedy improvement: pi'(s) = argmax_a [ R(s,a) + gamma * E[V(s')] ]."""
    # Compute Q(s,a) = R + gamma * sum_j P(s,a,j) V(ns_j)
    EV_next = (NS_prob * V[NS_idx]).sum(axis=2)      # (S, A)
    Q = R + gamma * EV_next                           # (S, A)
    pi_new = np.argmax(Q, axis=1).astype(int)         # greedy deterministic policy
    stable = (pi_old is not None) and np.array_equal(pi_new, pi_old)
```

```
    return pi_new, stable

# Main PI loop
max_pi_iters = 100
V = np.zeros(S)
for it in range(max_pi_iters):
    # Policy evaluation
    V = policy_evaluation(pi, V_init=V, tol=1e-6, max_iters=10000)
    # Policy improvement
    pi_new, stable = policy_improvement(V, pi_old=pi)
    print(f"[PI] Iter {it+1}: policy changed = {not stable}")
    pi = pi_new
    if stable:
        print("Policy iteration converged: policy stable.")
        break
else:
    print("Reached max_pi_iters without policy stability (may still be near-optimal).")

# ----- Visualization -----
V_grid = V.reshape(N_theta, N_thetadot)

fig, ax = plt.subplots(figsize=(7,5), dpi=120)
im = ax.imshow(
    V_grid,
    origin="lower",
    extent=[thetadot_grid.min(), thetadot_grid.max(),
            theta_grid.min(), theta_grid.max()],
    aspect="auto",
    cmap="viridis"
)
cbar = fig.colorbar(im, ax=ax)
cbar.set_label(r"$V^{\pi}(\theta,\dot{\theta})$ (final PI)")

ax.set_xlabel(r"$\dot{\theta}$")
ax.set_ylabel(r"$\theta$")
ax.set_title(r"State-value $V$ after Policy Iteration")

plt.tight_layout()
plt.show()

# Visualize the greedy action *value* (torque)
pi_grid = pi.reshape(N_theta, N_thetadot)          # action indices
action_values = u_grid[pi_grid]                    # map indices -> torques
```

```
plt.figure(figsize=(7,5), dpi=120)
im = plt.imshow(action_values,
           origin="lower",
           extent=[thetadot_grid.min(), thetadot_grid.max(),
                   theta_grid.min(), theta_grid.max()],
           aspect="auto", cmap="coolwarm")         # diverging colormap good for ± tor
cbar = plt.colorbar(im)
cbar.set_label("Greedy action value (torque)")

plt.xlabel(r"$\dot{\theta}$")
plt.ylabel(r"$\theta$")
plt.title("Greedy policy (torque) after PI")
plt.tight_layout()
plt.show()
```

Running the code produces the optimal value function shown in Fig. 1.5 and the optimal policy shown in Fig. 1.6.

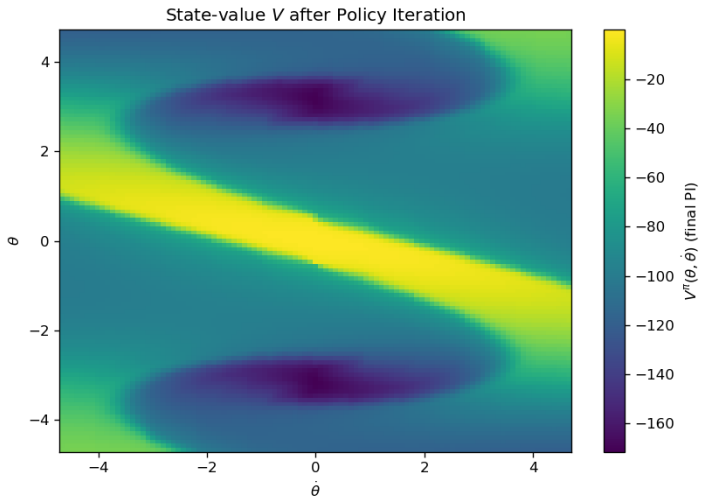

Figure 1.5: Optimal Value Function after Policy Iteration

We can apply the optimal policy to the pendulum with an initial state of $(-\pi, 0)$ (i.e., the bottomright position). Fig. 1.7 plots the rollout trajectory of $\theta, \dot{\theta}, u$. We can see that the optimal policy is capable of performing "bang-bang" control to accumulate energy before swinging up.

Fig. 1.8 overlays the trajectory on top of the optimal value function.

You can play with the code here.

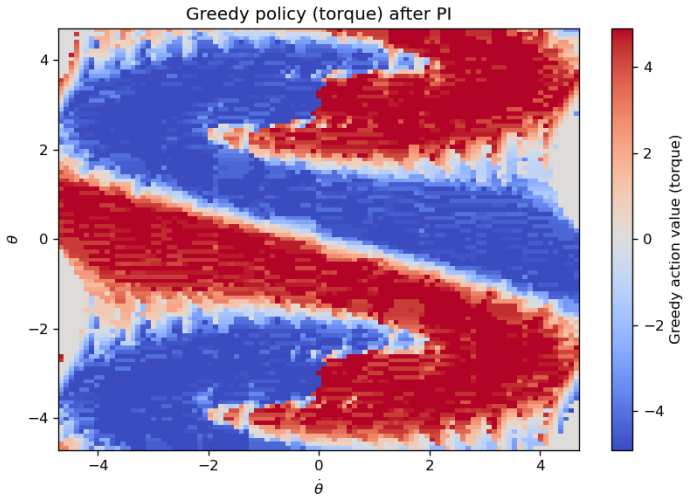

Figure 1.6: Optimal Policy after Policy Iteration

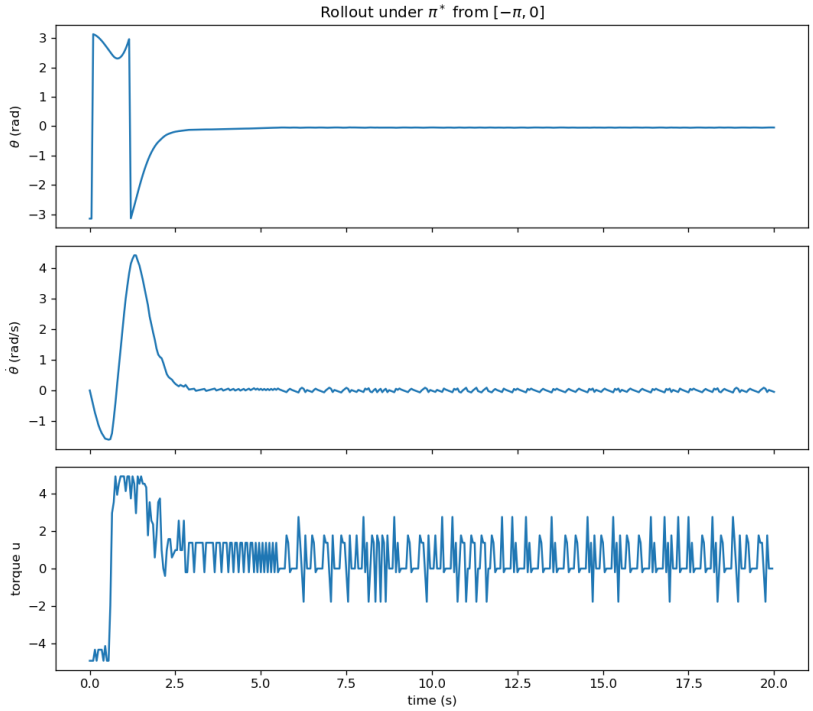

Figure 1.7: Optimal Trajectory of Pendulum Swing-Up

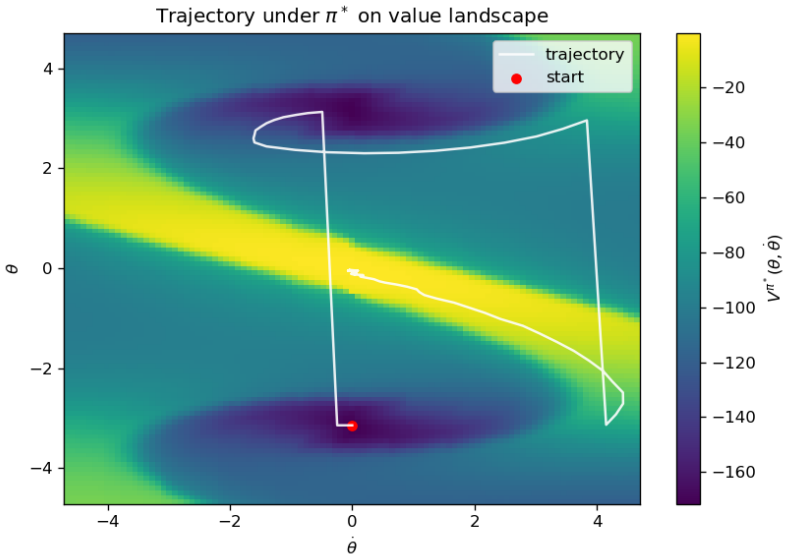

Figure 1.8: Optimal Trajectory of Pendulum Swing-Up Overlayed with Optimal Value Function

1.2.6 Value Iteration

Policy iteration—as the name suggests—iterates on *policies*: it alternates between (1) *policy evaluation* (computing V^π for the current policy π) and (2) *policy improvement* (making π greedy w.r.t. V^π).

An alternative, often very effective, method is *value iteration*. Unlike policy iteration, value iteration does *not* explicitly maintain a policy during its updates; it iterates directly on the value function toward the fixed point of the Bellman optimality* operator. Once the value function has (approximately) converged, the optimal policy is obtained by a single greedy extraction step. Note that intermediate value iterates need not correspond to the value of any actual policy.

The value iteration (VI) algorithm works as follows:

Initialization. Choose any $V_0 : \mathcal{S} \to \mathbb{R}$ (e.g., $V_0 \equiv 0$).
Iteration. For $k = 0, 1, 2, \ldots$,

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \Big[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V_k(s') \Big], \quad \forall s \in \mathcal{S}.$$

Stopping rule. Stop when $\|V_{k+1} - V_k\|_\infty \le \varepsilon$ (or any chosen tolerance).

Policy extraction (greedy):

$$\pi_{k+1}(s) \in \arg\max_{a \in \mathcal{A}} \Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V_{k+1}(s') \Big].$$

The following theorem states the convergence of value iteration.

Theorem 1.4 (Convergence of Value Iteration)**.** *Let $T^\star$ be the Bellman optimality operator,*

$$(T^\star V)(s) := \max_a \Big[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V(s') \Big].$$

For $\gamma \in [0,1)$ and finite $\mathcal{S}, \mathcal{A}$, $T^\star$ is a γ-contraction in the sup-norm. Hence, for any V_0,

$$V_k \;=\; (T^\star)^k V_0 \;\xrightarrow[k\to\infty]{}\; V^*,$$

the unique fixed point of $T^\star$. Moreover, the greedy policy π_k extracted from V_k converges to an optimal policy $\pi^\star$.

In addition, after k iterations, we have

$$\|V_k - V^*\|_\infty \;\le\; \gamma^k \, \|V_0 - V^*\|_\infty.$$

Finally, we apply value iteration to the inverted pendulum problem.

Example 1.5 (Value Iteration for Inverted Pendulum)**.** The following code performs value iteration for the inverted pendulum problem.

```
import numpy as np
import matplotlib.pyplot as plt

# ----- Physical & MDP parameters -----
g, l, m, c = 9.81, 1.0, 1.0, 0.1
dt = 0.05
gamma = 0.97
eps = 1e-8

# Grids
N_theta = 101
N_thetadot = 101
N_u = 51

theta_grid = np.linspace(-1.5*np.pi, 1.5*np.pi, N_theta)
thetadot_grid = np.linspace(-1.5*np.pi, 1.5*np.pi, N_thetadot)
u_max = 0.5 * m * g * l
u_grid = np.linspace(-u_max, u_max, N_u)

# Helpers to index/unwrap
def wrap_angle(x):
    return np.arctan2(np.sin(x), np.cos(x))
```

```
def state_index(i, j):
    return i * N_thetadot + j

def index_to_state(idx):
    i = idx // N_thetadot
    j = idx % N_thetadot
    return theta_grid[i], thetadot_grid[j]

S = N_theta * N_thetadot
A = N_u

# ----- Dynamics step (continuous -> one Euler step) -----
def step_euler(theta, thetadot, u):
    theta_next = wrap_angle(theta + dt * thetadot)
    thetadot_next = thetadot + dt * ((g/l) * np.sin(theta) + (1/(m*l*l))*u - c*thetadot
    # clip angular velocity to grid range (bounded MDP)
    thetadot_next = np.clip(thetadot_next, thetadot_grid[0], thetadot_grid[-1])
    return theta_next, thetadot_next

# ----- Find 3 nearest grid states and probability weights (inverse-distance) -----
grid_pts = np.stack(np.meshgrid(theta_grid, thetadot_grid, indexing='ij'), axis=-1).res

def nearest3_probs(theta_next, thetadot_next):
    x = np.array([theta_next, thetadot_next])
    dists = np.linalg.norm(grid_pts - x[None, :], axis=1)
    nn_idx = np.argpartition(dists, 3)[:3]      # three smallest (unordered)
    nn_idx = nn_idx[np.argsort(dists[nn_idx])]  # sort those 3 by distance
    d = dists[nn_idx]
    w = 1.0 / (d + eps)
    p = w / w.sum()
    return nn_idx.astype(int), p

# ----- Reward -----
def reward(theta, thetadot, u):
    return -(theta**2 + 0.1*thetadot**2 + 0.01*u**2)

# ----- Build tabular MDP: R[s,a] and sparse P[s,a,3] -----
R = np.zeros((S, A))
NS_idx = np.zeros((S, A, 3), dtype=int)   # next-state indices (3 nearest)
NS_prob = np.zeros((S, A, 3))             # their probabilities

for i, th in enumerate(theta_grid):
    for j, thd in enumerate(thetadot_grid):
        s = state_index(i, j)
```

```
        for a, u in enumerate(u_grid):
            R[s, a] = reward(th, thd, u)
            th_n, thd_n = step_euler(th, thd, u)
            nn_idx, p = nearest3_probs(th_n, thd_n)
            NS_idx[s, a, :] = nn_idx
            NS_prob[s, a, :] = p

# ========================
#       VALUE ITERATION
# ========================

# Bellman optimality update:
# V_{k+1}(s) = max_a [ R(s,a) + gamma * sum_j P(s,a,j) * V_k(ns_j) ]
V = np.zeros(S)
tol = 1e-6
max_vi_iters = 1000

for k in range(max_vi_iters):
    # Expected next V for every (s,a), given current V_k
    EV_next = (NS_prob * V[NS_idx]).sum(axis=2)   # shape (S, A)
    Q = R + gamma * EV_next                       # shape (S, A)
    V_new = np.max(Q, axis=1)                     # greedy backup over actions

    delta = np.max(np.abs(V_new - V))
    # Optional: a stopping rule aligned with policy loss bound could scale tol
    # e.g., stop when delta <= tol * (1 - gamma) / (2 * gamma)
    if delta < tol:
        V = V_new
        print(f"Value Iteration converged in {k+1} iterations (sup-norm change {delta:.2e}).")
        break
    V = V_new
else:
    print(f"Reached max_vi_iters={max_vi_iters} (last sup-norm change {delta:.2e}).")

# Greedy policy extraction from the final V
EV_next = (NS_prob * V[NS_idx]).sum(axis=2)   # recompute with final V
Q = R + gamma * EV_next
pi = np.argmax(Q, axis=1)                     # deterministic greedy policy (indices)

# ----- Visualization: Value function -----
V_grid = V.reshape(N_theta, N_thetadot)

fig, ax = plt.subplots(figsize=(7,5), dpi=120)
im = ax.imshow(
```

```
    V_grid,
    origin="lower",
    extent=[thetadot_grid.min(), thetadot_grid.max(),
            theta_grid.min(), theta_grid.max()],
    aspect="auto",
    cmap="viridis"
)
cbar = fig.colorbar(im, ax=ax)
cbar.set_label(r"$V^*(\theta,\dot{\theta})$ (Value Iteration)")

ax.set_xlabel(r"$\dot{\theta}$")
ax.set_ylabel(r"$\theta$")
ax.set_title(r"State-value $V$ after Value Iteration")

plt.tight_layout()
plt.show()

# ----- Visualization: Greedy torque field -----
pi_grid = pi.reshape(N_theta, N_thetadot)   # action indices
action_values = u_grid[pi_grid]             # map indices -> torques

plt.figure(figsize=(7,5), dpi=120)
im = plt.imshow(
    action_values,
    origin="lower",
    extent=[thetadot_grid.min(), thetadot_grid.max(),
            theta_grid.min(), theta_grid.max()],
    aspect="auto",
    cmap="coolwarm"   # good for ± torque
)
cbar = plt.colorbar(im)
cbar.set_label("Greedy action value (torque)")

plt.xlabel(r"$\dot{\theta}$")
plt.ylabel(r"$\theta$")
plt.title("Greedy policy (torque) extracted from Value Iteration")
plt.tight_layout()
plt.show()
```

Try it for yourself here!

You should obtain the same results as policy iteration.

Chapter 2

Value-based Reinforcement Learning

In Chapter 1, we introduced algorithms for policy evaluation, policy improvement, and computing optimal policies in the tabular setting when the model is known. These dynamic-programming methods are grounded in Bellman consistency and optimality and come with strong convergence guarantees.

A key limitation of the methods in Chapter 1 is that they require the transition dynamics $P(s' \mid s, a)$ to be known. While in some applications modeling the dynamics is feasible (e.g., the inverted pendulum), in many others it is costly or impractical to obtain an accurate model of the environment (e.g., a humanoid robot interacting with everyday objects).

This motivates relaxing the known-dynamics assumption and asking whether we can design algorithms that learn purely from interaction—i.e., by collecting data through environment interaction. This brings us to **model-free reinforcement learning**.

In this chapter we focus on **value-based** RL methods. The central idea is to learn the value functions—$V(s)$ and $Q(s, a)$—from interaction with the environment and then leverage these estimates to derive (approximately) optimal policies. We begin with tabular methods and then move to function-approximation approaches (e.g., neural networks) for problems where a tabular representation is intractable.

2.1 Tabular Methods

Consider an infinite-horizon Markov decision process (MDP)

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma),$$

with a discount factor $\gamma \in [0, 1)$. We focus on the *tabular setting* where both the state space $\mathcal{S}$ and the action space $\mathcal{A}$ are finite, with cardinalities $|\mathcal{S}|$ and $|\mathcal{A}|$, respectively.

A policy is a stationary stochastic mapping

$$\pi : \mathcal{S} \to \Delta(\mathcal{A}),$$

where $\pi(a \mid s)$ denotes the probability of selecting action a in state s.

Unlike in Chapter 1, here we do not assume knowledge of the transition dynamics P or the reward function R (other than that R is deterministic). Instead, we assume we can interact with the environment and obtain *trajectories* of the form

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots),$$

by following a policy π.

2.1.1 Policy Evaluation

We first consider the problem of estimating the value function of a given policy π. Recall the definition of the state-value function associated with π is:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s\right], \tag{2.1}$$

where the expectation is taken over the randomness of both the policy π and the transition dynamics P.

2.1.1.1 Monte Carlo Estimation

The basic idea of Monte Carlo (MC) estimation is to approximate the value function V^π by averaging *empirical returns* observed from sampled trajectories generated under policy π. Since the return is defined as the discounted sum of future rewards, MC methods replace the expectation in the definition of V^π with an average over sampled trajectories.

Episodic Assumption. To make Monte Carlo methods well-defined, we restrict attention to the *episodic setup*, where each trajectory terminates upon reaching a terminal state (and the rewards thereafter are always zero). This ensures that the return is finite and can be computed exactly for each trajectory.

Concretely, if an episode terminates at time T, the return starting from time t is

$$g_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t-1} r_{T-1}. \tag{2.2}$$

Algorithmic Form. Let $\mathcal{D}(s)$ denote the set of all time indices at which state s is visited across sampled episodes. Then the Monte Carlo estimate of the value function is

$$\hat{V}(s) = \frac{1}{|\mathcal{D}(s)|} \sum_{t \in \mathcal{D}(s)} g_t. \tag{2.3}$$

There are two common variants:

- **First-visit MC:** use only the first occurrence of s in each episode.
- **Every-visit MC:** use all occurrences of s within an episode.

Both variants converge to the same value function in the limit of infinitely many episodes.

Incremental Implementation. Monte Carlo can be written as an incremental stochastic-approximation update that uses the return g_t as the *target* and a *diminishing step size*. Let $N(s)$ be the number of (first- or every-) visits to state s that have been used to update $\hat{V}(s)$ so far, and let g_t be the return computed at a particular visit time $t \in \mathcal{D}(s)$. Then the MC update is

$$\hat{V}(s) \leftarrow \hat{V}(s) + \alpha_{N(s)} \left(g_t - \hat{V}(s)\right), \qquad \alpha_{N(s)} > 0 \text{ diminishing.} \tag{2.4}$$

A canonical choice is the *sample-average* step size $\alpha_{N(s)} = 1/N(s)$, which yields the recurrence

$$\hat{V}_N(s) = \hat{V}_{N-1}(s) + \tfrac{1}{N}(g_t - \hat{V}_{N-1}(s)) = \left(1 - \tfrac{1}{N}\right)\hat{V}_{N-1}(s) + \tfrac{1}{N}\, g_t \tag{2.5}$$

$$= \frac{N-1}{N} \frac{1}{N-1} \sum_{i=1}^{N-1} g_{t,i} + \frac{1}{N} g_t \tag{2.6}$$

$$= \frac{1}{N} \sum_{i=1}^{N} g_{t,i} \tag{2.7}$$

so that $\hat{V}_N(s)$ equals the average of the N observed returns for s (i.e., Eq. (2.3)). In the above equation, I have used $g_{t,i}$ to denote the i-th return before g_t was collected (and $g_t = g_{t,N}$). More generally, any diminishing schedule satisfying

$$\sum_{n=1}^{\infty} \alpha_n = \infty, \qquad \sum_{n=1}^{\infty} \alpha_n^2 < \infty$$

(e.g., $\alpha_n = c/(n+t_0)^p$ with $1/2 < p \leq 1$) also ensures consistency in the tabular setting. In first-visit MC, $N(s)$ increases by one per episode at most; in every-visit MC, $N(s)$ increases at each occurrence of s within an episode.

Theoretical Guarantees.

1. **Unbiasedness:** For any state s, the return g_t is an unbiased sample of $V^\pi(s)$.

$$\mathbb{E}[g_t \mid s_t = s] = V^\pi(s).$$

2. **Consistency:** By the law of large numbers, as the number of episodes grows,

$$\widehat{V}(s) \xrightarrow{\text{a.s.}} V^\pi(s).$$

3. **Asymptotic Normality:** The MC estimator converges at rate $O(1/\sqrt{N})$, where N is the number of episodes used for the estimation.

Limitations. Despite its conceptual simplicity, MC estimation suffers from several drawbacks:

- It requires *episodes to terminate*, making it unsuitable for continuing tasks without artificial truncation.
- It can only update value estimates *after an episode ends*, which is data-inefficient.
- While unbiased, MC estimates often have *high variance*, leading to slow convergence.

These limitations motivate the study of *Temporal-Difference (TD) learning*, which updates value estimates online and can handle continuing tasks.

2.1.1.2 Temporal-Difference Learning

While Monte Carlo methods estimate value functions by averaging full returns from complete episodes, Temporal-Difference (TD) learning provides an alternative approach that updates value estimates *incrementally* after each step of interaction with the environment. The key idea is to combine the sampling of Monte Carlo with the *bootstrapping* of dynamic programming.

High-Level Intuition. TD learning avoids waiting until the end of an episode by using the Bellman consistency equation as a basis for updates. Recall that for any policy π, the Bellman consistency equation reads:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}\left[R(s,a) + \gamma \mathbb{E}_{s'\sim P(s'|s,a)} V(s')\right]. \tag{2.8}$$

At a high level, TD learning turns the expectation in Bellman equation into sampling. At each step, it updates the current estimate of the value function toward a *one-step bootstrap target*: the immediate reward plus the discounted value of the next state. This makes TD methods more data-efficient and applicable to continuing tasks without terminal states.

Algorithmic Form. Suppose the agent is in state s_t, takes action $a_t \sim \pi(\cdot \mid s_t)$, receives reward r_t, and transitions to s_{t+1}. The TD(0) update rule is

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha[r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)], \tag{2.9}$$

where $\alpha \in (0, 1]$ is the learning rate.

The term inside the brackets,

$$\delta_t = r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t), \tag{2.10}$$

is called the TD error. It measures the discrepancy between the current value estimate and the bootstrap target. The algorithm updates $\hat{V}(s_t)$ in the direction of reducing this error.

Theoretical Guarantees.

1. **Convergence in the Tabular Case:** If each state is visited infinitely often and the learning rate sequence satisfies

   $$\sum_t \alpha_t = \infty, \ \sum_t \alpha_t^2 < \infty$$

   then TD(0) converges almost surely to the true value function V^π. For example, choosing $\alpha_t = 1/(t+1)$ satisfies this condition. Section 2.1.2 provides a detailed proof of the convergence of TD learning.

2. **Bias–Variance Tradeoff:**

   - The TD target uses the current estimate $\hat{V}(s_{t+1})$ rather than the true value, which introduces *bias*.
   - However, it has significantly *lower variance* than Monte Carlo estimates, often leading to faster convergence in practice.

   To see this, note that for TD(0), the target is a one-step bootstrap:

   $$y_t = r_t + \gamma \hat{V}(s_{t+1}).$$

   This replaces the true value $V^\pi(s_{t+1})$ with the *current estimate* $\hat{V}(s_{t+1})$. As a result, y_t is *biased* relative to the true return. However, since it depends only on the immediate reward and the next state, the variance of y_t is *much lower* than that of the Monte Carlo target.

Limitations.

- TD(0) relies on bootstrapping, which introduces bias relative to Monte Carlo methods.
- Convergence can be slow if the learning rate is not chosen carefully.

In summary, Temporal-Difference learning addresses the major limitations of Monte Carlo estimation: it works in *continuing tasks*, updates *online* at each step, and is generally more *sample-efficient*. However, it trades away unbiasedness for bias–variance efficiency, motivating further extensions such as multi-step TD and TD(λ).

2.1.1.3 Multi-Step TD Learning

Monte Carlo methods use the *full return* g_t, while TD(0) uses a *one-step bootstrap*. Multi-step TD learning generalizes these two extremes by using n-step returns as targets. In this way, multi-step TD interpolates between Monte Carlo and TD(0).

High-Level Intuition. The motivation is to balance the high variance of Monte Carlo with the bias of TD(0). Instead of waiting for a full return (MC) or using only one step of bootstrapping (TD(0)), multi-step TD uses partial returns spanning n steps of real rewards, followed by a bootstrap. This provides a flexible tradeoff between bias and variance.

Algorithmic Form. The n-step return starting from time t is defined as

$$g_t^{(n)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \widehat{V}(s_{t+n}). \tag{2.11}$$

The n-step TD update is

$$\widehat{V}(s_t) \leftarrow \widehat{V}(s_t) + \alpha[g_t^{(n)} - \widehat{V}(s_t)], \tag{2.12}$$

where $g_t^{(n)}$ replaces the one-step target in TD(0) (2.9).

- For $n = 1$: the method reduces to TD(0).
- For $n = T - t$ (the full episode length): the method reduces to Monte Carlo.

Theoretical Guarantees.

1. **Convergence in the Tabular Case:** With suitable learning rates and sufficient exploration, n-step TD converges to V^π.

2. **Bias–Variance Tradeoff:**
    - Larger n: lower bias, higher variance (closer to Monte Carlo).
    - Smaller n: higher bias, lower variance (closer to TD(0)).
    - Intermediate n provides a balance that often yields faster learning in practice.

Limitations.

- Choosing the right n is problem-dependent: too small and bias dominates; too large and variance grows.
- Requires storing n-step reward sequences before updating, which can increase memory and computation.

In summary, multi-step TD unifies Monte Carlo and TD(0) by introducing n-step returns. It allows practitioners to *tune the bias–variance tradeoff* by selecting n. Later, we will see how TD(λ) averages over all n-step returns in a principled way, further smoothing this tradeoff.

2.1.1.4 Eligibility Traces and TD(λ)

So far, we have seen that Monte Carlo methods use *full returns* g_t, while TD(0) uses a *one-step bootstrap*. Multi-step TD methods generalize between these two extremes by using n-step returns. However, a natural question arises: *can we combine information from all possible n-step returns in a principled way?*

This motivates TD(λ), which blends multi-step TD methods into a single algorithm using *eligibility traces*.

High-Level Intuition. TD(λ) introduces a parameter $\lambda \in [0, 1]$ that controls the weighting of n-step returns:

- $\lambda = 0$: reduces to TD(0), relying only on one-step bootstrapping.
- $\lambda = 1$: reduces to Monte Carlo, relying on full returns.
- $0 < \lambda < 1$: interpolates smoothly between these two extremes by averaging all n-step returns with exponentially decaying weights.

Formally, the λ-return is

$$g_t^{(\lambda)} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} g_t^{(n)}, \tag{2.13}$$

where $g_t^{(n)}$ is the n-step return defined in (2.11).

Remark. To make the λ-return well defined, we consider two cases.

Episodic Case: Well-posed. If an episode terminates at time T, let $N = T-t$ be the remaining steps. Then

$$\begin{aligned} g_t^{(\lambda)} &= (1-\lambda) \sum_{n=1}^{N-1} \lambda^{n-1} \, g_t^{(n)} \; + \; \lambda^{N-1} \, g_t^{(N)}, \\ &= (1-\lambda) \sum_{n=1}^{N} \lambda^{n-1} \, g_t^{(n)} \; + \; \lambda^{N} \, g_t^{(N)}, \end{aligned} \tag{2.14}$$

where $g_t^{(n)}$ is the n-step return (Eq. (2.11)) and $g_t^{(N)}$ is the *full* Monte Carlo return (Eq. (2.2)).

This expression is well-defined for all $\lambda \in [0, 1]$. Note that the weights form a convex combination:

$$(1-\lambda) \sum_{n=1}^{N-1} \lambda^{n-1} + \lambda^{N-1} = 1 - \lambda^{N-1} + \lambda^{N-1} = 1.$$

Continuing Case: Limit. Taking $\lambda \uparrow 1$ in (2.14) gives

$$\lim_{\lambda \uparrow 1} g_t^{(\lambda)} = g_t^{(N)} = g_t,$$

so the λ-return *reduces to the Monte Carlo return* at $\lambda = 1$. For continuing tasks (no terminal T), $\lambda = 1$ is conventionally defined by this same limiting argument, yielding the infinite-horizon discounted return when $\gamma < 1$.

Eligibility Traces. Naively computing $g_t^{(\lambda)}$ would require storing and combining infinitely many n-step returns, which is impractical. Instead, TD(λ) uses eligibility traces to implement this efficiently online.

An eligibility trace is a temporary record that tracks how much each state is "eligible" for updates based on how recently and frequently it has been visited. Specifically, for each state s, we maintain a trace $z_t(s)$ that evolves as

$$z_t(s) = \gamma \lambda z_{t-1}(s) + \mathbf{1}\{s_t = s\}, \tag{2.15}$$

where $\mathbf{1}\{s_t = s\}$ is an indicator that equals 1 if state s is visited at time t, and 0 otherwise.

TD(λ) Update Rule. At each time step t, we compute the TD error

$$\delta_t = r_t + \gamma \widehat{V}(s_{t+1}) - \widehat{V}(s_t),$$

as in (2.10). Then, for each state s, we update

$$\widehat{V}(s) \; \leftarrow \; \widehat{V}(s) + \alpha \, \delta_t \, z_t(s). \tag{2.16}$$

Thus, all states with nonzero eligibility traces are updated simultaneously, with the magnitude of the update determined by both the TD error and the eligibility trace. See Proposition 2.1 below for a justification.

Theoretical Guarantees.

1. In the tabular case, TD(λ) converges almost surely to the true value function V^π under the usual stochastic approximation conditions (sufficient exploration, decaying step sizes).
2. The parameter λ directly controls the bias–variance tradeoff:
   - Smaller λ: more bootstrapping, more bias but lower variance.
   - Larger λ: less bootstrapping, less bias but higher variance.
3. TD(λ) can be shown to converge to the fixed point of the λ-operator, which is itself a contraction mapping.

In summary, eligibility traces provide an elegant mechanism to combine the advantages of Monte Carlo and TD learning. TD(λ) introduces a spectrum of algorithms: at one end TD(0), at the other Monte Carlo, and in between a family of methods balancing bias and variance. In practice, intermediate values such as $\lambda \approx 0.9$ often work well.

Proposition 2.1 (Forward–Backward Equivalence)**.** *Consider one episode $s_0, a_0, r_0, \dots, s_T$ with $\hat{V}(s_T) = 0$. Let the **forward view** apply updates at the end of the episode:*

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha[g_t^{(\lambda)} - \hat{V}(s_t)], \quad t = 0, \dots, T-1,$$

where $g_t^{(\lambda)}$ is the λ-return in (2.13) *with the n-step returns $g_t^{(n)}$ from* (2.11)*, and where $\hat{V}$ is kept fixed while computing all $g_t^{(\lambda)}$.*

*Let the **backward view** run through the episode once, using the TD error δ_t from* (2.10) *and eligibility traces $z_t(s)$ from* (2.15)*, and then apply the cumulative update*

$$\Delta_{back}\hat{V}(s) \;=\; \alpha \sum_{t=0}^{T-1} \delta_t \, z_t(s).$$

Then, for every state s,

$$\Delta_{back}\hat{V}(s) \;=\; \alpha \sum_{t:\, s_t = s} [g_t^{(\lambda)} - \hat{V}(s_t)],$$

i.e., the net parameter change produced by (2.16) *equals that of the λ-return updates.*

Proof. Fix a state s. Using (2.15),

$$z_t(s) = \sum_{k=0}^{t} (\gamma\lambda)^{\,t-k}\, \mathbf{1}\{s_k = s\}.$$

Hence

$$\sum_{t=0}^{T-1} \delta_t z_t(s) = \sum_{t=0}^{T-1} \delta_t \sum_{k=0}^{t} (\gamma\lambda)^{\,t-k} \mathbf{1}\{s_k = s\} = \sum_{k:\, s_k = s} \sum_{t=k}^{T-1} (\gamma\lambda)^{\,t-k} \delta_t. \tag{1}$$

Write $\delta_t = r_t + \gamma \widehat{V}(s_{t+1}) - \widehat{V}(s_t)$ and split the inner sum:

$$\sum_{t=k}^{T-1} (\gamma\lambda)^{t-k} \delta_t = \underbrace{\sum_{t=k}^{T-1} \gamma^{t-k}\lambda^{t-k} r_t}_{\text{(A)}} + \underbrace{\sum_{t=k}^{T-1} \gamma^{t-k}\lambda^{t-k} (\gamma \widehat{V}(s_{t+1}) - \widehat{V}(s_t))}_{\text{(B)}}.$$

Term (B) telescopes. Shifting index in the first part of (B),

$$\sum_{t=k}^{T-1} \gamma^{t-k}\lambda^{t-k} \gamma \widehat{V}(s_{t+1}) = \sum_{t=k+1}^{T} \gamma^{t-k}\lambda^{t-1-k} \widehat{V}(s_t).$$

Therefore

$$\text{(B)} = -\widehat{V}(s_k) + \sum_{t=k+1}^{T-1} \gamma^{t-k}\lambda^{t-1-k}(1-\lambda)\widehat{V}(s_t) + \underbrace{\gamma^{T-k}\lambda^{T-1-k}\widehat{V}(s_T)}_{=\,0}. \tag{2}$$

Combining (A) and (2), and reindexing with $n = t - k$,

$$\sum_{t=k}^{T-1} (\gamma\lambda)^{t-k} \delta_t = -\widehat{V}(s_k) + \sum_{n=0}^{T-1-k} \gamma^n \lambda^n r_{k+n} + (1-\lambda) \sum_{n=1}^{T-1-k} \gamma^n \lambda^{n-1} \widehat{V}(s_{k+n}). \tag{3}$$

On the other hand, expanding the λ-return (2.13),

$$\begin{aligned} g_k^{(\lambda)} &= (1-\lambda) \sum_{n=1}^{T-k} \lambda^{n-1} \left(\sum_{m=0}^{n-1} \gamma^m r_{k+m} + \gamma^n \widehat{V}(s_{k+n}) \right) + \lambda^{T-k} g_k^{(T-k)} \\ &= \sum_{n=0}^{T-1-k} \gamma^n \lambda^n r_{k+n} + (1-\lambda) \sum_{n=1}^{T-1-k} \gamma^n \lambda^{n-1} \widehat{V}(s_{k+n}), \end{aligned} \tag{4}$$

where we used that $\widehat{V}(s_T) = 0$. Comparing (3) and (4) yields

$$\sum_{t=k}^{T-1} (\gamma\lambda)^{t-k} \delta_t = g_k^{(\lambda)} - \widehat{V}(s_k). \tag{5}$$

Substituting (5) into (1) and multiplying by α completes the proof. □

Example 2.1 (Policy Evaluation (MC and TD Family))**.** We consider the classic random-walk MDP with terminal states:

- **States:** $\{0, 1, 2, 3, 4, 5, 6\}$, where 0 and 6 are terminal; nonterminal states are 1:5.
- **Actions:** $\{-1, +1\}$ ("Left"/"Right").
- **Dynamics:** From a nonterminal state $s \in \{1, \dots, 5\}$, action -1 moves to $s-1$, and action $+1$ moves to $s+1$.
- **Rewards:** Transitioning into state 6 yields reward $+1$; all other transitions yield 0.
- **Discount:** $\gamma = 1$ (episodic task). Episodes start at state $s_0 = 3$ and terminate upon reaching $\{0, 6\}$.

We evaluate the *equiprobable policy* π that chooses Left/Right with probability 1/2 each at every nonterminal state. Under this policy, the true state-value function on nonterminal states $s \in \{1, \dots, 5\}$ is

$$V^{\pi}(s) \;=\; \frac{s}{6}. \tag{2.17}$$

We compare four *tabular policy-evaluation* methods:

1. **Monte Carlo (MC), first-visit** — using full returns as target.
2. **TD(0)** — one-step bootstrap.
3. ***n*-step TD** — here we use $n = 3$ (intermediate between MC and TD(0)).
4. **TD(λ)** — accumulating eligibility traces (we illustrate with $\lambda = 0.9$).

All methods estimate V^{π} from trajectories generated by π.

Error Metric. We report the *mean-squared error (MSE)* over nonterminal states after each episode:

$$\mathrm{MSE}_t \;=\; \frac{1}{5}\sum_{s=1}^{5} \big(\hat{V}_t(s) - V^{\pi}(s)\big)^2, \tag{2.18}$$

where V^{π} is given by (2.17). Curves are averaged over multiple random seeds.

Fixed Step Sizes. We first use a fixed step size $\alpha = 0.1$ for all methods. Fig. 2.1 shows the trajectories of MSE versus number of episodes. We can see that, when using a constant step size, these methods do not converge to exactly the true value function, but to a small neighborhood. In addition, if the algorithm initially decays very fast, then the final variance is larger. For example, MC initially decays very fast, but has a higher variance, whereas TD(0) initially decays slower, but has a lower final variance. This agrees with the theoretical analysis in (Kearns and Singh, 2000).

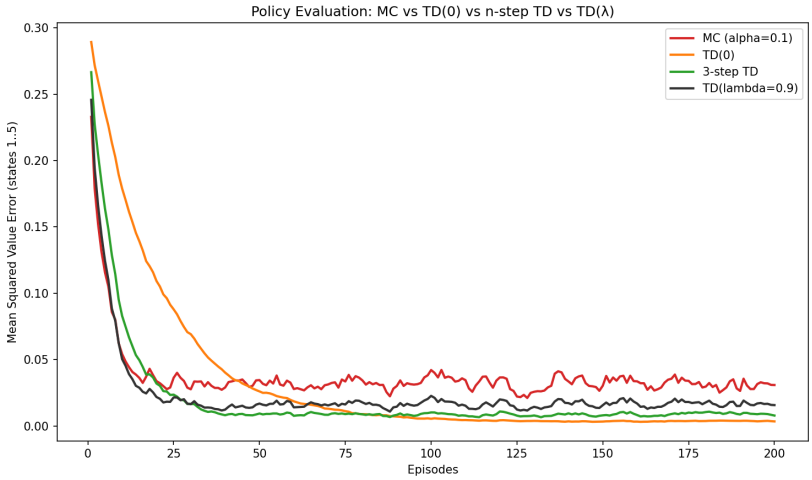

Figure 2.1: Policy Evaluation, MC versus TD Family, Fixed Step Size

Diminishing Step Sizes. We then use a diminishing step size for the TD family:

$$\alpha_t(s) \;=\; \frac{c}{\big(N_t(s)+t_0\big)^p}, \qquad \tfrac{1}{2} < p \le 1, \tag{2.19}$$

where $N_t(s)$ counts how many times $V(s)$ has been updated up to time t. A common choice is $p = 1$ with moderate $c > 0$ and $t_0 > 0$.

Fig. 2.2 shows the MSE versus episodes for MC, TD(0), 3-step TD, and TD(λ) under the diminishing step-size. Observe that all algorithms converge to the true value function under the diminishing step size schedule.

You are encouraged to play with the parameters of these algorithms in the code here.

2.1.2 Convergence Proof of TD Learning

Setup. Consider a tabular MDP with finite state space $\mathcal{S}$ and action space $\mathcal{A}$, and a discount factor $\gamma \in [0, 1)$. Assume the reward function is bounded, for example, $R(s, a) \in [0, 1]$ for any $(s, a) \in \mathcal{S} \times \mathcal{A}$. Let π be a stochastic policy and V^π be the true value function associated with π, the target we wish to estimate from interaction data. Denote

$$\mathcal{F}_t = \sigma(s_0, a_0, r_0, \dots, s_{t-1}, a_{t-1}, r_{t-1}),$$

as the σ algebra of all state-action-reward information up to time $t - 1$.

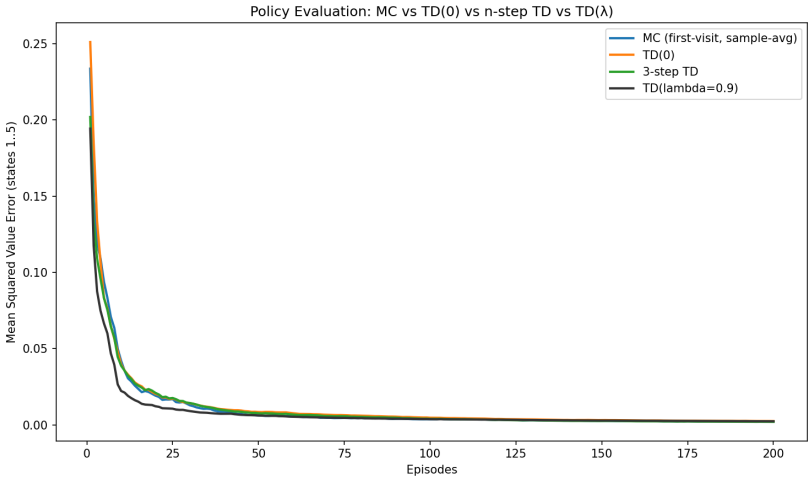

Figure 2.2: Policy Evaluation, MC versus TD Family, Diminishing Step Size

TD(0) Update. We maintain a tabular estimate V_t of the true value V^π. On visiting s_t and observing (s_t, a_t, r_t, s_{t+1}), the TD(0) algorithm performs

$$V_{t+1}(s_t) = V_t(s_t) + \alpha_t(s_t)\delta_t, \tag{2.20}$$

where δ_t is the TD error

$$\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t).$$

The update (2.20) only changes the value at s_t, leaving the value at other states unchanged.

Robbins–Monro Step Size. We assume the step size α satisfy the Robbins–Monro condition. That is, for any $s \in \mathcal{S}$:

$$\alpha_t(s) > 0, \quad \sum_{t:s_t=s} \alpha_t(s) = \infty, \quad \sum_{t:s_t=s} \alpha_t^2(s) < \infty.$$

Stationary Distribution. Assume the Markov chain over $\mathcal{S}$ induced by π is ergodic, then a unique stationary state distribution μ^π exists and satisfy:

$$\mu^\pi(s') = \sum_{s\in\mathcal{S}} \mu^\pi(s) \left(\sum_{a\in\mathcal{A}} \pi(a \mid s) P(s' \mid s, a) \right), \quad \forall s' \in \mathcal{S}. \tag{2.21}$$

If we denote

$$P^\pi(s' \mid s) = \sum_{a\in\mathcal{A}} \pi(a \mid s) P(s' \mid s, a), \tag{2.22}$$

as the π-induced state-only transition dynamics, then condition (2.21) is equivalent to

$$\mu^\pi(s') = \sum_{s \in \mathcal{S}} \mu^\pi(s) P^\pi(s' \mid s), \quad \forall s' \in \mathcal{S}. \tag{2.23}$$

See (2.47) for a generalization to continuous MDP.

Bellman Operator. For any $V : \mathcal{S} \to \mathbb{R}$, define the Bellman operator associated with π as $T^\pi V : \mathcal{S} \to \mathbb{R}$ by

$$(T^\pi V)(s) \;=\; \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \Big(R(s,a) + \gamma\, V(s') \Big). \tag{2.24}$$

We know that the operator T^π is a γ-contraction in $\|\cdot\|_\infty$. Hence it has a unique fixed point V^π satisfying $V^\pi = T^\pi V^\pi$.

The following theorem states the almost sure convergence of TD learning iterates to the true value function.

Theorem 2.1 (TD(0) Convergence (Tabular))**.** *Under the tabular MDP setup and assumptions above, the TD(0) iterates V_t generated by* (2.20) *converge almost surely to V^π.*

To prove this theorem, we need the following two lemmas.

Lemma 2.1 (Robbins-Siegmund Lemma)**.** *Let $(X_t)_{t \geq 0}$ be nonnegative and adapted to $(\mathcal{F}_t)$. Suppose there exist nonnegative $(\beta_t), (\gamma_t), (\xi_t)$ with $\sum_t \gamma_t < \infty$ and $\sum_t \xi_t < \infty$ such that*

$$\mathbb{E}[X_{t+1} \mid \mathcal{F}_t] \;\leq\; (1 + \gamma_t) X_t \;-\; \beta_t \;+\; \xi_t \qquad \textit{almost surely}$$

Then X_t converges almost surely to a finite random variable and $\sum_t \beta_t < \infty$ almost surely.

This lemma is from (Robbins and Siegmund, 1971).

Lemma 2.2. *Let μ^π be the stationary distribution in* (2.23)*, $D = \mathrm{diag}(\mu^\pi)$, and $w := V - V^\pi$. Then*

$$\langle w,\, D\,(T^\pi V - V) \rangle \;\leq\; -(1-\gamma)\, \|w\|_D^2,$$

where $\langle x, y \rangle = x^\top y$ and $\|w\|_D^2 = \sum_s \mu^\pi(s)\, w(s)^2$.

Proof. First, for any two value functions $V, U \in \mathbb{R}^{|\mathcal{S}|}$, we have

$$(T^\pi V)(s) - (T^\pi U)(s) = \gamma \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a)(V(s') - U(s)').$$

Therefore,

$$T^\pi V - T^\pi U = \gamma \widetilde{P}(V - U),$$

with

$$(\widetilde{P}u)(s) \coloneqq \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a)\, u(s'). \tag{2.25}$$

With this, we can write

$$\begin{aligned} T^\pi V - V &= T^\pi V - V^\pi + V^\pi - V \\ &= T^\pi V - T^\pi V^\pi - (V - V^\pi) \\ &= \gamma \widetilde{P}(V - V^\pi) - (V - V^\pi) \\ &= (\gamma \widetilde{P} - I)(V - V^\pi) \\ &= (\gamma \widetilde{P} - I) w. \end{aligned} \tag{2.26}$$

Thus,

$$\langle w, D(T^\pi V - V)\rangle = -\, w^\top D\,(I - \gamma \widetilde{P})\, w = -\|w\|_D^2 + \gamma\, \langle w, D\, \widetilde{P} w\rangle. \tag{2.27}$$

Next, we prove $\langle w, D\widetilde{P}w\rangle \le \|w\|_D^2$.

- First, we show $\|\widetilde{P}w\|_D \le \|w\|_D$. For any state $s \in \mathcal{S}$, from (2.25), we have

  $$(\widetilde{P}w)(s) = \sum_{s'} P^\pi(s' \mid s) w(s'),$$

  where $P^\pi(s' \mid s)$ is the π-induced state-only transition in (2.22). Since $P^\pi(\cdot \mid s)$ is a probability distribution, and $x \mapsto x^2$ is convex, we have

  $$((\widetilde{P}w)(s))^2 = \left(\sum_{s'} P^\pi(s' \mid s) w(s') \right)^2 \le \sum_{s'} P^\pi(s' \mid s) w^2(s').$$

  Therefore, we have

  $$\begin{aligned} \|\widetilde{P}w\|_D^2 &= \sum_s \mu^\pi(s) ((\widetilde{P}w)(s))^2 \\ &\le \sum_s \mu^\pi(s) \left(\sum_{s'} P^\pi(s' \mid s) w^2(s') \right) \\ &= \sum_{s'} \left(\sum_s \mu^\pi(s) P^\pi(s' \mid s) \right) w^2(s') \\ &= \sum_{s'} \mu^\pi(s') w^2(s') = \|w\|_D^2. \end{aligned} \tag{2.28}$$

  where the second-from-last equality holds because μ^π is the stationary distribution and satisfies (2.23).

- Second, we write

  $\langle w, D\widetilde{P}w\rangle = \langle D^{0.5}w, D^{0.5}\widetilde{P}w\rangle \le \|D^{0.5}w\| \cdot \|D^{0.5}\widetilde{P}w\| = \|w\|_D \cdot \|\widetilde{P}w\|_D \le \|w\|_D^2.$

Plugging this back to (2.27), we obtain

$$\langle w, D(T^\pi V - V)\rangle \le -\|w\|_D^2 + \gamma\|w\|_D^2,$$

proving the desired result in the lemma. □

We are now ready to prove Theorem 2.1.

Proof. **Step 1 (TD as stochastic approximation).** For the TD error

$$\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t),$$

we have the conditional expectation

$$\mathbb{E}[\delta_t \mid \mathcal{F}_t, s_t] = \sum_a \pi(a \mid s_t) \sum_{s'} P(s' \mid s_t, a)\Big(R(s_t,a)+\gamma V_t(s')\Big) - V_t(s_t) = (T^\pi V_t - V_t)(s_t).$$

Define the "noise":

$$\eta_{t+1} := \delta_t - \mathbb{E}[\delta_t \mid \mathcal{F}_t, s_t].$$

Then $\mathbb{E}[\eta_{t+1} \mid \mathcal{F}_t, s_t] = 0$ and the TD update is equivalent to

$$V_{t+1}(s_t) = V_t(s_t) + \alpha_t(s_t)\Big((T^\pi V_t - V_t)(s_t) + \eta_{t+1}\Big), \tag{2.29}$$

while learving all other coordinates unchanged. Because rewards are uniformly bounded, we know that V_t remains bounded. Hence, $\mathbb{E}[\eta_{t+1}^2 \mid \mathcal{F}_t, s_t]$ is uniformly bounded. Equation (2.29) shows that the TD update can be seen as a stochastic approximation to the Bellman operator (2.24).

Step 2 (Lyapunov drift). Let $D = \mathrm{diag}(\mu^\pi)$, a diagonal matrix whose diagonal entries are the probabilities in μ^π. Define the Lyapunov function

$$\mathcal{L}(V) = \frac{1}{2}\|V - V^\pi\|_D^2 = \frac{1}{2}\sum_s \mu^\pi(s)\,\big(V(s) - V^\pi(s)\big)^2.$$

Let $w_t := V_t - V^\pi$. Since only the s_t-coordinate changes at time t, we have

$$\begin{aligned}\mathcal{L}(V_{t+1}) - \mathcal{L}(V_t) &= \frac{1}{2}\mu^\pi(s_t)\Big((\underbrace{V_t(s_t) + \alpha_t\delta_t}_{V_{t+1}(s_t)} - V^\pi(s_t))^2 - (V_t(s_t) - V^\pi(s_t))^2\Big) \\ &= \mu^\pi(s_t)\,\alpha_t\,\delta_t\,w_t(s_t) \;+\; \frac{1}{2}\mu^\pi(s_t)\,\alpha_t^2\,\delta_t^2.\end{aligned} \tag{2.30}$$

Define $g_t := T^\pi V_t - V_t$. Taking conditional expectation given $\mathcal{F}_t$ and i.i.d. $s_t \sim \mu^\pi$,

$$\begin{aligned}\mathbb{E}[\mathcal{L}(V_{t+1}) - \mathcal{L}(V_t) \mid \mathcal{F}_t] &= \alpha_t\,\mathbb{E}[\mu^\pi(s_t)\,w_t(s_t)\,g_t(s_t) \mid \mathcal{F}_t] + \frac{1}{2}\alpha_t^2\,\mathbb{E}[\mu^\pi(s_t)\,\delta_t^2 \mid \mathcal{F}_t] \\ &= \alpha_t\sum_s \mu^\pi(s)\,w_t(s)\,g_t(s) \;+\; \frac{1}{2}\alpha_t^2\,C_t,\end{aligned} \tag{2.31}$$

where $C_t := \mathbb{E}[\mu^\pi(s_t)\,\delta_t^2 \mid \mathcal{F}_t]$ is finite because rewards are bounded and V_t stays bounded. Assume $C_t \le C$.

At the same time, by Lemma 2.2

$$\sum_s \mu^\pi(s)\, w_t(s)\, g_t(s) = \langle w_t, D g_t \rangle \le -(1-\gamma)\,\|w_t\|_D^2.$$

Plugging into (2.31) yields

$$\mathbb{E}[\mathcal{L}(V_{t+1}) \mid \mathcal{F}_t] \;\le\; \mathcal{L}(V_t) \;-\; \alpha_t\,(1-\gamma)\,\|w_t\|_D^2 \;+\; \frac{1}{2}\alpha_t^2\, C. \tag{2.32}$$

This is in Robbins–Siegmund form with

$$X_t := \mathcal{L}(V_t), \qquad \beta_t := (1-\gamma)\,\alpha_t\,\|w_t\|_D^2, \qquad \gamma_t := 0, \qquad \xi_t := \frac{1}{2} C\,\alpha_t^2.$$

We have $\sum_t \xi_t < \infty$ by $\sum_t \alpha_t^2 < \infty$. Therefore X_t converges a.s. and $\sum_t \beta_t < \infty$ a.s., which implies $\sum_t \alpha_t \|w_t\|_D^2 < \infty$. Since $\sum_t \alpha_t = \infty$, it must be that $\liminf_t \|w_t\|_D = 0$.
Finally, using (2.32) again and the continuity of the drift, one shows that any subsequential limit of V_t must satisfy $T^\pi V - V = 0$; by uniqueness of the fixed point, the only possible limit is V^π. Hence $V_t \to V^\pi$ almost surely. □

2.1.3 On-Policy Control

Monte Carlo (MC) estimation and the TD family evaluate policies directly from interaction—no model required. We now turn evaluation into control via generalized policy iteration (GPI): repeatedly (i) evaluate the current policy from data and (ii) improve it by acting greedily with respect to the new estimates. We first cover on-policy control methods, which estimate and improve the same (typically ε-greedy) policy, and then off-policy methods, which learn about a target policy while behaving with a different one.

2.1.3.1 Monte Carlo Control

High-level Intuition.

- **Goal.** Learn an (approximately) optimal policy by alternating *policy evaluation* and *policy improvement* using only sampled episodes.
- **Why action-values?** Estimating $Q^\pi(s,a)$ lets us improve the policy *without a model* by choosing "$\arg\max_a Q(s,a)$".

- **Exploration.** Pure greedy improvement can get stuck. MC control keeps the policy ε-*soft* (e.g., ε-greedy) so that every action has nonzero probability and all state-action pairs continue to be sampled. An ε-soft policy is one that never rules out any action: in every state s, each action a gets at least a small fraction of probability. Formally, in the tabular setup, we have that a policy π is ε-soft if and only if

$$\forall s, \forall a : \quad \pi(a \mid s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}, \quad \varepsilon \in (0, 1], \tag{2.33}$$

  where $\mathcal{A}(s)$ denotes the set of actions the agent can select at state s.

- **Coverage mechanisms.** Classic guarantees use either:

  1) **Exploring starts (ES):** start each episode from a randomly chosen (s, a) with nonzero probability; or

  2) **ε-soft / GLIE (Greedy in the Limit with Infinite Exploration):** use ε-greedy behavior with $\varepsilon_t \downarrow 0$ so every (s, a) is visited infinitely often while the policy becomes greedy in the limit.

Algorithmic Form. We maintain tabular action-value estimates $Q(s, a)$ and an ε-*soft* policy π (ε-greedy w.r.t. Q). After each episode we update Q from *empirical returns* and then improve π.

Return from time t:

$$g_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t} r_T = \sum_{k=0}^{T-t} \gamma^k r_{t+k}.$$

First-visit MC update (common choice):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_{N(s_t, a_t)}(g_t - Q(s_t, a_t)), \tag{2.34}$$

applied only on the first occurrence of (s_t, a_t) in the episode. *Sample-average* learning uses $\alpha_n = 1/n$ per pair; more generally, use diminishing stepsizes.

Policy improvement (ε-greedy):

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \dfrac{\varepsilon}{|\mathcal{A}(s)|}, & a \in \arg\max_{a'} Q(s, a'), \\ \dfrac{\varepsilon}{|\mathcal{A}(s)|}, & \text{otherwise.} \end{cases} \tag{2.35}$$

Theoretical Guarantees.

Assume a tabular episodic MDP and $\gamma \in [0, 1)$.

- **Convergence with Exploring Starts.** If every state–action pair has nonzero probability of being the *first* pair of an episode (using ES), and each $Q(s,a)$ is updated toward the true mean return from (s,a) (e.g., via sample averages), then repeated policy evaluation and greedy improvement converge with probability 1 to an optimal deterministic policy. (If one uses an ε-greedy improvement, then it converges to an optimal ε-soft policy.)
- **Convergence with ε-soft GLIE behavior.** If the behavior policy is GLIE—every (s,a) is visited infinitely often and $\epsilon_t \to 0$—and the step-sizes for each (s,a) satisfy the Robbins–Monro conditions $\sum_t \alpha_t(s,a) = \infty, \sum_t \alpha_t(s,a)^2 < \infty$, then $Q(s,a)$ converges to $Q^\star(s,a)$ for all pairs visited infinitely often, and the ε-greedy policy converges almost surely to an optimal policy.

Remark. **Unbiased but high-variance.** MC targets g_t are unbiased estimates of action values under the current policy, but can have high variance—especially for long horizons—so convergence can be slower than TD methods. Keeping $\varepsilon > 0$ ensures exploration but limits asymptotic optimality to the best ε-soft policy; hence $\varepsilon_t \downarrow 0$ (GLIE) is recommended for optimality.

2.1.3.2 SARSA (On-Policy TD Control)

High-level Intuition.

- **Goal.** Turn evaluation into control by updating action values online and improving the same policy that generates data.
- **Key idea.** Replace Monte Carlo returns with a bootstrapped target. After taking action a_t in state s_t and observing r_t, s_{t+1}, sample the next action a_{t+1} from the current policy and update toward $r_t + \gamma Q(s_{t+1}, a_{t+1})$.
- **On-policy nature.** SARSA evaluates the behavior policy itself, typically an ε-greedy policy w.r.t. Q.
- **Exploration.** Use ε-soft behavior so every action keeps nonzero probability. For optimality, let $\varepsilon_t \downarrow 0$ to obtain GLIE (Greedy in the Limit with Infinite Exploration).

Algorithmic Form.

Let Q be a tabular action-value function and π_t be ε_t-greedy w.r.t. Q_t.

TD target and error:

$$y_t = r_t + \gamma Q(s_{t+1}, a_{t+1}), \qquad \delta_t = y_t - Q(s_t, a_t). \tag{2.36}$$

SARSA update (one-step):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(s_t, a_t)\, \delta_t. \tag{2.37}$$

ε-greedy policy improvement:

$$\pi_{t+1}(a \mid s) = \begin{cases} 1 - \varepsilon_{t+1} + \dfrac{\varepsilon_{t+1}}{|\mathcal{A}(s)|}, & a \in \arg\max_{a'} Q_{t+1}(s, a'), \\ \dfrac{\varepsilon_{t+1}}{|\mathcal{A}(s)|}, & \text{otherwise.} \end{cases} \tag{2.38}$$

Variants.

- **Expected SARSA** replaces the sampled a_{t+1} by its expectation under π_t for lower variance:

$$y_t = r_t + \gamma \sum_a \pi_t(a \mid s_{t+1}) Q(s_{t+1}, a). \tag{2.39}$$

- **n-step SARSA** and **SARSA(λ)** blend multi-step targets; these trade bias and variance similarly to MC vs TD.

Convergence Guarantees.

Assume a finite MDP, $\gamma \in [0, 1)$, asynchronous updates, and that each state–action pair is visited infinitely often.

- **GLIE convergence to optimal policy.** If the behavior is GLIE, i.e., $\varepsilon_t \downarrow 0$ while ensuring infinite exploration, and stepsizes satisfy the Robbins–Monro conditions, then $Q_t \to Q^\star$ almost surely and the ε_t-greedy behavior becomes greedy in the limit, yielding an optimal policy almost surely.

2.1.4 Off-Policy Control

Off-policy methods learn about a *target* policy π while following a (potentially different) *behavior* policy b to gather data. This decoupling is useful when:

- you want to *reuse logged data* collected by some b (e.g., a rule-based controller or a past system),
- you need *safer exploration* by restricting behavior b while aiming to evaluate or improve a different π,
- you want to learn about the *greedy* policy without executing it, which motivates algorithms like Q-learning.

In this section we first cover off-policy policy evaluation with *importance sampling*, then show how it can be used to construct an off-policy *Monte Carlo control* scheme in the tabular case. Finally, we present Q-learning.

2.1.4.1 Importance Sampling for Policy Evaluation

Motivation. Suppose we have episodes generated by a behavior policy b, but we want the value of a different target policy π. For a state value this is $V^\pi(s) = \mathbb{E}_\pi[g_t \mid s_t = s]$, and for action values $Q^\pi(s,a) = \mathbb{E}_\pi[g_t \mid s_t = s, a_t = a]$, where

$$g_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k}.$$

Because the data come from b, the naive sample average is biased. Importance sampling (IS) reweights returns so that expectations under b equal those under π.

A basic *support condition* is required:

$$\text{If } \pi(a \mid s) > 0 \text{ then } b(a \mid s) > 0 \quad \text{for all visited } (s,a). \tag{2.40}$$

This ensures that π is absolutely continuous with respect to b on the experienced trajectories.

Importance Sampling (episode-wise). Consider a trajectory starting at time t:

$$\tau_t = (s_t, a_t, r_t, s_{t+1}, a_{t+1}, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T).$$

The probability of observing this trajectory conditioned on $s_t = s$, under policy π, is

$$\mathbb{P}_\pi[\tau_t \mid s_t = s] = \pi(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t) \pi(a_{t+1} \mid s_{t+1}) \cdots \pi(a_{T-1} \mid s_{T-1}) P(s_T \mid s_{T-1}, a_{T-1}).$$

The probability of observing the same trajectory conditioned on $s_t = s$, under policy b, is

$$\mathbb{P}_b[\tau_t \mid s_t = s] = b(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t) b(a_{t+1} \mid s_{t+1}) \cdots b(a_{T-1} \mid s_{T-1}) P(s_T \mid s_{T-1}, a_{T-1}).$$

Since the return g_t is a deterministic function of τ_t, i.e., applying the reward function R to state-action pairs, we have that

$$\begin{aligned} V^\pi(s) = \mathbb{E}_\pi[g_t \mid s_t = s] &= \sum_{\tau_t} g_t \mathbb{P}_\pi[\tau_t \mid s_t = s] \\ &= \sum_{\tau_t} g_t \mathbb{P}_b[\tau_t \mid s_t = s] \left(\frac{\mathbb{P}_\pi[\tau_t \mid s_t = s]}{\mathbb{P}_b[\tau_t \mid s_t = s]} \right) \\ &= \sum_{\tau_t} \left(\frac{\pi(a_t \mid s_t) \pi(a_{t+1} \mid s_{t+1}) \cdots \pi(a_{T-1} \mid s_{T-1})}{b(a_t \mid s_t) b(a_{t+1} \mid s_{t+1}) \cdots b(a_{T-1} \mid s_{T-1})} \right) g_t \mathbb{P}_b[\tau_t \mid s_t = s] \end{aligned} \tag{2.41}$$

Therefore, define the *likelihood ratio*

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(a_k \mid s_k)}{b(a_k \mid s_k)}, \tag{2.42}$$

we have

$$V^{\pi}(s) = \mathbb{E}_b \left[\rho_{t:T-1} g_t \mid s_t = s \right]. \tag{2.43}$$

Similarly, we have

$$Q^{\pi}(s,a) = \mathbb{E}_b [\rho_{t:T-1} g_t \mid s_t = s, a_t = a]. \tag{2.44}$$

Given n episodes, the ordinary IS estimator for Q^{π} at the first visit of (s,a) is

$$\hat{Q}_n^{\mathrm{IS}}(s,a) = \frac{1}{N_n(s,a)} \sum_{i=1}^{n} \mathbf{1}\{(s,a) \text{ visited}\} \, \rho_{t_i:T_i-1}^{(i)} \, g_{t_i}^{(i)},$$

where $N_n(s,a)$ counts the number of first visits of (s,a). In words, to estimate the Q value of the target policy π using trajectories of the behavior policy b, we need to reweight the return g_t by the likelihood ratio $\rho_{t:T-1}$. Note that the likelihood ratio does not require knowledge about the transition dynamics.

Algorithmic Form: Off-policy Monte Carlo Policy Evaluation.

Input: behavior b, target π, episodes from b
For each episode $(s_0, a_0, r_0, s_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$:

1. For $t = T-1, \ldots, 0$ compute episode-wise likelihood ratio $\rho_{t:T-1}$ and return g_t,
2. For first visits of (s_t, a_t), update

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha_{N(s_t,a_t)} (\rho_{t:T-1} g_t - Q(s_t,a_t)).$$

   Use sample averages $\alpha_n = 1/n$ or Robbins-Monro stepsizes.

Guarantees. Under the support condition and finite variance assumptions, ordinary IS is *unbiased* and converges almost surely to Q^{π}.

2.1.4.2 Off-Policy Monte Carlo Control

High-level Intuition. We wish to improve a target policy π toward optimality while behaving with a different exploratory policy b. We evaluate Q^{π} off-policy using IS on data from b, then set π greedy with respect to the updated Q. Keep b sufficiently exploratory (for coverage), for example ε-greedy with a fixed $\varepsilon > 0$ or a GLIE schedule.

Algorithmic Form.

1. Initialize $Q(s,a)$ arbitrarily. Set target π to be greedy w.r.t. Q. Choose an exploratory behavior b that ensures coverage, e.g., ε-greedy w.r.t. Q with $\varepsilon > 0$.

2. Loop over iterations $i = 0, 1, 2, ...$:

    1. Data collection under b: generate a batch of episodes using b.
    2. Off-policy evaluation of π: for each episode, compute IS targets for first visits of (s_t, a_t) and update Q using either ordinary IS
    3. Policy improvement: set for all states

$$\pi_{i+1}(s) \in \arg\max_a Q(s, a).$$

    4. Optionally update b to remain exploratory, for example $b \leftarrow \varepsilon$-greedy w.r.t. Q with a chosen ε or a GLIE decay.

Convergence Guarantees.

- **Evaluation step:** With the support condition and appropriate stepsizes, off-policy MC prediction converges almost surely to Q^π when using ordinary IS.
- **Control in the batch GPI limit:** If each evaluation step produces estimates that converge to the exact Q^{π_i} before improvement, then by the policy improvement theorem the sequence of greedy target policies π_i converges to an optimal policy in finite MDPs.

Remark. **Choice of b.** A common and simple choice is an ε-greedy behavior b w.r.t. current Q that maintains $\varepsilon > 0$ for coverage or uses GLIE so that $\varepsilon_t \downarrow 0$ while all pairs are still visited infinitely often.

2.1.4.3 Q-Learning

High-Level Intuition.

- **What it learns.** Q-Learning seeks the fixed point of the Bellman optimality operator

$$(\mathcal{T}^\star Q)(s, a) = \mathbb{E}\big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') \mid s_t = s, a_t = a\big],$$

  whose unique fixed point is $Q^\star$. Because $\mathcal{T}^\star$ is a γ-contraction in $\|\cdot\|_\infty$, repeatedly applying it converges to $Q^\star$ in the tabular case.
- **Why off-policy.** We can behave with any sufficiently exploratory policy b (e.g., ε-greedy w.r.t. current Q) but learn from the greedy target $\max_{a'} Q(s', a')$. No importance sampling is needed.

Algorithmic Form. Let Q be a tabular action-value function. At each step observe a transition (s_t, a_t, r_t, s_{t+1}) generated by a behavior policy b_t (typically ε_t-greedy w.r.t. Q_t).

- **Target and TD error**

$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'), \qquad \delta_t = y_t - Q(s_t, a_t).$$

- **Update**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(s_t, a_t)\, \delta_t.$$

- **Behavior (exploration)** Use ε_t-greedy with ε_t decaying (GLIE) or any scheme that ensures each (s, a) is updated infinitely often.

Convergence. In a finite MDP with $\gamma \in [0, 1)$, if each (s, a) is updated infinitely often (sufficient exploration) and stepsizes satisfy Robbins-Monro conditions, then Q-Learning converges to $Q^\star$ with probability 1.

2.1.4.4 Double Q-Learning

Motivation. Max operators tend to be optimistically biased when action values are noisy. Consider an example where in state s one can take two actions 1 and 2. The estimated Q function $\widehat{Q}(s, \cdot)$ has two values $+1$ and -1 with equal probability. In this case we have $Q(s, 1) = Q(s, 2) = \mathbb{E}[\widehat{Q}(s, \cdot)] = 0$. Therefore, $\max Q(s, a) = 0$. However, the noisy estimated $\widehat{Q}(s, \cdot)$ has four outcomes with equal probabilities:

$$(+1, -1), (+1, +1), (-1, +1), (-1, -1).$$

Therefore, we have

$$\mathbb{E}[\max_a \widehat{Q}(s, a)] = \frac{1}{4}(1 + 1 + 1 - 1) = 1/2 > \max_a Q(s, a),$$

which overestimates the max Q value. In general, we have

$$\mathbb{E}[\max_a \widehat{Q}(s, a)] \geq \max_a \mathbb{E}[\widehat{Q}(s, a)] = \max_a Q(s, a),$$

where the estimates $\widehat{Q}$ are noisy (try to prove this on your own). In Q-Learning the target

$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$$

can therefore overestimate action values and slow learning or push policies toward risky actions.

Double Q-Learning reduces this bias by decoupling selection from evaluation: maintain two independent estimators Q^A and Q^B. Use one to select the greedy action and the other to evaluate it, and alternate which table you update. This weakens the statistical coupling that creates overestimation.

Algorithmic Form. Keep two tables Q^A, Q^B. Use an ε-greedy behavior policy with respect to a combined estimate, e.g., $Q^{\text{avg}} = \frac{1}{2}(Q^A + Q^B)$ or $Q^A + Q^B$.

At each step observe (s_t, a_t, r_t, s_{t+1}). With probability 1/2 update Q^A, else update Q^B.

- **Update Q^A:**

$$a^\star = \arg\max_{a'} Q^A(s_{t+1}, a'), \qquad y_t = r_t + \gamma\, Q^B(s_{t+1}, a^\star),$$

$$Q^A(s_t, a_t) \leftarrow Q^A(s_t, a_t) + \alpha_t(s_t, a_t)[y_t - Q^A(s_t, a_t)].$$

- **Update Q^B:**

$$a^\star = \arg\max_{a'} Q^B(s_{t+1}, a'), \qquad y_t = r_t + \gamma\, Q^A(s_{t+1}, a^\star),$$

$$Q^B(s_t, a_t) \leftarrow Q^B(s_t, a_t) + \alpha_t(s_t, a_t)[y_t - Q^B(s_t, a_t)].$$

- **Behavior policy (ε-greedy):** choose $a_t \sim \varepsilon$-greedy with respect to $Q^{\text{avg}}(s_t, \cdot)$. A GLIE schedule $\varepsilon_t \downarrow 0$ is standard.
- **Acting and planning:** for greedy actions or plotting a single estimate, use $Q^{\text{avg}} = \frac{1}{2}(Q^A + Q^B)$.

Convergence.

- **Tabular setting.** In a finite MDP with $\gamma \in [0, 1)$, bounded rewards, sufficient exploration so that every (s, a) is updated infinitely often, and Robbins–Monro stepsizes for each pair. Double Q-Learning converges with probability 1 to $Q^\star$.

Example 2.2 (Value-based RL for Grid World)**.** Consider the following 5×5 grid with $(0, 4)$ being the goal and the terminal state. At every state, the agent can take four actions: left, right, up, and down. There is a wall in the gray area shown in Fig. 2.3. Upon hitting the wall, the agent stays in the original cell. Every action incurs a reward of -1. Once the agent arrives at the goal state, reward stays at 0.

We run Generalized Policy Iteration (GPI) with Monte Carlo (on-policy), SARSA, Expected SARSA, Q-Learning, and Double Q-Learning on this problem with diminishing learning rates.

Fig. 2.4 plots the error between the estimated Q values (of different algorithms) and the ground-truth optimal Q value (obtained from value iteration with known transition dynamics). Except Monte Carlo control which converges slowly, the other methods converge fast.

From the final estimated Q value, we can extract a greedy policy, visualized below.

You can play with the code here.

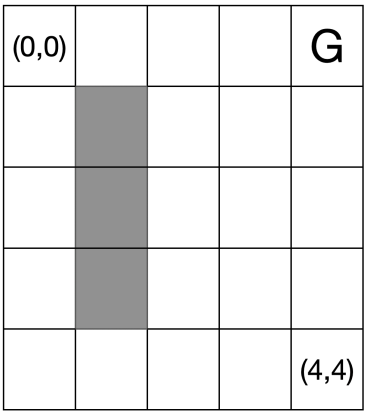

Figure 2.3: Grid World

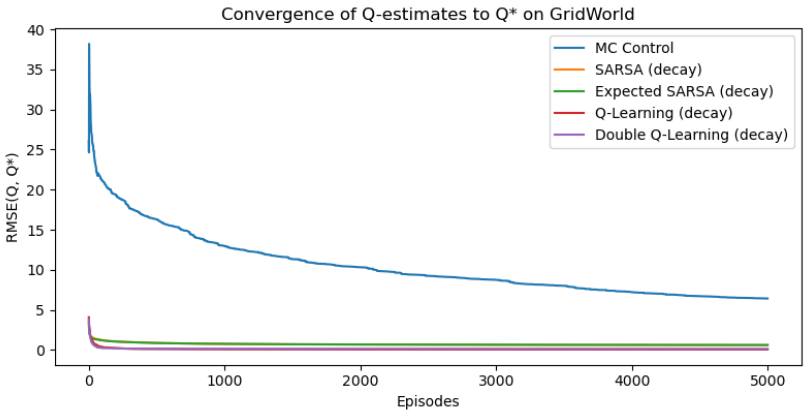

Figure 2.4: Convergence of Estimated Q Values.

```
MC Control:
 > > > > G
^ # ^ ^ ^
v # ^ ^ ^
v # > ^ ^
> > > > ^
SARSA:
 > > > > G
^ # > > ^
^ # ^ ^ ^
^ # ^ ^ ^
> > ^ ^ ^
Expected SARSA:
 > > > > G
^ # > > ^
^ # ^ ^ ^
^ # > > ^
> > ^ ^ ^
Q-Learning:
 > > > > G
^ # ^ ^ ^
^ # ^ ^ ^
^ # ^ ^ ^
> > ^ ^ ^
Double Q-Learning:
 > > > > G
^ # > ^ ^
^ # > ^ ^
^ # ^ ^ ^
> > > > ^
```

2.2 Function Approximation

Many reinforcement learning problems have continuous state spaces–think of mechanical systems like robot arms, legged locomotion, drones, and autonomous vehicles. In these domains the state s (e.g., joint angles/velocities, poses) lives in $\mathbb{R}^n$, which makes a tabular representation of the value functions impossible. In this case, we must approximate values with parameterized functions.

2.2.1 Basics of Continuous MDP

In a continuous MDP, at least one of the state space or the action space is a continuous space. Suppose $\mathcal{S} \subseteq \mathbb{R}^n$ and $\mathcal{A} \subseteq \mathbb{R}^m$ are both continuous spaces.

The environment kernel $P(\cdot \mid s, a)$ is a Markov kernel from $\mathcal{S} \times \mathcal{A}$ to $\mathcal{S}$: for each state-action pair (s, a), $P(\cdot \mid s, a)$ is a probability measure on $\mathcal{S}$. For each Borel set $B \subseteq \mathcal{S}$, the map $(s, a) \mapsto P(B \mid s, a)$ is measurable. For example, $P(\mathcal{S} \mid s, a) = 1$ for any (s, a).

The policy kernel $\pi(\cdot \mid s)$ is a stochastic kernel from $\mathcal{S}$ to $\mathcal{A}$: for each s, $\pi(\cdot \mid s)$ is a probability measure on $\mathcal{A}$.

Induced State-Transition Kernel. For notational convenience, given a policy and the environment kernel P, we define a state-only Markov kernel

$$P^\pi(B \mid s) := \int_{\mathcal{A}} P(B \mid s, a)\pi(da \mid s), \quad B \subseteq \mathcal{S}. \tag{2.45}$$

In words, $P^\pi(B \mid s)$ measures the probability of landing at a set B starting from state s, under all actions possible for the policy π.

If densities exist, i.e., $P(ds' \mid s, a) = p(s' \mid s, a)ds'$ and $\pi(da \mid s) = \pi(a \mid s)da$, then,

$$p^\pi(s' \mid s) := \int_{\mathcal{A}} p(s' \mid s, a)\pi(a \mid s)da \quad \text{and} \quad P^\pi(ds' \mid s) = p^\pi(s' \mid s)ds'. \tag{2.46}$$

Stationary State Distribution. A probability measure μ^π on $\mathcal{S}$ is called *stationary* for the state-transition kernel P^π if and only if

$$\mu^\pi(B) = \int_{\mathcal{S}} P^\pi(B \mid s)\mu^\pi(ds), \quad \forall B \subseteq \mathcal{S}. \tag{2.47}$$

If a density $\mu^\pi(s)$ exists, then the above equation is the followng condition

$$\mu^\pi(s') = \int_{\mathcal{S}} p^\pi(s' \mid s)\mu^\pi(s)ds. \tag{2.48}$$

In words, the state distribution μ^π does not change under the state-transition kernel P^π (e.g., if a state A has probability 0.1 of being visited at time t, the probability of visiting A in the next time step remains 0.1, under policy π). Under standard ergodicity assumptions, this stationary state distribution μ^π exists and is unique (after sufficient steps, the initial state distribution does not matter and the state distribution follows μ^π). Moreover, the empirical state distribution converge to μ^π.

2.2.2 Policy Evaluation

For simplicity, let us first relax the state space to be a continuous space $\mathcal{S} \subseteq \mathbb{R}^n$. We assume the action space $\mathcal{A}$ is still finite with $|\mathcal{A}|$ elements. We first consider the problem of policy evaluation, i.e., estimate the value functions associated with a policy π from interaction data with the environment.

Bellman Consistency. Given a policy $\pi : \mathcal{S} \mapsto \Delta(\mathcal{A})$, its associated state-value function V^π must satisfy the following Bellman Consistency equation

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a \mid s) \left[R(s,a) + \gamma \int_{\mathcal{S}} V(s') P(ds' \mid s,a) \right]. \tag{2.49}$$

Notice that since $\mathcal{S}$ is a continuous space, we need to replace "$\sum_{s'\in\mathcal{S}}$" with "$\int_{\mathcal{S}}$". If $P(ds' \mid s,a)$ has a density $p(s' \mid s,a)$, the above Bellman consistency equation also reads

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a \mid s) \left[R(s,a) + \gamma \int_{\mathcal{S}} V(s') p(s' \mid s,a) ds' \right]. \tag{2.50}$$

Bellman Operator. Define the Bellman operator T^π acting on any bounded measurable function $V : \mathcal{S} \to \mathbb{R}$ by

$$(T^\pi V)(s) = \sum_{a\in\mathcal{A}} \pi(a \mid s) \left[R(s,a) + \gamma \int_{\mathcal{S}} V(s') P(ds' \mid s,a) \right]. \tag{2.51}$$

Then V^π is the unique fixed point of T^π, i.e., $V^\pi = T^\pi V^\pi$. Moreover, when rewards are uniformly bounded and $\gamma \in [0,1)$, T^π is a γ-contraction under the sup-norm and is monotone.

Approximate Value Function. In large/continuous state spaces we restrict attention to a parametric family $V(\cdot;\theta) : \theta \in \mathbb{R}^d$ and learn θ from data. We use $\nabla_\theta V(s;\theta) \in \mathbb{R}^d$ to denote the gradient of V with respect to θ at state s.

A special and very important case is linear function approximation

$$V(s;\theta) = \theta^\top \phi(s), \tag{2.52}$$

where $\phi(s) = [\phi_1(s), \dots, \phi_d(s)]^\top$ are fixed basis functions (e.g., neural network last-layer features). When $V(s;\theta) = \theta^\top \phi(s)$, we have

$$\nabla_\theta V(s;\theta) = \phi(s).$$

When we restrict the value function to a function class (e.g., linear features or a neural network), it is generally not guaranteed that the unique fixed point of the Bellman operator (2.51), namely V^π, belongs to that class. This misspecification (or realizability gap) means we typically cannot recover V^π exactly; instead, we seek its *best approximation* according to a chosen criterion.

2.2.2.1 Monte Carlo Estimation

Given an episode $(s_t, a_t, r_t, \dots, s_T)$ collected by policy π, its discounted return

$$g_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t} r_T$$

is an unbiased estimate of the value at s_t, i.e., $V^\pi(s_t)$.

Therefore, Monte Carlo estimation follows the intuitive idea to make the approximate value function $V(\cdot, \theta)$ fit the returns from these episodes as close as possible:

$$\min_\theta \frac{1}{\mathcal{D}} \sum_{t \in \mathcal{D}} \frac{1}{2}(g_t - V(s_t, \theta))^2, \tag{2.53}$$

where $\mathcal{D}$ denotes the dataset of episodes collected under policy π. The formulation (2.53) is a *batch* formulation in the sense that it waits until all episodes are collected before performing the optimization.

In an online formulation, we can optimize after every episode the objective function

$$\min_\theta \frac{1}{2}(g_t - V(s_t, \theta))^2,$$

which leads to one step of gradient descent:

$$\theta \quad \leftarrow \quad \theta + \alpha_t(g_t - V(s_t; \theta))\nabla_\theta V(s_t; \theta). \tag{2.54}$$

To connect the above update back to the MC update (2.4) in the tabular case, we see that the term $g_t - V(s_t; \theta$ is similar as before the difference between the target and the current estimate. However, in the case of function approximation, the error is multiplied by the gradient $\nabla_\theta V(s_t; \theta)$.

It is worth noting that when using function approximation, the update on θ caused by one episode $(s_t, \dots)$ will affect the values at all other states even if the policy only visited state s_t.

Convergence Guarantees. Assume on-policy sampling under π, bounded rewards, and step sizes α_t satisfying Robbins–Monro conditions.

- For *linear* $V(s; \theta) = \theta^\top \phi(s)$ with full-rank features, i.e.,

  $$\mathbb{E}_{s \sim \mu^\pi}\left[\phi(s)\phi(s)^\top\right] \succ 0,$$

  and $\mathbb{E}_{s \sim \mu^\pi}\|\phi(s)\|^2 < \infty$, the iterates converge almost surely to the unique global minimizer of the convex objective

  $$\theta^\star_{\mathrm{MC}} \in \arg\min_\theta \ \frac{1}{2}\mathbb{E}_{s_t \sim \mu^\pi}\left[\left(V(s_t; \theta) - V^\pi(s_t)\right)^2\right], \tag{2.55}$$

  where the expectation is with respect to stationary state distribution μ^π under π.

- For *nonlinear* differentiable function classes with bounded gradients, the iterates converge almost surely to a stationary point of the same objective.

– Correction: Since Monte Carlo Estimation can be seen as performing Stochastic Gradient Descent on the objective in (2.55), to guarantee convergence to a first-order stationary point, we need some technical conditions: (a) diminishing step sizes satisfying the Robbins-Monro condition; (b) bounded second-order moment of the stochastic gradient; and (c) L-smoothness of the objective.

2.2.2.2 Semi-Gradient TD(0)

We know from previous discussion that MC uses the full return g_t as the target and thus can have high variance. A straightforward idea is to replace the MC target g_t in the update (2.54) by the one-step bootstrap target

$$r_t + \gamma V(s_{t+1}; \theta),$$

which yields the *semi-gradient TD(0)* update

$$\theta \;\leftarrow\; \theta \;+\; \alpha_t \left(r_t + \gamma V(s_{t+1}; \theta) - V(s_t; \theta)\right) \nabla_\theta V(s_t; \theta). \tag{2.56}$$

(At terminal s_{t+1}, use $V(s_{t+1}; \theta) = 0$ or equivalently set $\gamma = 0$ for that step.)

Why call it "semi-gradient"? Let the TD error be

$$\delta_t(\theta) \;:=\; r_t + \gamma V(s_{t+1}; \theta) - V(s_t; \theta).$$

Consider the per-sample squared TD error objective

$$\min_\theta \; \frac{1}{2}\, \delta_t(\theta)^2.$$

Its **true gradient** (a.k.a. the *residual gradient*) is

$$\nabla_\theta \frac{1}{2} \delta_t(\theta)^2 \;=\; \delta_t(\theta) \left(\gamma \nabla_\theta V(s_{t+1}; \theta) - \nabla_\theta V(s_t; \theta)\right).$$

Thus a **true-gradient (residual-gradient) TD(0)** step would be

$$\theta \;\leftarrow\; \theta \;-\; \alpha_t\, \delta_t(\theta) \left(\gamma \nabla_\theta V(s_{t+1}; \theta) - \nabla_\theta V(s_t; \theta)\right). \tag{2.57}$$

By contrast, the semi-gradient TD(0) step in (2.56) ignores the dependence of the target on θ (i.e., it drops the $\gamma \nabla_\theta V(s_{t+1}; \theta)$ term) and treats the target $r_t + \gamma V(s_{t+1}; \theta)$ as a *constant* when differentiating. Concretely,

$$\nabla_\theta \frac{1}{2} \big(\text{target} - V(s_t; \theta)\big)^2 \;\approx\; -\big(\text{target} - V(s_t; \theta)\big)\, \nabla_\theta V(s_t; \theta).$$

This approximation yields the simpler update (2.56).

Convergence Guarantees. When using linear approximation, the Monte Carlo estimator converges to $\theta^\star_{\text{MC}}$ in (2.55). We now study what the semi-gradient TD(0) updates (2.56) converge to.

Projected Bellman Operator. Fix a weighting/visitation distribution μ on $\mathcal{S}$ (e.g., the stationary distribution μ^π) and the associated inner product

$$\langle f, g\rangle_\mu := \mathbb{E}_{s\sim\mu}[\, f(s)g(s)\,], \qquad \|f\|_\mu := \sqrt{\langle f, f\rangle_\mu}.$$

Let $\mathcal{V} := \{V(s;\theta) = \theta^\top\phi(s) \;:\; \theta \in \mathbb{R}^d\}$ be the linear function class spanned by features $\phi : \mathcal{S} \to \mathbb{R}^d$. The *$\mu$-orthogonal projection* $\Pi_\mu : \mathcal{F} \to \mathcal{V}$ is

$$\Pi_\mu f \;:=\; \arg\min_{V\in\mathcal{V}} \|V - f\|_\mu.$$

In words, given any function $f \in \mathcal{F} : \mathcal{S} \mapsto \mathbb{R}$, $\Pi_\mu f$ returns the closest function V to f that belongs to the subset of linearly representable functions $\mathcal{V}$, where the "closest" is defined by the weighting distribution μ. The Projected Bellman Operator is the composition

$$\mathcal{T}^\pi_{\text{proj}} \;:=\; \Pi_\mu\, T^\pi, \qquad \text{i.e.,} \qquad (\mathcal{T}^\pi_{\text{proj}} V)(\cdot) \;=\; \Pi_\mu[\, T^\pi V\,](\cdot). \tag{2.58}$$

- T^π is the Bellman operator defined in (2.51).
- Π_μ projects any function onto $\mathcal{V}$ using the μ-weighted L^2 norm.
- In discrete $\mathcal{S}$, write $\Phi \in \mathbb{R}^{|\mathcal{S}|\times d}$ with rows $\phi(s)^\top$ and $D = \text{diag}(\mu(s))$. Then

$$\Pi_\mu f \;=\; \Phi\,(\Phi^\top D\Phi)^{-1}\Phi^\top D f.$$

- T^π is a γ-contraction under $\|\cdot\|_\mu$, and Π_μ is nonexpansive under $\|\cdot\|_\mu$, hence $\mathcal{T}^\pi_{\text{proj}}$ is a γ-contraction:

$$\|\Pi_\mu T^\pi V - \Pi_\mu T^\pi U\|_\mu \;\le\; \|T^\pi V - T^\pi U\|_\mu \;\le\; \gamma\|V - U\|_\mu.$$

Therefore, (2.58), the projected Bellman equation (PBE), has a unique fixed point $V^\star_{\text{TD}} \in \mathcal{V}$ satisfying

$$V^\star_{\text{TD}} \;=\; \Pi_\mu\, T^\pi V^\star_{\text{TD}}. \tag{2.59}$$

Semi-gradient TD(0) Converges to the PBE Fixed Point (linear case). Assume on-policy sampling under an ergodic chain, bounded second moments, Robbins–Monro stepsizes, and full-rank features under $\mu = \mu^\pi$. In the linear case $V(s;\theta) = \theta^\top\phi(s)$, define

$$\delta_t(\theta) := r_t + \gamma\theta^\top\phi(s_{t+1}) - \theta^\top\phi(s_t).$$

The semi-gradient TD(0) update (2.56) becomes

$$\theta \leftarrow \theta + \alpha_t\, \delta_t(\theta)\, \phi(s_t).$$

Taking conditional expectation w.r.t. the stationary visitation (and using the Markov property) yields the mean update:

$$\mathbb{E}[\delta_t(\theta)\, \phi(s_t)] \;=\; b \;-\; A\theta,$$

with the standard TD system

$$A \;:=\; \mathbb{E}_\mu[\phi(s_t)(\phi(s_t) - \gamma\phi(s_{t+1}))^\top], \qquad b \;:=\; \mathbb{E}_\mu[r_t\,\phi(s_t)]. \tag{2.60}$$

Thus, in expectation, TD(0) performs a stochastic approximation to the ODE

$$\dot{\theta} \;=\; b - A\theta,$$

whose unique globally asymptotically stable equilibrium is

$$\theta^\star_{\mathrm{TD}} \;=\; A^{-1}b,$$

provided the symmetric part of A is positive definite (guaranteed on-policy with full-rank features). Standard stochastic approximation theory then gives

$$\theta_t \xrightarrow{\text{a.s.}} \theta^\star_{\mathrm{TD}}.$$

Finally, one can show the equivalence with the PBE: $V(\cdot;\theta) \in \mathcal{V}$ satisfies $V(\cdot;\theta) = \Pi_\mu T^\pi V(\cdot;\theta)$ if and only if $A\theta = b$ (see a proof below). Hence the almost-sure limit $V(\cdot;\theta^\star_{\mathrm{TD}})$ is exactly the fixed point (2.59).

Proof. The projected Bellman equation reads

$$V(\cdot;\theta) = \Pi_\mu T^\pi V(\cdot;\theta).$$

Since $V(\cdot;\theta) = \theta^\top\phi(\cdot)$ is the orthogonal projection of $T^\pi V(\cdot;\theta)$ onto $\mathcal{V}$ weighted by μ, we have that

$$\mathbb{E}_\mu\left[\phi(s_t)(T^\pi V(s_t;\theta) - V(s_t;\theta))\right] = 0 = \mathbb{E}_\mu\left[\phi(s_t)(r_t + \gamma\theta^\top\phi(s_{t+1}) - \theta^\top\phi(s_t))\right],$$

which reduces to $A\theta = b$ with A and b defined in (2.60). □

What does convergence to the PBE fixed point imply?

- **Best fixed point in the feature subspace (good).** $V^\star_{\mathrm{TD}}$ is the unique function in $\mathcal{V}$ whose Bellman update $T^\pi V$ projects back to itself under Π_μ. If $V^\pi \in \mathcal{V}$ (realisable case), then $V^\star_{\mathrm{TD}} = V^\pi$.
- **Different target than least squares (mixed).** TD(0) solves the Projected Bellman Equation (2.58); Monte Carlo least-squares solves

  $$\min_{V\in\mathcal{V}} \frac{1}{2}\|V - V^\pi\|^2_\mu.$$

  When $V^\pi \notin \mathcal{V}$, these solutions generally differ. Either can have lower μ-weighted prediction error depending on features and dynamics; in practice TD often wins due to lower variance and online bootstrapping.

2.2.3 On-Policy Control

2.2.3.1 Semi-Gradient SARSA(0)

High-level Intuition. Semi-gradient SARSA(0) is an on-policy value-based control method. It learns an action-value function $Q(s, a; \theta)$ by bootstrapping one step ahead and using the next action actually selected by the current behavior policy (e.g., ε-greedy). Because the target uses $Q(s_{t+1}, a_{t+1}; \theta)$, SARSA trades some bias for substantially lower variance than Monte Carlo, updates online from each transition, and naturally couples policy evaluation (of the current policy) with policy improvement (by making the policy greedy/soft-greedy w.r.t. the current Q).

Algorithmic Form (On-policy, Finite $\mathcal{A}$). Let the behavior policy at time t be $\pi_t(\cdot \mid s)$ (e.g., ε_t-greedy w.r.t. $Q(\cdot, \cdot; \theta_t)$). For each step:

1. Given s_t, pick $a_t \sim \pi_t(\cdot \mid s_t)$; observe r_t and s_{t+1}.
2. Pick the next action $a_{t+1} \sim \pi_t(\cdot \mid s_{t+1})$.
3. Form the TD error
$$\delta_t \;=\; r_t \;+\; \gamma\, Q(s_{t+1}, a_{t+1}; \theta) \;-\; Q(s_t, a_t; \theta). \tag{2.61}$$
4. Update parameters with a semi-gradient step
$$\theta \;\leftarrow\; \theta \;+\; \alpha_t\, \delta_t\, \nabla_\theta Q(s_t, a_t; \theta). \tag{2.62}$$
For terminal s_{t+1}, use $Q(s_{t+1}, a_{t+1}; \theta) = 0$ (equivalently, set $\gamma = 0$ on terminal transitions).

- **Linear special case.** If $Q(s, a; \theta) = \theta^\top \phi(s, a)$, then $\nabla_\theta Q(s_t, a_t; \theta) = \phi(s_t, a_t)$ and the update becomes
$$\theta \leftarrow \theta + \alpha_t\, \delta_t\, \phi(s_t, a_t).$$
- **Expected SARSA (variance reduction).** Replace the sample bootstrap by its expectation under π_t:
$$\delta_t^{\text{exp}} \;=\; r_t \;+\; \gamma \sum_{a' \in \mathcal{A}} \pi_t(a' \mid s_{t+1})\, Q(s_{t+1}, a'; \theta) \;-\; Q(s_t, a_t; \theta), \tag{2.63}$$
then update $\theta \leftarrow \theta + \alpha_t\, \delta_t^{\text{exp}}\, \nabla_\theta Q(s_t, a_t; \theta)$.

5. Update the next policy to be ε_{t+1}-greedy w.r.t. the new Q value $Q(\cdot, \cdot; \theta_{t+1})$. ($\varepsilon_t$ follows GLIE.)

Example 2.3 (Semi-Gradient SARSA for Mountain Car)**.** Consider the Mountain Car problem from Gym illustrated in Fig. 2.5. The state space $\mathcal{S} \subset \mathbb{R}^2$ is continuous and contains the position of the car along the x-axis as well as the car's velocity. The action space $\mathcal{A}$ is discrete and contains three elements: "0: Accelerate to the left", "1: Don't accelerate", and "2: Accelerate to the right". The transition dynamics of the mountain car is:

$$\begin{aligned} v_{t+1} &= v_t + (a_t - 1)F - \cos(3p_t)g \\ p_{t+1} &= p_t + v_{t+1} \end{aligned} \tag{2.64}$$

where (p_t, v_t) denotes the state at time t with position and velocity, a_t denotes the action at time t, $F = 0.001$ is the force and $g = 0.0025$ is the gravitational constant.

The goal is for the mountain car to reach the flag placed on top of the right hill as quickly as possible. Therefore, the agent is penalised with a reward of -1 for each timestep. The position of the car is assigned a uniform random value in $[-0.6, -0.4]$. The starting velocity of the car is always assigned to 0. In every episode, the agent is allowed a maximum of 200 steps (therefore, the worst per-episode return is -200).

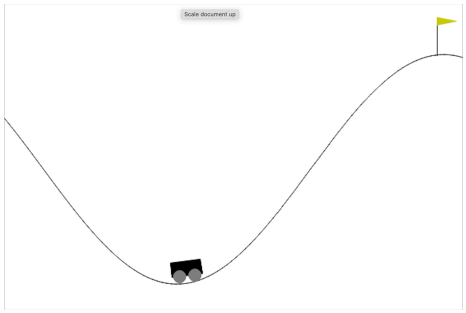

Figure 2.5: Mountain Car from Gym

Naive Semi-Gradient SARSA. We first apply the semi-gradient SARSA algorithm introduced above to the mountain car problem. We parameterize the action value Q as a 2-layer multi-layer perceptron (MLP). Fig. 2.6 shows the average return per episode as training progreses. Clearly, the return stagnates at -200 and the algorithm failed to learn. The rollout in Fig. 2.7 confirms that the final policy is not able to achieve the goal.

You can find code for the naive semi-gradient SARSA algorithm here.

Semi-Gradient SARSA with Experience Replay. Inspired by the technique of experience replay (ER) popularized by DQN (see 2.2.4.2), we incorporated ER into semi-gradient SARSA, which breaks its on-policy nature. Fig. 2.8 displays the learning curve, which shows steady increase of the per-episode return. Applying the final learned policy to the mountain car yields a successful trajectory to the top of the mountain.

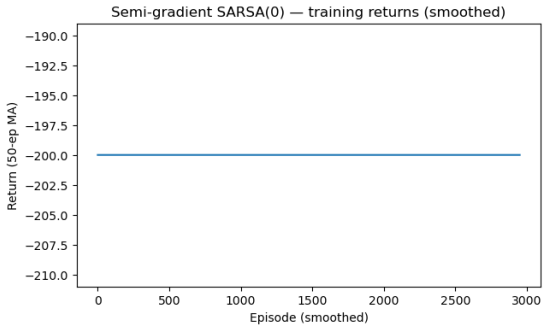

Figure 2.6: Average return w.r.t. episode (Semi-Gradient SARSA)

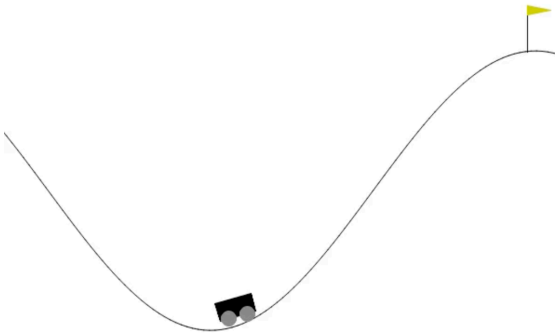

Figure 2.7: Example rollout (Semi-Gradient SARSA)

You can find code for the semi-gradient SARSA with experience replay algorithm here.

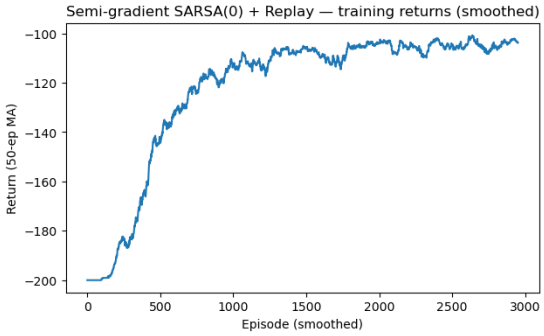

Figure 2.8: Average return w.r.t. episode (Semi-Gradient SARSA + Experience Replay)

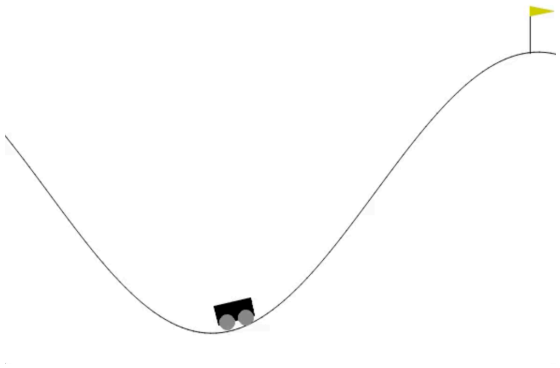

Figure 2.9: Example rollout (Semi-Gradient SARSA + Experience Replay)

2.2.4 Off-Policy Control

Off-policy control seeks to learn the optimal action–value function while collecting data under a different behavior policy (e.g., an ε-soft policy). As in the tabular setting, Q-learning is the canonical off-policy control method. With function approximation, however, off-policy control becomes substantially harder than in the tabular case.

To illustrate why, we first present off-policy semi-gradient TD(0) for policy evaluation and use Baird's counterexample (Baird et al., 1995) to highlight the deadly triad: bootstrapping + function approximation + off-policy sampling can cause divergence.

We then turn to the Deep Q-Network (DQN) (Mnih et al., 2015), which stabilizes Q-learning with two key mechanisms—experience replay and a target network—leading to landmark Atari results. Finally, we connect DQN to fitted Q-iteration (FQI) (Riedmiller, 2005), a batch method with theoretical guarantees, to clarify why these stabilizations work (Fan et al., 2020).

2.2.4.1 Off-Policy Semi-Gradient TD(0)

Setup. We aim to estimate the state-value function of a target policy π using a different behavior policy b. Since the state space is continuous, we employ function approximation to represent the value function as $V(s;\theta)$ with $\theta \in \mathbb{R}^d$. In the case of linear approximation, we have $V(s;\theta) = \theta^\top \phi(s)$ where $\phi(s)$ is a function that featurizes the state.

Semi-Gradient TD(0). Given a transition (s_t, a_t, r_t, s_{t+1}) collected under the behavior policy b, form the TD error

$$\delta_t = r_t + \gamma V(s_{t+1};\theta) - V(s_t;\theta).$$

The off-policy Semi-Gradient TD(0) update reads

$$\theta \quad \leftarrow \quad \theta + \alpha_t \rho_t \delta_t \nabla_\theta V(s_t;\theta), \tag{2.65}$$

where ρ_t is the likelihood ratio as in (2.42):

$$\rho_t = \frac{\pi(a_t \mid s_t)}{b(a_t \mid s_t)}.$$

This off-policy semi-gradient TD(0) update (2.65) looks perfectly reasonable. However, the following Baird's counterexample illustrates the instability of the algorithm.

Baird's Counterexample. Consider an MDP with 7 states containing 6 upper states and 1 lower state, as shown in Fig. 2.10 (the figure comes from (Sutton and Barto, 1998)). There are two actions, one called "solid" and the other called "dashed". If the agent picks "solid" at any state, then the system transitons to the lower state with probability 1. If the agent picks "dashed" at any state, then the system transitions to any one of the upper states with equal probability. All rewards are zero, and the discount factor is $\gamma = 0.99$.

The target policy π always picks "solid", while the behavior policy b picks "solid" with probability 1/7 and "dashed" with probability 6/7.

Consider the case of linear function approximation where $V(s;w) = w^\top \phi(s)$ where $w \in \mathbb{R}^8$. For the upper states, the feature $\phi(s)$ leads to $V(s;w) = 2w_1 + w_8$, and for the lower state, the feature $\phi(s)$ leads to $V(s;w) = w_7 + 2w_8$.

This Python script implements the off-policy semi-gradient TD(0) algorithm with importance sampling for policy evaluation.

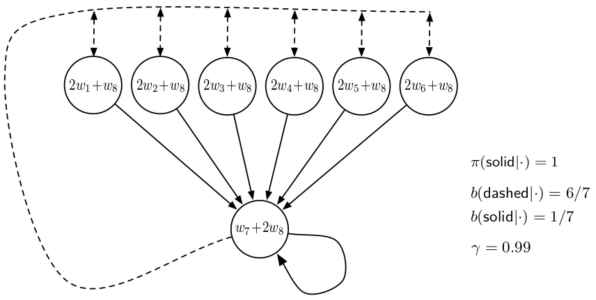

Figure 2.10: Baird Counterexample

Fig. 2.11 plots $\|w\|_2$, the magnitude of w, with respect to iterations. Clearly, we see the parameter w diverges under the off-policy semi-gradient TD(0) algorithm.

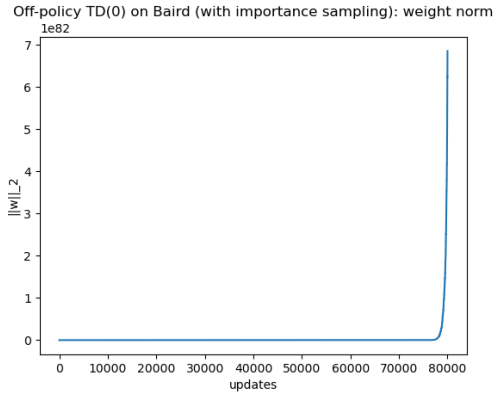

Figure 2.11: Baird Counterexample: divergence

The Deadly Triad. Three ingredients were used together in Baird's example:

- Off-policy: a different behavior policy b is used to collect data for the evaluation of the target policy π;
- Function approximation: the value function employs function approximation;
- Bootstrapping: the TD error uses the bootstrapped target "$r_t + \gamma V(s_{t+1}; \theta)$" instead of the full return as in Monte Carlo.

The "deadly triad" is used to illustrate that using all three ingredients together will lead to the potential divergence of policy evaluation.

Why? Recall that, using linear approximation, the on-policy Semi-Gradient TD(0) algorithm guarantees convergence to the unique fixed point of the projected Bellman equation (PBE) (2.59), restated here

$$V_{\mathrm{TD}}^{\star} = \Pi_{\mu} T^{\pi} V_{\mathrm{TD}}^{\star},$$

where μ is the stationary distribution induced by the policy π. A central reason for the guaranteed convergence is that the projected Bellman operator

$$\Pi_{\mu} T^{\pi}$$

is a γ-contraction and has a unique fixed point. Therefore, the on-policy Semi-Gradient TD(0) algorithm—can be seen as a stochastic approximation of the projected Bellman operator—enjoys convergence guarantees.

However, in the off-policy case, the orthogonal projection Π_{μ} needs to be modified as Π_{ν}, where ν is the stationary distribution induced by the behavior policy b. The new operator

$$\Pi_{\nu} T^{\pi}$$

is not guaranteed to be a γ-contraction, due to the mismatch between ν—induced by b—and the target policy π. Therefore, divergence can potentially happen.

How to Fix? Multiple algorithms have been proposed to fix the deadly triad. Notable examples include the gradient TD (GTD) family of algorithms (Sutton et al., 2008), (Sutton et al., 2009), and the Emphatic TD (ETD) learning algorithm (Mahmood et al., 2015). They are influential and widely cited, but they are not (yet) mainstream in deep RL practice. Their main appeal is theoretical—they provide off-policy evaluation algorithms with convergence guarantees under linear function approximation. Moreover, GTD/TDC require two-time-scale step-sizes and ETD's emphatic weights can have high variance, making them less attractive for large-scale control with neural networks. I encourage you to read the papers to understand the algorithms. However, in the next, I will explain the deep Q network (DQN) approach that is more popular in practice.

2.2.4.2 Deep Q Network

We consider continuous state spaces with a finite action set, and a parametric action–value function $Q(s, a; \theta)$.

Naive (semi-gradient) Q-Learning with Function Approximation. The goal is to learn $Q^{\star}$ and act ε-greedily w.r.t. $Q(\cdot, \cdot; \theta)$. The update uses a bootstrapped optimality target built from the current network.

$$\begin{aligned} y_t &= r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) \\ \theta &\leftarrow \theta + \alpha \left(y_t - Q(s_t, a_t; \theta) \right) \nabla_{\theta} Q(s_t, a_t; \theta). \end{aligned} \tag{2.66}$$

The transitions (s_t, a_t, r_t, s_{t+1}) are generated using a ε-greedy policy with respect to $Q(s, a; \theta)$.

This naive variant is **off-policy + bootstrapping + function approximation** (i.e., the deadly triad) and thus can be unstable.

Deep Q Network (DQN) with Experience Replay (ER) and Target Network (TN). DQN augments the above naive Q learning with two stabilizers:

- **Experience Replay (ER):** store transitions in a buffer $\mathcal{D}$; train on i.i.d.-like mini-batches to decorrelate updates and reuse data.
- **Target Network (TN):** maintain a delayed copy $Q(\cdot, \cdot; \theta^-)$ to compute targets
$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-),$$
keeping the target fixed for many gradient steps.

The full DQN algorithm is presented below.

- Initialize replay buffer $\mathcal{D}$ with capacity N
- Initialize approximate Q value $Q(s, a; \theta)$
- Initialize target $Q_T(s, a; \theta^-)$ with $\theta^- = \theta$
- For episode $= 1, \ldots, M$ do:
  - Initialize s_0
  - For $t = 0, \ldots, T$ do:
    - ε-greedy policy:
      - With probability ε select a random action $a_t \in \mathcal{A}$
      - Otherwise select $a_t = \arg\max_a Q(s_t, a_t; \theta)$
    - Observe transition $\tau_t = (s_t, a_t, r_t, s_{t+1})$
    - Put τ_t inside replay buffer $\mathcal{D}$
    - Sample a random minibatch of transitions $\{(s_i, a_i, r_i, s_{i+1})\}_{i \in \mathcal{J}}$ from $\mathcal{D}$
    - For $i \in \mathcal{J}$ do:
      - Set target $y_i = r_i + \gamma \max_a Q_T(s_{i+1}, a; \theta^-)$ using the target network
      - Update $\theta \leftarrow \theta + \alpha(y_i - Q(s_i, a_i; \theta))\nabla_\theta Q(s_i, a_i; \theta)$
    - Every C steps synchronize the target network with the Q net: $\theta^- = \theta$

Although the naive Q-leanring with function approximation can be unstable, DQN has achieved great success and is also very efficient (Mnih et al., 2015).

Fitted Q Iteration (FQI). To understand why DQN can achieve stabilized training compared to naive Q-learning with function approximation. It is insightful to look at the fitted Q iteration (FQI) algorithm, presented below.

- Initialize $Q^{(0)} = Q(s, a; \theta_0)$
- For $k = 0, 1, 2, \ldots, K-1$ do:
  - Sample i.i.d. transitions $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^{N}$ with (s_i, a_i) drawn from a distribution μ
  - Compute targets $y_i = r_i + \gamma \max_a Q^{(k)}(s_{t+1}, a; \theta_k), i = 1, \ldots, N$
  - Update the action-value function:

$$Q^{(k+1)} = Q(s, a; \theta_{k+1}), \quad \theta_{k+1} \in \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(s_i, a_i; \theta))^2$$

The FQI algorithm samples trajectories from a fixed distribution μ and optimizes the parameter using targets generated from those trajectories. Under reasonable coverage and approximation assumptions, it converges and admits finite-sample error bounds (Antos et al., 2007), (Munos and Szepesvári, 2008).

Connection to DQN. DQN is similar to FQI in the following aspects.

- **Frozen targets:** DQN's target network $Q(\cdot, \cdot; \theta^-)$ plays the role of $Q^{(k-1)}$ in FQI.
- **Supervised fit:** DQN's mini-batch loss minimizes $\sum (Q(s_i, a_i; \theta) - y_i)^2$, just like FQI's regression step.
- **Data usage:** FQI trains on a fixed dataset; DQN's replay buffer approximates training on an (ever-growing) *quasi-fixed* dataset by repeatedly sampling past transitions.
- **Iteration vs. updates:** FQI alternates *full* regressions and *target recomputation*; DQN alternates *many SGD steps* with *periodic target updates*. In the limit of many SGD steps per target update and a large replay buffer, DQN $\approx$ online, incremental FQI.

This perspective explains why **ER + TN** make DQN far more stable than naive Q-learning with function approximation: they make the optimization behave like a sequence of supervised fits to **fixed** targets drawn from a nearly **stationary** dataset.

Example 2.4 (DQN for Mountain Car)**.** Consider again the Mountain car problem from Example 2.3.

Naive Q-Learning with Function Approximation. As shown in Fig. 2.12 and Fig. 2.13, naive Q-leanring with function approximation fails to learn a good policy.

DQN with Experience Replay and Target Network. Adding ER and TN to Q-learning leads to steady learning (Fig. 2.14) and a successful final policy (Fig. 2.15).

You can find code for these experiments here.

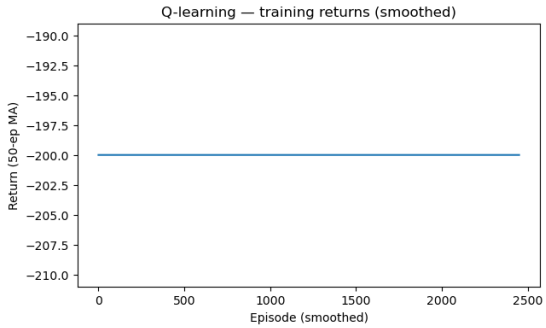

Figure 2.12: Average return w.r.t. episode (Naive Q Learning)

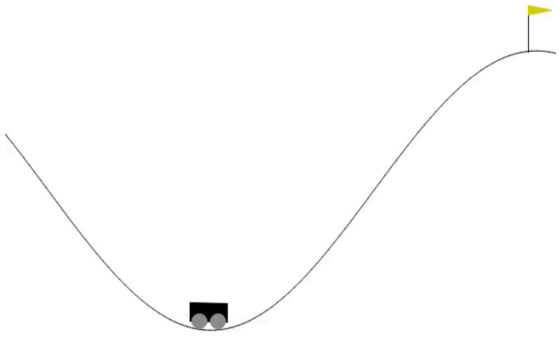

Figure 2.13: Example rollout (Naive Q Learning)

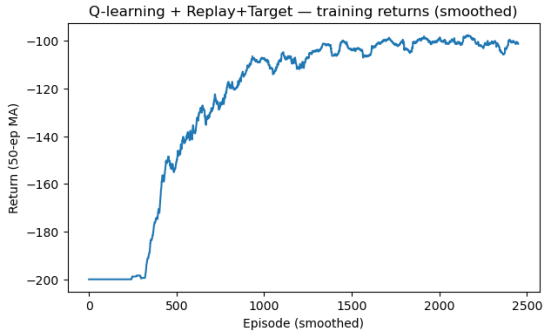

Figure 2.14: Average return w.r.t. episode (DQN)

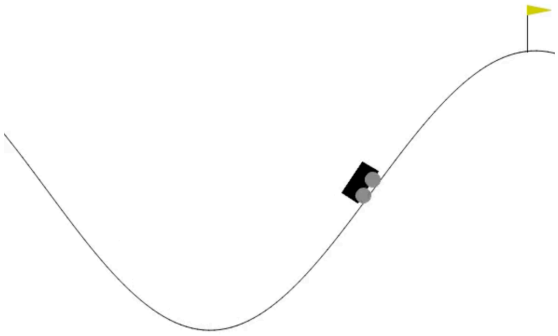

Figure 2.15: Example rollout (DQN)

Chapter 3

Policy Gradient Methods

In Chapter 2, we relaxed two key assumptions of the MDP introduced in Chapter 1:

- **Unknown dynamics**: the transition function P was no longer assumed to be known.
- **Continuous states**: the state space $\mathcal{S}$ was extended from finite to continuous.

When only the dynamics are unknown but the MDP remains tabular, we introduced generalized versions of policy iteration (e.g., SARSA) and value iteration (e.g., Q-learning). These algorithms can recover near-optimal value functions with strong convergence guarantees.

When both the dynamics are unknown and the state space is continuous, tabular methods become infeasible. In this setting, we employed function approximation to represent value functions, and generalized SARSA and Q-learning accordingly. We also introduced stabilization techniques such as experience replay and target networks to ensure more reliable learning.

In this chapter, we relax a third assumption: the action space $\mathcal{A}$ is also continuous. This setting captures many important real-world systems, such as autonomous vehicles and robots. Handling continuous actions requires a departure from the value-based methods of Chapter 2. The key difficulty is that even if we had access to a near-optimal action-value function $Q(s, a)$, selecting the control action requires solving

$$\max_a Q(s, a),$$

which is often computationally expensive and can lead to suboptimal solutions.

To address this challenge, we introduce a new paradigm: policy gradient methods. Rather than learning value functions to derive policies indirectly, we directly optimize parameterized policies using gradient-based methods.

We begin this chapter by reviewing the fundamentals of gradient-based optimization, and then build upon them to develop algorithms for searching optimal policies via policy gradients.

3.1 Gradient-based Optimization

Gradient-based optimization is the workhorse behind most modern machine learning algorithms, including policy gradient methods. The central idea is to iteratively update the parameters of a model in the direction that most improves an objective function.

3.1.1 Basic Setup

Suppose we have a differentiable objective function $J(\theta)$, where $\theta \in \mathbb{R}^d$ represents the parameter vector. The goal is to find

$$\theta^\star \in \arg\max_\theta J(\theta).$$

The gradient of the objective with respect to the parameters,

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} & \frac{\partial J}{\partial \theta_2} & \cdots & \frac{\partial J}{\partial \theta_d} \end{bmatrix}^\top,$$

provides the local direction of steepest ascent. Gradient-based optimization uses this direction to iteratively update the parameters. Note that modern machine learning software tools such as PyTorch allow the user to conveniently query the gradient of any function J defined by neural networks.

3.1.2 Gradient Ascent and Descent

The simplest method is **gradient ascent** (for maximization):

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta_k),$$

where $\alpha > 0$ is the learning rate.
For minimization, the update rule uses **gradient descent**:

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta J(\theta_k).$$

The choice of learning rate α is critical:

- Too large α can cause divergence.
- Too small α leads to slow convergence.

3.1.2.1 Convergence Guarantees

For convex functions $J(\theta)$, gradient descent (or ascent) can be shown to converge to the **global optimum** under appropriate conditions on the learning rate.

For non-convex functions—which are common in reinforcement learning—gradient methods may only find so-called **first-order stationary points**, i.e., points θ at which the gradient $\nabla_\theta J(\theta) = 0$. Nevertheless, they remain effective in practice.

TODO: graph different stationary points

We now formalize the convergence speed of Gradient Descent (GD) for minimizing a smooth convex function. We switch to the minimization convention and write the objective as $f : \mathbb{R}^d \to \mathbb{R}$ (to avoid sign confusions with J used for maximization). We assume exact gradients $\nabla f(\theta)$ are available.

Setup and Assumptions.

- (**Convexity**) For all $\theta, \vartheta \in \mathbb{R}^d$,

$$f(\vartheta) \;\geq\; f(\theta) + \nabla f(\theta)^\top (\vartheta - \theta). \tag{3.1}$$

- (L**-smoothness**) The gradient is L-Lipschitz: for all θ, ϑ,

$$\|\nabla f(\vartheta) - \nabla f(\theta)\| \;\leq\; L\|\vartheta - \theta\|. \tag{3.2}$$

  Equivalently (the **descent lemma**), for all θ, Δ,

$$f(\theta + \Delta) \;\leq\; f(\theta) + \nabla f(\theta)^\top \Delta + \frac{L}{2}\|\Delta\|^2. \tag{3.3}$$

Consider Gradient Descent with a constant stepsize $\alpha > 0$:

$$\theta_{k+1} \;=\; \theta_k \;-\; \alpha\, \nabla f(\theta_k).$$

Theorem 3.1 (GD on smooth convex function)**.** *Let f be convex and L-smooth with a minimizer*

$$\theta^\star \in \arg\min_\theta f(\theta).$$

and the global minimum $f^\star = f(\theta^\star)$. If $0 < \alpha \leq \frac{2}{L}$, then the GD iterates satisfy for all $k \geq 0$:

$$f(\theta_k) - f^\star \leq \frac{2(f(\theta_0) - f^\star)\|\theta_0 - \theta^\star\|^2}{2\|\theta_0 - \theta^\star\|^2 + k\alpha(2 - L\alpha)(f(\theta_0) - f^\star)} \tag{3.4}$$

In particular, choosing $\alpha = \frac{1}{L}$ yields the canonical $O(1/k)$ convergence rate in suboptimality:

$$f(\theta_k) - f^\star \leq \frac{2L\|\theta_0 - \theta^\star\|^2}{k+4} \tag{3.5}$$

Proof. See Theorem 2.1.14 and Corollary 2.1.2 in (Nesterov, 2018). □

Strongly Convex Case (Linear Rate). If, in addition, f is μ-strongly convex ($\mu > 0$), i.e., for all $\theta, \vartheta \in \mathbb{R}^d$,

$$f(\vartheta) \;\geq\; f(\theta) + \nabla f(\theta)^\top(\vartheta - \theta) \;+\; \frac{\mu}{2}\|\vartheta - \theta\|^2. \tag{3.6}$$

Then, GD with $0 < \alpha \leq \frac{2}{\mu+L}$ enjoys a **linear** (geometric) rate:

Theorem 3.2 (GD on smooth strongly convex function)**.** *If f is L-smooth and μ-strongly convex, then for $0 < \alpha \leq \frac{2}{\mu+L}$,*

$$\|\theta_k - \theta^\star\|^2 \leq \left(1 - \frac{2\alpha\mu L}{\mu + L}\right)^k \|\theta_0 - \theta^\star\|^2. \tag{3.7}$$

If $\alpha = \frac{2}{\mu+L}$, then

$$\begin{aligned} \|\theta_k - \theta^\star\| &\leq \left(\frac{Q_f - 1}{Q_f + 1}\right)^k \|\theta_0 - \theta^\star\| \\ f(\theta_k) - f^\star &\leq \frac{L}{2}\left(\frac{Q_f - 1}{Q_f + 1}\right)^{2k} \|\theta_0 - \theta^\star\|^2, \end{aligned} \tag{3.8}$$

where $Q_f = L/\mu$.

Proof. See Theorem 2.1.15 in (Nesterov, 2018). □

Practical Notes.

- The step size $\alpha = \frac{1}{L}$ is **optimal among fixed stepsizes** for the above worst-case bounds on smooth convex f.
- In practice, backtracking line search or adaptive schedules can approach similar behavior without knowing L.
- For policy gradients (which maximize J), apply the results to $f = -J$ and flip the update sign (gradient ascent). The smooth/convex assumptions rarely hold globally in RL, but these results calibrate expectations about step sizes and motivate variance reduction and curvature-aware methods used later.

3.1.3 Stochastic Gradients

In reinforcement learning and other large-scale machine learning problems, computing the exact gradient $\nabla_\theta J(\theta)$ is often infeasible. Instead, we use an unbiased estimator $\hat{\nabla}_\theta J(\theta)$ computed from a subset of data (or trajectories in RL). The update becomes

$$\theta_{k+1} = \theta_k + \alpha \hat{\nabla}_\theta J(\theta_k).$$

This approach, known as **stochastic gradient ascent/descent (SGD)**, trades off exactness for computational efficiency. Variance in the gradient estimates plays an important role in convergence speed and stability.

3.1.3.1 Convergence Guarantees

We now turn to the convergence guarantees of stochastic gradient methods, which replace exact gradients with unbiased noisy estimates. Throughout this section we consider the minimization problem $\min_\theta f(\theta)$ and assume ∇f is available only through a stochastic oracle.

Setup and Assumptions.

Let $f : \mathbb{R}^d \to \mathbb{R}$ be differentiable. At iterate θ_k, we observe a random vector g_k such that

$$\mathbb{E}[\, g_k \mid \theta_k \,] = \nabla f(\theta_k) \quad \text{and} \quad \mathbb{E}[\|g_k - \nabla f(\theta_k)\|^2 \mid \theta_k] \le \sigma^2.$$

We will also use one of the following standard regularity conditions:

- (**Convex + L-smooth**) f is convex and the gradient is L-Lipschitz.
- (**Strongly convex + L-smooth**) f is μ-strongly convex and L-smooth.

We consider the SGD update

$$\theta_{k+1} \;=\; \theta_k - \alpha_k\, g_k,$$

and define the **averaged iterate**

$$\bar{\theta}_K := \frac{1}{K+1} \sum_{k=0}^{K} \theta_k.$$

Theorem 3.3 (SGD on smooth convex function)**.** *Assume f is convex and L-smooth. Suppose there exists $G>0$ with $\mathbb{E}\|g_k\|^2 \le G^2$ for all k.*

- *Choose a constant stepsize $\alpha_k = \alpha > 0$. Then for all $K \ge 1$,*

$$\mathbb{E}[f(\bar{\theta}_K)] - f^\star \le \frac{\|\theta_0 - \theta^\star\|^2}{2\alpha(K+1)} + \frac{\alpha G^2}{2}. \tag{3.9}$$

- *Choose a diminishing step size* $\alpha_k = \frac{\|\theta_0 - \theta^\star\|}{G\sqrt{k+1}}$*, then*

$$\mathbb{E}[f(\bar{\theta}_K)] - f^\star \leq \frac{\|\theta_0 - \theta^\star\| G}{\sqrt{K+1}} = \mathcal{O}\left(\frac{1}{\sqrt{K}}\right). \tag{3.10}$$

Proof. See this lecture note and (Garrigos and Gower, 2023). □

Remarks.

- The bound is on the *averaged* iterate $\bar{\theta}_K$ (the last iterate may be worse by constants without further assumptions).
- Replacing the second-moment bound by a variance bound σ^2 yields the same rate with G^2 replaced by $\sigma^2 + \sup_k \|\nabla f(\theta_k)\|^2$.
- With a constant stepsize, SGD converges $\mathcal{O}(1/k)$ up to a neighborhood set by the gradient noise.

The next theorem states the convergence rate of SGD for minimizing strongly convex functions.

Theorem 3.4 (SGD on smooth strongly convex function)**.** *Assume* f *is* μ*-strongly convex and* L*-smooth, and* $\mathbb{E}[\|g_k\|^2] \leq G^2$*.*
With stepsize $\alpha_k = \frac{1}{\mu(k+1)}$*, the SGD iterates satisfy for all* $K \geq 1$*,*

$$\begin{aligned} \mathbb{E}[f(\bar{\theta}_K)] - f^\star &\leq \frac{G^2}{2\mu(K+1)}(1 + \log(K+1)), \\ \mathbb{E}\|\bar{\theta}_K - \theta^\star\|^2 &\leq \frac{Q}{K+1}, \quad Q = \max\left(\frac{G^2}{\mu^2}, \|\theta_0 - \theta^\star\|^2\right). \end{aligned} \tag{3.11}$$

Proof. See this lecture note and (Garrigos and Gower, 2023). □

Practical Takeaways for Policy Gradients.

- Use **diminishing stepsizes** for theoretical convergence ($\alpha_k \propto 1/\sqrt{k}$ for general convex, $\alpha_k \propto 1/k$ for strongly convex surrogates).
- With **constant stepsizes**, expect fast initial progress down to a variance-limited plateau; lowering variance (e.g., via baselines/advantage estimation) is as important as tuning α.

TODO: graph the different trajectories between minimizing a convex function using GD and SGD.

3.1.4 Beyond Vanilla Gradient Methods

Several refinements to basic gradient updates are widely used:

- **Momentum methods**: incorporate past gradients to smooth updates and accelerate convergence.
- **Adaptive learning rates (Adam, RMSProp, AdaGrad)**: adjust the learning rate per parameter based on historical gradient magnitudes.
- **Second-order methods**: approximate or use curvature information (the Hessian) for more informed updates, though often impractical in high dimensions.

3.2 Policy Gradients

Policy gradients optimize a *parameterized stochastic policy* directly, without requiring an explicit action-value maximization step. They are applicable to both finite and continuous action spaces and are especially useful when actions are continuous or when "arg max" over $Q(s, a)$ is costly or ill-posed.

3.2.1 Setup

We consider a Markov decision process (MDP) with (possibly continuous) state space $\mathcal{S}$, action space $\mathcal{A}$, unknown dynamics P, reward function $R(s, a)$, and discount factor $\gamma \in [0, 1)$. Let $\pi_\theta(a \mid s)$ be a differentiable stochastic policy with parameters $\theta \in \mathbb{R}^d$.

- **Trajectory.** A state-action trajectory is $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$ with probability density/mass

$$p_\theta(\tau) = \rho(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t \mid s_t)\, P(s_{t+1} \mid s_t, a_t), \tag{3.12}$$

  where ρ is the initial state distribution and T is the (random or fixed) episode length.

- **Return.** Define the (discounted) return

$$R(\tau) \;=\; \sum_{t=0}^{T-1} \gamma^t R(s_t, a_t), \tag{3.13}$$

  and the return-to-go

$$g_t \;=\; \sum_{t'=t}^{T-1} \gamma^{t'-t} R(s_{t'}, a_{t'}). \tag{3.14}$$

- **Optimization objective.** The goal is to maximize the expected return

$$J(\theta) \;\equiv\; \mathbb{E}_{\tau\sim p_\theta}[R(\tau)] \;=\; \mathbb{E}\left[\sum_{t=0}^{T-1}\gamma^t R(s_t,a_t)\right], \tag{3.15}$$

  where the expectation is taken over the randomness in (i) the initial state $s_0 \sim \rho$, (ii) the policy π_θ, and (iii) the transition dynamics P.

3.2.1.1 Policy models

- **Finite action spaces ($\mathcal{A}$ discrete).** A common choice is a **softmax (categorical) policy** over a score (logit) function $f_\theta(s,a)$:

$$\pi_\theta(a \mid s) \;=\; \frac{\exp\{f_\theta(s,a)\}}{\sum_{a'\in\mathcal{A}}\exp\{f_\theta(s,a')\}}. \tag{3.16}$$

  Here we use $\exp\{f_\theta(s,a)\} = e^{f_\theta(s,a)}$ for pretty formatting. Typically f_θ is a neural network or a linear function over features.

- **Continuous action spaces ($\mathcal{A} \subseteq \mathbb{R}^m$).** A standard choice is a **Gaussian policy**:

$$\pi_\theta(a \mid s) \;=\; \mathcal{N}(a;\, \mu_\theta(s),\, \Sigma_\theta(s)), \tag{3.17}$$

  where $\mu_\theta(s)$ and (often diagonal) covariance $\Sigma_\theta(s)$ are differentiable functions (e.g., neural networks) parameterized by θ. The policy $\pi_\theta(a \mid s)$ samples actions from the Gaussian parameterized by $\mu_\theta(s)$ and $\Sigma_\theta(s)$. Other choices include squashed Gaussians (e.g., tanh) or Beta distributions for bounded actions.

3.2.2 The Policy Gradient Lemma

With the gradient-based optimization machinery from Section 3.1, a natural strategy for the policy optimization problem in (3.15) is gradient ascent on the objective $J(\theta)$. Consequently, the central task is to characterize the ascent direction, i.e., to compute $\nabla_\theta J(\theta)$.

The policy gradient lemma, stated below, provides exactly this characterization. Crucially, it expresses $\nabla_\theta J(\theta)$ in terms of the policy's score function $\nabla_\theta \log \pi_\theta(a \mid s)$ and returns, without differentiating through the environment dynamics. This likelihood-ratio form makes policy optimization feasible even when the transition model is unknown or non-differentiable.

Theorem 3.5 (Policy Gradient Lemma)**.** *Let $J(\theta) = \mathbb{E}_{\tau\sim p_\theta}[R(\tau)]$ as defined in* (3.15) *Then:*

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{\tau\sim p_\theta}\Big[R(\tau)\,\nabla_\theta \log p_\theta(\tau)\Big] \;=\; \mathbb{E}_{\tau\sim p_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t \mid s_t)\; R(\tau)\right]. \tag{3.18}$$

By causality (future action does not affect past reward), the full return can be replaced by return-to-go:

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{\tau\sim p_\theta}\left[\sum_{t=0}^{T-1}\gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\; g_t\right]. \tag{3.19}$$

Equivalently, using value functions,

$$\nabla_\theta J(\theta) \;=\; \frac{1}{1-\gamma}\mathbb{E}_{s\sim d_\theta,\, a\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a \mid s)\, Q^{\pi_\theta}(s,a)\Big], \tag{3.20}$$

where d_θ is the (discounted) on-policy state visitation distribution for infinite-horizon MDPs:

$$d_\theta(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t \Pr_\theta(s_t = s). \tag{3.21}$$

Proof. We prove the three equivalent forms step by step. Throughout, we assume θ parameterizes only the policy π_θ (not the dynamics P nor the initial distribution ρ), and that interchanging ∇_θ with the trajectory integral/sum is justified (e.g., bounded rewards and finite horizon or standard dominated-convergence conditions). Let the return-to-go g_t be defined as in (3.14).

Step 1 (Log-derivative trick). Write the objective as an expectation over trajectories:

$$J(\theta) \;=\; \int R(\tau)\, p_\theta(\tau)\, d\tau.$$

Differentiate under the integral and use

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) \tag{3.22}$$

we can write:

$$\nabla_\theta J(\theta) = \int R(\tau)\,\nabla_\theta p_\theta(\tau)\, d\tau = \int R(\tau)\, p_\theta(\tau)\,\nabla_\theta \log p_\theta(\tau)\, d\tau = \mathbb{E}_{\tau\sim p_\theta}[R(\tau)\,\nabla_\theta \log p_\theta(\tau)],$$

which is (3.18) up to expanding $\log p_\theta(\tau)$. To see why (3.22) is true, write

$$\nabla_\theta \log p_\theta(\tau) = \frac{1}{p_\theta(\tau)}\nabla_\theta p_\theta(\tau),$$

using the chain rule.

Step 2 (Policy-only dependence). Factor the trajectory likelihood/mass:

$$p_\theta(\tau) = \rho(s_0)\prod_{t=0}^{T-1}\pi_\theta(a_t \mid s_t)\, P(s_{t+1} \mid s_t, a_t).$$

Since ρ and P do not depend on θ,

$$\log p_\theta(\tau) = \text{const} + \sum_{t=0}^{T-1}\log \pi_\theta(a_t \mid s_t) \quad\Rightarrow\quad \nabla_\theta \log p_\theta(\tau) \;=\; \sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t \mid s_t).$$

Substitute into Step 1 to obtain the second equality in (3.18):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, R(\tau)\right].$$

Step 3 (Causality ⇒ return-to-go). Expand $R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_t$ (with $r_t := R(s_t, a_t)$) and swap sums:

$$\mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, R(\tau)\right] = \sum_{t=0}^{T-1}\sum_{t'=0}^{T-1} \mathbb{E}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \gamma^{t'} r_{t'}\big].$$

For $t' < t$, the factor $\gamma^{t'} r_{t'}$ is measurable w.r.t. the history $\mathcal{F}_t = \sigma(s_0, a_0, \dots, s_t)$, while

$$\mathbb{E}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \,\big|\, \mathcal{F}_t\big] = \sum_a \pi_\theta(a \mid s_t)\, \nabla_\theta \log \pi_\theta(a \mid s_t) = \nabla_\theta \sum_a \pi_\theta(a \mid s_t) = \nabla_\theta 1 = 0,$$

(and analogously with integrals for continuous $\mathcal{A}$). Hence by the tower property,

$$\mathbb{E}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \gamma^{t'} r_{t'}\big] = 0 \quad \text{for all } t' < t.$$

Therefore only the terms with $t' \geq t$ survive, and

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \mathbb{E}\Big[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T-1} \gamma^{t'} r_{t'}\Big] = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, g_t\right],$$

which is (3.19).

Step 4 (Value-function form). Condition on (s_t, a_t) and use the definition of the action-value function:

$$Q^{\pi_\theta}(s_t, a_t) \;\equiv\; \mathbb{E}[g_t \,|\, s_t, a_t]\,.$$

Taking expectations then yields

$$\mathbb{E}\big[\gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, g_t\big] = \mathbb{E}\big[\gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, Q^{\pi_\theta}(s_t, a_t)\big].$$

Summing over t and collecting terms with the (discounted) on-policy state visitation distribution d_θ (for the infinite-horizon case, e.g., $d_\theta(s) = (1-\gamma)\sum_{t=0}^{\infty} \gamma^t \Pr_\theta(s_t = s)$; for finite T, use the corresponding finite-horizon weighting), we obtain

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{s\sim d_\theta,\, a\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a \mid s)\, Q^{\pi_\theta}(s, a)\Big],$$

which is (3.20).

Conclusion. Combining Steps 1–4 proves all three stated forms of the policy gradient. □

3.2.3 REINFORCE

The policy gradient lemma immediately gives us an algorithm. Specifically, the gradient receipe in (3.18) tells us that if we generate one trajectory τ by following the policy π, then

$$\widehat{\nabla_\theta J} = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) R(\tau) \tag{3.23}$$

is an unbiased estimator of the true gradient.

With this sample gradient estimator, we obtain the classical REINFORCE algorithm.

Single-Trajectory (Naive) REINFORCE

1. Initialize θ_0 for the initial policy $\pi_{\theta_0}(a \mid s)$
2. For $k = 0, 1, \ldots$, do:

- Obtain a trajectory $\tau \sim p_{\theta_k}$
- Compute the stochastic gradient g_k as in (3.23)
- Update $\theta_{k+1} = \theta_k + \alpha_k g_k$

To reduce variance of the gradient estimator, we can use a minibatch of trajectories. For example, given a batch of N trajectories $\{\tau^{(i)}\}_{i=1}^N$ collected by π_θ, define for each timestep the return-to-go

$$g_t^{(i)} = \sum_{t'=t}^{T^{(i)}-1} \gamma^{t'-t} R\big(s_{t'}^{(i)}, a_{t'}^{(i)}\big) .$$

An unbiased gradient estimator, from (3.19) is

$$\widehat{\nabla_\theta J} \;=\; \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T^{(i)}-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) g_t^{(i)}. \tag{3.24}$$

This leads to the following minibatch REINFORCE algorithm.

Minibatch REINFORCE

1. Initialize θ_0 for the initial policy $\pi_{\theta_0}(a \mid s)$
2. For $k = 0, 1, \ldots$, do:

- Obtain N trajectories $\{\tau^{(i)}\}_{i=1}^N \sim p_{\theta_k}$
- Compute the stochastic gradient g_k as in (3.24)
- Update $\theta_{k+1} = \theta_k + \alpha_k g_k$

We apply both the single-trajectory (naive) REINFORCE and a minibatch variant to the CartPole-v1 balancing task. The results show that variance reduction via minibatching is crucial for stable learning and for obtaining strong policies with policy-gradient methods.

Example 3.1 (REINFORCE for Cart-Pole Balancing)**.** Consider the cart-pole balancing task illustrated in Fig. 3.1. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

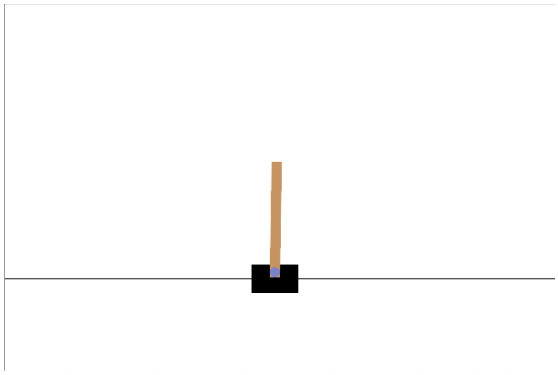

Figure 3.1: Cart Pole balance.

State Space. The state of the cart-pole system is denoted by $s \in \mathcal{S} \subset \mathbb{R}^4$, containing the position and velocity of the cart, as well as the angle and angular velocity of the pole.

Action Space. The action space $\mathcal{A}$ is discrete and contains two elements: pushing to the left and pushing to the right.

The dynamics of the MDP is provided by the Gym simulator and is described in the original paper (Barto et al., 2012). At the beginning of the episode, all state variables are randomly initialized in $[-0.05, 0.05]$ and the goal for the agent is to apply the actions to balance the cart-pole for as long as possible—the agent gets a reward of $+1$ every step if (1) the pole angle remains between -12° and $+12^\circ$ and (2) the cart position remains between -2.4 and 2.4. The maximum episode length is 500.

We design a policy network in the form of (3.16) since the action space is finite.

REINFORCE. We first apply the naive REINFORCE algorithm where the gradient estimator is computed from a single trajectory as in (3.23). Fig. 3.2 shows the learning curve, which indicates that the REINFORCE algorithm was not able to learn a good policy after 2000 episodes.

Minibatch REINFORCE. We then apply the minibatch REINFORCE algorithm where the gradient estimator is computed from multiple (20 in our case)

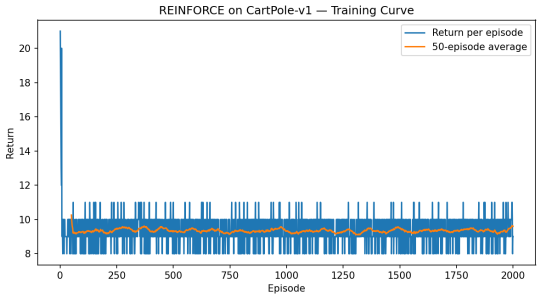

Figure 3.2: Learning curve (Naive REINFORCE).

trajectories as in (3.24). Fig. 3.3 shows the learning curve, which shows steady increase in the per-episode return that eventually gets close to the maximum per-episode return 500.

Fig. 3.4 shows a rollout video of applying the policy training from minibatch REINFORCE. We can see the policy nicely balances the cart-pole system.

You can play with the code here.

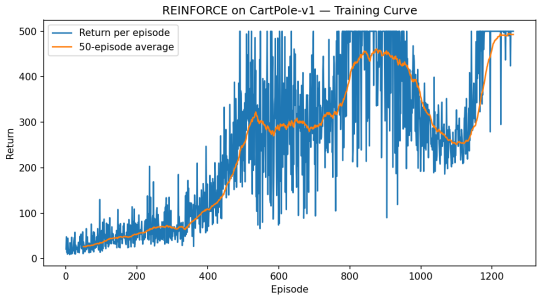

Figure 3.3: Learning curve (Minibatch REINFORCE).

3.2.4 Baselines and Variance Reduction

From the REINFORCE experiments above, we have seen firsthand that **variance reduction** is critical for stable policy-gradient learning.

A natural question is: *what framework can we use to systematically reduce the variance of the gradient estimator while preserving unbiasedness?*

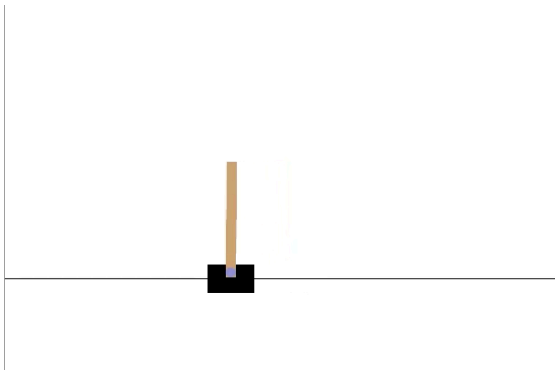

Figure 3.4: Policy rollout (Minibatch REINFORCE).

3.2.4.1 Baseline

A key device is a **baseline** $b : \mathcal{S} \to \mathbb{R}$ added at each timestep:

$$\hat{g} \;=\; \sum_{t=0}^{T-1} \gamma^t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, (g_t - b(s_t)). \tag{3.25}$$

The only difference between (3.25) and the original gradient estimator (3.19) is that the baseline $b(s_t)$ is subtracted from the return-to-go g_t. The next theorem states that any state-only baseline does not change the expectation of the gradient estimator.

Theorem 3.6 (Baseline Invariance)**.** *Let $b : \mathcal{S} \to \mathbb{R}$ be any function independent of the action a_t. Then*

$$\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, b(s_t)\right] = 0,$$

and thus

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, (g_t - b(s_t))\right]. \tag{3.26}$$

Equivalently, using action-values,

$$\nabla_\theta J(\theta) = \frac{1}{1-\gamma} \, \mathbb{E}_{s \sim d_\theta,\, a \sim \pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a \mid s) \, (Q^{\pi_\theta}(s,a) - b(s))\Big]. \tag{3.27}$$

Proof. We prove (i) the baseline term has zero expectation, (ii) the baseline-subtracted estimator in (3.26) is unbiased, and (iii) the equivalent Q-value form (3.27).

Throughout we assume standard conditions ensuring interchange of expectation and differentiation (e.g., bounded rewards with finite horizon or discounted infinite horizon, and a differentiable policy).

Step 1 (Score-function expectation is zero). Fix a state $s \in \mathcal{S}$. The **score function** integrates/sums to zero under the policy:

$$\begin{aligned}\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a \mid s)] &= \sum_{a\in\mathcal{A}} \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s) = \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a \mid s) \\ &= \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(a \mid s) = \nabla_\theta 1 = 0,\end{aligned} \tag{3.28}$$

with the obvious replacement of sums by integrals for continuous $\mathcal{A}$. This identity is the standard "score has zero mean" property.

Step 2 (Baseline term has zero expectation). Let $\mathcal{F}_t := \sigma(s_0, a_0, \dots, s_t)$ be the history up to time t and recall that $b(s_t)$ is **independent of** a_t. Using iterated expectations:

$$\mathbb{E}[\gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, b(s_t)] = \mathbb{E}\left[\gamma^t\, b(s_t)\, \underbrace{\mathbb{E}[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \mid s_t]}_{=0 \text{ by Step 1}}\right] = 0.$$

Summing over t yields

$$\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, b(s_t)\right] = 0.$$

Step 3 (Unbiasedness of the baseline-subtracted estimator). By the policy gradient lemma (likelihood-ratio form with return-to-go; see (3.19)),

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, g_t\right].$$

Subtract and add the baseline term inside the expectation:

$$\begin{aligned}\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, g_t\right] = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, (g_t - b(s_t))\right] + \\ \underbrace{\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, b(s_t)\right]}_{=0 \text{ by Step 2}}.\end{aligned}$$

Therefore (3.26) holds, proving that **any** state-only baseline preserves unbiasedness.

Step 4 (Equivalent Q-value form). Condition on (s_t, a_t) and use the definition $Q^{\pi_\theta}(s_t, a_t) := \mathbb{E}[g_t \mid s_t, a_t]$:

$$\mathbb{E}\big[\gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\,(g_t - b(s_t))\big] = \mathbb{E}\Big[\gamma^t\, \mathbb{E}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\,(g_t - b(s_t)) \mid s_t\big]\Big].$$

Inside the inner expectation (over $a_t \sim \pi_\theta(\cdot \mid s_t)$) and using $b(s_t)$'s independence from a_t,

$$\mathbb{E}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\,(g_t - b(s_t)) \mid s_t\big] = \mathbb{E}_{a \sim \pi_\theta(\cdot \mid s_t)}\Big[\nabla_\theta \log \pi_\theta(a \mid s_t)\,(Q^{\pi_\theta}(s_t, a) - b(s_t))\Big].$$

Summing over t with discount γ^t and collecting terms with the (discounted) on-policy state-visitation distribution d_θ (cf. (3.21)) yields the infinite-horizon identity

$$\nabla_\theta J(\theta) = \frac{1}{1-\gamma}\, \mathbb{E}_{s \sim d_\theta,\, a \sim \pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a \mid s)\,(Q^{\pi_\theta}(s, a) - b(s))\Big],$$

which is (3.27). □

3.2.4.2 Optimal Baseline and Advantage

Among all state-only baselines $b(s)$, which one minimizes the variance of the gradient estimator?

Theorem 3.7 (Variance-Minimizing Baseline (per-state))**.** *For the estimator*

$$g(s, a) = \nabla_\theta \log \pi_\theta(a \mid s)\,(Q^\pi(s, a) - b(s)),$$

the $b(s)$ minimizing $\mathrm{Var}[g \mid s]$ *is*

$$b^\star(s) = \frac{\mathbb{E}_{a \sim \pi_\theta}\big[\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2\, Q^\pi(s, a)\big]}{\mathbb{E}_{a \sim \pi_\theta}\big[\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2\big]}.$$

Assuming that the norm factor $\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2$ varies slowly with a, then

$$b^\star(s) \approx V^\pi(s).$$

Proof. Let $s \in \mathcal{S}$ be fixed and write

$$u(a \mid s) \;\equiv\; \nabla_\theta \log \pi_\theta(a \mid s) \in \mathbb{R}^d, \qquad w(a \mid s) \;\equiv\; \|u(a \mid s)\|^2 \;\geq 0.$$

Consider the vector-valued random variable

$$g(s, a) \;=\; u(a \mid s)\,(Q^\pi(s, a) - b(s)),$$

where the randomness is over $a \sim \pi_\theta(\cdot \mid s)$.

We aim to choose $b(s) \in \mathbb{R}$ to minimize the **conditional variance**

$$\mathrm{Var}[g \mid s] \;=\; \mathbb{E}\big[\|g(s,a) - \mathbb{E}[g \mid s]\|^2 \,\big|\, s\big]\,.$$

Using the identity $\mathrm{Var}[X] = \mathbb{E}\|X\|^2 - \|\mathbb{E}X\|^2$ (for vector X with Euclidean norm), we have

$$\mathrm{Var}[g \mid s] \;=\; \underbrace{\mathbb{E}[\|g(s,a)\|^2 \mid s]}_{\text{depends on } b(s)} \;-\; \underbrace{\big\|\mathbb{E}[g \mid s]\big\|^2}_{\text{independent of } b(s)}\,.$$

We first show that the mean term is independent of $b(s)$. Indeed,

$$\mathbb{E}[g \mid s] = \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\big[u(a \mid s)\,(Q^\pi(s,a) - b(s))\big] = \mathbb{E}\big[u(a \mid s)\,Q^\pi(s,a)\big] - b(s)\,\underbrace{\mathbb{E}\big[u(a \mid s)\big]}_{=0},$$

where $\mathbb{E}[u(a \mid s)] = \sum_a \pi_\theta(a \mid s)\nabla_\theta \log \pi_\theta(a \mid s) = \nabla_\theta \sum_a \pi_\theta(a \mid s) = \nabla_\theta 1 = 0$ (replace sums by integrals in the continuous case). Therefore $\mathbb{E}[g \mid s]$ does **not** depend on $b(s)$.

Consequently, minimizing $\mathrm{Var}[g \mid s]$ is equivalent to minimizing the conditional **second moment**

$$\mathbb{E}[\|g(s,a)\|^2 \mid s] = \mathbb{E}\Big[\|u(a \mid s)\|^2\,(Q^\pi(s,a) - b(s))^2 \,\Big|\, s\Big] = \mathbb{E}\Big[w(a \mid s)\,(Q^\pi(s,a) - b(s))^2 \,\Big|\, s\Big]\,.$$

The right-hand side is a convex quadratic in the scalar $b(s)$. Differentiate w.r.t. $b(s)$ and set to zero:

$$\frac{\partial}{\partial b(s)}\mathbb{E}\Big[w(a \mid s)\,(Q^\pi(s,a) - b(s))^2 \,\Big|\, s\Big] = -2\,\mathbb{E}\big[w(a \mid s)\,(Q^\pi(s,a) - b(s)) \,\big|\, s\big] = 0.$$

Hence,

$$\mathbb{E}[w(a \mid s)\,Q^\pi(s,a) \mid s] = b(s)\,\mathbb{E}[w(a \mid s) \mid s]\,,$$

and provided $\mathbb{E}[w(a \mid s) \mid s] > 0$ (i.e., the Fisher information at s is non-degenerate), the unique minimizer is

$$b^\star(s) = \frac{\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\big[\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2\,Q^\pi(s,a)\big]}{\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\big[\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2\big]},$$

which is (3.7). If $\mathbb{E}[w(a \mid s) \mid s] = 0$ (e.g., a locally deterministic policy), then $g \equiv 0$ almost surely and any $b(s)$ attains the minimum.

Finally, when the weight $w(a \mid s) = \|\nabla_\theta \log \pi_\theta(a \mid s)\|^2$ varies slowly with a (or is approximately constant) for a fixed s, the ratio simplifies to

$$b^\star(s) \;\approx\; \frac{\mathbb{E}[c(s)\,Q^\pi(s,a) \mid s]}{\mathbb{E}[c(s) \mid s]} \;=\; \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\big[Q^\pi(s,a)\big] \;=\; V^\pi(s),$$

so that the baseline-subtracted target becomes the **advantage** $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. □

When using $V^\pi(s)$ as the baseline, the baseline-subtracted target is called the **advantage function**

$$A^{\pi_\theta}(s,a) \;=\; Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s). \tag{3.29}$$

The corresponding minibatch gradient estimator becomes

$$\widehat{\nabla_\theta J} \;=\; \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T^{(i)}-1} \gamma^t\, \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})\, \widehat{A}_t^{(i)}, \quad \widehat{A}_t^{(i)} \approx g_t^{(i)} - V_\phi(s_t^{(i)}), \tag{3.30}$$

where V_ϕ is a learned approximation to V^{π_θ}.

3.2.4.3 Intuition for the Advantage

The advantage

$$A^\pi(s,a) \;=\; Q^\pi(s,a) - V^\pi(s)$$

measures how much *better or worse* action a is at state s *relative to the policy's average action quality* $V^\pi(s) = \mathbb{E}_{a\sim\pi}[Q^\pi(s,a) \mid s]$.
Hence $\mathbb{E}_{a\sim\pi}[A^\pi(s,a) \mid s] = 0$: it is a *relative* score.

With a value baseline, the policy-gradient update is

$$\nabla_\theta J(\theta) \;=\; \frac{1}{1-\gamma}\,\mathbb{E}_{s\sim d_\pi,\, a\sim\pi}\big[\nabla_\theta \log \pi_\theta(a \mid s)\, A^\pi(s,a)\big].$$

- If $A^\pi(s,a) > 0$: the term $\nabla_\theta \log \pi_\theta(a \mid s)\, A^\pi(s,a)$ **increases** $\log \pi_\theta(a \mid s)$ (and thus $\pi_\theta(a \mid s)$)—the policy puts **more** probability mass on actions that outperformed its average at s.
- If $A^\pi(s,a) < 0$: it **decreases** $\log \pi_\theta(a \mid s)$—the policy puts **less** probability mass on actions that underperformed at s.
- If $A^\pi(s,a) \approx 0$: the action performed about as expected; the update at that (s,a) is **negligible**.

Subtracting $V^\pi(s)$ centers returns *per state*, so the update depends only on *relative* goodness. This:

- preserves unbiasedness (baseline invariance),
- reduces variance (no large, shared offset),
- focuses learning on which actions at s should get more/less probability.

3.2.4.4 REINFORCE with a Learned Value Baseline

Recall that in Section 2.2, we have introduced multiple algorithms that can learn an approximate value function for policy evaluation. For example, we can use Monte Carlo estimation.

We now combine REINFORCE with a learned baseline $V_\phi(s) \approx V^{\pi_\theta}(s)$, yielding a lower-variance update while keeping the estimator unbiased.

Minibatch REINFORCE with a Learned Value Baseline

Inputs: policy $\pi_\theta(a \mid s)$, value $V_\phi(s)$, discount $\gamma \in [0, 1)$, stepsizes $\alpha_\theta, \alpha_\phi > 0$, batch size N.
Convergence controls: tolerance $\varepsilon > 0$, maximum inner steps $K_{\max}$ (value-fit loop), optional patience P.

1. **Collect trajectories.** Roll out N on-policy trajectories $\{\tau^{(i)}\}_{i=1}^{N}$ using π_θ.
   For each trajectory i and timestep t, record $(s_t^{(i)}, a_t^{(i)}, r_t^{(i)})$.

2. **Compute returns-to-go.** For each i, t,
$$g_t^{(i)} \;=\; \sum_{t'=t}^{T^{(i)}-1} \gamma^{\,t'-t}\, r_{t'}^{(i)}.$$

3. **Fit the value to convergence (critic inner loop).** Define the batch regression loss
$$\mathcal{L}_V(\phi) \;=\; \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T^{(i)}-1} \big(g_t^{(i)} - V_\phi(s_t^{(i)})\big)^2.$$
   Perform gradient steps on ϕ **until convergence** on this fixed batch:
$$\phi \leftarrow \phi - \alpha_\phi \, \nabla_\phi \mathcal{L}_V(\phi).$$
   Repeat for $k = 1, \ldots, K_{\max}$ or until
$$\frac{\mathcal{L}_V^{(k-1)} - \mathcal{L}_V^{(k)}}{\max\{1, |\mathcal{L}_V^{(k-1)}|\}} < \varepsilon$$
   for M consecutive checks. Denote the (approximately) converged parameters by $\phi^\star$.

4. **Form (optionally standardized) advantages using the converged value.**
$$\widehat{A}_t^{(i)} \;=\; g_t^{(i)} - V_{\phi^\star}(s_t^{(i)}), \qquad \widetilde{A}_t^{(i)} \;=\; \frac{\widehat{A}_t^{(i)} - \mu_A}{\sigma_A + \delta} \;\; \text{(optional, batch-wise)},$$
   where μ_A, σ_A are the mean and std of $\{\widehat{A}_t^{(i)}\}$ over the **whole** batch, and $\delta > 0$ is a small constant.

5. **Single policy (actor) update.** Using the converged baseline, take **one** ascent step:

$$\theta \leftarrow \theta + \alpha_\theta \cdot \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T^{(i)}-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) \, \tilde{A}_t^{(i)}.$$

(If not standardizing, use $\widehat{A}_t^{(i)}$ in place of $\tilde{A}_t^{(i)}$.)

6. **Repeat** from Step 1 with the updated policy.

Notes.

- By baseline invariance, subtracting $V_{\phi^\star}(s)$ keeps the policy-gradient unbiased while reducing variance.
- Converging the critic on each fixed batch (Steps 3–4) approximates the variance-minimizing baseline for that batch before a single actor step, often stabilizing learning in high-variance settings.

Example 3.2 (REINFORCE with a Learned Value Baseline for Cart-Pole)**.** Consider the same cart-pole balancing task in Example 3.1. We use minibatch REINFORCE with a learned value baseline (batch size 50), the algorithm described above.

Fig. 3.5 shows the learning curve. The algorithm is able to steadily increase the per-episode returns.

Fig. 3.6 shows a rollout of the system trajectory under the learned policy.

You can play with the code here.

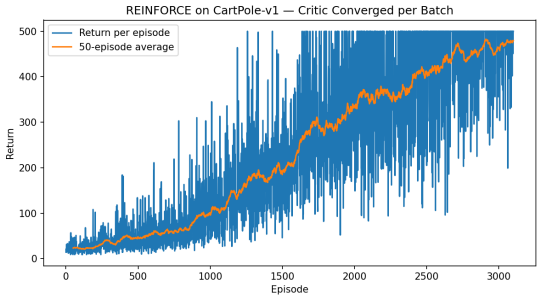

Figure 3.5: Learning curve (Minibatch REINFORCE with a Learned Value Baseline).

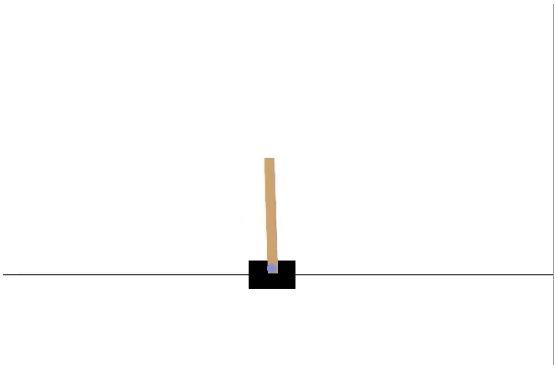

Figure 3.6: Policy rollout (Minibatch REINFORCE with a Learned Value Baseline).

3.3 Actor–Critic Methods

Actor–critic (AC) algorithms marry **policy gradients** (the *actor*) with **value function learning** (the *critic*). The critic reduces variance by supplying low-noise estimates of action quality (values or advantages), while the actor updates the policy using these estimates. In contrast to pure Monte Carlo baselines, actor–critic **bootstraps** from its own predictions, enabling online, incremental, and often more sample-efficient learning.

3.3.1 Anatomy of an Actor–Critic

- **Actor (policy):** a differentiable policy $\pi_\theta(a \mid s)$.
- **Critic (value):** an approximator for $V_\phi(s)$, $Q_\psi(s,a)$, or directly the advantage $A_\eta(s,a)$.
- **Update coupling:** the actor ascends a baseline-subtracted log-likelihood objective using *advantage-like* targets supplied by the critic.

3.3.2 On-Policy Actor–Critic with TD(0)

We first learn a state value function $V_\phi(s)$ with a **one-step bootstrapped** TD(0) target:

$$\delta_t \;\equiv\; r_t + \gamma\, V_\phi(s_{t+1}) - V_\phi(s_t), \qquad \mathcal{L}_V(\phi) \;=\; \frac{1}{2}\,\delta_t^2.$$

If $V_\phi \approx V^\pi$, then $\mathbb{E}[\delta_t \mid s_t, a_t] \approx A^\pi(s_t, a_t)$, so δ_t serves as a low-variance **advantage** target for the actor:

$$\widehat{\nabla_\theta J} \;=\; \frac{1}{|\mathcal{B}|} \sum_{(s_t,a_t)\in\mathcal{B}} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \underbrace{\delta_t}_{\text{advantage target}}.$$

(Practical: normalize $\{\delta_t\}_{\mathcal{B}}$ to mean 0 and unit variance within a batch; clip gradients for stability.)

On-Policy Actor–Critic with One-Step Bootstrap (TD(0))

Inputs: policy $\pi_\theta(a \mid s)$, value $V_\phi(s)$, discount $\gamma \in [0, 1)$, stepsizes $\alpha_\theta, \alpha_\phi > 0$, rollout length K, minibatch size $|\mathcal{B}|$.

For iterations $k = 0, 1, 2, ...$:

1. **Collect on-policy rollouts.** Run π_θ for K steps (optionally across parallel envs), storing transitions $\{(s_t, a_t, r_t, s_{t+1}\}$.
2. **Compute TD errors.** For each transition, compute the TD error
$$\delta_t \leftarrow r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t).$$
3. **Critic update (value).** Minimize $\sum_{t\in\mathcal{B}} \frac{1}{2}\,\delta_t^2$: perform multiple steps of
$$\phi \leftarrow \phi - \alpha_\phi \,\nabla_\phi \Big(\frac{1}{|\mathcal{B}|} \sum_{t\in\mathcal{B}} \frac{1}{2}\,\delta_t^2 \Big).$$
4. **Actor advantages.** Set $\widehat{A}_t \leftarrow \delta_t$ (optionally normalize over $\mathcal{B}$).
5. **Actor update (policy gradient).**
$$\theta \leftarrow \theta + \alpha_\theta \,\frac{1}{|\mathcal{B}|} \sum_{t\in\mathcal{B}} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\,\widehat{A}_t.$$
6. **Repeat** from step 1.

We apply the on-policy actor-critic algorithm to the cart-pole balancing task.

Example 3.3 (Actor–Critic with One-Step Bootstrap for Cart-Pole)**.** Consider the same cart-pole balancing control task as before, and this time apply the on-policy actor-critic with one-step bootstrap.

Fig. 3.7 shows the learning curve.

Fig. 3.8 shows an example rollout of the policy.

You can play with the code here.

3.3.3 Generalized Advantage Estimation (GAE)

In REINFORCE with a learned value baseline (Section 3.2.4.4), we used the full Monte Carlo return g_t as the target for value function approximation; while in

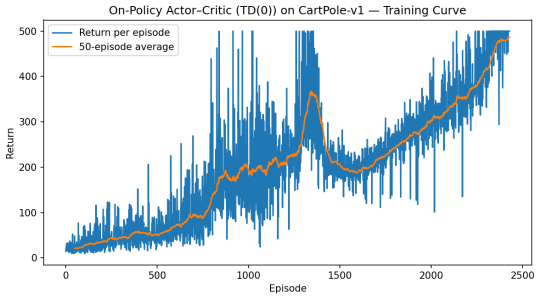

Figure 3.7: Learning curve (Actor–Critic with One-Step Bootstrap).

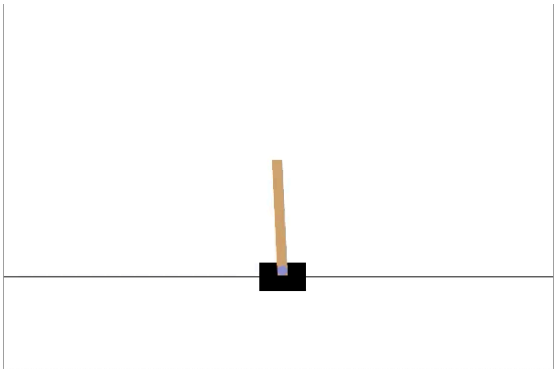

Figure 3.8: Policy rollout (Actor–Critic with One-Step Bootstrap).

on-policy Actor-Critic with TD(0) (Section 3.3.2), we used the one-step bootstrap return $r_t + \gamma V_\phi(s_{t+1})$ as the target for value function estimation.

Recall in policy evaluation (Section 2.1.1), we have introduced a spectrum of methods that sit in between Monte Carlo and TD(0): they are methods that leverage the n-step bootstrap return that balance bias and variance. (Section 2.1.1.3 and 2.1.1.4).

In particular, recall the definition of an n-step bootstrap return

$$g_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} \; + \; \gamma^n \, V_\phi(s_{t+n}), \tag{3.31}$$

where V_ϕ denotes the approximate value function. The λ-return (with $\lambda \in [0, 1]$) performs a convex combination of all the n-step returns

$$g_t^{(\lambda)} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \, g_t^{(n)}. \tag{3.32}$$

The Generalized Advantage Estimation (GAE) algorithm (Schulman et al., 2015b) is an Actor-Critic type of policy gradient method that leverages the λ-return as the target for fitting the critic (i.e., the approximate value function).

GAE-λ Advantage. Start from the TD residual

$$\delta_t \; = \; r_t + \gamma \, V_\phi(s_{t+1}) - V_\phi(s_t),$$

and define the GAE-λ advantage as the exponentially-weighted sum of future TD residuals:

$$\widehat{A}_t^{(\lambda)} \; = \; \sum_{\ell=0}^{T-1-t} (\gamma\lambda)^\ell \, \delta_{t+\ell}.$$

This admits an efficient backward recursion:

$$\widehat{A}_t^{(\lambda)} \; = \; \delta_t \; + \; \gamma\lambda \, \widehat{A}_{t+1}^{(\lambda)}, \quad \widehat{A}_T^{(\lambda)} = 0 \text{ (at terminal)}.$$

From Advantage to Return. A key identity (obtained by expanding the sum of TD residuals and grouping terms) is

$$\sum_{\ell=0}^{\infty} (\gamma\lambda)^\ell \, \delta_{t+\ell} \; = \; \Big(1-\lambda\Big) \sum_{n=1}^{\infty} \lambda^{n-1} \Big(g_t^{(n)} - V_\phi(s_t) \Big).$$

The left-hand side is the GAE-λ advantage, and the right-hand side is $g_t^{(\lambda)} - V_\phi(s_t)$. Therefore,

$$\widehat{A}_t^{(\lambda)} \; = \; g_t^{(\lambda)} \; - \; V_\phi(s_t), \qquad \text{and hence} \qquad g_t^{(\lambda)} \; = \; \widehat{A}_t^{(\lambda)} + V_\phi(s_t).$$

In GAE, we use

$$\widehat{V}_t^{\text{targ}} \;=\; \widehat{A}_t^{(\lambda)} \;+\; V_\phi(s_t),$$

as the target for fitting V_ϕ.

GAE Policy Gradient. The true on-policy policy gradient can be written as

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, A^\pi(s_t, a_t)\right].$$

An estimator remains unbiased if we replace A^π by any $\widehat{A}$ satisfying

$$\mathbb{E}\Big[\widehat{A}_t \,\Big|\, s_t, a_t\Big] \;=\; A^\pi(s_t, a_t).$$

When the critic is exact, $V_\phi \equiv V^\pi$, each n-step bootstrap return has expectation

$$\mathbb{E}\Big[g_t^{(n)} \,\Big|\, s_t, a_t\Big] \;=\; Q^\pi(s_t, a_t),$$

so by linearity and (3.32),

$$\mathbb{E}\Big[g_t^{(\lambda)} \,\Big|\, s_t, a_t\Big] \;=\; Q^\pi(s_t, a_t).$$

Using $\widehat{A}_t^{(\lambda)} = g_t^{(\lambda)} - V^\pi(s_t)$ gives

$$\mathbb{E}\Big[\widehat{A}_t^{(\lambda)} \,\Big|\, s_t, a_t\Big] \;=\; Q^\pi(s_t, a_t) - V^\pi(s_t) \;=\; A^\pi(s_t, a_t),$$

which satisfies (3.3.3). Plugging $\widehat{A}_t^{(\lambda)}$ into (3.3.3) thus yields an unbiased policy-gradient estimator.

The pseudocode for GAE is presented below.

On-Policy Actor–Critic with Generalized Advantage Estimation (GAE)

Inputs: policy $\pi_\theta(a \mid s)$, value $V_\phi(s)$, discount $\gamma \in [0, 1)$, GAE parameter $\lambda \in [0, 1]$; stepsizes $\alpha_\theta, \alpha_\phi > 0$; rollout length T; minibatch size B.

For iterations $k = 0, 1, 2, \ldots$:

1. **Collect rollouts.** Run π_θ to collect B trajectories and each trajectory has T steps , storing $(s_t, a_t, r_t, s_{t+1}, \text{done}_t)$.
2. **Values & residuals.** Compute

   $$v_t \leftarrow V_\phi(s_t), \quad v_{t+1} \leftarrow V_\phi(s_{t+1}), \quad m_t \leftarrow 1 - \text{done}_t, \quad \delta_t \leftarrow r_t + \gamma m_t v_{t+1} - v_t.$$

3. **Backward GAE.** Set $\widehat{A}_T \leftarrow 0$, and for $t = T - 1$ to 0 do:

   $$\widehat{A}_t \leftarrow \delta_t + \gamma\lambda m_t \widehat{A}_{t+1}.$$

   (Optionally normalize $\{\widehat{A}_t\}$ within the minibatch.)

4. **Critic target (λ-return).** Set critic target

$$\widehat{V}_t^{\text{targ}} \leftarrow \widehat{A}_t + v_t \;\; (= g_t^{(\lambda)}).$$

5. **Critic update.** Gradient descent:

$$\phi \leftarrow \phi - \alpha_\phi \nabla_\phi \frac{1}{B} \sum_t \left(V_\phi(s_t) - \widehat{V}_t^{\text{targ}} \right)^2.$$

(Often take several critic steps here.)

6. **Actor update.** Gradient ascent

$$\theta \leftarrow \theta + \alpha_\theta \frac{1}{B} \sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \widehat{A}_t.$$

The next example applies GAE to the cart-pole balancing problem.

Example 3.4 (GAE for Cart-Pole Balancing)**.** Fig. 3.9 shows the learning curve using Actor-Critic with GAE and Fig. 3.10 shows a sample rollout of the trained policy.

The Python code can be found here.

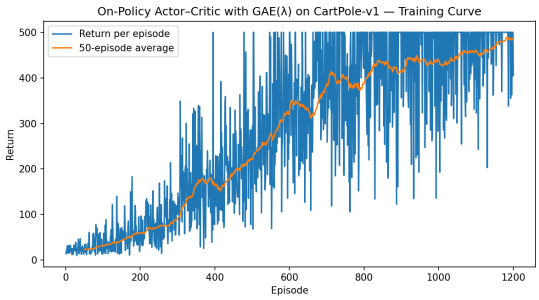

Figure 3.9: Learning curve (Actor–Critic with GAE).

3.3.4 Off-Policy Actor–Critic

On-policy actor–critic discards data after a single update. *Off-policy* methods decouple the behavior policy (that collects data) from the target policy (that we improve), enabling replay buffers and better sample efficiency.

Off-Policy Policy Gradient. When data come from a behavior policy $b \neq \pi_\theta$, define the per-decision likelihood ratio

$$\rho_t \;=\; \frac{\pi_\theta(a_t \mid s_t)}{b(a_t \mid s_t)}.$$

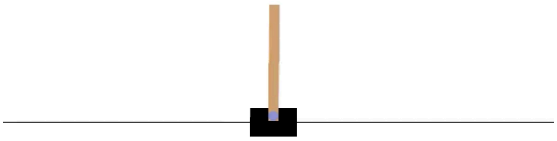

Figure 3.10: Policy rollout (Actor–Critic with GAE).

A basic off-policy policy gradient with an advantage target $\widehat{A}_t$ is

$$\widehat{\nabla_\theta J} \;=\; \mathbb{E}\Big[\rho_t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, \widehat{A}_t\Big] \,. \tag{3.33}$$

In practice we often clip the ratio to control variance:

$$\bar{\rho}_t \;=\; \min\{\rho_t,\; c\}, \quad c \geq 1, \qquad \widehat{\nabla_\theta J} \approx \mathbb{E}\Big[\bar{\rho}_t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, \widehat{A}_t\Big] \,.$$

Clipping introduces small bias but usually reduces variance.

Off-policy Critic. A convenient choice is an *action-value critic* $Q_\psi(s, a)$ trained with an expected SARSA style target under the current π_θ:

$$\begin{aligned} y_t &= r_t \;+\; \gamma \, \mathbb{E}_{a' \sim \pi_\theta(\cdot \mid s_{t+1})} \Big[Q_{\bar{\psi}}(s_{t+1}, a') \Big] \,, \\ \psi &\leftarrow \arg\min_{\psi} \; \mathbb{E}\big[(Q_\psi(s_t, a_t) - y_t)^2\big], \end{aligned}$$

where $Q_{\bar{\psi}}$ is a target network used to stabilize bootstrapping (i.e., mitigate the deadly triad). For discrete actions, the expectation is an exact sum $\sum_{a'} \pi_\theta(a' \mid s') Q_{\bar{\psi}}(s', a')$; for continuous, we approximate the expectation with a few samples $a' \sim \pi_\theta(\cdot \mid s')$.

Advantage. Given Q_ψ, we can estimate the advantage by

$$\widehat{A}_t \;=\; Q_\psi(s_t, a_t) \;-\; V_\psi(s_t), \quad V_\psi(s) \;\equiv\; \mathbb{E}_{a \sim \pi_\theta(\cdot \mid s)}[Q_\psi(s, a)].$$

Again, for discrete actions, we can compute V_ψ exactly; for continuous actions, we approximate using a few samples.

Pseudocode for off-policy actor-critic is presented below.

Experience-Replay Off-Policy Actor–Critic

Inputs: target policy π_θ, Q-critic Q_ψ (and target $Q_{\bar{\psi}}$), discount γ, stepsizes $\alpha_\theta, \alpha_\psi$, replay buffer $\mathcal{D}$, IS clip $c \geq 1$, minibatch size B.

Initialize: $\bar{\psi} \leftarrow \psi$. Behavior policy b can be π_θ with exploration (e.g., ε-greedy).

For iterations $k = 0, 1, 2, ...$:

1. **Interact & store.** Use b to step the env and append to $\mathcal{D}$ tuples $(s_t, a_t, r_t, s_{t+1}, \text{done}_t,\ p_t^b)$, where $p_t^b = b(a_t \mid s_t)$ (store this to compute ρ_t).

2. **Sample minibatch** Sample transitions $\{(s, a, r, s', d, p^\mu)\}_{i=1}^B$ from the replay buffer $\mathcal{D}$.

3. **Critic target (expected SARSA).**

   - Compute $\pi_\theta(a' \mid s')$ and $Q_{\bar{\psi}}(s', a')$.
   - Set $y \leftarrow r + \gamma(1 - \text{done}_t) \sum_{a'} \pi_\theta(a' \mid s')\, Q_{\bar{\psi}}(s', a')$. (for continuous actions: perform sample average)

4. **Critic update.**

$$\psi \leftarrow \psi - \alpha_\psi \nabla_\psi \frac{1}{B} \sum_{i=1}^{B} \left(Q_\psi(s_i, a_i) - y_i\right)^2.$$

   (Optionally clip gradients; perform multiple critic steps.)

5. **Actor advantage.**

   - Compute $V_\psi(s) = \sum_a \pi_\theta(a \mid s)\, Q_\psi(s, a)$ (or sample-average for continuous actions).

   - Set $\widehat{A} = Q_\psi(s, a) - V_\psi(s)$; optionally normalize $\widehat{A}$ within the batch.

6. **Importance ratios (clipped).**

$$\rho \leftarrow \frac{\pi_\theta(a \mid s)}{p^b}, \qquad \bar{\rho} \leftarrow \min\{\rho,\ c\}.$$

7. **Actor update.**

$$\theta \leftarrow \theta + \alpha_\theta \frac{1}{B} \sum_{i=1}^{B} \bar{\rho}_i \nabla_\theta \log \pi_\theta(a_i \mid s_i)\, \widehat{A}_i.$$

8. **Target network (moving average).**

$$\bar{\psi} \leftarrow \tau\, \psi + (1 - \tau)\, \bar{\psi}.$$

Notes & Variants.

- **Unbiased vs. biased:** Without clipping and with a correct critic/advantage, (3.33) is unbiased; clipping $\bar{\rho}$ adds bias but improves variance.

- **Critic options:** You can learn V_ϕ instead of Q_ψ using off-policy TD with IS; using Q with an expected SARSA target avoids IS in the critic while keeping evaluation under π_θ.

- **Behavior refresh:** Periodically update b toward π_θ (reduce exploration) to keep ratios well-behaved.

The next example applies off-policy actor-critic to cart-pole balancing.

Example 3.5 (Off-Policy Actor-Critic for Cart-Pole Balancing)**.** Fig. 3.11 shows the learning curve of applying off-policy actor-critic to cart-pole balancing.

Fig. 3.12 shows a sample rollout of the learned policy.

The Python code can be found here.

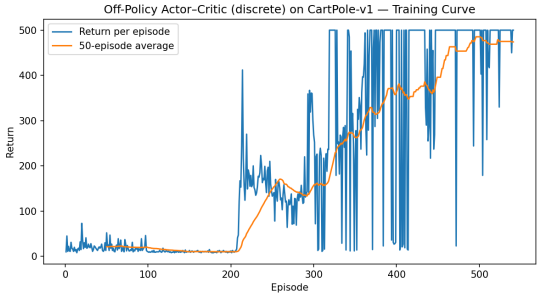

Figure 3.11: Learning curve (Off-Policy Actor–Critic).

The next example applies off-policy actor-critic to a control problem with a continuous action space.

Example 3.6 (Off-Policy Actor-Critic for Inverted Pendulum)**.** Consider the Inverted Pendulum problem illustrated in Fig. 3.13. The state of the pendulum is $s = (\theta, \dot{\theta})$, or equivalently, $s = (x, y, \dot{\theta})$ with $x = \cos(\theta), y = \sin(\theta)$. The action space is continuous: $\tau \in \mathcal{A} = [-2, 2]$.

The dynamics of the pendulum is specified by Gym, and the reward is

$$R(s, \tau) = -(\theta^2 + 0.1\dot{\theta}^2 + 0.001\tau^2).$$

The episode truncates at 200 time steps.

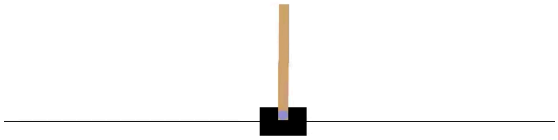

Figure 3.12: Policy rollout (Off-Policy Actor–Critic).

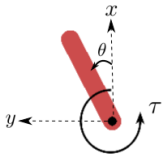

Figure 3.13: Illustration of Inverted Pendulum in Gym.

Fig. 3.14 shows the learning curve of applying off-policy actor-critic to the pendulum problem.

Fig. 3.15 shows a sample rollout of the learned policy.

You can find the Python code here.

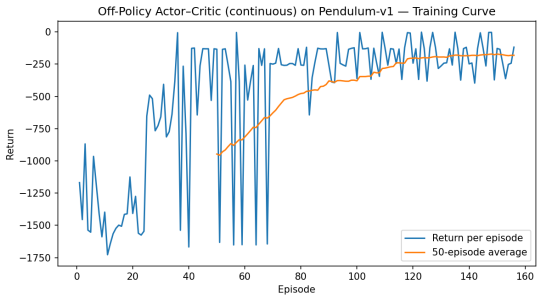

Figure 3.14: Learning curve (Off-Policy Actor-Critic).

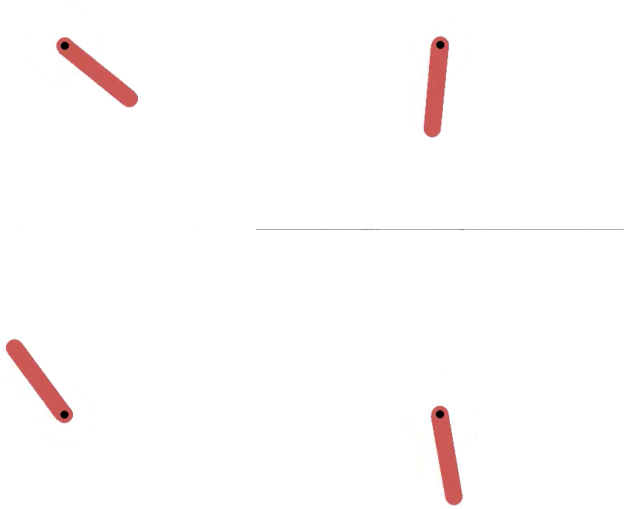

Figure 3.15: Policy rollout (Off-Policy Actor-Critic).

3.4 Advanced Policy Gradients

3.4.1 Revisiting Generalized Policy Iteration

Recall from Chapter 2 that **generalized policy iteration (GPI)** extends tabular policy iteration (with known dynamics) to unknown-dynamics settings. At a high level, GPI iterates over policies; at iteration k it performs:

1. **Policy evaluation.** Use the current policy π_k to generate N trajectories and estimate either the Q-function $\widehat{Q}^{\pi_k}(s,a)$ or the advantage function $\widehat{A}^{\pi_k}(s,a)$, using function approximation. This can be done, for example, with the GAE algorithm introduced in Section 3.3.3, and is the "critic" in the Actor–Critic family of methods.

2. **Policy improvement.** Construct a new policy π_{k+1} that (approximately) prefers actions deemed better by $\widehat{Q}^{\pi_k}$ or $\widehat{A}^{\pi_k}$:

$$\pi_{k+1}(s) \approx \arg\max_a \widehat{Q}^{\pi_k}(s,a) \;=\; \arg\max_a \widehat{A}^{\pi_k}(s,a).$$

   In policy gradients, we approximate $\arg\max_a \widehat{A}^{\pi_k}(s,a)$ via gradient ascent in a, i.e., using

$$\nabla_\theta J(\theta) = \frac{1}{1-\gamma}\,\mathbb{E}_{s\sim d_{\pi_k},\, a\sim\pi_k}\big[\nabla_\theta \log \pi_\theta(a \mid s)\, \widehat{A}^{\pi_k}(s,a)\big]. \tag{3.34}$$

A key observation is that we use an advantage estimate obtained from data generated by π_k (the old policy) to produce a new policy. In the tabular case, this improvement step guarantees monotonic improvement of π_{k+1} over π_k, because the evaluation produces a value (or advantage) estimate over the entire state space. In continuous state spaces, this no longer holds: we typically can only obtain an advantage estimate that is accurate *along the state–action distribution induced by* π_k rather than globally over $\mathcal{S}\times\mathcal{A}$. (If, however, we use off-policy data, the expectation here can be different.)

The question "*how much better is* π_{k+1} *than* π_k*?*" motivates a relation between the performances of two policies that explicitly accounts for distribution shift.

3.4.2 Performance Difference Lemma

The following performance difference lemma (PDL) expresses the return gap between two policies in terms of the (old) policy's advantage and the (new) policy's state-action visitation:

Theorem 3.8 (Performance Difference Lemma)**.** *Let π and π' be two stationary policies in a discounted MDP with $\gamma \in [0, 1)$. Then*

$$J(\pi') - J(\pi) \;=\; \frac{1}{1-\gamma}\, \mathbb{E}_{s\sim d^{\pi'},\, a\sim\pi'}[A^{\pi}(s,a)]\,, \tag{3.35}$$

where $d^{\pi}(s) = (1-\gamma)\sum_{t=0}^{\infty} \gamma^t \Pr_{\pi}(s_t = s)$ is the (discounted) state-visitation distribution generated by policy π and $A^{\pi} = Q^{\pi} - V^{\pi}$ is the advantage.

Interpretation. The performance difference lemma highlights **distribution shift**: the advantage is evaluated under policy π, while the expectation is taken over the state–action distribution induced by π'. In policy gradients, when performing a step using (3.34), we are approximately maximizing the surrogate

$$\mathcal{L}_{\pi}(\pi') := \frac{1}{1-\gamma}\, \mathbb{E}_{s\sim d^{\pi},\, a\sim\pi'}[A^{\pi}(s,a)],$$

where the state distribution is d_{π}, not $d_{\pi'}$. To guarantee improvement, we want this surrogate to reflect the true gain $J(\pi') - J(\pi)$. The two coincide when $d^{\pi'} \approx d^{\pi}$. Hence, **keep π' close to π** so state visitation does not change dramatically, making the surrogate reliable (to some extent, off-policy versions of actor–critic aim to achieve this). This "stay local" principle underpins TRPO, NPG, and PPO.

3.4.3 Trust Region Constraint

How to enforce the new policy $\pi_{\theta_{k+1}}$ to be close to the old policy π_{θ_k}?

KL Divergence. The Kullback–Leibler (KL) divergence is a type of statistical distance: a measure of how much an approximating probability distribution Q is different from a true probability distribution P. Formally, let P and Q be two probability distributions supported on $\mathcal{X}$, the KL divergence between P and Q is

$$D_{\mathrm{KL}}(P\|Q) = \sum_{x\in\mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right) = \mathbb{E}_{x\sim P(x)}\left[\log\left(\frac{P(x)}{Q(x)}\right)\right]. \tag{3.36}$$

For example, when $P = Q$, we have $D_{\mathrm{KL}}(P\|Q) = 0$. Indeed, $D_{\mathrm{KL}}(P\|Q) \geq 0$ and the equality holds if and only if $P = Q$.

Trust Region Constraint. We now augment the usual policy optimization problem with a trust region constraint defined by the KL divergence. In particular, we wish to improve the current policy π_{θ_k} **locally** by maximizing a **surrogate advantage objective** while constraining the **expected KL divergence** from the old policy. This keeps the new policy π_{θ} close to π_{θ_k}, so the surrogate built under $d^{\pi_{\theta_k}}$ remains predictive of true improvement.

Formally, let θ_k denote the current policy parameters. Define the importance ratio

$$\rho_\theta(s,a) = \frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}.$$

We aim to maximize the on-policy surrogate

$$L_{\theta_k}(\theta) \;=\; \mathbb{E}_{s\sim d^{\pi_{\theta_k}},\, a\sim\pi_{\theta_k}}\big[\rho_\theta(s,a)\, \widehat{A}^{\pi_{\theta_k}}(s,a)\big], \tag{3.37}$$

subject to an expected KL constraint measured under the old state distribution:

$$\bar{D}_{\mathrm{KL}}(\theta_k \,\|\, \theta) \;:=\; \mathbb{E}_{s\sim d^{\pi_{\theta_k}}}\Big[D_{\mathrm{KL}}(\pi_{\theta_k}(\cdot \mid s) \,\|\, \pi_\theta(\cdot \mid s))\Big] \;\le\; \delta, \tag{3.38}$$

with a small radius $\delta > 0$. In summary, we are now interested in the following constrained policy optimization problem:

$$\begin{aligned} \max_{\theta} \quad & L_{\theta_k}(\theta) \\ \text{subject to} \quad & \bar{D}_{\mathrm{KL}}(\theta_k \,\|\, \theta) \le \delta. \end{aligned} \tag{3.39}$$

3.4.4 Natural Policy Gradient

The natural policy gradient method (Kakade, 2001) can be seen as first performing a linear approximation to the objective of (3.39) and a quadratic approximation to the constraint of (3.39), and then solve the resulting approximate problem in closed form.

Leading-Order Approximation. To maximize the surrogate $L_{\theta_k}(\theta)$ in (3.37) subject to the KL trust-region constraint (3.38), we linearize the surrogate around θ_k and quadratically approximate the KL trust region constraint. This leads to the following convex quadratic program (QP)

$$\max_{\Delta\theta} \quad g^\top \Delta\theta \quad \text{s.t.} \quad \frac{1}{2}\,\Delta\theta^\top F(\theta_k)\,\Delta\theta \;\le\; \delta, \tag{3.40}$$

where

$$g \;=\; \nabla_\theta L_{\theta_k}(\theta)\big|_{\theta=\theta_k} = \mathbb{E}_{s\sim d^{\pi_{\theta_k}},\, a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta \log \pi_{\theta_k}(a \mid s)\, \widehat{A}(s,a)\big] \tag{3.41}$$

is the policy gradient, and

$$F(\theta_k) \;=\; \mathbb{E}_{s\sim d^{\pi_{\theta_k}},\, a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta \log \pi_{\theta_k}(a \mid s)\, \nabla_\theta \log \pi_{\theta_k}(a \mid s)^\top\big] \tag{3.42}$$

is the (empirical) *Fisher information* of the policy under the old distribution. See a proof in Section 3.4.5.

One can show that the QP (3.40) has a closed-form solution:

$$p_{\mathrm{NPG}} \;=\; F(\theta_k)^{-1}\, g, \qquad \Delta\theta_{\mathrm{NPG}} \;=\; \sqrt{\frac{2\delta}{g^\top F(\theta_k)^{-1} g}}\; p_{\mathrm{NPG}}, \tag{3.43}$$

where p_{NPG} is called the *natural policy gradient*, for the reason that the usual policy gradient g is pre-multiplied by $F(\theta_k)^{-1}$, which contains the second-order curvature of the KL constraint. In practice, p_{NPG} is computed with conjugate gradient (CG) using Fisher–vector products; no matrices are formed. In (3.43),

$$\alpha = \sqrt{\frac{2\delta}{g^\top F(\theta_k)^{-1} g}} = \sqrt{\frac{2\delta}{p_{\text{NPG}}^\top F(\theta_k) p_{\text{NPG}}}}$$

is often called the trust-region step size.

The following pseudocode implements NPG with GAE as the critic.

Natural Policy Gradient (with GAE advantages)

Inputs: initial policy θ_0; value/critic ϕ_0; discount γ; GAE parameter λ; KL radius δ (or learning rate η); CG iterations K_{cg}; (optional) damping $\xi > 0$.

For iterations $k = 0, 1, 2, \ldots$:

1. **Collect rollouts (on-policy).** Run π_{θ_k} to obtain a batch $\{(s_t, a_t, r_t, s_{t+1}, \text{done}_t)\}_{t=1}^N$; cache $\log \pi_{\theta_k}(a_t \mid s_t)$.
2. **Critic / advantages (GAE).**
   Compute TD residuals $\delta_t = r_t + \gamma(1 - \text{done}_t)V_\phi(s_{t+1}) - V_\phi(s_t)$;
   backward recursion $\widehat{A}_t = \delta_t + \gamma\lambda(1 - \text{done}_t)\widehat{A}_{t+1}$, with $\widehat{A}_T = 0$;
   (optionally) standardize $\widehat{A}$; set value targets $\widehat{V}_t^{\text{targ}} = \widehat{A}_t + V_\phi(s_t)$.
3. **Value update.** Fit V_ϕ by minimizing $\sum_t (V_\phi(s_t) - \widehat{V}_t^{\text{targ}})^2$ (one or several epochs).
4. **Surrogate gradient.**
$$g \;=\; \frac{1}{N} \sum_t \nabla_\theta \log \pi_{\theta_k}(a_t \mid s_t)\, \widehat{A}_t.$$
5. **Fisher–vector product (FvP).** Define the empirical KL $\bar{D}_{\text{KL}}(\theta_k \,\|\, \theta)$. Implement $v \mapsto Fv$ as the **Hessian–vector product** of $\bar{D}_{\text{KL}}$ at θ_k (optionally use **damping** $F \leftarrow F + \xi I$ to make sure F is positive definite).
6. **Conjugate gradient (CG).** Approximately solve $(F)\, p = g$ to obtain $p_{\text{NPG}} \approx F^{-1} g$.
7. **Step size.**
   - **Trust-region scaling:** set $\alpha \leftarrow \sqrt{\frac{2\delta}{p_{\text{NPG}}^\top F p_{\text{NPG}}}}$ and update $\theta_{k+1} \leftarrow \theta_k + \alpha p_{\text{NPG}}$.
   - **Fixed-rate natural step:** choose $\eta > 0$ and set $\theta_{k+1} \leftarrow \theta_k + \eta p_{\text{NPG}}$ (monitor empirical KL for safety).

3.4.5 Proof of Fisher Information

Let the expected KL trust-region constraint (measured under the old policy's state distribution) be

$$\bar{D}_{\mathrm{KL}}(\theta_k \,\|\, \theta) \;:=\; \mathbb{E}_{s\sim d^{\pi_{\theta_k}}}\Big[D_{\mathrm{KL}}(\pi_{\theta_k}(\cdot \mid s) \,\|\, \pi_\theta(\cdot \mid s))\Big].$$

Write $\theta = \theta_k + \Delta\theta$ and define, for a fixed state s,

$$f_s(\theta) \;=\; D_{\mathrm{KL}}(\pi_{\theta_k}(\cdot \mid s) \,\|\, \pi_\theta(\cdot \mid s)) = \mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\big[\log \pi_{\theta_k}(a \mid s) - \log \pi_\theta(a \mid s)\big].$$

We will show that the *second-order Taylor expansion* of $\bar{D}_{\mathrm{KL}}$ around θ_k is

$$\bar{D}_{\mathrm{KL}}(\theta_k \,\|\, \theta_k{+}\Delta\theta) \;=\; \frac{1}{2}\,\Delta\theta^\top \underbrace{\mathbb{E}_{s\sim d^{\pi_{\theta_k}},\, a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta \log \pi_{\theta_k}(a \mid s)\, \nabla_\theta \log \pi_{\theta_k}(a \mid s)^\top\big]}_{F(\theta_k)\ \text{(Fisher information)}}\, \Delta\theta + \mathcal{O}(\|\Delta\theta\|^3).$$

Step 1: Zeroth- and first-order terms vanish at $\theta = \theta_k$. For each s,

$$f_s(\theta_k) = D_{\mathrm{KL}}(\pi_{\theta_k} \,\|\, \pi_{\theta_k}) = 0.$$

The gradient (holding the expectation under π_{θ_k}) is

$$\nabla_\theta f_s(\theta) = -\mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta \log \pi_\theta(a \mid s)\big].$$

Evaluating at $\theta = \theta_k$,

$$\begin{aligned}\nabla_\theta f_s(\theta_k) &= -\mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta \log \pi_{\theta_k}(a \mid s)\big] \\ &= -\sum_a \pi_{\theta_k}(a \mid s)\, \nabla_\theta \log \pi_{\theta_k}(a \mid s) = -\nabla_\theta \sum_a \pi_{\theta_k}(a \mid s) = 0,\end{aligned}$$

using the normalization $\sum_a \pi_{\theta_k}(a \mid s) = 1$. Hence both the value and the first-order term are zero.

Step 2: The Hessian equals the (per-state) Fisher information. The Hessian of f_s is

$$\nabla_\theta^2 f_s(\theta) = -\mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta^2 \log \pi_\theta(a \mid s)\big].$$

At $\theta = \theta_k$, apply the *information identity* (a.k.a. Bartlett identity):

$$-\mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta^2 \log \pi_{\theta_k}(a \mid s)\big] = \mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta \log \pi_{\theta_k}(a \mid s)\, \nabla_\theta \log \pi_{\theta_k}(a \mid s)^\top\big].$$

Proof sketch of the identity: start from $\sum_a \pi_\theta(a \mid s) = 1$, differentiate once to get $\mathbb{E}_{a\sim\pi_\theta}[\nabla \log \pi_\theta] = 0$; differentiate again and use the product rule to obtain $\mathbb{E}_{a\sim\pi_\theta}[\nabla^2 \log \pi_\theta + (\nabla \log \pi_\theta)(\nabla \log \pi_\theta)^\top] = 0$.

Thus,

$$\nabla_\theta^2 f_s(\theta_k) = \mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\big[\nabla_\theta \log \pi_{\theta_k}(a \mid s)\, \nabla_\theta \log \pi_{\theta_k}(a \mid s)^\top\big] \;=:\; F_s(\theta_k).$$

Step 3: Second-order Taylor expansion and averaging over states. For each s,

$$f_s(\theta_k + \Delta\theta) = \frac{1}{2}\,\Delta\theta^\top F_s(\theta_k)\,\Delta\theta \;+\; \mathcal{O}(\|\Delta\theta\|^3).$$

Taking expectation over $s \sim d^{\pi_{\theta_k}}$ gives

$$\bar{D}_{\mathrm{KL}}(\theta_k \,\|\, \theta_k{+}\Delta\theta) = \mathbb{E}_{s\sim d^{\pi_{\theta_k}}}[f_s(\theta_k{+}\Delta\theta)] = \frac{1}{2}\,\Delta\theta^\top \underbrace{\mathbb{E}_{s\sim d^{\pi_{\theta_k}}}[F_s(\theta_k)]}_{F(\theta_k)}\,\Delta\theta + \mathcal{O}(\|\Delta\theta\|^3).$$

Conclusion. The Fisher information $F(\theta_k)$ is exactly the Hessian of the expected KL at θ_k. Therefore, the KL trust-region constraint admits the quadratic local approximation

$$\bar{D}_{\mathrm{KL}}(\theta_k \,\|\, \theta_k + \Delta\theta) \;\approx\; \frac{1}{2}\,\Delta\theta^\top F(\theta_k)\,\Delta\theta,$$

which yields the TRPO/NPG quadratic constraint and identifies $F(\theta_k)$ as the local metric tensor of the policy manifold.

3.4.6 Trust Region Policy Optimization

The NPG algorithm presented above leverages a leading-order approximation of the KL-constrained policy optimization problem (3.39).

In Trust Region Policy Optimization (Schulman et al., 2015a), we still use the leading-order approximation to obtain the natural policy gradient direction, but additionally, we perform a *backtracking line search* to enforce the true (nonlinear) KL constraint and surrogate improvement.

The following pseudocode implements TRPO with GAE as the critic.

TRPO (with GAE advantages)

Inputs: initial policy θ_0; value/critic parameters ϕ_0; discount γ; GAE parameter λ; KL radius δ; CG iterations K_{cg}; backtrack factor $\beta \in (0,1)$; max backtracks M.

For iterations $k = 0, 1, 2, \ldots$:

1. **Collect rollouts (on-policy).** Run π_{θ_k} to obtain trajectories; build a batch $\{(s_t, a_t, r_t, s_{t+1}, \mathrm{done}_t)\}_{t=1}^{N}$.

2. **Critic / advantages (GAE).**
   Compute TD residuals $\delta_t = r_t + \gamma(1-\text{done}_t)V_\phi(s_{t+1}) - V_\phi(s_t)$;
   backward recursion $\widehat{A}_t = \delta_t + \gamma\lambda(1-\text{done}_t)\widehat{A}_{t+1}$, with $\widehat{A}_T = 0$;
   (optionally) standardize $\widehat{A}$ within the batch; set value targets $\widehat{V}_t^{\text{targ}} = \widehat{A}_t + V_\phi(s_t)$.

3. **Value function update.** Fit V_ϕ by minimizing $\sum_t (V_\phi(s_t) - \widehat{V}_t^{\text{targ}})^2$ (one or several epochs).

4. **Policy gradient at θ_k.**

$$g \;=\; \nabla_\theta L_{\theta_k}(\theta)\big|_{\theta=\theta_k} \approx \frac{1}{N}\sum_t \nabla_\theta \log \pi_{\theta_k}(a_t \mid s_t)\,\widehat{A}_t.$$

5. **Fisher–vector product (FvP).** Define the Fisher information under π_{θ_k}:

$$F(\theta_k) \;=\; \mathbb{E}\Big[\nabla_\theta \log \pi_{\theta_k}(a \mid s)\,\nabla_\theta \log \pi_{\theta_k}(a \mid s)^\top\Big].$$

   Implement $v \mapsto Fv$ via the **Hessian-vector product** of the empirical KL.

6. **Conjugate gradient (CG) solve.** Approximately solve $Fp = g$ with K_{cg} CG iterations to get the natural direction $p_{\text{NPG}} \approx F^{-1}g$.

7. **Compute step size for the quadratic trust region.**

$$\alpha \;\leftarrow\; \sqrt{\frac{2\delta}{p_{\text{NPG}}^\top F p_{\text{NPG}}}}.$$

   Candidate update: $\theta^\star \leftarrow \theta_k + \alpha p_{\text{NPG}}$.

8. **Backtracking line search (feasibility + improvement).** Repeatedly set $\theta^\star \leftarrow \theta_k + \beta^j \alpha p_{\text{NPG}}$ for $j = 0, 1, \ldots, M$ until both hold on the batch:

   - **KL constraint:** $\bar{D}_{\text{KL}}(\theta_k \,\|\, \theta^\star) \le \delta$.
   - **Surrogate improvement:** $L_{\theta_k}(\theta^\star) \ge L_{\theta_k}(\theta_k)$.

   Accept the first $\theta^\star$ that satisfies both; set $\theta_{k+1} \leftarrow \theta^\star$.

3.4.6.1 Backtracking Line Search

Batch-only evaluation. During TRPO's line search you **do not collect new trajectories**. All checks are computed on the same batch gathered with the old policy π_{θ_k} (i.e., under $d^{\pi_{\theta_k}}$).

Given: a candidate update $\theta^\star = \theta_k + \beta^j \alpha p_{\text{NPG}}$.

1. **Empirical KL constraint (nonlinear, "true" KL).** Compute the state-wise KL between the full action distributions of the old and candidate policies and average over the batch states:

$$\widehat{\widehat{D}}_{\mathrm{KL}}(\theta_k \,\|\, \theta^\star) \;=\; \frac{1}{|\mathcal{B}|}\sum_{s\in\mathcal{B}} D_{\mathrm{KL}}(\pi_{\theta_k}(\cdot \mid s) \,\|\, \pi_{\theta^\star}(\cdot \mid s)).$$

   - **Categorical policy:**

$$D_{\mathrm{KL}}(\pi_{\theta_k} \,\|\, \pi_{\theta^\star}) \;=\; \sum_a \pi_{\theta_k}(a \mid s)\Big[\log \pi_{\theta_k}(a \mid s) - \log \pi_{\theta^\star}(a \mid s)\Big].$$

   - **Gaussian policy** (mean $\mu(s)$, covariance $\Sigma(s)$; use pre-squash distribution if actions are squashed):

$$D_{\mathrm{KL}}(\mathcal{N}(\mu_k,\Sigma_k) \,\|\, \mathcal{N}(\mu_\star,\Sigma_\star)) = \frac{1}{2}\Big(\operatorname{tr}(\Sigma_\star^{-1}\Sigma_k)+(\mu_\star-\mu_k)^\top\Sigma_\star^{-1}(\mu_\star-\mu_k)-d+\log\tfrac{\det\Sigma_\star}{\det\Sigma_k}\Big).$$

   Feasibility test: accept if $\widehat{\widehat{D}}_{\mathrm{KL}}(\theta_k \,\|\, \theta^\star) \le \delta$ (cf. (3.38)).

2. **Surrogate improvement.** Evaluate the TRPO surrogate $L_{\theta_k}(\theta)$ (cf. (3.37)) on the same batch using importance ratios from $\theta^\star$:

$$\widehat{L}_{\theta_k}(\theta^\star) \;=\; \frac{1}{|\mathcal{B}|}\sum_{(s,a)\in\mathcal{B}} \frac{\pi_{\theta^\star}(a \mid s)}{\pi_{\theta_k}(a \mid s)}\,\widehat{A}^{\pi_{\theta_k}}(s,a), \qquad \widehat{L}_{\theta_k}(\theta_k) = \frac{1}{|\mathcal{B}|}\sum_{(s,a)} \widehat{A}^{\pi_{\theta_k}}(s,a).$$

   Improvement test: accept if $\widehat{L}_{\theta_k}(\theta^\star) \ge \widehat{L}_{\theta_k}(\theta_k)$.

3. **Backtracking loop (on-batch).** Decrease the step by $\beta \in (0,1)$ until both tests pass or a maximum of M backtracks is reached:

$$\theta^\star \leftarrow \theta_k + \beta^j \alpha p_{\mathrm{NPG}}, \quad j = 0, 1, \ldots, M.$$

   If successful, set $\theta_{k+1} \leftarrow \theta^\star$; otherwise keep $\theta_{k+1} \leftarrow \theta_k$.

3.4.7 Proximal Policy Optimization

While NPG/TRPO are stable, they may be computationally heavier due to constrained solves or natural-step systems. Proximal Policy Optimization (PPO) (Schulman et al., 2017) replaces the hard constraint with a *penalized (regularized) objective* and optimizes it with standard first-order SGD:

$$\ell_k(\theta) \;=\; \mathbb{E}_{s\sim d^{\pi_{\theta_k}},a\sim\pi_{\theta_k}}\big[\,\rho_\theta(s,a)\,\widehat{A}^{\pi_{\theta_k}}(s,a)\,\big] \;-\; \lambda\,\mathbb{E}_{s\sim d^{\pi_{\theta_k}},a\sim\pi_{\theta_k}}\left[\log\frac{\pi_{\theta_k}(a \mid s)}{\pi_\theta(a \mid s)}\right], \tag{3.44}$$

where $\lambda > 0$ and the second term is the *per-sample KL penalty* that discourages large departures from π_{θ_k}. Conceptually, this is a *Lagrangian relaxation* of TRPO's trust region, where the hard constraint is moved to the objective function as a soft penalty.

3.4.7.1 Gradient of the KL–Regularized Surrogate

Treat $\widehat{A}^{\pi_{\theta_k}}$ and the sampling distribution as fixed during the policy update. Using $\nabla_\theta \rho_\theta = \rho_\theta \nabla_\theta \log \pi_\theta$ and $\nabla_\theta \log \frac{\pi_{\theta_k}}{\pi_\theta} = -\nabla_\theta \log \pi_\theta$, the gradient of the KL-regularized objective (3.44) is

$$\nabla_\theta \ell_k(\theta) \;=\; \mathbb{E}_{s\sim d^{\pi_{\theta_k}}, a\sim\pi_{\theta_k}}\Big[\nabla_\theta \log \pi_\theta(a \mid s)\, \underbrace{\big(\rho_\theta(s,a)\, \widehat{A}(s,a) - \lambda\big)}_{\text{effective advantage}}\Big].$$

This shows the KL penalty shifts the effective advantage by $-\lambda$.

3.4.7.2 From the Lagrangian Relaxation to PPO Updates

There are two standard PPO realizations:

1. **PPO–KL (penalty version).** Directly ascend $\ell_k(\theta)$ with minibatch SGD:

$$\theta \leftarrow \theta + \alpha\, \frac{1}{B} \sum_{(s,a)\in\mathcal{B}} \nabla_\theta \log \pi_\theta(a \mid s)\, (\rho_\theta(s,a)\, \widehat{A}(s,a) - \lambda).$$

   After each epoch, measure the empirical KL $\widehat{\bar{D}}_{\mathrm{KL}}(\theta_k \,\|\, \theta)$ on the batch; increase λ if KL is too high (tighten the region), decrease λ if it is too low.

2. **PPO–Clip (clipping version).** Replace the penalty with a *hard* trust region on the ratio ρ_θ. When $\widehat{A} > 0$, forbid $\rho_\theta > 1+\varepsilon$; when $\widehat{A} < 0$, forbid $\rho_\theta < 1-\varepsilon$. This yields the clipped objective

$$\ell_k^{\mathrm{CLIP}}(\theta) \;=\; \mathbb{E}_{s\sim d^{\pi_{\theta_k}}, a\sim\pi_{\theta_k}}\Big[\min\big(\rho_\theta(s,a)\, \widehat{A}(s,a),\; \mathrm{clip}(\rho_\theta(s,a),\, 1-\varepsilon,\, 1+\varepsilon)\, \widehat{A}(s,a)\big)\Big], \tag{3.45}$$

   which is a first-order proxy to the Lagrangian/TRPO trust region: the min/clip term cancels the incentive to move ρ_θ outside $[1-\varepsilon, 1+\varepsilon]$ in directions that would further increase the objective.

Both versions are typically combined with a value-function loss and an entropy bonus to encourage exploration:

$$\mathcal{L}^{\mathrm{PPO}}(\theta,\phi) = -\ell_k^{\mathrm{PG}}(\theta) + c_v\, \mathbb{E}\big[(V_\phi(s) - \widehat{V}^{\,\mathrm{targ}})^2\big] - c_e\, \mathbb{E}\big[\mathcal{H}(\pi_\theta(\cdot \mid s))\big],$$

where ℓ_k^{PG} is either ℓ_k^{CLIP} or ℓ_k.

Why PPO "forbids" ρ_θ from leaving $[1-\varepsilon,\, 1+\varepsilon]$. Let $r \equiv \rho_\theta(s,a) = \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ and $\widehat{A} = \widehat{A}^{\pi_{\theta_k}}(s,a)$. The PPO–Clip objective for one sample is

$$L^{\mathrm{CLIP}}(r, \widehat{A}) = \min(r\, \widehat{A},\; \mathrm{clip}(r, 1-\varepsilon, 1+\varepsilon)\, \widehat{A}).$$

Let's do a case analysis, as shown in Fig. 3.16.

- If $\widehat{A} > 0$: increasing r (i.e., increasing $\pi_\theta(a \mid s)$) raises the *unclipped* term $r\,\widehat{A}$.
  The clipped term equals $(1+\varepsilon)\widehat{A}$ whenever $r > 1+\varepsilon$. Hence

$$L^{\text{CLIP}}(r, \widehat{A}) = \begin{cases} r\,\widehat{A}, & r \le 1+\varepsilon, \\ (1+\varepsilon)\widehat{A}, & r > 1+\varepsilon, \end{cases}$$

  so $\frac{\partial L^{\text{CLIP}}}{\partial r} = \widehat{A}$ for $r \le 1+\varepsilon$ and 0 for $r > 1+\varepsilon$. There is no further gain by pushing r beyond $1+\varepsilon$; the gradient vanishes.
  Intuitively: don't increase an action's probability *too much* even if it looks good—stay proximal.

- If $\widehat{A} < 0$: decreasing r (i.e., reducing $\pi_\theta(a \mid s)$) lowers the unclipped term $r\,\widehat{A}$. The clipped term equals $(1-\varepsilon)\widehat{A}$ whenever $r < 1-\varepsilon$. Thus

$$L^{\text{CLIP}}(r, \widehat{A}) = \begin{cases} r\,\widehat{A}, & r \ge 1-\varepsilon, \\ (1-\varepsilon)\widehat{A}, & r < 1-\varepsilon, \end{cases}$$

  so $\frac{\partial L^{\text{CLIP}}}{\partial r} = \widehat{A}(< 0)$ for $r \ge 1-\varepsilon$ and 0 for $r < 1-\varepsilon$. There is no incentive to shrink r below $1-\varepsilon$; the gradient goes to zero.

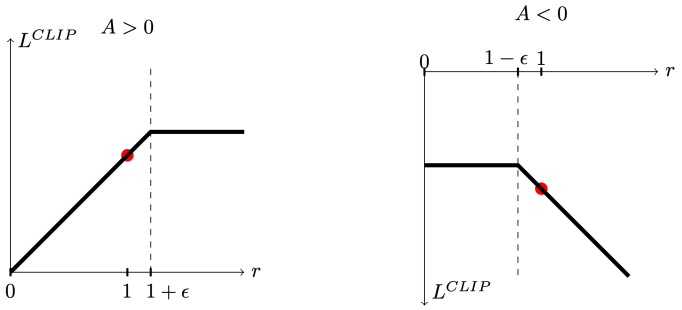

Figure 3.16: The clipped objective function in PPO (from the original PPO paper).

Therefore, the min with a clipped ratio creates flat regions where the objective stops improving in the "profitable" outward direction. This removes the optimization incentive to move r outside $[1-\varepsilon, 1+\varepsilon]$, implementing a per-sample trust region on the probability ratio while retaining the standard policy-gradient inside the bracket.

The following pseudocode implements PPO (clipped version) with GAE.

Proximal Policy Optimization (PPO–Clip)

Inputs: policy π_θ, value V_ϕ, discount γ, GAE λ, clip ε, coefficients c_v, c_e, learning rate α, epochs K_{epoch}, minibatch size B.

For iterations $k = 0, 1, 2, ...$:

1. **Collect on-policy data.** Roll out π_{θ_k} to get trajectories $\{(s_t, a_t, r_t, s_{t+1}, \text{done}_t)\}$. Cache $\log \pi_{\theta_k}(a_t \mid s_t)$.

2. **Compute GAE advantages and value targets.**
   $\delta_t = r_t + \gamma(1 - \text{done}_t)V_\phi(s_{t+1}) - V_\phi(s_t)$
   $\widehat{A}_t = \delta_t + \gamma\lambda(1 - \text{done}_t)\widehat{A}_{t+1}$, with $\widehat{A}_T = 0$.
   $\widehat{V}_t^{\text{targ}} = \widehat{A}_t + V_\phi(s_t)$.
   (Optionally standardize $\{\widehat{A}_t\}$ within the batch.)

3. **Policy/Value optimization (multiple epochs).**
   For $e = 1, \ldots, K_{\text{epoch}}$:

   - Split the batch into minibatches $\mathcal{B}$ of size B.
   - For each $\mathcal{B}$:

$$\rho_\theta(s,a) = \exp(\log \pi_\theta(a \mid s) - \log \pi_{\theta_k}(a \mid s)),$$

$$\ell_{\mathcal{B}}^{\text{CLIP}}(\theta) = \frac{1}{B} \sum_{(s,a)\in\mathcal{B}} \min(\rho_\theta \widehat{A},\ \text{clip}(\rho_\theta, 1-\varepsilon, 1+\varepsilon)\, \widehat{A}),$$

$$\ell_{\mathcal{B}}^{\text{VAL}}(\phi) = \frac{1}{B} \sum_{s\in\mathcal{B}} (V_\phi(s) - \widehat{V}^{\text{targ}})^2, \quad \mathcal{H}_{\mathcal{B}}(\theta) = \frac{1}{B} \sum_{s\in\mathcal{B}} \mathcal{H}(\pi_\theta(\cdot \mid s)).$$

   - The total loss to be minimized is

$$\mathcal{J}_{\mathcal{B}}(\theta, \phi) = -\ell_{\mathcal{B}}^{\text{CLIP}}(\theta) + c_v\, \ell_{\mathcal{B}}^{\text{VAL}}(\phi) - c_e\, \mathcal{H}_{\mathcal{B}}(\theta).$$

   - Take an optimizer step on $\mathcal{J}_{\mathcal{B}}$ (e.g., Adam with learning rate α).

4. **(Optional) Early stopping by KL.**
   Estimate $\widehat{D}_{\text{KL}}(\theta_k \,\|\, \theta)$ on the whole batch; stop inner epochs early if it exceeds a threshold.

3.4.8 Soft Actor–Critic

Standard actor–critic methods maximize expected return. Soft Actor–Critic (SAC) augments the objective with an entropy bonus that explicitly encourages exploration and robustness while remaining off-policy and sample efficient (Haarnoja et al., 2018). We first introduce a minimal implementation of SAC for discrete actions, then present full SAC with additional techniques for continuous actions.

3.4.8.1 SAC for Discrete Actions

Entropy of A Probability Distribution. Given a probability distribution P supported on the set $\mathcal{X}$, the entropy of the distribution is defined as

$$\mathcal{H}(P) = -\sum_{x\in\mathcal{X}} P(x)\log P(x) = -\mathbb{E}_{x\sim P}\log P(x). \tag{3.46}$$

Since $0 \le P(x) \le 1$ for any x, it is clear that $\mathcal{H}(P) \ge 0$ for any distribution P.

Suppose the set $\mathcal{X}$ has N elements $x_1, \dots, x_N$, and suppose $P(x_i) = p_i \ge 0, i = 1, \dots, N$. We claim that the distribution $P^\star$ that maximizes $\mathcal{H}(P)$ is such that $p_i^\star = \frac{1}{N}, i = 1, \dots, N$.

To show this, consider the function $\log t$ that is concave for $t > 0$. Using Jensen's inequality, we have that

$$\begin{aligned}\mathcal{H}(P) &= -\sum_x P(x)\log P(x) = \sum_x P(x)\log\frac{1}{P(x)} \\ &\le \log\left(\sum_x P(x)\frac{1}{P(x)}\right) \\ &= \log N,\end{aligned} \tag{3.47}$$

with the equality holds if and only if $P(x_1) = P(x_2) = \cdots = P(x_N) = \frac{1}{N}$. Therefore, maximizing the entropy $\mathcal{H}(P)$ encourages the distribution P to have a density function that spreads out evenly over the set $\mathcal{X}$.

Maximum-Entropy Objective. SAC maximizes the soft objective

$$\begin{aligned}J(\pi) &= \mathbb{E}\left[\sum_t \gamma^t\Big(R(s_t,a_t) \;+\; \alpha\,\mathcal{H}(\pi(\cdot\mid s_t))\Big)\right], \\ \mathcal{H}(\pi(\cdot\mid s)) &= -\mathbb{E}_{a\sim\pi}[\log\pi(a\mid s)],\end{aligned} \tag{3.48}$$

where the entropy function $\mathcal{H}(\cdot)$ encourages the policy to explore, and the temperature $\alpha > 0$ balances reward maximization against exploration.

Given a trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, \dots)$, define the soft return:

$$g_t = \sum_{t=0}\gamma^t\left(R(s_t,a_t) - \alpha\log\pi(a_t\mid s_t)\right).$$

This leads to the "soft" state value and soft action value associated with π:

$$\begin{aligned}V^\pi(s) &= \mathbb{E}_{a\sim\pi}\left[Q^\pi(s,a) \;-\; \alpha\log\pi(a\mid s)\right], \\ Q^\pi(s,a) &= R(s,a) + \gamma\,\mathbb{E}_{s'}\left[V^\pi(s')\right].\end{aligned} \tag{3.49}$$

Combining the two equations above, we obtain a soft Bellman Consistency equation on the Q value:

$$Q^\pi(s,a) = R(s,a) + \gamma\mathbb{E}_{s'}\left[\mathbb{E}_{a'\sim\pi}\left[Q^\pi(s',a') - \alpha\log\pi(a'\mid s')\right]\right]. \tag{3.50}$$

Critic Update. For a replay sample (s, a, r, s'), assuming discrete actions, we can compute the target Q value following the soft Bellman Consistency equation (3.50)

$$y \;=\; r + \gamma \sum_{a'} \pi_\theta(a' \mid s') \left(Q_{\bar{\psi}}(s', a') - \alpha \log \pi_\theta(a' \mid s') \right) \tag{3.51}$$

where $Q_{\bar{\psi}}$ is the target Q network inspired by DQN to mitigate the deadly triad. The critic loss is therefore

$$\mathcal{L}_Q(\psi) \;=\; \mathbb{E}_{(s,a)\sim\mathcal{D}}\big[\big(Q_\psi(s,a) - y\big)^2\big], \tag{3.52}$$

where the expectation is taken over a minibatch drawn from the replay buffer.

Actor Update. Given the learned critic Q_ψ and replay state distribution $s \sim \mathcal{D}$, the SAC policy improvement step chooses π_θ to **minimize**, for each state, the soft advantage–regularized objective

$$J_\pi(\theta) \;=\; \mathbb{E}_{s\sim\mathcal{D}} \left[\sum_a \pi_\theta(a \mid s) \left(\alpha \, \log \pi_\theta(a \mid s) - Q_\psi(s,a) \right) \right]. \tag{3.53}$$

For discrete actions, the expectation over a is a finite sum—no action sampling is required.

Differentiating (3.53) yields the policy gradient

$$\nabla_\theta J_\pi(\theta) = \mathbb{E}_{s\sim\mathcal{D}} \left[\sum_a \nabla_\theta \pi_\theta(a \mid s) \, \Big(\alpha \left[1 + \log \pi_\theta(a \mid s) \right] - Q_\psi(s,a) \Big) \right]. \tag{3.54}$$

The following pseudocode implements a basic SAC algorithm with discrete actions.

Soft Actor–Critic (Discrete Actions, Single Q + Single Target)

Inputs: replay buffer $\mathcal{D}$; policy $\pi_\theta(a \mid s)$ over K actions; single critic $Q_\psi(s, \cdot)$ (returns a K-vector); target critic parameters $\bar{\psi}$; discount γ; temperature α (learned or fixed); Polyak $\tau \in (0, 1]$; batch size B; stepsizes $\alpha_\theta, \alpha_\psi$.

Initialize: $\bar{\psi} \leftarrow \psi$.

For iterations $k = 0, 1, 2, \ldots$:

1. **Interaction.**
   Observe s_t. Sample $a_t \sim \pi_\theta(\cdot \mid s_t)$; step env to get $(s_t, a_t, r_t, s_{t+1}, \text{done}_t)$; push to $\mathcal{D}$.

2. **Sample minibatch.**
   Draw B transitions $\{(s, a, r, s', d)\}$ from $\mathcal{D}$.

3. **Target computation (single target network).**

- Compute $\pi_\theta(\cdot \mid s')$ and $\log \pi_\theta(\cdot \mid s')$.
- Evaluate target critic $Q_{\bar{\psi}}(s', \cdot)$.
- Soft value target:

$$V_{\text{tgt}}(s') = \Big\langle \pi_\theta(\cdot \mid s'),\ Q_{\bar{\psi}}(s', \cdot) - \alpha \log \pi_\theta(\cdot \mid s') \Big\rangle.$$

- **Bellman target:**

$$y \leftarrow r + \gamma(1-d)\, V_{\text{tgt}}(s').$$

(Matches (3.51) *with one target network.)*

4. **Critic update.** Minimize the squared error (cf. (3.52)):

$$\psi \leftarrow \psi - \alpha_\psi \nabla_\psi \frac{1}{B} \sum\nolimits_{(s,a,r,s',d)} \big(Q_\psi(s,a) - y\big)^2.$$

5. **Actor update.** Minimize (cf. (3.53)):

$$\theta \leftarrow \theta - \alpha_\theta \nabla_\theta \frac{1}{B} \sum_s \sum_a \pi_\theta(a \mid s) \Big(\alpha \log \pi_\theta(a \mid s) - Q_\psi(s,a)\Big).$$

6. **Target critic (Polyak).**

$$\bar{\psi} \ \leftarrow \ \tau\, \psi \ + \ (1-\tau)\, \bar{\psi}.$$

The next example applies the SAC algorithm above to the cart-pole problem.

Example 3.7 (SAC for Cart-pole Balancing)**.** We use a fixed temperature $\alpha = 0.2$.

Fig. 3.17 shows the learning curve of SAC.

Fig. 3.18 shows a sample rollout of the learned policy.

You can find the code here. Play with the temperature parameter.

3.4.8.2 SAC for Continuous Actions

In continuous action spaces we cannot sum over actions. SAC therefore:

1) samples actions from the current policy using a *reparameterization* trick (low-variance gradients), and

2) computes the soft Bellman target with those sampled actions and a *twin-target minimum* to reduce overestimation.

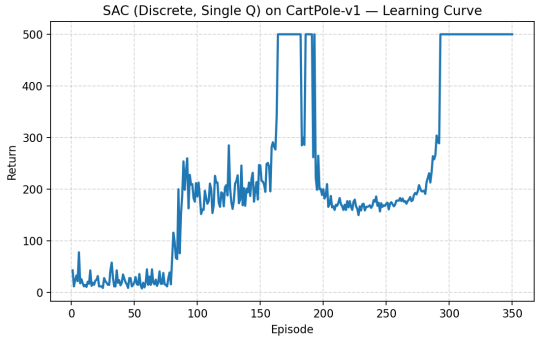

Figure 3.17: Learning curve (Soft Actor–Critic).

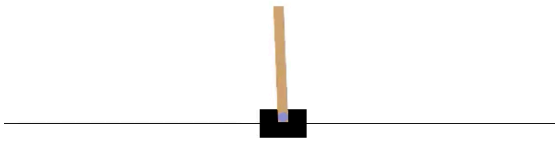

Figure 3.18: Policy rollout (Soft Actor–Critic).

Reparameterization (pathwise) Gradient. Let the stochastic policy be a Gaussian in unconstrained space, squashed by `tanh` to the action bounds:

$$u \;=\; \mu_\theta(s) \;+\; \sigma_\theta(s) \odot \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, I), \qquad a \;=\; \tanh(u) \cdot a_{\text{scale}} + a_{\text{bias}},$$

where $\sigma_\theta(s)$ outputs per-dimension standard deviation. This gives a differentiable map $a = f_\theta(s, \varepsilon)$. Expectations over $a \sim \pi_\theta(\cdot \mid s)$ are then written as expectations over ε, so gradients can flow through f_θ (the *pathwise derivative*). The correct log-density under the squashed policy uses change-of-variables:

$$\log \pi_\theta(a \mid s) \;=\; \log \mathcal{N}(u; \mu_\theta(s), \sigma_\theta^2(s)) \;-\; \sum_i \log(1 - \tanh^2(u_i)) \;+\; \text{constant}.$$

The intuition here is that the `tanh` function is a nonlinear transformation that distorts the original Gaussian distribution. This "tanh correction" is crucial for stable training.

Critic Update. Maintain two critics Q_{ψ_1}, Q_{ψ_2} and their target copies $Q_{\bar{\psi}_1}, Q_{\bar{\psi}_2}$. For a replay minibatch (s, a, r, s', d), form the target by drawing a next action from the current policy:

$$a' \sim \pi_\theta(\cdot \mid s'), \qquad y \;=\; r \;+\; \gamma(1-d)\Big(\min_{j=1,2} Q_{\bar{\psi}_j}(s', a') \;-\; \alpha \log \pi_\theta(a' \mid s')\Big). \tag{3.55}$$

Each critic minimizes the squared error to y (with stop-grad on y):

$$\mathcal{L}_Q(\psi_j) \;=\; \mathbb{E}\big[(Q_{\psi_j}(s, a) - y)^2\big], \quad j = 1, 2,$$

where the expectation is taken over the distribution in the replay buffer.

Actor Update. The actor minimizes the soft objective under the replay state distribution:

$$J_\pi(\theta) \;=\; \mathbb{E}_{s \sim \mathcal{D},\, \varepsilon}\left[\alpha \log \pi_\theta(f_\theta(s, \varepsilon) \mid s) - \min_{j=1,2} Q_{\psi_j}(s, f_\theta(s, \varepsilon))\right]. \tag{3.56}$$

By reparameterization, the gradient flows through both the explicit $\log \pi_\theta$ term and the path $a = f_\theta(s, \varepsilon)$. Particularly, denote $Q_\psi(\cdot, \cdot) = \min_{j=1,2} Q_{\psi_j}(\cdot, \cdot)$, we have that

$$\nabla_\theta J_\pi(\theta) = \mathbb{E}_{s,\varepsilon}\left[\alpha \nabla_\theta \log \pi_\theta(a \mid s) + \big(\alpha \nabla_a \log \pi_\theta(a \mid s) - \nabla_a Q_\psi(s, a)\big) \nabla_\theta f_\theta(s, \varepsilon)\right]_{a = f_\theta(s,\varepsilon)}.$$

In code, you typically just write the loss

$$\mathbb{E}_{s,\varepsilon}\left[\alpha \log \pi_\theta(a \mid s) - Q_\psi(s, a)\right], \quad a = f_\theta(s, \varepsilon),$$

and autodiff will automatically compute the correct gradient.

Tuning α (Temperature). α trades off reward pursuit vs. policy entropy. A fixed α is problem-dependent. SAC treats α as a dual variable to enforce a target entropy $\bar{\mathcal{H}}$ (often $-\dim(\mathcal{A})$):

$$J(\alpha) \;=\; \mathbb{E}_{s \sim \mathcal{D},\, a \sim \pi_\theta}\Big[-\alpha\big(\log \pi_\theta(a \mid s) + \bar{\mathcal{H}}\big)\Big], \quad \log \alpha \leftarrow \log \alpha - \alpha_\alpha \nabla_{\log \alpha} J(\alpha).$$

This adapts exploration automatically across tasks and training phases.

Soft Actor–Critic (Continuous Actions, Twin Critics + Twin Targets)

Inputs: replay buffer $\mathcal{D}$; policy $\pi_\theta(a \mid s)$ reparameterized by $a = f_\theta(s, \varepsilon)$ with tanh-squashed Gaussian; twin critics Q_{ψ_1}, Q_{ψ_2}; twin target critics with params $\bar{\psi}_1, \bar{\psi}_2$; discount γ; temperature α (learned or fixed); Polyak $\tau \in (0, 1]$; batch size B; stepsizes $\alpha_\theta, \alpha_\psi, \alpha_\alpha$.

Initialize: $\bar{\psi}_j \leftarrow \psi_j$ for $j \in \{1, 2\}$.

For iterations $k = 0, 1, 2, ...$:

1. **Interaction.**
   Observe s_t. Sample $\varepsilon_t \sim \mathcal{N}(0, I)$, set $a_t = f_\theta(s_t, \varepsilon_t)$; step env to get $(s_t, a_t, r_t, s_{t+1}, \text{done}_t)$; push to $\mathcal{D}$.
2. **Sample minibatch.**
   Draw B transitions $\{(s, a, r, s', d)\}$ from $\mathcal{D}$.
3. **Target computation (twin targets, reparameterized next action).**
   - Sample $\varepsilon' \sim \mathcal{N}(0, I)$, set $a' = f_\theta(s', \varepsilon')$.
   - Compute $\log \pi_\theta(a' \mid s')$ with **tanh correction**.
   - Evaluate target critics $Q_{\bar{\psi}_1}(s', a')$, $Q_{\bar{\psi}_2}(s', a')$; let $Q_{\min}(s', a') = \min\{Q_{\bar{\psi}_1}, Q_{\bar{\psi}_2}\}$.
   - **Bellman target:**

     $$y \leftarrow r + \gamma(1 - d)\left(Q_{\min}(s', a') - \alpha \log \pi_\theta(a' \mid s')\right).$$

     (Stop gradient through y.)
4. **Critic updates (both heads).**

   $$\psi_j \leftarrow \psi_j - \alpha_\psi \nabla_{\psi_j} \frac{1}{B} \sum (Q_{\psi_j}(s, a) - y)^2, \quad j = 1, 2.$$

5. **Actor update (reparameterized).**
   - For each s in the batch, sample ε, set $a = f_\theta(s, \varepsilon)$.
   - **Actor objective:**

     $$J_\pi(\theta) = \frac{1}{B} \sum_s \Big(\alpha \log \pi_\theta(a \mid s) - \min_j Q_{\psi_j}(s, a)\Big).$$

   - Update:

     $$\theta \leftarrow \theta - \alpha_\theta \nabla_\theta J_\pi(\theta).$$

6. **Temperature (optional).**
   With target entropy $\bar{\mathcal{H}}$ and parameter $\log \alpha$:

   $$J(\alpha) = \frac{1}{B} \sum_s \left[-\alpha(\log \pi_\theta(a \mid s) + \bar{\mathcal{H}})\right], \quad \log \alpha \leftarrow \log \alpha - \alpha_\alpha \nabla_{\log \alpha} J(\alpha), \quad \alpha \leftarrow e^{\log \alpha}.$$

7. **Target critics (Polyak).** For $j = 1, 2$:

$$\bar{\psi}_j \leftarrow \tau \psi_j + (1 - \tau) \bar{\psi}_j.$$

The next example applies SAC to Inverted Pendulum.

Example 3.8 (SAC for Inverted Pendulum)**.** Fig. 3.19 plots the learning curve.

Fig. 3.20 visualizes two sample rollouts of the policy.

Code can be found here.

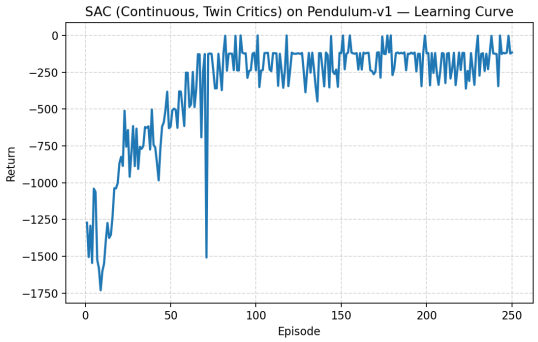

Figure 3.19: Learning curve (Soft Actor–Critic).

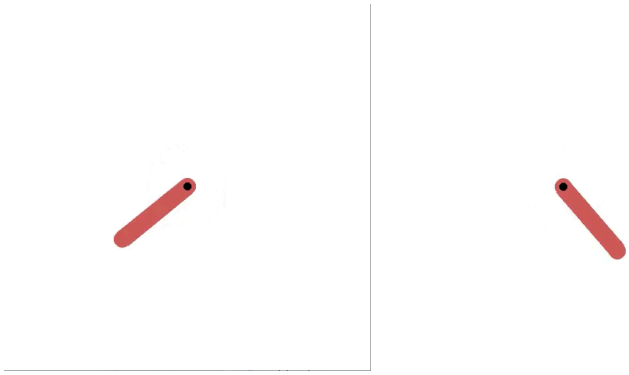

Figure 3.20: Policy rollout (Soft Actor–Critic).

3.4.9 Deterministic Policy Gradient

In continuous-control tasks, sampling or integrating over actions inside policy gradients is costly and noisy. The Deterministic Policy Gradient (DPG)

framework (Silver et al., 2014) replaces the stochastic policy $\pi_\theta(a \mid s)$ with a *deterministic* actor

$$a = \mu_\theta(s) \in \mathbb{R}^m.$$

Its state–action value and discounted state visitation measure are

$$Q^{\mu_\theta}(s,a) \;=\; \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t,a_t) \,\middle|\, s_0 = s,\; a_0 = a,\; a_{t>0} = \mu_\theta(s_t)\right],$$

$$\rho^{\pi}(s) \;=\; \sum_{t=0}^{\infty} \gamma^t \Pr(s_t = s \mid a_t \sim \pi(\cdot \mid s_t)), \qquad \rho^{\mu_\theta} \equiv \rho^{\pi=\mu_\theta}.$$

We consider two objectives:

- **On-policy objective**:

$$J(\theta) \;=\; \mathbb{E}_{s\sim\rho^{\mu_\theta}}\big[\, Q^{\mu_\theta}(s,\mu_\theta(s))\,\big]. \tag{3.57}$$

- **Off-policy surrogate** (with behavior policy β):

$$J_\beta(\theta) \;=\; \mathbb{E}_{s\sim\rho^{\beta}}\big[\, Q^{\mu_\theta}(s,\mu_\theta(s))\,\big]. \tag{3.58}$$

The on-policy objective (3.57) is the usual RL objective in policy gradient methods, as $Q^{\mu_\theta}(s,\mu_\theta(s)) = V^{\mu_\theta}(s)$ by definition.

A key result in Deterministic Policy Gradient is that under mild conditions, optimizating the surrogate off-policy objective (3.58) is the same as optimizating the original on-policy objective.

To see this, assume

1. R and $P(\cdot \mid s,a)$ (the transition dynamics) are bounded/measurable; Q^{μ_θ} exists and is continuously differentiable in a;
2. $\mu_\theta(s)$ is continuously differentiable in θ;
3. Interchange of integration and differentiation is valid (e.g., dominated convergence).

Then, the on-policy and off-policy deterministic policy gradients (DPG) are:

- **On-policy DPG.**

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{s\sim\rho^{\mu_\theta}}\Big[\nabla_\theta \mu_\theta(s)\, \nabla_a Q^{\mu_\theta}(s,a)\big|_{a=\mu_\theta(s)}\Big]. \tag{3.59}$$

- **Off-policy DPG.** For any behavior policy β with visitation ρ^β,

$$\nabla_\theta J_\beta(\theta) \;=\; \mathbb{E}_{s\sim\rho^{\beta}}\Big[\nabla_\theta \mu_\theta(s)\, \nabla_a Q^{\mu_\theta}(s,a)\big|_{a=\mu_\theta(s)}\Big]. \tag{3.60}$$

In particular, the off-policy DPG (3.60) can be estimated from replay sampled under β without action-importance ratios; only the state weighting changes.

The following result states that the on-policy and off-policy objectives share the same stationary points.

Theorem 3.9 (Common First-Order Optima)**.** *Let*

$$g(s;\theta) \;:=\; \nabla_\theta \mu_\theta(s)\, \nabla_a Q^{\mu_\theta}(s,a)\big|_{a=\mu_\theta(s)} \in \mathbb{R}^d, \tag{3.61}$$

where d is the dimension of θ.

Suppose ρ^β has coverage of the on-policy support, i.e.,

$$\mathit{supp}(\rho^{\mu_\theta}) \;\subseteq\; \mathit{supp}(\rho^\beta), \quad \textit{and} \quad \rho^\beta(s) > 0 \ \textit{a.e. on } \mathit{supp}(\rho^{\mu_\theta}).$$

If $g(s;\theta^\star) = 0$ for $\rho^{\mu_{\theta^\star}}$-almost every s (in particular, if $\mu_{\theta^\star}$ is greedy w.r.t. $Q^{\mu_{\theta^\star}}$, so $\nabla_a Q^{\mu_{\theta^\star}}(s,a)|_{a=\mu_{\theta^\star}(s)} = 0$ for all s), then

$$\nabla_\theta J(\theta^\star) = 0 \quad \textit{and} \quad \nabla_\theta J_\beta(\theta^\star) = 0.$$

Thus any deterministic policy satisfying the first-order optimality condition (greedy w.r.t. its own Q) is a stationary point of both J *and J_β, regardless of the (covered) state weighting.*

If additionally $\mathit{supp}(\rho^{\mu_\theta}) = \mathit{supp}(\rho^\beta)$ and both are strictly positive on that support, then

$$\nabla_\theta J(\theta) = 0 \iff \nabla_\theta J_\beta(\theta) = 0.$$

Remarks.

- The off-policy objective J_β changes only the **weights** over states; the **per-state improvement direction** $g(s;\theta)$ is identical. With sufficient coverage, ascent on J_β improves J and shares its stationary points.
- In practice, DDPG uses exploration noise to expand support of ρ^β and target networks to stabilize Q^{μ_θ}, making the off-policy gradient estimate reliable.

From DPG to DDPG (Deep DPG). DDPG (Lillicrap et al., 2015) implements DPG with deep networks + standard stabilizers:

- **Replay buffer** $\mathcal{D}$ for off-policy sample efficiency.
- **Target networks** $\mu_{\bar\theta}, Q_{\bar\psi}$ with Polyak averaging to stabilize TD targets.
- **Exploration noise** added to the deterministic action: $a_t = \mu_\theta(s_t) + \varepsilon_t$ (original paper used Ornstein–Uhlenbeck noise; Gaussian works well too).

High-Level Algorithm (DDPG).

1. **Interact off-policy.** Act with exploration: $a_t = \mu_\theta(s_t) + \varepsilon_t$. Store $(s_t, a_t, r_t, s_{t+1}, \text{done}_t)$ in $\mathcal{D}$.

2. **Critic TD(0).** For a minibatch from $\mathcal{D}$,

$$y = r + \gamma(1 - \text{done})\, Q_{\bar{\psi}}(s',\, \mu_{\bar{\theta}}(s')), \qquad \min_{\psi} \frac{1}{B} \sum (Q_\psi(s,a) - y)^2.$$

3. **Actor DPG step.**

$$\max_{\theta} \frac{1}{B} \sum Q_\psi(s,\, \mu_\theta(s)) \quad \Longleftrightarrow \quad \nabla_\theta J \approx \frac{1}{B} \sum \nabla_\theta \mu_\theta(s)\, \nabla_a Q_\psi(s,a)\big|_{a=\mu_\theta(s)}.$$

4. **Targets Polyak update.**

$$\bar{\theta} \leftarrow \tau\, \theta + (1-\tau)\bar{\theta}, \quad \bar{\psi} \leftarrow \tau\, \psi + (1-\tau)\bar{\psi}.$$

Remarks.

- No entropy bonus or log-probabilities (in contrast to SAC). Exploration comes from additive noise.
- Overestimation and sensitivity to hyperparameters can appear; target networks, small actor steps, and proper normalization help.

The following pseudocode implements DDPG.

Deep Deterministic Policy Gradient (DDPG)

Inputs: replay buffer $\mathcal{D}$; deterministic actor $\mu_\theta(s)$; critic $Q_\psi(s,a)$; target networks $\mu_{\bar{\theta}}, Q_{\bar{\psi}}$; discount $\gamma \in [0,1)$; Polyak $\tau \in (0,1]$; batch size B; stepsizes $\alpha_\theta, \alpha_\psi$; exploration noise process $\varepsilon_t \sim \mathcal{N}(0, \sigma^2 I)$ (or Ornstein–Uhlenbeck).

Initialize: $\bar{\theta} \leftarrow \theta,\ \bar{\psi} \leftarrow \psi$. Fill $\mathcal{D}$ with a short random warm-up.

For iterations $k = 0, 1, 2, \ldots$:

1. **Interaction (off-policy).**
   Observe s_t. Compute action with noise

$$a_t \leftarrow \text{clip}(\mu_\theta(s_t) + \varepsilon_t,\ a_{\min}, a_{\max}).$$

   Step env to get $(s_t, a_t, r_t, s_{t+1}, \text{done}_t)$. Push into $\mathcal{D}$.

2. **Sample minibatch.**
   Draw B transitions $\{(s, a, r, s', d)\}$ from $\mathcal{D}$.

3. **Critic target.**

$$a' \leftarrow \mu_{\bar{\theta}}(s'), \qquad y \leftarrow r + \gamma(1-d)\, Q_{\bar{\psi}}(s', a').$$

(Stop gradient through y.)

4. **Critic update.**

$$\psi \leftarrow \psi - \alpha_\psi \nabla_\psi \frac{1}{B} \sum (Q_\psi(s,a) - y)^2.$$

5. **Actor update (DPG).**

$$\theta \leftarrow \theta + \alpha_\theta \frac{1}{B} \sum \Big[\nabla_\theta \mu_\theta(s)\ \nabla_a Q_\psi(s,a)\big|_{a=\mu_\theta(s)} \Big].$$

(Equivalently, ascend $\frac{1}{B}\sum Q_\psi(s, \mu_\theta(s))$ by backprop.)

6. **Target networks (Polyak).**

$$\bar{\theta} \leftarrow \tau\,\theta + (1-\tau)\,\bar{\theta}, \qquad \bar{\psi} \leftarrow \tau\,\psi + (1-\tau)\,\bar{\psi}.$$

The next example applies DDPG to Inverted Pendulum.

Example 3.9 (DDPG for Inverted Pendulum)**.** Fig. 3.21 plots the learning curve of DDPG.

Fig. 3.22 visualizes sample rollouts of the learned policy.

Code can be found here.

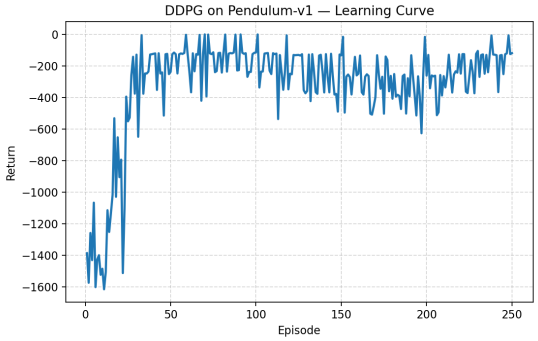

Figure 3.21: Learning curve (DDPG).

Figure 3.22: Policy rollout (DDPG).

3.5 Model-based Policy Optimization

Model-based policy optimization (MBPO) (Janner et al., 2019) sits between pure model-free methods (high variance, data hungry) and "plan-only" model-based control (sensitive to model bias, to be introduced in Chapter 4). The key idea is to learn a dynamics model and then use only *short* rollouts from that model to create extra training data for a strong off-policy learner (usually SAC).

Consider an MDP with unknown dynamics $s_{t+1} \sim P(\cdot \mid s_t, a_t)$. MBPO learns an ensemble $\{\hat{f}_{\psi_k}\}_{k=1}^{K}$ that predicts the next state (often the *delta-state* Δs). Rather than planning far ahead inside the learned model, MBPO:

1. Collects real transitions $\mathcal{D}_{\text{env}}$ by interacting with the environment.
2. Fits the dynamics ensemble on $\mathcal{D}_{\text{env}}$.
3. Periodically generates short rollouts (e.g., horizon $H = 1 \dots 5$) starting from *real* states by simulating with a *random* member of the ensemble, producing model transitions $\mathcal{D}_{\text{model}}$.
4. Trains an off-policy actor-critic (e.g., SAC) on a *mixture* of $\mathcal{D}_{\text{env}}$ and $\mathcal{D}_{\text{model}}$, typically with a high fraction of model data but *short* H to limit bias.

This yields the sample-efficiency benefits of model-based learning while maintaining the robustness of model-free policy optimization.

The following pseudocode implements MBPO with SAC as the off-policy learner.

Model-based Policy Optimization (SAC as Off-Policy Learner)

Inputs: environment $\mathcal{E}$, policy $\pi_\theta(a \mid s)$, twin critics Q_{ϕ_1}, Q_{ϕ_2} with targets, temperature α (auto-tuned), dynamics ensemble $\{\hat{f}_{\psi_k}\}_{k=1}^{K}$, real buffer $\mathcal{D}_{\text{env}}$, model buffer $\mathcal{D}_{\text{model}}$. Rollout horizon H, model ratio $p_{\text{model}} \in [0, 1]$, update counts $G_{\text{dyn}}, G_{\text{rl}}$.

1. **Warm-up & data collection.**
   - Interact with $\mathcal{E}$ using π_θ (or random for a short warm-up).
   - Store (s, a, r, s') in $\mathcal{D}_{\text{env}}$.
2. **Fit dynamics.** For G_{dyn} steps:
   - Sample minibatch $B \subset \mathcal{D}_{\text{env}}$.
   - Update each ψ_k to predict $\Delta s = s' - s$ (and optionally r) by minimizing, e.g, mean squared error.
3. **Short model rollouts (data generation).**
   - Sample seed states S_0 from recent $\mathcal{D}_{\text{env}}$.
   - For each $s \in S_0$
     - for $h = 1 \dots H$:
       * Sample $a \sim \pi_\theta(\cdot \mid s)$.
       * Pick random ensemble member k; predict $\hat{\Delta} s \leftarrow \hat{f}_{\psi_k}(s, a)$; set $\hat{s}' = s + \hat{\Delta} s$.
       * Compute r via a learned reward model or a known formula (when available).
       * Push $(s, a, r, \hat{s}', \texttt{done} = 0)$ into $\mathcal{D}_{\text{model}}$.
       * Set $s \leftarrow \hat{s}'$; break if time-limit reached.
4. **Off-policy RL updates (SAC).** For G_{rl} steps:
   - Form a minibatch by drawing a fraction p_{model} from $\mathcal{D}_{\text{model}}$ and $1 - p_{\text{model}}$ from $\mathcal{D}_{\text{env}}$.
   - **Critic targets:**

     $$y = r + \gamma(1 - d)\big[\min_j Q_{\phi_j^-}(s', a') - \alpha \log \pi_\theta(a' \mid s')\big],$$

     where $a' \sim \pi_\theta(\cdot \mid s')$.
   - **Critic update:** regress Q_{ϕ_j} to y (both heads).
   - **Actor update:** minimize $J_\pi = \mathbb{E}_s[\alpha \log \pi_\theta(a \mid s) - \min_j Q_{\phi_j}(s, a)]$, with $a \sim \pi_\theta$.
   - **Temperature update (optional):** adjust α towards target entropy.
   - Soft-update target critics.
5. **Repeat** steps 1–4 until convergence or iteration limits.

Notes.

- Ensembles capture epistemic uncertainty; random-member rollouts implicitly regularize toward pessimism.
- Keeping H *short* (e.g., 1–5) is crucial to prevent model error explosion.

- Use recent real states as rollout seeds to stay on-distribution.

The next example applies MBPO to Inverted Pendulum.

Example 3.10 (MBPO for Inverted Pendulum)**.** Fig. 3.23 plots the learning curve.

Fig. 3.24 visualizes sample rollouts of the policy.

Code can be found here.

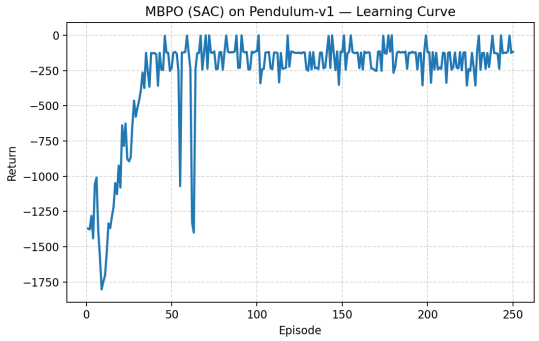

Figure 3.23: Learning curve (MBPO).

Figure 3.24: Policy rollout (MBPO).

Chapter 4

Model-based Planning and Optimization

In Chapter 1, we studied tabular MDPs with known dynamics where both the state and action spaces are finite, and we saw how dynamic-programming methods—Policy Iteration (PI) and Value Iteration (VI)—recover optimal policies.

Chapters 2 and 3 generalized these ideas to unknown dynamics and continuous spaces by introducing function approximation for value functions and policies. Those methods are *model-free*: they assume no access to the transition model and rely solely on data collected from interaction.

This chapter turns to the complementary regime: **known dynamics** with **continuous state and action spaces**. Our goal is to develop model-based planning and optimization methods that exploit the known dynamics to compute high-quality decisions efficiently.

We proceed in three steps:

1. **Linear-quadratic systems.** For linear dynamics and quadratic stage/terminal rewards (or costs), dynamic programming yields a linear optimal policy that can be computed efficiently via Riccati recursions. This setting serves as a tractable, illuminating baseline and a recurring building block in more general algorithms.
2. **Trajectory optimization (TO) for nonlinear systems.** When dynamics are nonlinear, we focus on planning an optimal state–action trajectory from a given initial condition, maximizing the cumulative reward (or minimizing cumulative cost) subject to dynamics and constraints. Unlike RL, which seeks an optimal *feedback policy* valid for all states, TO computes an *open-loop plan* (often with a time-varying local feedback around

a nominal trajectory). Although less ambitious, TO naturally accommodates state/control constraints—common in motion planning under safety, actuation, and environmental limits—and is widely used in robotics and control.

3. **Model predictive control (MPC).** MPC converts open-loop plans into a feedback controller by repeatedly solving a short-horizon TO problem at each time step, applying only the first action, and receding the horizon. This receding-horizon strategy brings robustness to disturbances and model mismatch while retaining the constraint-handling benefits of TO.

We adopt the standard discrete-time dynamical system notation

$$x_{t+1} = f_t(x_t, u_t, w_t), \tag{4.1}$$

where $x_t \in \mathbb{R}^n$ is the state, $u_t \in \mathbb{R}^m$ is the control/action, $w_t \in \mathbb{R}^d$ is a (possibly stochastic) disturbance, and f_t is a known transition function, potentially nonlinear or nonsmooth. The goal is to find a policy that maximizes a sum of stage rewards $r(x_t, u_t)$ and optional terminal reward $r_T(x_T)$. We will often use the cost-minimization form $c = -r$. State and action constraints are written as

$$x_t \in \mathcal{X}, \qquad u_t \in \mathcal{U}.$$

4.1 Linear Quadratic Regulator

In this section, we focus on the case when f_t is a linear function, and the rewards/costs are quadratic in x and u. This family of problems is known as linear quadratic regulator (LQR).

4.1.1 Finite-Horizon LQR

Consider a linear discrete-time dynamical system

$$x_{k+1} = A_k x_k + B_k u_k + w_k, \quad k = 0, 1, \dots, N-1, \tag{4.2}$$

where $x_k \in \mathbb{R}^n$ the state, $u_k \in \mathbb{R}^m$ the control, $w_k \in \mathbb{R}^n$ the independent, zero-mean disturbance with given probability distribution that does not depend on x_k, u_k, and $A_k \in \mathbb{R}^{n\times n}, B_k \in \mathbb{R}^{n\times m}$ are known matrices determining the transition dynamics.

We want to solve the following optimal control problem

$$\min_{\mu_0,\dots,\mu_{N-1}} \mathbb{E}\left\{ x_N^\top Q_N x_N + \sum_{k=0}^{N-1} \left(x_k^\top Q_k x_k + u_k^\top R_k u_k\right) \right\}, \tag{4.3}$$

where $\mu_0, \dots, \mu_{N-1}$ are feedback policies/controllers that map states to actions and the expectation is taken over the randomness in $w_0, \dots, w_{N-1}$. In (4.3), $\{Q_k\}_{k=0}^{N}$ are positive semidefinite matrices, and $\{R_k\}_{k=0}^{N-1}$ are positive definite matrices. The formulation (4.3) is known as the linear quadratic regulator (LQR) problem because the dynamics is linear, the cost is quadratic, and the formulation can be considered to "regulate" the system around the origin $x = 0$.

The Bellman Optimality condition introduced in Theorem 1.1 still holds for continuous state and action spaces. Therefore, we will try to follow the dynamic programming (DP) algorithm in Section 1.1.4 to solve for the optimal policy.

The DP algorithm computes the optimal cost-to-go backwards in time. The terminal cost is

$$J_N(x_N) = x_N^\top Q_N x_N$$

by definition.

The optimal cost-to-go at time $N-1$ is equal to

$$\begin{aligned} J_{N-1}(x_{N-1}) = \min_{u_{N-1}} \mathbb{E}_{w_{N-1}} \{ & x_{N-1}^\top Q_{N-1} x_{N-1} + u_{N-1}^\top R_{N-1} u_{N-1} + \\ & \| \underbrace{A_{N-1} x_{N-1} + B_{N-1} u_{N-1} + w_{N-1}}_{x_N} \|_{Q_N}^2 \} \end{aligned} \tag{4.4}$$

where $\|v\|_Q^2 = v^\top Q v$ for $Q \succeq 0$. Now observe that the objective in (4.4) is

$$\begin{aligned} & x_{N-1}^\top Q_{N-1} x_{N-1} + u_{N-1}^\top R_{N-1} u_{N-1} + \|A_{N-1} x_{N-1} + B_{N-1} u_{N-1}\|_{Q_N}^2 + \\ & \mathbb{E}_{w_{N-1}} \left[2(A_{N-1} x_{N-1} + B_{N-1} u_{N-1})^\top Q_{N-1} w_{N-1} \right] + \\ & \mathbb{E}_{w_{N-1}} \left[w_{N-1}^\top Q_N w_{N-1} \right] \end{aligned} \tag{4.5}$$

where the second line is zero due to $\mathbb{E}[w_{N-1}] = 0$ and the third line is a constant with respect to u_{N-1}. Consequently, the optimal control $u_{N-1}^\star$ can be computed by setting the derivative of the objective with respect to u_{N-1} equal to zero

$$u_{N-1}^\star = -\left[\left(R_{N-1} + B_{N-1}^\top Q_N B_{N-1} \right)^{-1} B_{N-1}^\top Q_N A_{N-1} \right] x_{N-1}. \tag{4.6}$$

Plugging the optimal controller $u_{N-1}^\star$ back to the objective of (4.4) leads to

$$J_{N-1}(x_{N-1}) = x_{N-1}^\top S_{N-1} x_{N-1} + \mathbb{E}\left[w_{N-1}^\top Q_N w_{N-1} \right], \tag{4.7}$$

with

$$S_{N-1} = Q_{N-1} + A_{N-1}^\top \left[Q_N - Q_N B_{N-1} \left(R_{N-1} + B_{N-1}^\top Q_N B_{N-1} \right)^{-1} B_{N-1}^\top Q_N \right] A_{N-1}.$$

We note that S_{N-1} is positive semidefinite (this is an exercise for you to convince yourself).

Now we realize that something surprising and nice has happened.

1. The optimal controller $u_{N-1}^\star$ in (4.6) is a linear feedback policy of the state x_{N-1}, and

2. The optimal cost-to-go $J_{N-1}(x_{N-1})$ in (4.7) is quadratic in x_{N-1}, just the same as $J_N(x_N)$.

This implies that, if we continue to compute the optimal cost-to-go at time $N-2$, we will again compute a linear optimal controller and a quadratic optimal cost-to-go. This is the rare nice property for the LQR problem, that is,

> The (representation) complexity of the optimal controller and cost-to-go does not grow as we run the DP recursion backwards in time.

We summarize the solution for the LQR problem (4.3) as follows.

Proposition 4.1 (Solution of Discrete-Time Finite-Horizon LQR)**.** *The optimal controller for the LQR problem* (4.3) *is a linear state-feedback policy*

$$\mu_k^\star(x_k) = -K_k x_k, \quad k = 0, \dots, N-1. \tag{4.8}$$

The gain matrix K_k can be computed as

$$K_k = \left(R_k + B_k^\top S_{k+1} B_k\right)^{-1} B_k^\top S_{k+1} A_k,$$

where the matrix S_k satisfies the following backwards recursion

$$\begin{aligned} S_N &= Q_N \\ S_k &= Q_k + A_k^\top \left[S_{k+1} - S_{k+1} B_k \left(R_k + B_k^\top S_{k+1} B_k\right)^{-1} B_k^\top S_{k+1}\right] A_k, k = N-1, \dots, 0. \end{aligned} \tag{4.9}$$

The optimal cost-to-go is given by

$$J_0(x_0) = x_0^\top S_0 x_0 + \sum_{k=0}^{N-1} \mathbb{E}\left[w_k^\top S_{k+1} w_k\right].$$

The recursion (4.9) *is called the* discrete-time Riccati equation.

Proposition 4.1 states that, to evaluate the optimal policy (4.8), one can first run the backwards Riccati equation (4.9) to compute all the positive definite matrices S_k, and then compute the gain matrices K_k. For systems of reasonable dimensions, evalutating the matrix inversion in (4.9) should be fairly efficient.

4.1.2 Infinite-Horizon LQR

We now switch to the infinite-horizon LQR problem

$$\min_{\mu} \quad \sum_{k=0}^{\infty} (x_k^\top Q x_k + u_k^\top R u_k) \tag{4.10}$$

$$\text{subject to} \quad x_{k+1} = A x_k + B u_k, \quad k = 0, \dots, \infty, \tag{4.11}$$

where $Q \succeq 0$, $R \succ 0$, A, B are constant matrices, and we seek a stationary policy μ that maps states to actions. Note that here we remove the disturbance w_k because in general adding w_k will make the objective function unbounded. To handle w_k, we will have to either add a discount factor γ, or switch to an average cost objective function.

For infinite-horizon problems, the Bellman Optimality condition changes from a recursion to an equation. Specifically, according to Theorem 1.2 and equation (1.34), the optimal value function should satisfy the following Bellman optimality equation, restated for the case of cost minimization instead of reward maximization:

$$J^\star(x) = \min_u \left[c(x,u) + \sum_{x'} P(x' \mid x, u) J^\star(x') \right], \quad \forall x, \tag{4.12}$$

where $c(x,u)$ is the cost function.

Guess A Solution. Based on our derivation in the finite-horizon case, we might as well guess that the optimal value function is a quadratic function:

$$J(x) = x^\top S x, \quad \forall x,$$

for some positive definite matrix S. Then, our guessed solution must satisfy the Bellman optimality stated in (4.12):

$$x^\top S x = J(x) = \min_u \left\{ x^\top Q x + u^\top R u + \| \underbrace{Ax + Bu}_{x'} \|_S^2 \right\}. \tag{4.13}$$

The minimization over u in (4.13) can again be solved in closed-form by setting the gradient of the objective with respect to u to be zero

$$u^\star = - \underbrace{\left[(R + B^\top S B)^{-1} B^\top S A \right]}_{K} x. \tag{4.14}$$

Plugging the optimal $u^\star$ back into (4.13), we see that the matrix S has to satisfy the following equation

$$S = Q + A^\top \left[S - SB \left(R + B^\top S B \right)^{-1} B^\top S \right] A. \tag{4.15}$$

Equation (4.15) is known as the *discrete algebraic Riccati equation* (DARE).

So the question boils down to if the DARE has a solution S that is positive definite?

Proposition 4.2 (Solution of Discrete-Time Infinite-Horizon LQR)**.** *Consider a linear system*

$$x_{k+1} = Ax_k + Bu_k,$$

with (A, B) controllable (see Section 4.1.3). Let $Q \succeq 0$ in (4.10) be such that Q can be written as $Q = C^\top C$ with (A, C) observable.

Then the optimal controller for the infinite-horizon LQR problem (4.10) is a stationary linear policy

$$\mu^\star(x) = -Kx,$$

with

$$K = \left(R + B^\top S B\right)^{-1} B^\top S A.$$

The matrix S is the unique positive definite matrix that satisfies the discrete algebraic Riccati equation

$$S = Q + A^\top \left[S - SB\left(R + B^\top S B\right)^{-1} B^\top S\right] A. \tag{4.16}$$

Moreover, the closed-loop system

$$x_{k+1} = Ax_k + B(-Kx_k) = (A - BK)x_k$$

is stable, i.e., the eigenvalues of the matrix $A - BK$ are strictly within the unit circle (see Appendix B.1.2).

Remark. The assumptions of (A, B) being controllable and (A, C) being observable can be relaxted to (A, B) being stabilizable and (A, C) being detectable (for definitions of stabilizability and detectability, see Appendix B).

We have not discussed how to solve the algebraic Riccati equation (4.16). It is clear that (4.16) is not a linear system of equations in S. In fact, the numerical algorithms for solving the algebraic Riccati equation can be highly nontrivial, for example see (Arnold and Laub, 1984). Fortunately, such algorithms are often readily available, and as practitioners we do not need to worry about solving the algebraic Riccati equation by ourselves. For example, the Matlab `dlqr` and the Python `scipy.linalg.solve_discrete_are` function computes the K and S matrices from A, B, Q, R.

Let us now apply the infinite-horizon LQR solution to stabilizing a simple pendulum.

Example 4.1 (Pendulum Stabilization by LQR)**.** Consider the simple pendulum in Fig. 4.1 with dynamics

$$x = \begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix}, \quad \dot{x} = f(x, u) = \begin{bmatrix}\dot{\theta} \\ -\frac{1}{ml^2}(b\dot{\theta} + mgl\sin\theta) + \frac{1}{ml^2}u\end{bmatrix} \tag{4.17}$$

where m is the mass of the pendulum, l is the length of the pole, g is the gravitational constant, b is the damping ratio, and u is the torque applied to the pendulum.

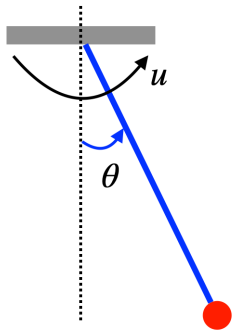

Figure 4.1: A Simple Pendulum.

We are interested in applying the LQR controller to balance the pendulum in the upright position $x_d = [\pi, 0]^\top$ with a zero velocity.

Let us first shift the dynamics so that "0" is the upright position. This can be done by defining a new variable $z = x - x_d = [\theta - \pi, \dot{\theta}]^\top$, which leads to

$$\dot{z} = \dot{x} = f(x,u) = f(z + x_d, u) = \begin{bmatrix} z_2 \\ \frac{1}{ml^2}\left(u - bz_2 + mgl\sin z_1\right) \end{bmatrix} = f'(z,u). \tag{4.18}$$

We then linearize the nonlinear dynamics $\dot{z} = f'(z,u)$ at the point $z^\star = 0, u^\star = 0$:

$$\dot{z} \approx f'(z^\star, u^\star) + \left(\frac{\partial f'}{\partial z}\right)_{z^\star,u^\star}(z - z^\star) + \left(\frac{\partial f'}{\partial u}\right)_{z^\star,u^\star}(u - u^\star) \tag{4.19}$$

$$= \begin{bmatrix} 0 & 1 \\ \frac{g}{l}\cos z_1 & -\frac{b}{ml^2} \end{bmatrix}_{z^\star,u^\star} z + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} u \tag{4.20}$$

$$= \underbrace{\begin{bmatrix} 0 & 1 \\ \frac{g}{l} & -\frac{b}{ml^2} \end{bmatrix}}_{A_c} z + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix}}_{B_c} u. \tag{4.21}$$

Finally, we convert the continuous-time dynamics to discrete time with a fixed discretization h

$$z_{k+1} = \dot{z}_k \cdot h + z_k = \underbrace{(h \cdot A_c + I)}_{A} z_k + \underbrace{(h \cdot B_c)}_{B} u_k.$$

We are now ready to implement the LQR controller. In the formulation (4.10), we choose $Q = I$, $R = I$, and compute the gain matrix K by solving the DARE.

Fig. 4.2 shows the simulation result for $m = 1, l = 1, b = 0.1$, $g = 9.8$, and $h = 0.01$, with an initial condition $z^0 = [0.1, 0.1]^\top$. We can see that the LQR controller successfully stabilizes the pendulum at $z^\star$, the upright position.

You can play with the Python code here.

Alternatively, the Matlab code can be found here.

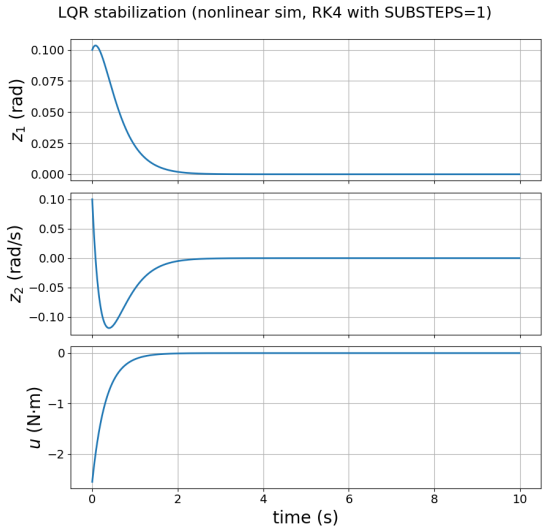

Figure 4.2: LQR stabilization of a simple pendulum.

4.1.3 Linear System Basics

Consider the discrete-time linear time-invariant (LTI) system

$$x_{k+1} = Ax_k + Bu_k, \qquad y_k = Cx_k + Du_k,$$

with $x_k \in \mathbb{R}^n$, $u_k \in \mathbb{R}^m$, $y_k \in \mathbb{R}^p$.

We provide a very brief review of linear system theory to understand Proposition 4.2. More details can be found in Appendix B.

Stability. The autonomous system $x_{k+1} = Ax_k$ is (asymptotically) stable if for every x_0 we have $x_k \to 0$ as $k \to \infty$.

Equivalent characterizations.

- A is **Schur**: all eigenvalues satisfy $|\lambda_i(A)| < 1$.
- Lyapunov: $\exists P \succ 0$ s.t. $A^\top PA - P \prec 0$.

Controllability (reachability). The pair (A, B) is controllable if for any x_0, x_f there exists a *finite* input sequence $\{u_0, \dots, u_{N-1}\}$ that drives the state from x_0 to $x_N = x_f$.

Kalman controllability matrix.

$$\mathcal{C} \;=\; [\,B\;AB\;A^2B\;\cdots\;A^{n-1}B\,], \quad (A,B) \text{ controllable} \iff \operatorname{rank}(\mathcal{C}) = n.$$

Popov-Belevitch-Hautus (PBH) test.

$$(A,B) \text{ controllable} \iff \operatorname{rank}\begin{bmatrix}\lambda I - A & B\end{bmatrix} = n \text{ for all } \lambda \in \mathbb{C}.$$

It suffices to check λ equal to the eigenvalues of A.

Observability. The pair (A, C) is observable if the initial state x_0 can be uniquely determined from a finite sequence of outputs (and known inputs), e.g., from $\{y_0, \dots, y_{n-1}\}$.

Observability matrix.

$$\mathcal{O} \;=\; \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix}, \quad (A,C) \text{ observable} \iff \operatorname{rank}(\mathcal{O}) = n.$$

PBH test.

$$(A,C) \text{ observable} \iff \operatorname{rank}\begin{bmatrix}\lambda I - A^\top & C^\top\end{bmatrix} = n \text{ for all } \lambda \in \mathbb{C}.$$

Dual to controllability: (A, C) observable $\Leftrightarrow$ $(A^\top, C^\top)$ controllable.

4.2 LQR Trajectory Tracking

Classical LQR delivers an optimal linear state-feedback when dynamics are linear and the objective is quadratic. In many planning problems, however, we do not seek a single stationary feedback for *all* states but rather a *local stabilizer around a given (possibly time-varying) trajectory*—for instance, a motion plan from a trajectory optimizer or MPC's rolling nominal (see Section 4.3). LQR Tracking (also called time-varying LQR, TVLQR) provides exactly this: a *time-varying* linear feedback that stabilizes the system near a nominal state–control sequence and rejects small disturbances.

Problem Setup. Let the nominal (i.e., ignoring the disturbance) discrete-time system be

$$x_{t+1} \;=\; f_t(x_t, u_t), \qquad t = 0, \dots, N-1,$$

and suppose we have a *nominal trajectory* $\{(\bar{x}_t, \bar{u}_t)\}_{t=0}^{N-1}$ satisfying

$$\bar{x}_{t+1} \;=\; f_t(\bar{x}_t, \bar{u}_t).$$

Our goal is to design a controller that can stabilize the system with disturbance, i.e., $x_{t+1} = f_t(x_t, u_t, w_t)$, around the nominal trajectory.

Towards this, we define *deviations* from the nominal trajectory as

$$\delta x_t := x_t - \bar{x}_t, \qquad \delta u_t := u_t - \bar{u}_t.$$

If the true system is linear time-varying (or we linearize a nonlinear system along the nominal), we obtain the *deviation dynamics*

$$\delta x_{t+1} \;\approx\; A_t\,\delta x_t + B_t\,\delta u_t, \quad A_t := \left.\frac{\partial f_t}{\partial x}\right|_{(\bar{x}_t,\bar{u}_t)}, \quad B_t := \left.\frac{\partial f_t}{\partial u}\right|_{(\bar{x}_t,\bar{u}_t)}.$$

We penalize deviations with a quadratic cost

$$J \;=\; \delta x_N^\top Q_N \delta x_N \;+\; \sum_{t=0}^{N-1} \Big(\delta x_t^\top Q_t \delta x_t \;+\; \delta u_t^\top R_t \delta u_t\Big), \quad Q_t \succeq 0,\; R_t \succ 0.$$

LQR Tracking Algorithm. The tracking controller takes the *affine* form

$$u_t \;=\; \bar{u}_t \;-\; K_t\,(x_t - \bar{x}_t),$$

where $\{K_t\}_{t=0}^{N-1}$ are time-varying gains computed by a backward Riccati recursion on the deviation system (4.2) with cost (4.2).

From Proposition 4.1, we know the gains can be computed as follows.

Initialize at terminal time:

$$S_N \;=\; Q_N.$$

For $t = N-1,\, N-2,\, \ldots,\, 0$:

$$\begin{aligned} K_t &= \Big(R_t + B_t^\top S_{t+1} B_t\Big)^{-1} B_t^\top S_{t+1} A_t, \\ S_t &= Q_t \;+\; A_t^\top\Big(S_{t+1} - S_{t+1}B_t(R_t + B_t^\top S_{t+1} B_t)^{-1} B_t^\top S_{t+1}\Big)A_t. \end{aligned} \tag{4.22}$$

Given the gains $\{K_t\}$, apply at runtime:

$$u_t \;=\; \bar{u}_t - K_t\,(x_t - \bar{x}_t), \qquad t = 0, \ldots, N-1. \tag{4.23}$$

The following pseudocode summarizes the algorithm.

Algorithm: LQR Trajectory Tracking (TVLQR)

Inputs: nominal $\{(\bar{x}_t, \bar{u}_t)\}_{t=0}^{N-1}$, weights $\{Q_t, R_t\}$, terminal Q_N.

1. **Linearize** along the nominal to get A_t, B_t via (4.2).
2. **Backward pass:** compute K_t and S_t via (4.22).

3. **Apply feedback:** $u_t = \bar{u}_t - K_t(x_t - \bar{x}_t)$ as in (4.23).

Output: time-varying gains $\{K_t\}$ giving a local stabilizer around the trajectory.

We now apply TVLQR to a vehicle trajectory tracking problem.

Example 4.2 (LQR Trajectory Tracking for Unicyle)**.** We (i) define the dynamics in continuous and discrete time, (ii) specify a circular *nominal trajectory*, (iii) linearize the nonlinear dynamics *along the nominal*, (iv) state the deviation-cost weights Q, R (and terminal Q_N), and (v) list the *experiment setup* (discretization and horizon length).

Unicycle Dynamics (Continuous and Discrete).

State and input.

$$x = \begin{bmatrix} p_x \\ p_y \\ \theta \end{bmatrix} \in \mathbb{R}^3, \qquad u = \begin{bmatrix} v \\ \omega \end{bmatrix} \in \mathbb{R}^2,$$

where (p_x, p_y) is planar position, θ is heading, v is forward speed, and ω is yaw rate.

Continuous time:

$$\dot{p}_x = v\cos\theta, \qquad \dot{p}_y = v\sin\theta, \qquad \dot{\theta} = \omega. \tag{4.24}$$

Discrete time (forward Euler with step $h > 0$):

$$x_{t+1} \;=\; f(x_t, u_t) := \begin{bmatrix} p_{x,t} + h\, v_t \cos\theta_t \\ p_{y,t} + h\, v_t \sin\theta_t \\ \theta_t + h\, \omega_t \end{bmatrix}. \tag{4.25}$$

Nominal Trajectory (Circular Motion). We track a circle of radius $R = \dfrac{\bar{v}}{\bar{\omega}}$ using *constant nominal inputs*

$$\bar{u}_t \equiv \begin{bmatrix} \bar{v} \\ \bar{\omega} \end{bmatrix}, \qquad t = 0, \dots, N-1, \tag{4.26}$$

and generate the *nominal state* recursively under the discrete dynamics (4.25):

$$\bar{x}_{t+1} \;=\; f(\bar{x}_t, \bar{u}_t), \qquad \bar{x}_0 = \begin{bmatrix} R \\ 0 \\ \frac{\pi}{2} \end{bmatrix}. \tag{4.27}$$

We define *deviations* from the nominal:

$$\delta x_t := x_t - \bar{x}_t, \qquad \delta u_t := u_t - \bar{u}_t.$$

Linearization Along the Nominal. Linearize (4.25) at $(\bar{x}_t, \bar{u}_t)$ to obtain the deviation dynamics

$$\delta x_{t+1} \;\approx\; A_t\,\delta x_t + B_t\,\delta u_t, \tag{4.28}$$

with Jacobians (using $c_t := \cos\bar{\theta}_t,\ s_t := \sin\bar{\theta}_t$)

$$A_t \;=\; \frac{\partial f}{\partial x}\Big|_{(\bar{x}_t,\bar{u}_t)} = \begin{bmatrix} 1 & 0 & -h\,\bar{v}\,s_t \\ 0 & 1 & h\,\bar{v}\,c_t \\ 0 & 0 & 1 \end{bmatrix}, \qquad B_t \;=\; \frac{\partial f}{\partial u}\Big|_{(\bar{x}_t,\bar{u}_t)} = \begin{bmatrix} h\,c_t & 0 \\ h\,s_t & 0 \\ 0 & h \end{bmatrix}. \tag{4.29}$$

Deviation Cost (Weights Q, R, Q_N). We penalize deviations with a quadratic stage/terminal cost

$$J \;=\; \delta x_N^\top Q_N\,\delta x_N \;+\; \sum_{t=0}^{N-1} \Big(\delta x_t^\top Q\,\delta x_t + \delta u_t^\top R\,\delta u_t\Big),$$

using the weights:

$$Q = \mathrm{diag}(30,\ 30,\ 5), \qquad Q_N = \mathrm{diag}(60,\ 60,\ 8), \qquad R = \mathrm{diag}(0.2,\ 0.2). \tag{4.30}$$

These reflect a stronger emphasis on position tracking, a moderate penalty on heading error, and a mild penalty on control *deviations* from the nominal.

Experiment Setup.

- **Discretization step:** $h = 0.02$ s.
- **Horizon length:** $T_{\text{final}} = 12$ s.
- **Number of steps:** $N = T_{\text{final}}/h = \mathbf{600}$.
- **Nominal inputs:** $\bar{v} = 1.2$ m/s, $\bar{\omega} = 0.4$ rad/s (radius $R = \bar{v}/\bar{\omega}$).
- **Initialization:** the nominal starts at $\bar{x}_0 = [\,R,\ 0,\ \pi/2\,]^\top$; the actual system may start with a small offset (see code).

With (A_t, B_t) from (4.29) and weights (4.30), the TVLQR backward Riccati recursion (Section 4.2) yields gains K_t. We then apply the **local feedback**

$$u_t \;=\; \bar{u}_t - K_t\,(x_t - \bar{x}_t),$$

to robustly track the circular nominal under small disturbances.

Disturbances. To test robustness, we inject additive process disturbances into the discrete dynamics:

$$x_{t+1} \;=\; f(x_t, u_t) \;+\; w_t, \qquad t = 0, \ldots, N-1,$$

where

$$w_t \;=\; \underbrace{\eta_t}_{\text{i.i.d. Gaussian noise}} \;+\; \underbrace{g_t}_{\text{deterministic gust}}.$$

1) Small i.i.d. Gaussian process noise. We draw $\eta_t \sim \mathcal{N}(0, W)$ independently at each step with

$$W \;=\; \mathrm{diag}\big(\sigma_x^2,\ \sigma_y^2,\ \sigma_\theta^2\big), \qquad \sigma_x = \sigma_y = 0.01 \text{ m}, \quad \sigma_\theta = \mathrm{deg2rad}(0.2).$$

This noise perturbs the post-update state components (p_x, p_y, θ).

2) Finite-duration "gust" impulse. In addition to η_t, we apply a brief deterministic bias over a window

$$t \in [t_g,\ t_g + \Delta] \;=\; [\,4.0\,\mathrm{s},\ 4.8\,\mathrm{s}\,),$$

implemented at the discrete indices $\{t_g, \dots, t_g + \Delta\}$. During this window we set

$$g_t \;=\; \begin{bmatrix} \delta p_x \\ 0 \\ \delta\theta \end{bmatrix}, \qquad \delta p_x = 0.01 \text{ m per step}, \quad \delta\theta = \mathrm{deg2rad}(1.8) \text{ per step},$$

and $g_t = \mathbf{0}$ otherwise. This models a short-lived lateral drift and a heading kick.

Results. Fig. 4.3 visualizes the nominal trajectory (the dotted circle) and the TVLQR-stabilized trajectory in blue. To clearly see the impact of closed-loop LQR tracking, we also plotted the open-loop trajectory, i.e., the system's trajectory if no feedback is applied. We can observe that the TVLQR controller effectively rejects the disturbances and stabilizes the closed-loop trajectory along the nominal path.

Fig. 4.4 visualizes the state tracking error (position and heading error) as well as compares the closed-loop control with open-loop control.

You can play with the code here.

4.3 Trajectory Optimization

In Section 4.2 we saw that TVLQR gives a powerful *local stabilizer* around a nominal state–control sequence $(\bar{x}_t, \bar{u}_t)$. This raises a natural question:

> Where do nominal trajectories come from?

In many robotics tasks (maneuvering a car, landing a rocket, walking with a robot), we must compute a *feasible, high-quality open-loop plan* that respects the dynamics and constraints. **Trajectory Optimization (TO)** does exactly this: it searches over sequences $\{x_t, u_t\}$ to minimize a cumulative cost while satisfying the system dynamics and constraints.

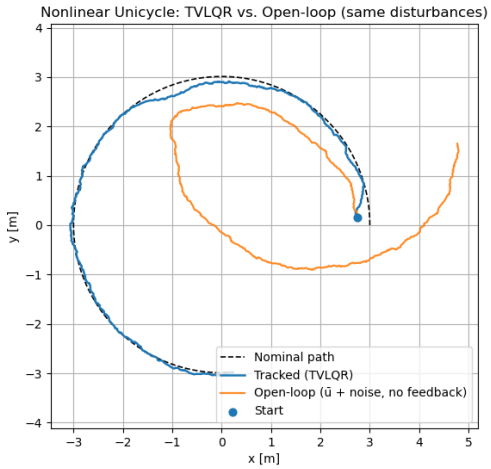

Figure 4.3: LQR Trajectory Tracking for Unicyle: comparison between nominal trajectory, open-loop trajectory, and closed-loop trajectory with feedback.

Moreover, if we can solve TO **quickly** (or approximately, but reliably), then by re-solving over a short horizon at each time step and applying only the first control, we obtain **Model Predictive Control (MPC)**—a feedback controller that blends optimization with robustness (see Section 4.4 later). Thus, TO is both a **planner** and the engine behind **feedback via MPC**.

General Nonlinear Trajectory Optimization Problem. We adopt the standard discrete-time nonlinear system

$$x_{t+1} = f_t(x_t, u_t), \qquad t = 0, \dots, N-1,$$

with state $x_t \in \mathbb{R}^n$ and control $u_t \in \mathbb{R}^m$. A generic finite-horizon TO problem is

$$\begin{aligned} \min_{\{x_t, u_t\}} \quad & \Phi(x_N) + \sum_{t=0}^{N-1} \ell_t(x_t, u_t) \\ \text{s.t.} \quad & x_{t+1} = f_t(x_t, u_t), \qquad t = 0, \dots, N-1, \\ & x_0 = \hat{x}_0 \quad \text{(given)}, \\ & x_t \in \mathcal{X}_t, \quad u_t \in \mathcal{U}_t \quad \text{(bounds)}, \\ & g_t(x_t, u_t) \le 0, \quad h_t(x_t, u_t) = 0 \quad \text{(path/terminal constraints)}. \end{aligned} \tag{4.31}$$

Here ℓ_t and Φ encode performance (e.g., energy, time, tracking error), $\mathcal{X}_t, \mathcal{U}_t$ capture box limits and safety sets, and g_t, h_t represent additional nonlinear constraints (obstacles, terminal goals, etc.).

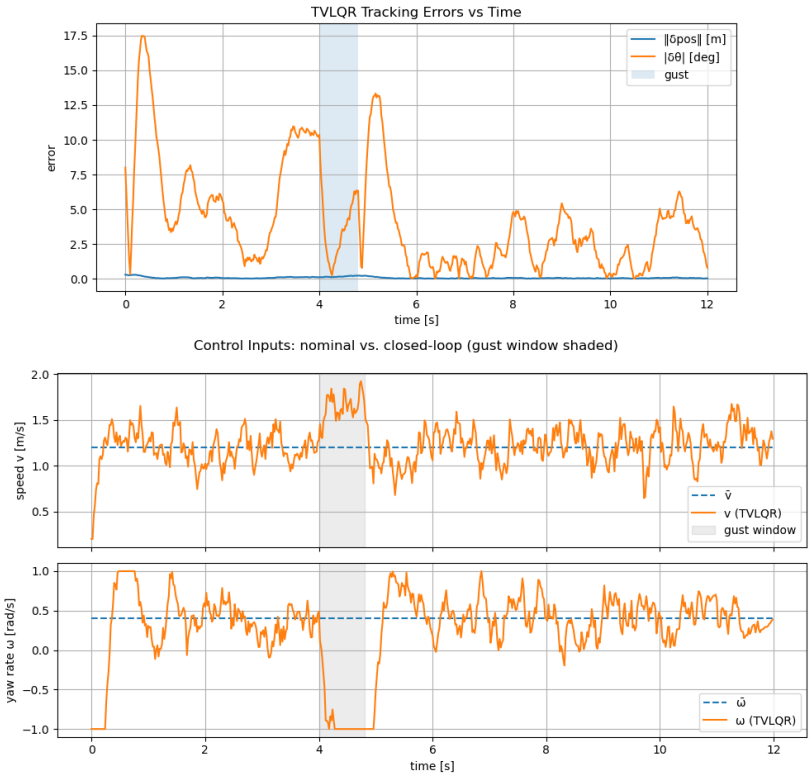

Figure 4.4: LQR Trajectory Tracking for Unicyle: state tracking error (top) and control signal (bottom).

Solving (4.31) directly is difficult in general. A widely used strategy is to iteratively approximate it by *quadratic* subproblems that can be solved efficiently. This leads to **iLQR** and its second-order cousin **DDP** (see Section 4.3.2).

4.3.1 Iterative LQR

High-level intuition. iLQR (iterative LQR) alternates between:

1. **Local modeling:** around a *current* nominal trajectory $\{(\bar{x}_t, \bar{u}_t)\}$,
   - **linearize** the dynamics,
   - **quadratically approximate** the cost.
2. **LQR step:** solve the resulting *time-varying LQR* subproblem to obtain a *time-varying affine policy*

   $$\delta u_t = k_t + K_t\, \delta x_t, \quad \delta x_t := x_t - \bar{x}_t,\ \delta u_t := u_t - \bar{u}_t,$$

   which gives both a feedforward step k_t (to change the nominal control) and a feedback gain K_t (to stabilize the rollout).
3. **Forward rollout + line search:** apply $u_t^{\text{new}} = \bar{u}_t + \alpha k_t + K_t(x_t^{\text{new}} - \bar{x}_t)$ to the true nonlinear dynamics, producing a new nominal trajectory $\{(\bar{x}_t, \bar{u}_t)\}$. Here we choose $\alpha \in (0, 1]$ to reduce the cost and respect constraints.
4. **Repeat** until convergence (cost decrease and dynamics residuals are small).

iLQR can be viewed as a Gauss–Newton method on trajectories: it uses first-order dynamics and second-order cost, capturing the dominant curvature while remaining numerically robust and fast.

4.3.1.1 LQR Subproblem (one iLQR outer iteration)

Given a nominal trajectory $\{(\bar{x}_t, \bar{u}_t)\}_{t=0}^{N-1}$ with $\bar{x}_{t+1} = f_t(\bar{x}_t, \bar{u}_t)$, define deviations

$$\delta x_t := x_t - \bar{x}_t, \qquad \delta u_t := u_t - \bar{u}_t, \qquad \delta x_0 \text{ given.}$$

Linearized Dynamics. We linearize the dynamics along the nominal trajectory

$$\delta x_{t+1} \;\approx\; A_t\, \delta x_t + B_t\, \delta u_t, \quad A_t := \left.\frac{\partial f_t}{\partial x}\right|_{(\bar{x}_t, \bar{u}_t)}, \quad B_t := \left.\frac{\partial f_t}{\partial u}\right|_{(\bar{x}_t, \bar{u}_t)}.$$

Quadratic Cost Approximation. We perform a quadratic approximation of the objective function about $(\bar{x}_t, \bar{u}_t)$

$$\ell_t(x_t, u_t) \approx \ell_t + \ell_{x,t}^\top \delta x_t + \ell_{u,t}^\top \delta u_t + \frac{1}{2} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^\top \begin{bmatrix} \ell_{xx,t} & \ell_{xu,t} \\ \ell_{ux,t} & \ell_{uu,t} \end{bmatrix} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix},$$
$$\Phi(x_N) \approx \Phi + \Phi_x^\top \delta x_N + \frac{1}{2} \delta x_N^\top \Phi_{xx} \, \delta x_N.$$

The LQR Subproblem. With (4.3.1.1)–(4.3.1.1), the iLQR subproblem at this outer iteration is the **finite-horizon linear–quadratic program in deviations**:

$$\min_{\{\delta x_t, \delta u_t\}} \quad \underbrace{\frac{1}{2} \delta x_N^\top \Phi_{xx} \, \delta x_N + \Phi_x^\top \delta x_N}_{\text{terminal}} + \underbrace{\sum_{t=0}^{N-1} \left(\frac{1}{2} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^\top \begin{bmatrix} \ell_{xx,t} & \ell_{xu,t} \\ \ell_{ux,t} & \ell_{uu,t} \end{bmatrix} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} + \ell_{x,t}^\top \delta x_t + \ell_{u,t}^\top \delta u_t \right)}_{\text{stage}}$$
$$\text{s.t.} \quad \delta x_{t+1} = A_t \, \delta x_t + B_t \, \delta u_t, \qquad t = 0, \dots, N-1,$$
$$\delta x_0 \text{ given.}$$

Notes. The iLQR subproblem (4.3.1.1) is slightly different from the previous finite-horizon LQR formulation (4.3) in the sense that the objective function of (4.3.1.1) also contains **linear terms** in $\delta x_t, \delta u_t$, and those linear terms come from the Taylor expansion of the original nonlinear objective fuctions. In this case, we will see in the following that the optimal policy is **affine** (feedforward k_t + feedback K_t).

4.3.1.2 Solving the Subproblem by Dynamic Programming

We posit a **quadratic value approximator** at each time:

$$V_t(\delta x_t) \;\approx\; V_t + V_{x,t}^\top \delta x_t + \frac{1}{2} \delta x_t^\top V_{xx,t} \, \delta x_t, \qquad V_{x,N} = \Phi_x, \; V_{xx,N} = \Phi_{xx}.$$

Note that this quadratic value approximator also contains linear and constant terms because the objective function contains linear terms.

Define the local Q-function at stage t by substituting the linear dynamics into the next-step value (this is our familiar Q-value in RL):

$$Q_t(\delta x_t, \delta u_t) \;=\; \ell_t(x_t, u_t) + V_{t+1}(A_t \delta x_t + B_t \delta u_t),$$

which, after collecting terms, yields the iLQR blocks

$$\begin{aligned} Q_{x,t} &= \ell_{x,t} + A_t^\top V_{x,t+1}, \qquad Q_{u,t} = \ell_{u,t} + B_t^\top V_{x,t+1}, \\ Q_{xx,t} &= \ell_{xx,t} + A_t^\top V_{xx,t+1} A_t, \quad Q_{ux,t} = \ell_{ux,t} + B_t^\top V_{xx,t+1} A_t, \\ Q_{uu,t} &= \ell_{uu,t} + B_t^\top V_{xx,t+1} B_t. \end{aligned}$$

The iLQR blocks assemble into a big matrix such that

$$Q_t(\delta x_t, \delta u_t) \;=\; \frac{1}{2} \begin{bmatrix} 1 \\ \delta x_t \\ \delta u_t \end{bmatrix}^\top \begin{bmatrix} 2c_t & Q_{x,t}^\top & Q_{u,t}^\top \\ Q_{x,t} & Q_{xx,t} & Q_{xu,t} \\ Q_{u,t} & Q_{ux,t} & Q_{uu,t} \end{bmatrix} \begin{bmatrix} 1 \\ \delta x_t \\ \delta u_t \end{bmatrix}.$$

where c_t collects all stage/terminal constants e.g., $\ell_t + V_{t+1}$.

Solving the local Q (backward pass). Set the first-order condition w.r.t. δu:

$$0 \;=\; \partial_{\delta u} Q_t \;=\; Q_{u,t} + Q_{ux,t}\delta x + Q_{uu,t}\delta u.$$

Solve for the affine control law

$$\delta u_t^\star \;=\; k_t + K_t\, \delta x, \qquad k_t = -Q_{uu,t}^{-1} Q_{u,t}, \quad K_t = -Q_{uu,t}^{-1} Q_{ux,t},$$

which is exactly the LQR solution for the quadratic Q_t.

Substitute $\delta u_t^\star$ back into (4.3.1.2). The minimized Q becomes a quadratic in δx with coefficients given by

$$\begin{aligned} V_{x,t} &= Q_{x,t} + Q_{xu,t} k_t + K_t^\top Q_{uu,t} k_t + K_t^\top Q_{u,t}, \\ V_{xx,t} &= Q_{xx,t} + Q_{xu,t} K_t + K_t^\top Q_{ux,t} + K_t^\top Q_{uu,t} K_t, \end{aligned}$$

with terminal

$$V_{x,N} = \Phi_x, V_{xx,N} = \Phi_{xx}.$$

Forward Pass (apply the computed policy). Given $\{k_t, K_t\}$, produce a candidate trajectory on the *true* nonlinear dynamics using a line search $\alpha \in (0, 1]$:

$$\begin{aligned} u_t^{\text{cand}} &= \bar{u}_t + \alpha k_t + K_t(x_t^{\text{cand}} - \bar{x}_t), \\ x_{t+1}^{\text{cand}} &= f_t(x_t^{\text{cand}}, u_t^{\text{cand}}), \qquad x_0^{\text{cand}} = \hat{x}_0. \end{aligned}$$

Choose α (e.g., $\{1, \frac{1}{2}, \frac{1}{4}, \dots\}$) to reduce the **true** cost and respect constraints, then update the nominal:

$$(\bar{x}_t, \bar{u}_t) \;\leftarrow\; (x_t^{\text{cand}}, u_t^{\text{cand}}).$$

The following pseudocode summarizes iLQR.

Algorithm: iLQR (Trajectory Generation)

Inputs: dynamics f_t, initial state $\hat{x}_0$, horizon N, stage/terminal costs ℓ_t, Φ, initial guess $\{\bar{u}_t\}$.

1. **Initialize** nominal rollout $\{\bar{x}_t, \bar{u}_t\}$ from $\hat{x}_0$.
2. **Linearize & quadratize** at $\{(\bar{x}_t, \bar{u}_t)\}$: build A_t, B_t and cost derivatives.
3. **Backward pass (TVLQR):** compute $\{k_t, K_t\}$ using (4.3.1.2) and update $V_{x,t}, V_{xx,t}$ via (4.3.1.2).
4. **Forward rollout:** apply $u_t^{\text{new}} = \bar{u}_t + \alpha k_t + K_t(x_t^{\text{new}} - \bar{x}_t)$ on the **true** dynamics, pick α by line search.
5. **Convergence check:** stop if the cost decrease and dynamics residuals fall below thresholds; otherwise, set the new nominal and **repeat** from Step 2.

The next example applies iLQR to trajectory generation for rocket landing.

Example 4.3 (iLQR for Rocket Landing)**.** We model a planar (2D) rocket with state and control

$$x = \begin{bmatrix} p_x & p_y & v_x & v_y & \theta & \omega \end{bmatrix}^\top, \qquad u = \begin{bmatrix} T & \tau \end{bmatrix}^\top,$$

where (p_x, p_y) is position, (v_x, v_y) is velocity, θ is attitude (pitch) and ω its angular rate. The thrust $T \geq 0$ acts **along the body axis** (pointing out of the engine), and τ is a planar torque about the center of mass. Continuous-time dynamics are

$$\begin{aligned} \dot{p}_x &= v_x, & \dot{p}_y &= v_y, \\ \dot{v}_x &= \frac{T}{m}\sin\theta, & \dot{v}_y &= \frac{T}{m}\cos\theta - g, \\ \dot{\theta} &= \omega, & \dot{\omega} &= \frac{\tau}{I_{zz}}. \end{aligned}$$

In simulation we use RK4 with stepsize h to propagate the true dynamics (4.3). For iLQR's local subproblems we form the continuous Jacobians $(A_c, B_c) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial u})$ at the current nominal and use the standard first-order discrete map

$$A_t \;\approx\; I + h\, A_c(\bar{x}_t, \bar{u}_t), \qquad B_t \;\approx\; h\, B_c(\bar{x}_t, \bar{u}_t).$$

Soft-Landing Objective. The goal is a **soft, upright landing** at the origin:

$$x_{\text{goal}} = \mathbf{0} \quad \Longleftrightarrow \quad p_x = p_y = 0,\ v_x = v_y = 0,\ \theta = 0,\ \omega = 0.$$

We penalize deviations from this goal along the entire trajectory and especially at the terminal state to encourage low touchdown velocities and an upright attitude.

Cost Function. With horizon N and step h, the discrete objective is

$$J \;=\; \frac{1}{2}(x_N - x_g)^\top Q_f (x_N - x_g) \;+\; \sum_{t=0}^{N-1} \Big[\frac{1}{2}(x_t - x_g)^\top Q (x_t - x_g) \;+\; \frac{1}{2} u_t^\top R u_t\Big],$$

where $x_g = \mathbf{0}$. In the example:

$$\begin{aligned} Q &= \text{diag}(1,\ 2,\ 0.5,\ 0.5,\ 2,\ 0.5), \\ Q_f &= \text{diag}(200,\ 300,\ 50,\ 50,\ 300,\ 50), \\ R &= \text{diag}(10^{-3},\ 10^{-3}). \end{aligned}$$

These weights place strong emphasis on terminal altitude and attitude (p_y, θ), moderate emphasis on velocities and lateral position, and a light regularization on the controls.

Experiment Setup.

- **Physical parameters.** Gravity $g = 9.81\,\mathrm{m/s^2}$, mass $m = 1.0\,\mathrm{kg}$, planar inertia $I_{zz} = 0.2\,\mathrm{kg\,m^2}$.
- **Discretization.** Stepsize $h = 0.05\,\mathrm{s}$; horizon $T = 6.0\,\mathrm{s}$; number of steps $N = T/h = 120$.
- **Initial state.**

$$x_0 = \left[\, 5.0,\ 10.0,\ -0.5,\ -1.0,\ \mathrm{deg2rad}(10),\ 0 \,\right]^\top,$$

  i.e., 10 m altitude, lateral offset, small descent and slight pitch.
- **Initial nominal controls.** Constant hover thrust and zero torque:

$$\bar{u}_t = \left[\, mg,\ 0 \,\right]^\top, \qquad t = 0, \ldots, N-1.$$

- **iLQR procedure.** Each outer iteration:

  1) Linearize dynamics and quadratize the cost along the current nominal ((4.3), (4.3));

  2) Solve the **time-varying LQR** subproblem to get affine updates $\delta u_t = k_t + K_t\,\delta x_t$;

  3) **Forward rollout** on the nonlinear RK4 dynamics with

$$u_t^{\mathrm{new}} = \bar{u}_t + \alpha\, k_t + K_t(x_t^{\mathrm{new}} - \bar{x}_t),$$

     using a backtracking line search over $\alpha \in \{1, \frac{1}{2}, \frac{1}{4}, \ldots\}$ (note: α scales only the **feedforward** k_t, not the feedback K_t);

  4) Update the nominal and repeat until cost reduction is small.

Fig. 4.5 plots the **initial**, **intermediate**, and **final** trajectories, and render the rocket as oriented rectangles (boxes) using (p_x, p_y, θ) to visualize attitude along the descent. We can see iLQR successfully generated a soft landing trajectory.

You can play with the code here.

4.3.2 Differential Dynamic Programming

Similar to iLQR, skipped for now.

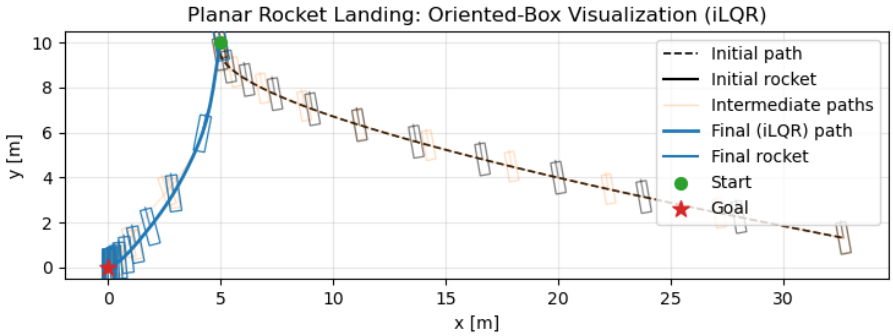

Figure 4.5: iLQR Trajectory Generation for Rocket Landing.

4.3.3 Quadratic Programming

Trajectory optimization (TO) with nonlinear dynamics and objectives is well served by iLQR/DDP: at each outer iteration, they *linearize* the dynamics and *quadratize* the objective, then solve a *time-varying LQR* subproblem. This works remarkably well for *unconstrained* or softly constrained problems.

However, many TO tasks are **constrained**—e.g., obstacle avoidance, actuator limits, keep-out zones, terminal envelopes. Hard constraints are awkward for iLQR/DDP (they typically enter via penalties or saturations), and feasibility can be fragile. For such cases, it is often more natural to frame TO as a **nonlinear program (NLP)**—an optimization problem with general nonlinear objective and constraints. This brings the full machinery of modern numerical optimization (see, e.g., (Nocedal and Wright, 1999)).

As a first step, we study the **convex** special case where the linearization already yields linear dynamics, affine constraints, and a quadratic objective (these are typically known as "constrained LQR" problems). This leads to **Quadratic Programming (QP)**, a cornerstone problem class with mature, efficient solvers. In the next section, we will lift these ideas to **Sequential Quadratic Programming (SQP)** to handle *general* constrained TO.

4.3.3.1 From Trajectory Optimization to Quadratic Programming

Start from the constrained TO template in (4.31). Suppose:

- **Linear (time-varying) dynamics** (from linearization or an intrinsically linear model):

$$x_{t+1} = A_t x_t + B_t u_t + a_t, \qquad t = 0, \dots, N-1,$$

  with given $x_0 = \hat{x}_0$.
- **Quadratic objective** (from quadratization or an Linear-Quadratic tracking design):

$$\Phi(x_N) + \sum_{t=0}^{N-1} \ell_t(x_t, u_t) \;\equiv\; \frac{1}{2} x_N^\top Q_N x_N + q_N^\top x_N + \sum_{t=0}^{N-1} \Big(\frac{1}{2} [x_t; u_t]^\top H_t [x_t; u_t] + h_t^\top [x_t; u_t] \Big),$$

  with $Q_N \succeq 0$, $H_t = \begin{bmatrix} Q_t & S_t \\ S_t^\top & R_t \end{bmatrix} \succeq 0$ and $R_t \succ 0$ for convexity.
- **Affine path/terminal constraints** (from linearized safety/goal constraints):

$$G_t^x x_t + G_t^u u_t \le g_t, \qquad F_x x_N \le f.$$

Define the stacked decision vector

$$z := \left[x_0^\top, u_0^\top, x_1^\top, u_1^\top, \dots, x_{N-1}^\top, u_{N-1}^\top, x_N^\top\right]^\top.$$

Then the horizon-wide problem is a **QP**:

$$\begin{aligned} \min_{z} \quad & \frac{1}{2} z^\top H z \ + \ h^\top z \\ \text{s.t.} \quad & A_{\text{dyn}} z = b_{\text{dyn}} \quad \text{(stacked dynamics and } x_0 = \hat{x}_0\text{)}, \\ & G z \le g \qquad\quad \text{(stacked affine path/terminal constraints)} \end{aligned}$$

Here $H \succeq 0$ is block-sparse (banded) due to the stagewise structure; the constraint matrices are also sparse/banded because each dynamic constraint couples only (x_t, u_t, x_{t+1}).

Exercise 4.1. Can you write down the blocks in H, A, and G, as functions of $H_t, A_t, B_t, G_t^x, G_t^u$? Then, observe the block-sparsity patterns.

Convexity. If all stage Hessians $H_t \succeq 0$ and $Q_N \succeq 0$, (4.3.3.1) is a **convex QP** with a **unique** minimizer when H is positive definite on the feasible subspace (e.g., via $R_t \succ 0$).

4.3.3.2 Solving the Quadratic Program

We now discuss how to solve a general convex quadratic program (QP) containing both equality and inequality constraints:

$$\begin{aligned} \min_{z \in \mathbb{R}^n} \quad & \frac{1}{2} z^\top H z \ + \ h^\top z \\ \text{s.t.} \quad & Az = b \\ & Gz \le g \end{aligned}$$

where $z \in \mathbb{R}^n$ is the decision variable, $H \in \mathbb{S}^n, h \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, G \in \mathbb{R}^{p \times n}, g \in \mathbb{R}^p$ are given problem data (e.g., generated as in Section 4.3.3.1). We assume $H \succeq 0$ is positive semidefinite.

There are multiple popular algorithms to solve (4.3.3.2), e.g., the active set algorithm, the interior point algorithm, and the alternating direction method of multipliers. Here we only present the *primal–dual interior point method* (PD-IPM) due to its generality and robustness. Before presenting the PD-IPM algorithm, it is beneficial to review Newton's method for solving a system of equations.

Newton's Method

Given a function $f : \mathbb{R} \to \mathbb{R}$ that is continuously differentiable, Newton's method is designed to find a root of

$$f(x) = 0.$$

Given an initial iterate $x^{(0)}$, Newton's method works as follows

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})},$$

where $f'(x^{(k)})$ denotes the derivative of f at the current iterate $x^{(k)}$. This simple algorithm is indeed the most important foundation of modern numerical optimization. Under mild conditions, Newton's method has at least quadratic convergence rate, that is to say, if $|x^{(k)} - x^\star| = \epsilon$, then $|x^{(k+1)} - x^\star| = O(\epsilon^2)$ (it should be noted that there exist pathological cases where even linear convergence is not guaranteed, e.g., when $f'(x^\star) = 0$).

Newton's method can be generalized to find a point at which multiple functions vanish simultaneously. Given a function $F : \mathbb{R}^n \to \mathbb{R}^n$ that is continuously differentiable, and an initial iterate $x^{(0)}$, Newton's method reads

$$x^{(k+1)} = x^{(k)} - J_F(x^{(k)})^{-1} F(x^{(k)}), \tag{4.32}$$

where $J_F(\cdot)$ denotes the Jacobian of F. Iteration (4.32) is equivalent to

$$\begin{aligned} J_F(x^{(k)})\Delta x^{(k)} &= -F(x^{(k)}) \\ x^{(k+1)} &= x^{(k)} + \Delta x^{(k)} \end{aligned} \tag{4.33}$$

i.e., one first solves a linear system of equations to find an update direction $\Delta x^{(k)}$, and then take a step along the direction.

As we will see, PD-IPMs for solving convex QPs can be seen as applying Newton's method to the perturbed KKT system of optimality conditions.

Slacks, Lagrangian, and KKT Optimality. Introduce $s \in \mathbb{R}^p$ so that

$$Gz + s = g, \qquad s \geq 0.$$

Let $y \in \mathbb{R}^m$ be the Lagrangian multipliers for $Az = b$, $\nu \in \mathbb{R}^p$ for the equality $Gz + s = g$, and $w \in \mathbb{R}^p$ (with $w \geq 0$) for the inequality $s \geq 0$. The Lagrangian for the QP (4.3.3.2) is

$$\mathcal{L}(z, s, y, \nu, w) = \frac{1}{2} z^\top H z + h^\top z + y^\top (Az - b) + \nu^\top (Gz + s - g) - w^\top s.$$

From the Lagrangian, we can derive the KKT optimality conditions, i.e., under technical conditions (such as constraint qualification), a point (z, s) is a local minimizer of the QP if and only if there exists dual variables (ν, w) satisfying:

$$\begin{aligned} \text{(stationarity)}: \quad & \nabla_z \mathcal{L} = Hz + h + A^\top y + G^\top \nu = 0, \\ & \nabla_s \mathcal{L} = \nu - w = 0 \implies \nu = w, \\ \text{(primal feasibility)}: \quad & Az - b = 0, \qquad Gz + s - g = 0, \qquad s \geq 0, \\ \text{(dual feasibility)}: \quad & w \geq 0, \\ \text{(complementarity)}: \quad & s_i w_i = 0, \quad i = 1, \ldots, p. \end{aligned}$$

Using $\nu = w$ we eliminate ν and keep variables (z, s, y, w). Since the QP is convex, we know that any local minimizer is a global minimizer. Hence, if we can solve the KKT system (4.3.3.2), we can find an optimal solution of the QP.

If you are not familiar with the Lagrangian and the KKT optimality conditions, make sure to review Appendix A.1.3 and A.1.4.

Central Path and Residuals. Replace complementarity by the *perturbed* condition

$$SW\mathbf{1} = \sigma\mu\,\mathbf{1}, \qquad \mu := \frac{1}{p}\,s^\top w, \qquad \sigma \in (0,1)$$

with $S = \operatorname{diag}(s)$, $W = \operatorname{diag}(w)$. At a current iterate (z, s, y, w) define residuals

$$\begin{aligned} r_{\text{dual}} &= Hz + h + A^\top y + G^\top w, \\ r_{\text{pe}} &= Az - b, \\ r_{\text{pi}} &= Gz + s - g, \\ r_{\text{cent}} &= SW\mathbf{1} - \sigma\mu\,\mathbf{1}. \end{aligned}$$

Newton System (primal–dual step). Now that we have arrived at the perturbed KKT system of equations in (4.3.3.2) where we aim to drive all the residuals to zero. This is a system of $(n + m + p + p)$ nonlinear equations in $(n + m + p + p)$ variables. Therefore, we can apply Newton's method to solve the system of equations.

Note that we actually want to solve the original KKT system with $\sigma\mu = 0$. However, this system is ill-conditioned and directly applying Newton's method would lead to instability. Therefore, we solve the perturbed KKT system with $\sigma\mu > 0$ and at each iteration we move closer to the original KKT system with $\sigma \in (0,1)$ so that in the limit we will converge (arbitrarily close) to a solution of the original KKT system.

Solve for the Newton direction $(\Delta z, \Delta s, \Delta y, \Delta w)$:

$$\begin{aligned} H\,\Delta z + A^\top \Delta y + G^\top \Delta w &= -\,r_{\text{dual}}, \\ A\,\Delta z &= -\,r_{\text{pe}}, \\ G\,\Delta z + \Delta s &= -\,r_{\text{pi}}, \\ W\,\Delta s + S\,\Delta w &= -\,r_{\text{cent}}. \end{aligned}$$

Eliminate $\Delta s = -r_{\text{pi}} - G\Delta z$ in the last equation to get

$$S\,\Delta w = -\,r_{\text{cent}} + W\,r_{\text{pi}} + W\,G\,\Delta z \quad \Longrightarrow \quad \Delta w = S^{-1}\big(-r_{\text{cent}} + W r_{\text{pi}} + WG\Delta z\big)\,.$$

Substitute into the first equation to obtain the **reduced symmetric system** in $(\Delta z, \Delta y)$:

$$\begin{bmatrix} H + G^\top DG & A^\top \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta y \end{bmatrix} = - \begin{bmatrix} r_{\text{dual}} - G^\top S^{-1}(r_{\text{cent}} + W r_{\text{pi}}) \\ r_{\text{pe}} \end{bmatrix}$$

with $D := S^{-1}W$. Then recover

$$\Delta w = S^{-1}\big(-r_{\text{cent}} + W r_{\text{pi}} + WG\Delta z\big), \qquad \Delta s = -r_{\text{pi}} - G\Delta z.$$

Step Lengths. Choose step sizes to preserve positivity of s, w:

$$\alpha_{\text{pri}} = \min\Big(1,\ \eta \min_{\Delta s_i<0} \frac{-s_i}{\Delta s_i}\Big), \qquad \alpha_{\text{du}} = \min\Big(1,\ \eta \min_{\Delta w_i<0} \frac{-w_i}{\Delta w_i}\Big),$$

with $\eta \in (0,1)$ (e.g., 0.99). Update both primal and dual variables

$$z \leftarrow z + \alpha_{\text{pri}}\,\Delta z, \qquad s \leftarrow s + \alpha_{\text{pri}}\,\Delta s,$$
$$y \leftarrow y + \alpha_{\text{du}}\,\Delta y, \qquad w \leftarrow w + \alpha_{\text{du}}\,\Delta w.$$

Initialization and Stopping.

- **Initialization.** Find any z satisfying $Az = b$ (e.g., least-squares projection). Set
$$s := \max\{\mathbf{1},\, g - Gz\}, \quad w := \mathbf{1},$$
to ensure strict positivity ($s > 0, w > 0$); choose $y := 0$.
- **Stopping.** Terminate when
$$\|r_{\text{dual}}\|_\infty \le \varepsilon, \quad \|r_{\text{pe}}\|_\infty \le \varepsilon, \quad \|r_{\text{pi}}\|_\infty \le \varepsilon, \quad \mu \le \varepsilon,$$
for a small tolerance ε (e.g., 10^{-6}).

The following pseudocode implements the PD-IPM algorithm for solving convex QP.

Algorithm: Primal–Dual Interior-Point for Convex QP

Input: $H \succeq 0, h, A, b, G, g$; tolerance ε; $\eta = 0.99$; $\sigma \in (0,1)$

1. Initialize z with $Az = b$; set $s > 0, w > 0$ (e.g., $s = \max\{1, g - Gz\}$, $w = \mathbf{1}$); set $y = 0$.
2. Repeat until convergence:
    1. Compute residuals $r_{\text{dual}}, r_{\text{pe}}, r_{\text{pi}}$, $\mu = (s^\top w)/p$.
    2. Solve the reduced system (4.3.3.2) to obtain Newton direction.
    3. Compute $\alpha_{\text{pri}}, \alpha_{\text{du}}$ by (4.3.3.2).
    4. Update (z, s, y, w).
    5. Check stopping criteria; if satisfied, **return** $z^\star$.

Remarks.

- For QPs obtained from trajectory optimization problems, the matrices are typically sparse (e.g., time-banded sparsity). This sparsity can be leveraged when forming and solving the Newton direction.

- In practice, Mehrotra's predictor–corrector method is used to improve the robustness and convergence of PD-IPM.

Software. It is important to understand the high-level algorithmic idea for solving a convex QP. However, in practice, there are many mature QP solvers and it takes just a few lines of code to call your favorite QP solver.

The following code snippet shows how to define a QP in `cvxpy` and then solve it using MOSEK (which implements PD-IPM).

```
# Minimal dense QP with CVXPY
#   minimize    (1/2) x^T P x + q^T x
#   subject to  Ax <= b, 1^T x = 1
#
# pip install cvxpy

import numpy as np
import cvxpy as cp

# ----- QP data (dense) -----
P = np.array([
    [4.0, 1.0, 0.5],
    [1.0, 2.0, 0.3],
    [0.5, 0.3, 1.5]
], dtype=float)
# Make sure P is symmetric positive definite
P = 0.5 * (P + P.T) + 1e-9 * np.eye(3)

q = np.array([-1.0, -2.0, -3.0])

A = np.array([
    [1.0, -2.0, 1.0],   # linear inequality:  x1 - 2 x2 + x3   2
    [-1.0, 0.0, 0.0],   # x1   0  ->  -x1   0
    [0.0, -1.0, 0.0],   # x2   0  ->  -x2   0
    [0.0, 0.0, -1.0],   # x3   0  ->  -x3   0
    [1.0, 0.0, 0.0],    # x1   1.5
    [0.0, 1.0, 0.0],    # x2   1.5
    [0.0, 0.0, 1.0],    # x3   1.5
])
b = np.array([2.0, 0.0, 0.0, 0.0, 1.5, 1.5, 1.5])

# Equality: sum(x) = 1
e = np.ones((1, 3))
d = np.array([1.0])
```

```python
# ----- CVXPY problem -----
x = cp.Variable(3)

objective = cp.Minimize(0.5 * cp.quad_form(x, P) + q @ x)
constraints = [
    A @ x <= b,
    e @ x == d
]

prob = cp.Problem(objective, constraints)
# You can choose a solver; OSQP is common for QPs. ECOS/SCS also work.
prob.solve(solver=cp.MOSEK, verbose=True)

print("Status:", prob.status)
print("Optimal value:", prob.value)
print("x* =", x.value.round(6))

# (Optional) check constraints
ineq_res = (A @ x.value - b)
eq_res = (e @ x.value - d)
print("Max inequality residual (<=0):", np.max(ineq_res))
print("Equality residual ( 0):", eq_res.item())
```

Running the code produces the following output. You should now be able to interpret the iterations of MOSEK.

```
-----------------------------------------------------------------------------
                                 Numerical solver
-----------------------------------------------------------------------------
(CVXPY) Nov 04 12:06:51 PM: Invoking solver MOSEK  to obtain a solution.


(CVXPY) Nov 04 12:06:52 PM: Problem
(CVXPY) Nov 04 12:06:52 PM:   Name                   :
(CVXPY) Nov 04 12:06:52 PM:   Objective sense        : maximize
(CVXPY) Nov 04 12:06:52 PM:   Type                   : CONIC (conic optimization proble
(CVXPY) Nov 04 12:06:52 PM:   Constraints            : 4
(CVXPY) Nov 04 12:06:52 PM:   Affine conic cons.     : 0
(CVXPY) Nov 04 12:06:52 PM:   Disjunctive cons.      : 0
(CVXPY) Nov 04 12:06:52 PM:   Cones                  : 1
(CVXPY) Nov 04 12:06:52 PM:   Scalar variables       : 13
(CVXPY) Nov 04 12:06:52 PM:   Matrix variables       : 0
(CVXPY) Nov 04 12:06:52 PM:   Integer variables      : 0
(CVXPY) Nov 04 12:06:52 PM:
```

```
(CVXPY) Nov 04 12:06:52 PM: Optimizer started.
(CVXPY) Nov 04 12:06:52 PM: Presolve started.
(CVXPY) Nov 04 12:06:52 PM: Linear dependency checker started.
(CVXPY) Nov 04 12:06:52 PM: Linear dependency checker terminated.
(CVXPY) Nov 04 12:06:52 PM: Eliminator started.
(CVXPY) Nov 04 12:06:52 PM: Freed constraints in eliminator : 1
(CVXPY) Nov 04 12:06:52 PM: Eliminator terminated.
(CVXPY) Nov 04 12:06:52 PM: Eliminator started.
(CVXPY) Nov 04 12:06:52 PM: Freed constraints in eliminator : 0
(CVXPY) Nov 04 12:06:52 PM: Eliminator terminated.
(CVXPY) Nov 04 12:06:52 PM: Eliminator - tries                  : 2                 time
(CVXPY) Nov 04 12:06:52 PM: Lin. dep.  - tries                  : 1                 time
(CVXPY) Nov 04 12:06:52 PM: Lin. dep.  - primal attempts        : 1                 successes
(CVXPY) Nov 04 12:06:52 PM: Lin. dep.  - dual attempts          : 0                 successes
(CVXPY) Nov 04 12:06:52 PM: Lin. dep.  - primal deps.           : 0                 dual deps.
(CVXPY) Nov 04 12:06:52 PM: Presolve terminated. Time: 0.00
(CVXPY) Nov 04 12:06:52 PM: Optimizer  - threads                : 12
(CVXPY) Nov 04 12:06:52 PM: Optimizer  - solved problem         : the primal
(CVXPY) Nov 04 12:06:52 PM: Optimizer  - Constraints            : 3
(CVXPY) Nov 04 12:06:52 PM: Optimizer  - Cones                  : 1
(CVXPY) Nov 04 12:06:52 PM: Optimizer  - Scalar variables       : 10                conic
(CVXPY) Nov 04 12:06:52 PM: Optimizer  - Semi-definite variables: 0                 scalarized
(CVXPY) Nov 04 12:06:52 PM: Factor     - setup time             : 0.00
(CVXPY) Nov 04 12:06:52 PM: Factor     - dense det. time        : 0.00              GP order time
(CVXPY) Nov 04 12:06:52 PM: Factor     - nonzeros before factor : 6                 after factor
(CVXPY) Nov 04 12:06:52 PM: Factor     - dense dim.             : 0                 flops
(CVXPY) Nov 04 12:06:52 PM: ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ              DOBJ
(CVXPY) Nov 04 12:06:52 PM: 0   1.3e+00  3.0e+00  2.0e+00  0.00e+00   -2.000000000e+00  -1.00000
(CVXPY) Nov 04 12:06:52 PM: 1   2.2e-01  5.3e-01  2.4e-01  1.05e-01   -1.810182709e+00  -1.72313
(CVXPY) Nov 04 12:06:52 PM: 2   3.7e-02  8.9e-02  1.6e-02  9.48e-01   -2.154687486e+00  -2.13541
(CVXPY) Nov 04 12:06:52 PM: 3   1.0e-02  2.5e-02  2.2e-03  1.01e+00   -2.236775806e+00  -2.23083
(CVXPY) Nov 04 12:06:52 PM: 4   2.8e-03  6.7e-03  3.2e-04  1.02e+00   -2.250313751e+00  -2.24864
(CVXPY) Nov 04 12:06:52 PM: 5   7.7e-04  1.8e-03  4.5e-05  1.00e+00   -2.256166317e+00  -2.25570
(CVXPY) Nov 04 12:06:52 PM: 6   2.1e-05  5.1e-05  2.1e-07  1.00e+00   -2.256863865e+00  -2.25685
(CVXPY) Nov 04 12:06:52 PM: 7   1.0e-07  2.4e-07  6.8e-11  1.00e+00   -2.256896295e+00  -2.25689
(CVXPY) Nov 04 12:06:52 PM: 8   8.5e-09  2.0e-08  1.6e-12  1.00e+00   -2.256896526e+00  -2.25689
(CVXPY) Nov 04 12:06:52 PM: Optimizer terminated. Time: 0.00
(CVXPY) Nov 04 12:06:52 PM:
(CVXPY) Nov 04 12:06:52 PM:
(CVXPY) Nov 04 12:06:52 PM: Interior-point solution summary
(CVXPY) Nov 04 12:06:52 PM:   Problem status  : PRIMAL_AND_DUAL_FEASIBLE
(CVXPY) Nov 04 12:06:52 PM:   Solution status : OPTIMAL
(CVXPY) Nov 04 12:06:52 PM:   Primal.  obj: -2.2568965259e+00   nrm: 3e+00    Viol.  con: 2e-08
(CVXPY) Nov 04 12:06:52 PM:   Dual.    obj: -2.2568965208e+00   nrm: 1e+00    Viol.  con: 0e+00
-------------------------------------------------------------------------------
                                Summary
```

```
-----------------------------------------------------------------------------
(CVXPY) Nov 04 12:06:52 PM: Problem status: optimal
(CVXPY) Nov 04 12:06:52 PM: Optimal value: -2.257e+00
(CVXPY) Nov 04 12:06:52 PM: Compilation took 3.196e-03 seconds
(CVXPY) Nov 04 12:06:52 PM: Solver (including time spent in interface) took 1.100e+00 s
Status: optimal
Optimal value: -2.256896525744356
x* = [0.       0.068914 0.931086]
Max inequality residual (<=0): -1.938693250380652e-08
Equality residual ( 0): 0.0
```

4.3.4 Sequential Quadratic Programming

We have seen that quadratic programs (QPs) gracefully handle *constrained LQR–style* trajectory optimization: convex quadratic costs, linear dynamics, and affine path/terminal constraints. With time-stacked sparsity, these problems are solved efficiently (e.g., via interior-point methods), making QP a strong tool for that regime.

The natural next step is the general trajectory optimization problem in (4.31), which features nonlinear dynamics, *nonconvex* objectives, and *nonlinear* constraints. To tackle this, we turn to *Sequential Quadratic Programming (SQP)*—a Newton-like framework that *iteratively* linearizes the dynamics/constraints and quadratizes the objective to form a sequence of QP subproblems. Each QP is solved to produce a step and updated multipliers; with globalization and appropriate Hessian modeling, the sequence converges to a *locally optimal* solution (a KKT point) of the original nonlinear TO problem. In short: QP handles the convex linearized case; SQP extends that logic to the full nonlinear setting by repeatedly building and solving the right QP at the current iterate. For an in-depth presentation of SQP for nonlinear programming, we refer to Chapter 18 of (Nocedal and Wright, 1999).

4.3.4.1 Problem Statement

We will consider the following general nonlinear program (NLP)

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \quad (c : \mathbb{R}^n \to \mathbb{R}^m), \\ & d(x) \le 0 \quad (d : \mathbb{R}^n \to \mathbb{R}^p), \end{aligned}$$

with objective function $f(x)$, equality constraints $c(x) = 0$, and inequality constraints $d(x) \le 0$. To obtain the NLP formulation (4.3.4.1) from the TO template (4.31), one needs to define the set constraints $x_t \in \mathcal{X}_t$ and $u_t \in \mathcal{U}_t$ as general equality and inequality constraints. The decision variable x contains the entire sequence of states and actions.

> While in many cases the functions f, c, d are defined analytically, technically speaking, we only need zero-order and first-order oracles of these functions to implement numerical algorithms. That is, given a point x, we need to evaluate $f(x), c(x), d(x)$ and their first-order derivatives.

With dual variables $\lambda \in \mathbb{R}^m$ and $\mu \in \mathbb{R}^p$, define the Lagrangian of (4.3.4.1) as

$$\mathcal{L}(x, \lambda, \mu) \;=\; f(x) + \lambda^\top c(x) + \mu^\top d(x), \qquad \mu \geq 0,$$

and Jacobians $J_c(x) \coloneqq \nabla c(x) \in \mathbb{R}^{m \times n}$, $J_d(x) \coloneqq \nabla d(x) \in \mathbb{R}^{p \times n}$.

4.3.4.2 High-Level Intuition

At a current iterate x_k, we (1) Linearize the constraints and (2) quadratize the Lagrangian to build a local QP. Solving this QP yields a primal step p_k and updated multipliers $(\lambda_{k+1}, \mu_{k+1})$ (from QP duals). A line-search (or trust-region) with a merit or filter globalization ensures convergence from remote starts.

Key ingredients:

- A Hessian (or quasi-Newton) approximation $H_k \approx \nabla^2_{xx} \mathcal{L}(x_k, \lambda_k, \mu_k)$.
- A QP subproblem capturing first-order feasibility and second-order optimality locally.
- A globalization mechanism (ℓ_1 merit or filter) + optional second-order correction (SOC) to mitigate linearization error in active inequalities.

4.3.4.3 The SQP QP Subproblem

Given the current iterate (x_k, λ_k, μ_k), define

$$g_k \coloneqq \nabla f(x_k), \qquad c_k \coloneqq c(x_k), \qquad d_k \coloneqq d(x_k), \qquad A_k \coloneqq J_c(x_k), \qquad G_k \coloneqq J_d(x_k).$$

Let H_k be a symmetric approximation to $\nabla^2_{xx} \mathcal{L}(x_k, \lambda_k, \mu_k)$ (see Section 4.3.4.4).

The QP subproblem in this step is

$$\begin{aligned} \min_{p \in \mathbb{R}^n} \quad & \frac{1}{2} p^\top H_k \, p + g_k^\top p \\ \text{s.t.} \quad & A_k \, p + c_k = 0, \\ & G_k \, p + d_k \leq 0. \end{aligned}$$

This defines a local quadratic model of (4.3.4.1): constraints are linearized; the objective is the second-order Taylor model of the Lagrangian (up to a constant).

Solving the QP subproblem (4.3.4.3) returns:

- **Primal step** p_k.
- **Dual estimates** $\lambda_{k+1}^{\mathrm{QP}}, \mu_{k+1}^{\mathrm{QP}}$ (the QP multipliers), which we use as new multipliers.

4.3.4.4 Hessian Approximation

The most natural choice for H_k in the QP subproblem (4.3.4.3) is the exact Hessian:

$$H_k = \nabla_{xx}^2 \mathcal{L}(x_k, \lambda_k, \mu_k) = \nabla_{xx}^2 f(x_k) + \sum_i \lambda_{k,i} \nabla_{xx}^2 c_i(x_k) + \sum_i \mu_{k,i} \nabla_{xx}^2 d_i(x_k).$$

However, two potential issues with the exact Hessian are (i) it can be costly to build and store the analytic Hessians $\nabla_{xx}^2 f, \nabla_{xx}^2 c_i, \nabla_{xx}^2 d_i$; (ii) the exact Hessian H_k may not be positive semidefinite, which may lead to failure of convexity in the QP subproblem.

A cornerstone result in numerical optimization, due to Broyden–Fletcher–Goldfarb–Shanno (BFGS), is to build an approximate Hessian from first-order derivatives. In particular, given two consecutive primal iterates x_{k+1}, x_k (and their associated dual variables) and first-order gradients of the Lagrangian $\nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}, \mu_{k+1})$, $\nabla_x \mathcal{L}(x_k, \lambda_k, \mu_k)$, define

$$s_k := x_{k+1} - x_k, \qquad y_k := \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}, \mu_{k+1}) - \nabla_x \mathcal{L}(x_k, \lambda_k, \mu_k).$$

The BFGS quasi-Newton method updates H_{k+1} from H_k as follows

$$H_{k+1} = H_k - \frac{H_k s_k s_k^\top H_k}{s_k^\top H_k s_k} + \frac{y_k y_k^\top}{s_k^\top y_k}.$$

One can show that if $y_k^\top s_k > 0$ holds, the BFGS Hessian approximation is always positive definite (provided $H_0 \succ 0$). Therefore, the BFGS Hessian approximation ensures the QP subproblem is convex. If the curvature condition $y_k^\top s_k > 0$ fails to hold, one can resort to damped BFGS, see (Nocedal and Wright, 1999).

> There is a broad family of quasi-Newton methods with BFGS being one of the most popular instances. For example, the symmetric rank-one (SR1) method is another popular quasi-Newton variant. In addition, the "limited memory" version of quasi-Newton methods (e.g., limited memory BFGS (Liu and Nocedal, 1989)) can further reduce the price of Hessian approximation by only looking at the history of a small amount of gradients.

4.3.4.5 Globalization: Merit or Filter

To accept a step, we assess optimality improvement + feasibility improvement:

- ℓ_1 **merit** (exact-penalty style): define the merit function

$$\phi_\rho(x) = f(x) + \rho(\|c(x)\|_1 + \|d^+(x)\|_1), \quad d^+ := \max(d, 0),$$

with penalty ρ large enough (ρ can also be adaptive with respect to iterations). Use backtracking line search on $\phi_\rho(x_k + \alpha p_k)$ to ensure decrease of the merit function.

- **Filter** method: Maintain a set of pairs (feasibility, objective). Accept steps that reduce **either** feasibility or objective sufficiently without worsening the other beyond the filter. Often paired with Second-order correction, see more details in (Nocedal and Wright, 1999).

The following pseudocode implements a basic line-search SQP with quasi-Newton Hessian approximation.

Algorithm: Line-Search SQP

Inputs: x_0, multipliers $(\lambda_0, \mu_0 \geq 0)$, initial Hessian $H_0 \succ 0$ (e.g., γI), globalization parameters.

For $k = 0, 1, 2, ...$

1. **Linearize & build QP**: form g_k, A_k, G_k, c_k, d_k and H_k, then solve the QP (4.3.4.3) to get p_k and QP multipliers $(\widehat{\lambda}_{k+1}, \widehat{\mu}_{k+1} \geq 0)$.
2. **Globalization**: Choose step size $\alpha_k \in (0, 1]$ by backtracking on the ℓ_1-merit.
3. **Update**:

$$x_{k+1} = x_k + \alpha_k p_k, \qquad \lambda_{k+1} = \widehat{\lambda}_{k+1}, \qquad \mu_{k+1} = \Pi_{\geq 0}(\widehat{\mu}_{k+1}).$$

4. **Hessian (quasi-Newton) update**: define

$$s_k := x_{k+1} - x_k, \qquad y_k := \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}, \mu_{k+1}) - \nabla_x \mathcal{L}(x_k, \lambda_k, \mu_k).$$

update

$$H_{k+1} = H_k - \frac{H_k s_k s_k^\top H_k}{s_k^\top H_k s_k} + \frac{y_k y_k^\top}{s_k^\top y_k}.$$

5. **Stopping:** if KKT residuals (stationarity, primal feasibility, complementarity) are below tolerance, **terminate**.

Notes.

- **Trust-region SQP.** An alternative globalization: add $\|p\| \leq \Delta$ or a quadratic trust region, and update Δ by comparing predicted vs. actual reduction in a composite model.

- **Software.** The `scipy` package in Python implements SLSQP, which is basically the line-search SQP we presented above. A more advanced version of SQP is provided by the SNOPT commercial software. The CRISP software provides a C++ implementation of an SQP algorithm. Additionally, Matlab's `fmincon` provides an implementation of SQP.

Example 4.4 (Trajectory Optimization with SQP)**.** We formulate a trajectory optimization (TO) problem for a unicycle robot that must travel from a start pose A to a goal pose B while avoiding circular (ball-shaped) obstacles.

System Model. We use the standard unicycle (Dubins-like) kinematics in continuous time:

$$\dot{x}(t) = \begin{bmatrix} \dot{p}_x \\ \dot{p}_y \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v(t)\cos\theta(t) \\ v(t)\sin\theta(t) \\ \omega(t) \end{bmatrix}, \qquad x = [p_x, p_y, \theta]^\top, \quad u = [v, \omega]^\top.$$

We discretize on a uniform grid $t_k = kh,\ k = 0, \dots, N$ with step $h > 0$ by forward Euler:

$$x_{k+1} \;=\; f_h(x_k, u_k) \;:=\; \begin{bmatrix} p_{x,k} + h\, v_k \cos\theta_k \\ p_{y,k} + h\, v_k \sin\theta_k \\ \theta_k + h\, \omega_k \end{bmatrix}.$$

Decision Variables. We optimize over the state–control sequence

$$\{x_k\}_{k=0}^{N}, \quad \{u_k\}_{k=0}^{N-1},$$

and collect them into a single vector

$$z = \left[x_0^\top,\ u_0^\top,\ x_1^\top,\ u_1^\top,\ \dots,\ x_{N-1}^\top,\ u_{N-1}^\top,\ x_N^\top\right]^\top \;\in\; \mathbb{R}^{(3+2)N+3}.$$

Constraints. We impose the following constraints.

(i) Initial condition.

$$x_0 = A \in \mathbb{R}^3.$$

(ii) System dynamics (equality constraints). For $k = 0, \dots, N-1$,

$$x_{k+1} - f_h(x_k, u_k) = 0.$$

(iii) Obstacle avoidance (inequalities). Let the set of circular obstacles be $\mathcal{O} = \{(c_j, r_j)\}_{j=1}^{n_{\text{obs}}}$ with centers $c_j = [c_{x,j}, c_{y,j}]^\top$ and radii $r_j > 0$. We require the robot's position to stay outside each inflated disk of radius $r_j + \delta$ (safety margin $\delta \geq 0$) at every knot:

$$\underbrace{(p_{x,k} - c_{x,j})^2 + (p_{y,k} - c_{y,j})^2}_{\text{dist}^2(x_k, \text{center}_j)} \;\geq\; (r_j + \delta)^2, \qquad \forall k = 0, \dots, N,\ \forall j.$$

In "$c(x) \le 0$" form (e.g., for `fmincon`):

$$c_{j,k}(x_k) \;:=\; (r_j+\delta)^2 - \big((p_{x,k}-c_{x,j})^2 + (p_{y,k}-c_{y,j})^2\big) \;\le\; 0.$$

(iv) Simple bounds. Box limits on controls (and possibly states):

$$v_{\min} \le v_k \le v_{\max}, \qquad \omega_{\min} \le \omega_k \le \omega_{\max}, \qquad k = 0, \dots, N-1.$$

Objective function. We use a smooth quadratic objective combining (a) terminal goal tracking, (b) control effort, and (c) control smoothness (temporal regularization):

- **Terminal goal tracking** to a desired pose $B = [p_x^\star, p_y^\star, \theta^\star]^\top$:

$$J_{\text{goal}} \;=\; w_{\text{pos}} \big\| x_N^{\text{pos}} - B^{\text{pos}} \big\|_2^2 \;+\; w_\theta\,(\theta_N - \theta^\star)^2, \quad x_N^{\text{pos}} = [p_{x,N}, p_{y,N}]^\top.$$

- **Control effort**:

$$J_u \;=\; \sum_{k=0}^{N-1} w_u\,\|u_k\|_2^2 \;=\; \sum_{k=0}^{N-1} w_u\,(v_k^2 + \omega_k^2).$$

- **Control smoothness** (discrete total-variation-like quadratic):

$$J_{\Delta u} \;=\; \sum_{k=0}^{N-2} w_{\Delta u}\,\|u_{k+1} - u_k\|_2^2.$$

The complete cost is

$$J(x_{0:N}, u_{0:N-1}) = \frac{1}{2}\Big(J_{\text{goal}} + J_u + J_{\Delta u}\Big),$$

where the outer factor $\frac{1}{2}$ is conventional in quadratic objectives.

Complete optimization problem. Given $A, B, \{(c_j, r_j)\}, \delta, h, N$, choose $\{x_k\}_{k=0}^{N}, \{u_k\}_{k=0}^{N-1}$ to

$$\min_{\{x_k,u_k\}} \quad \frac{1}{2}\Big(w_{\text{pos}}\|x_N^{\text{pos}} - B^{\text{pos}}\|_2^2 + w_\theta(\theta_N-\theta^\star)^2 + \sum_{k=0}^{N-1} w_u\|u_k\|_2^2 + \sum_{k=0}^{N-2} w_{\Delta u}\|u_{k+1}-u_k\|_2^2\Big)$$

$$\begin{aligned}
\text{s.t.}\quad & x_0 = A,\\
& x_{k+1} = f_h(x_k, u_k) \quad \text{for } k = 0, \dots, N-1,\\
& (r_j+\delta)^2 - \big((p_{x,k}-c_{x,j})^2 + (p_{y,k}-c_{y,j})^2\big) \le 0, \quad \forall j,\ \forall k,\\
& v_{\min} \le v_k \le v_{\max}, \quad \omega_{\min} \le \omega_k \le \omega_{\max}, \quad k = 0, \dots, N-1.
\end{aligned}$$

This is a smooth, sparse nonlinear program (NLP).

Experimental setup.

- Horizon $N = 60$, step $h = 0.1$ s.
- Start $A = [0, 0, 0]^\top$, goal $B = [6, 5, 0]^\top$.
- Obstacles $\{(c_j, r_j)\}$ with a margin $\delta = 0.25$ m.
- Control bounds $v \in [-1.5, 1.5]$ m/s, $\omega \in [-2, 2]$ rad/s.
- Weights $w_{\text{pos}} = 400,\ w_\theta = 20,\ w_u = 0.05,\ w_{\Delta u} = 0.2$.
- **Initialization:** straight-line interpolation of positions from A to B, heading toward the line, constant v, $\omega = 0$.

Two obstacles (success). Fig. 4.6 shows the results for trajectory optimization with two obstacles. As we can see, the SQP algorithm generates a smooth trajectory that avoids the two circular obstacles, despite the fact that the initial guess crosses both obstacles.

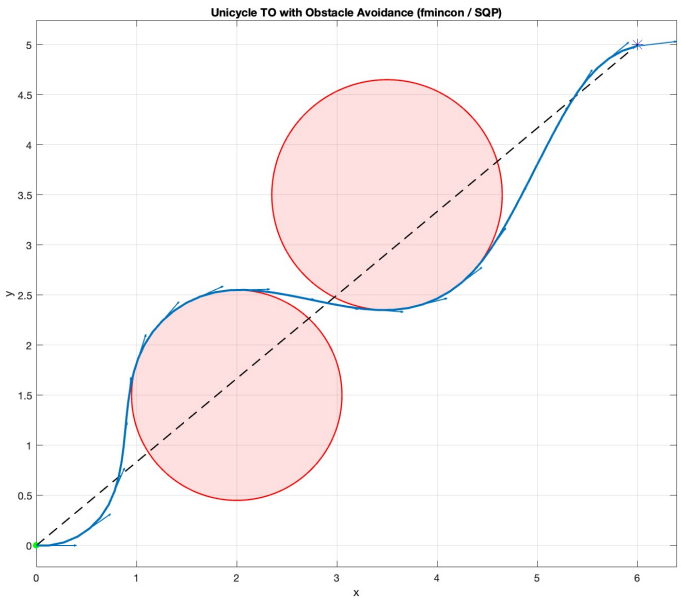

Figure 4.6: Trajectory optimization for unicyle using SQP (two obstacles). Dotted line: initial guess; solid line: optimized trajectory.

Three obstacles (failure). However, when we add a third obstacle, Fig. 4.7 shows that the SQP algorithm converges to an infeasible solution that collides with the obstacles.

You can play with the Matlab code here.

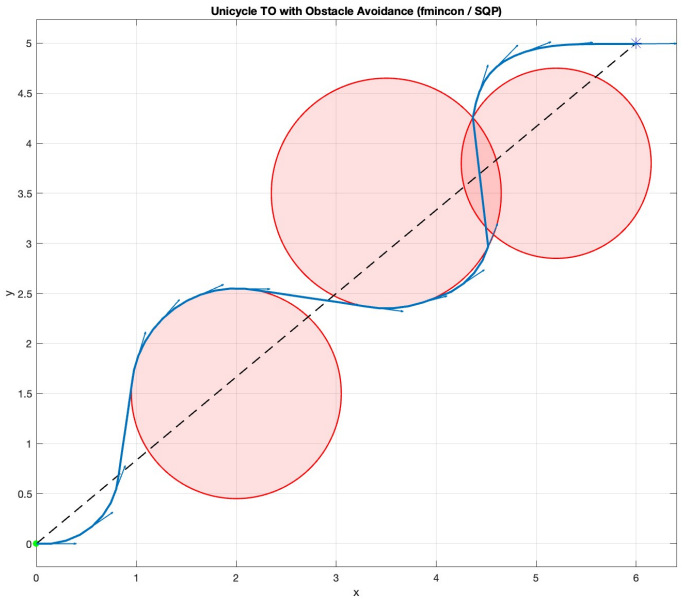

Figure 4.7: Trajectory optimization for unicyle using SQP (three obstacles). Dotted line: initial guess; solid line: optimized trajectory.

The example above highlights both the strengths and the limitations of solving TO with numerical NLP methods such as SQP. Because the TO problem is generally *nonconvex*, a *local* method's outcome depends strongly on the *initialization*. With a *good* initial guess, local NLP solvers often converge quickly to a high-quality solution (e.g., Fig. 4.6). With a *poor* initialization—especially when the landscape has many local minima—the solver may settle in a suboptimal basin or even fail to find a feasible trajectory (e.g., Fig. 4.7).

Global optimization methods can avoid initialization sensitivity and provide **global optimality guarantees**. These techniques can be substantially more expensive and require additional structure or reformulation, but when applicable they yield powerful *initialization-free* solutions; see, e.g., (Kang et al., 2024) and references therein.

4.3.5 Interior-Point Methods

We have seen that Primal–Dual Interior-Point Methods (PD-IPM) efficiently solve convex QPs (Section 4.3.3). The key idea was to write the KKT optimality

conditions as a system of nonlinear equations and apply Newton's method. We now extend this idea to the general NLP in (4.3.4.1):

$$\min_{x\in\mathbb{R}^n} f(x) \quad \text{s.t.} \quad c(x)=0,\;\; d(x)\le 0,$$

where $f:\mathbb{R}^n\to\mathbb{R}, c:\mathbb{R}^n\to\mathbb{R}^m$ and $d:\mathbb{R}^n\to\mathbb{R}^p$ are smooth.

4.3.5.1 Lagrangian and KKT

Introduce slack variables $s\in\mathbb{R}^p$ to convert inequalities to equalities:

$$c(x)=0, \qquad d(x)+s=0, \qquad s\ge 0.$$

With equality multipliers $y\in\mathbb{R}^m, \nu\in\mathbb{R}^p$ and inequality multipliers $\mu\in\mathbb{R}^p$ (with $\lambda\ge 0$), the **Lagrangian** of the slack-form problem is

$$\mathcal{L}(x,s,y,\nu,\lambda) \;=\; f(x)+y^\top c(x)+\nu^\top(d(x)+s)-\lambda^\top s.$$

The KKT optimality conditions are (after eliminating ν)

$$\begin{aligned}
\text{stationarity:} &\quad \nabla f(x)+J_c(x)^\top y+J_d(x)^\top\lambda=0,\\
\text{primal feasibility:} &\quad c(x)=0,\;\; d(x)+s=0,\;\; s\ge 0,\\
\text{dual feasibility:} &\quad \lambda\ge 0,\\
\text{complementarity:} &\quad s_i\,\lambda_i=0, \quad i=1,\dots,p,
\end{aligned}$$

where $J_c=\nabla c(x)\in\mathbb{R}^{m\times n}$, $J_d=\nabla d(x)\in\mathbb{R}^{p\times n}$.

4.3.5.2 Two Equivalent Views

There are two equivalent ways to derive primal–dual IPMs.

Homotopy / Perturbed KKT (primal–dual view). Replace the complementarity slackness condition in the KKT system (4.3.5.1) by a *perturbed* relation that defines the *central path*:

$$S\,\lambda \;=\; \mu\,\mathbf{1}, \qquad S:=\operatorname{diag}(s),\;\; \mu>0.$$

Driving the parameter $\mu\downarrow 0$ yields iterates approaching a KKT point.

Barrier (log-barrier view). Solve the barrier subproblem

$$\min_{x,s>0}\; f(x)-\mu\sum_{i=1}^{p}\log s_i \quad \text{s.t.} \quad c(x)=0,\;\; d(x)+s=0,$$

then reduce μ. The KKT conditions of (4.3.5.2) imply $S\lambda=\mu\mathbf{1}$, hence the barrier and homotopy views are equivalent (different perspectives on the same central path).

4.3.5.3 Primal–Dual Residuals and Newton System

Define residuals at (x, s, y, λ):

$$\begin{aligned} r_{\text{dual}} &= \nabla f(x) + J_c^\top y + J_d^\top \lambda, \\ r_{\text{pe}} &= c(x), \\ r_{\text{pi}} &= d(x) + s, \\ r_{\text{cent}} &= S\lambda - \mu\,\mathbf{1}. \end{aligned}$$

Let

$$H \;:=\; \nabla^2_{xx}\mathcal{L}(x, s, y, \lambda) = \nabla^2 f(x) + \sum_{i=1}^{m} y_i \nabla^2 c_i(x) + \sum_{j=1}^{p} \lambda_j \nabla^2 d_j(x)$$

be the exact Hessian of the Lagrangian with respect to x.

A primal–dual Newton step $(\Delta x, \Delta s, \Delta y, \Delta\lambda)$ solves

$$\begin{aligned} H\,\Delta x + J_c^\top \Delta y + J_d^\top \Delta\lambda &= -\,r_{\text{dual}}, \\ J_c\,\Delta x &= -\,r_{\text{pe}}, \\ J_d\,\Delta x + \Delta s &= -\,r_{\text{pi}}, \\ S\,\Delta\lambda + M\,\Delta s &= -\,r_{\text{cent}}, \qquad M := \operatorname{diag}(\lambda). \end{aligned}$$

Eliminating $\Delta s = -r_{\text{pi}} - J_d \Delta x$ gives

$$\Delta\lambda = S^{-1}\big(-r_{\text{cent}} + M\,r_{\text{pi}} + M\,J_d\,\Delta x\big)\,.$$

Substitute into the first line to obtain the reduced symmetric system in $(\Delta x, \Delta y)$:

$$\begin{bmatrix} H + J_d^\top D\, J_d & J_c^\top \\ J_c & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} r_{\text{dual}} + J_d^\top S^{-1}(-r_{\text{cent}} + M r_{\text{pi}}) \\ r_{\text{pe}} \end{bmatrix}$$

with $D := S^{-1}M$. Then recover $\Delta\lambda$ via (4.3.5.3) and $\Delta s = -r_{\text{pi}} - J_d\Delta x$.

4.3.5.4 Line-search IPM

Step lengths (fraction-to-the-boundary). Choose positive step sizes that keep strict interiority:

$$\alpha_{\text{pri}} = \min\Big(1,\ \eta \min_{\Delta s_i < 0} \frac{-s_i}{\Delta s_i}\Big), \quad \alpha_{\text{du}} = \min\Big(1,\ \eta \min_{\Delta \lambda_i < 0} \frac{-\lambda_i}{\Delta \lambda_i}\Big), \quad \eta \in (0,1).$$

Merit (or filter) globalization. Use a barrier merit for backtracking,

$$\Phi_\mu(x, s) \;=\; f(x) - \mu \sum_i \log s_i + \frac{\rho}{2}\,\|c(x)\|_2^2 + \frac{\rho}{2}\,\|d(x) + s\|_2^2,$$

or a filter that accepts steps reducing either infeasibility or the barrier objective.

Mehrotra predictor–corrector (recommended).

1. **Predictor (affine) step:** solve (4.3.5.3) with $\mu = 0$ to get $(\Delta x^{\text{aff}}, \Delta s^{\text{aff}}, \Delta y^{\text{aff}}, \Delta \lambda^{\text{aff}})$, and affine step sizes $\alpha_{\text{pri}}^{\text{aff}}, \alpha_{\text{du}}^{\text{aff}}$.
2. **Centering:** set

$$\tau_{\text{aff}} = \frac{(s + \alpha_{\text{pri}}^{\text{aff}} \Delta s^{\text{aff}})^\top (\lambda + \alpha_{\text{du}}^{\text{aff}} \Delta \lambda^{\text{aff}})}{p}, \qquad \sigma = \left(\frac{\tau_{\text{aff}}}{\frac{s^\top \mu}{p}} \right)^3,$$

   and replace the complementarity RHS by $\mu = \sigma \, \frac{s^\top \lambda}{p}$.
3. **Corrector:** resolve (4.3.5.3) with the corrected central residual

$$r_{\text{cent}}^{\text{corr}} = S\lambda - \mu \, \mathbf{1} - \Delta S^{\text{aff}} \Delta \lambda^{\text{aff}} \mathbf{1}.$$

4. **Line search & update:** use (4.3.5.4) and backtrack on Φ_μ.
5. **Stopping:** terminate inner loop when $\|r_{\text{dual}}\|_\infty, \|r_{\text{pe}}\|_\infty, \|r_{\text{pi}}\|_\infty$ and the average complementarity $\frac{s^\top \mu}{p}$ are below tolerance; then reduce μ and repeat.

Hessian Modeling. Use $H = \nabla^2_{xx} \mathcal{L}$, or a damped (L-)BFGS / Gauss–Newton model as in Section 4.3.4.4.

4.3.5.5 Trust-Region IPM

The trust-region (TR) IPM solves the barrier subproblem (4.3.5.2) *inexactly* within a TR globalization that is scaled to the slacks to avoid steps that approach the boundary too aggressively.

The TR-IPM subproblem is the local quadratic model of the barrier problem together with a metric that respects distance to the boundary.

1. **Quadratic model of the barrier objective.** Start from the barrier problem

$$\min_{x, s>0} \; f(x) - \mu \sum_i \log s_i \quad \text{s.t.} \quad c(x) = 0, \; d(x) + s = 0.$$

   A second-order Taylor model at (x, s) gives (constants omitted)

$$m(p_x, p_s) \;\approx\; \nabla f^\top p_x + \frac{1}{2} p_x^\top H \, p_x \;-\; \mu \, \mathbf{1}^\top S^{-1} p_s \;+\; \frac{1}{2} \, \mu \, \|S^{-1} p_s\|_2^2,$$

where $H = \nabla^2_{xx}\mathcal{L}$ and $S = \mathrm{diag}(s)$. The linear term $-\mu\,\mathbf{1}^\top S^{-1}p_s$ is exactly the gradient of the log-barrier in the slack coordinates. The quadratic curvature in p_s is μS^{-2}.

Many implementations move (part of) this curvature into the TR metric, replacing the explicit $+\frac{1}{2}\,\mu\|S^{-1}p_s\|^2$ by the *scaled trust region* below. This avoids double-counting and makes the subproblem simpler while keeping the right geometry.

2. **Linearized constraints for a consistent local step.**

$$J_c\,p_x + c(x) = 0, \qquad J_d\,p_x + p_s + d(x) = 0.$$

These are the first-order feasibility conditions of the barrier constraints.

3. **Scaled trust region $\|(p_x,\ S^{-1}p_s)\| \le \Delta$.** The scaling by S^{-1} measures p_s relative to the current distance to the boundary. If a slack s_i is tiny, even a small absolute change $p_{s,i}$ is risky; the scaled norm automatically shrinks the allowable step in that direction. This yields:

   - **Boundary awareness:** steps cannot run into $s_i \le 0$ unless the trust region is (incorrectly) large.
   - **Scale invariance:** the step is insensitive to units or simple rescalings of the inequalities.
   - **Numerical stability:** the local subproblem remains well conditioned near the boundary.

4. **Fraction-to-the-boundary safeguard $p_s \ge -\tau s$.** This is a simple interiority constraint ensuring $s + \alpha p_s > 0$ for admissible step sizes.

Putting these choices together yields the subproblem

$$\min_{p_x,p_s}\ \nabla f^\top p_x + \frac{1}{2}p_x^\top H p_x - \mu\,\mathbf{1}^\top S^{-1}p_s \quad \text{s.t.} \quad \begin{cases} J_c p_x + c(x) = 0, \\ J_d p_x + p_s + d(x) = 0, \\ \|(p_x, S^{-1}p_s)\| \le \Delta,\ p_s \ge -\tau s, \end{cases}$$

which is exactly a trust-region SQP step on the barrier problem in the barrier metric.

The outer loop adapts the trust-region radius Δ and the barrier parameter μ as follows.

- **Updating Δ (model–reality agreement).** After computing a trial step $p = (p_x, p_s)$ from the scaled TR subproblem (4.3.5.5), compare the

predicted reduction from the quadratic model to the *actual* reduction in the barrier merit:

$$\rho \;=\; \frac{\text{ared}}{\text{pred}} \;=\; \frac{\Phi_\mu(x,s) - \Phi_\mu(x+p_x,\, s+p_s)}{\text{model}(0) - \text{model}(p)}.$$

With thresholds $0 < \eta_1 < \eta_2 < 1$ and factors $\gamma_{\text{dec}} \in (0,1)$, $\gamma_{\text{inc}} > 1$:

- If $\rho \geq \eta_2$: accept the step and enlarge the radius, $\Delta \leftarrow \min(\gamma_{\text{inc}}\Delta, \Delta_{\max})$.
- If $\eta_1 \leq \rho < \eta_2$: accept and keep Δ.
- If $\rho < \eta_1$: reject the step and shrink the radius, $\Delta \leftarrow \gamma_{\text{dec}}\Delta$, then resolve the TR subproblem.

- **Updating μ (centrality progress).** Decrease the barrier parameter μ only when the current barrier subproblem is solved to an accuracy commensurate with μ. Using KKT residuals and average complementarity

$$\tau \;:=\; \frac{s^\top \lambda}{p},$$

require (for some constants $\kappa_{\text{dual}}, \kappa_{\text{pe}}, \kappa_{\text{pi}}, \kappa_{\text{cent}} > 0$)

$$\|r_{\text{dual}}\|_\infty \leq \kappa_{\text{dual}}\,\mu, \quad \|r_{\text{pe}}\|_\infty \leq \kappa_{\text{pe}}\,\mu, \quad \|r_{\text{pi}}\|_\infty \leq \kappa_{\text{pi}}\,\mu, \quad |\tau - \mu| \leq \kappa_{\text{cent}}\,\mu.$$

When these hold, reduce μ (e.g., $\mu \leftarrow \theta\,\mu$ with $\theta \in (0,1)$). If the tests are not met, hold μ fixed and continue improving the inner TR solve (possibly with an updated Δ).

When to favor TR-IPM. TR globalization is robust for nonconvex H, allows inexact linear solves (e.g., Krylov), and integrates naturally with limited-memory updates and scaling.

Software. Implementing a robust interior-point method that reliably solves *general* nonlinear programs is nontrivial. Fortunately, the open-source solver IPOPT (Wächter and Biegler, 2006) is a mature, widely used option that exploits sparsity and supports exact or quasi-Newton Hessians. In what follows, we formulate a trajectory-optimization problem and solve it with IPOPT to illustrate end-to-end modeling and solver usage.

Example 4.5 (Trajectory Optimization with IPOPT)**.** We solve a unicycle trajectory optimization (TO) problem (same as the one in Example 4.4) from start pose A to goal pose B while avoiding circular obstacles, now using IPOPT via `cyipopt` in Python.

Dynamics. With state $x = [p_x, p_y, \theta]^\top$ and control $u = [v, \omega]^\top$,

$$\dot{x}(t) = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \omega \end{bmatrix}.$$

We discretize at $t_k = kh$ by forward Euler:

$$x_{k+1} = f_h(x_k, u_k) := \begin{bmatrix} p_{x,k} + h\, v_k \cos\theta_k \\ p_{y,k} + h\, v_k \sin\theta_k \\ \theta_k + h\, \omega_k \end{bmatrix}, \quad k = 0, \dots, N-1.$$

Decision vector. We stack the state and control trajectories into a single decision vector

$$z = \left[x_0^\top, u_0^\top, x_1^\top, u_1^\top, \dots, x_{N-1}^\top, u_{N-1}^\top, x_N^\top\right]^\top \in \mathbb{R}^{(3+2)N+3}.$$

Constraints. The state and control trajectories need to satisfy the following constraints:

- Initial condition: $x_0 = A$.
- Dynamics equalities: $x_{k+1} - f_h(x_k, u_k) = 0$ for $k = 0, \dots, N-1$.
- Obstacle inequalities: for each circular obstacle $(c_j = [c_{x,j}, c_{y,j}]^\top, r_j)$ and all $k = 0, \dots, N$,

  $$c_{j,k}(x_k) := (r_j + \delta)^2 - \left((p_{x,k} - c_{x,j})^2 + (p_{y,k} - c_{y,j})^2\right) \le 0,$$

  where $\delta > 0$ is a safety margin.
- Box bounds on controls:

  $$v_{\min} \le v_k \le v_{\max}, \qquad \omega_{\min} \le \omega_k \le \omega_{\max}.$$

In total, there are $m = 3 + 3N$ equality constraints and $p = (N+1)O$ inequality constraints, where O is the number of obstacles.

Objective. A smooth quadratic cost with terminal goal tracking, control effort, and control smoothness:

$$J = \frac{1}{2}\Big(w_{\text{pos}} \|x_N^{\text{pos}} - B^{\text{pos}}\|_2^2 + w_\theta (\theta_N - \theta^\star)^2 + \sum_{k=0}^{N-1} w_u \|u_k\|_2^2 + \sum_{k=0}^{N-2} w_{\Delta u} \|u_{k+1} - u_k\|_2^2 \Big), \quad (4.34)$$

with $x_N^{\text{pos}} = [p_{x,N}, p_{y,N}]^\top$ the terminal position and $B = [B_x, B_y, \theta^\star]^\top$ the terminal pose. w_{pos}, w_θ, w_u, and $w_{\Delta u}$ are positive weights.

Jacobians. We derive the gradient of the objective and the Jacobian of the constraints with respect to the stacked decision vector

$$z \;=\; \left[x_0^\top,\; u_0^\top,\; x_1^\top,\; u_1^\top,\; \dots,\; x_{N-1}^\top,\; u_{N-1}^\top,\; x_N^\top\right]^\top \in \mathbb{R}^{(3+2)N+3},$$

where $x_k = [p_{x,k},\, p_{y,k},\, \theta_k]^\top$ and $u_k = [v_k,\, \omega_k]^\top$.

Objective gradient $\nabla_z J$. Recall the objective function from (4.34). Let $e_{\text{pos}} := x_N^{\text{pos}} - B^{\text{pos}}$ and $e_\theta := \theta_N - \theta^\star$, we have

$$\frac{\partial J}{\partial p_{x,N}} = w_{\text{pos}}\, e_{\text{pos},x}, \qquad \frac{\partial J}{\partial p_{y,N}} = w_{\text{pos}}\, e_{\text{pos},y}, \qquad \frac{\partial J}{\partial \theta_N} = w_\theta\, e_\theta.$$

All other states x_k for $k < N$ do not appear in the objective, so

$$\frac{\partial J}{\partial x_k} = 0, \quad k = 0, \dots, N-1.$$

The control effort term in the objective has gradients w.r.t. controls:

$$J_u \;=\; \frac{1}{2}\sum_{k=0}^{N-1} w_u \|u_k\|^2 \Rightarrow \; \frac{\partial J_u}{\partial u_k} \;=\; w_u\, u_k \quad \text{for } k = 0, \dots, N-1.$$

The control smoothness term in the objective reads:

$$J_{\Delta u} \;=\; \frac{1}{2}\sum_{k=0}^{N-2} w_{\Delta u}\, \|u_{k+1} - u_k\|_2^2.$$

By collecting contributions from the two adjacent differences that contain u_k, we obtain the gradient

$$\frac{\partial J_{\Delta u}}{\partial u_k} = \begin{cases} w_{\Delta u}\,(u_0 - u_1), & k = 0, \\ w_{\Delta u}\,(2u_k - u_{k-1} - u_{k+1}), & k = 1, \dots, N-2, \\ w_{\Delta u}\,(u_{N-1} - u_{N-2}), & k = N-1. \end{cases}$$

Combining the above derivations, the only nonzero blocks of $\nabla_z J$ are:

- the **terminal state block** x_N: entries shown above for $p_{x,N}, p_{y,N}, \theta_N$;
- the **control blocks** u_k: $w_u u_k$ plus the smoothness terms (4.5).

All other entries are zero. Thus $\nabla_z J$ is extremely sparse.

Constraint Jacobian $\nabla_z g(z)$. We stack constraints as

$$g(z) = \begin{bmatrix} g^{\text{init}} \\ g^{\text{dyn}} \\ g^{\text{obs}} \end{bmatrix} \in \mathbb{R}^{3+3N+(N+1)O}$$

in the order:

1. **Initial condition** $g^{\text{init}} = x_0 - A = 0$.

2. **Dynamics** for $k = 0, \dots, N-1$ (forward Euler with step h):

$$\begin{aligned} g^{\text{dyn}}_{x,k} &:= p_{x,k+1} - (p_{x,k} + h\, v_k \cos\theta_k) = 0, \\ g^{\text{dyn}}_{y,k} &:= p_{y,k+1} - (p_{y,k} + h\, v_k \sin\theta_k) = 0, \\ g^{\text{dyn}}_{\theta,k} &:= \theta_{k+1} - (\theta_k + h\, \omega_k) = 0. \end{aligned}$$

3. **Obstacle inequalities** for each obstacle $j = 1, \dots, O$ and each knot $k = 0, \dots, N$:

$$g^{\text{obs}}_{j,k} \;:=\; (r_j + \delta)^2 - \Big((p_{x,k} - c_{x,j})^2 + (p_{y,k} - c_{y,j})^2\Big) \;\le\; 0.$$

Below we list nonzero partial derivatives; all missing entries are 0.

(i) Initial condition $g^{\text{init}} = x_0 - A$

$$\frac{\partial g^{\text{init}}}{\partial x_0} = I_3.$$

(ii) Dynamics rows at time k

- **x-row $g^{\text{dyn}}_{x,k}$:**

$$\frac{\partial g^{\text{dyn}}_{x,k}}{\partial p_{x,k+1}} = 1, \quad \frac{\partial g^{\text{dyn}}_{x,k}}{\partial p_{x,k}} = -1, \quad \frac{\partial g^{\text{dyn}}_{x,k}}{\partial \theta_k} = h\, v_k \sin\theta_k, \quad \frac{\partial g^{\text{dyn}}_{x,k}}{\partial v_k} = -h\cos\theta_k.$$

- **y-row $g^{\text{dyn}}_{y,k}$:**

$$\frac{\partial g^{\text{dyn}}_{y,k}}{\partial p_{y,k+1}} = 1, \quad \frac{\partial g^{\text{dyn}}_{y,k}}{\partial p_{y,k}} = -1, \quad \frac{\partial g^{\text{dyn}}_{y,k}}{\partial \theta_k} = -h\, v_k \cos\theta_k, \quad \frac{\partial g^{\text{dyn}}_{y,k}}{\partial v_k} = -h\sin\theta_k.$$

- **Heading row $g^{\text{dyn}}_{\theta,k}$:**

$$\frac{\partial g^{\text{dyn}}_{\theta,k}}{\partial \theta_{k+1}} = 1, \quad \frac{\partial g^{\text{dyn}}_{\theta,k}}{\partial \theta_k} = -1, \quad \frac{\partial g^{\text{dyn}}_{\theta,k}}{\partial \omega_k} = -h.$$

All other partials in each row are zero. Each dynamics triple only touches the **local block** (x_k, u_k, x_{k+1}), yielding a **banded, block-sparse** Jacobian in time.

(iii) Obstacle row for (j,k)

$$g^{\text{obs}}_{j,k} = (r_j + \delta)^2 - ((p_{x,k} - c_{x,j})^2 + (p_{y,k} - c_{y,j})^2).$$

Nonzeros:

$$\frac{\partial g^{\text{obs}}_{j,k}}{\partial p_{x,k}} = -2\,(p_{x,k} - c_{x,j}), \qquad \frac{\partial g^{\text{obs}}_{j,k}}{\partial p_{y,k}} = -2\,(p_{y,k} - c_{y,j}).$$

This row depends only on the position of x_k, so each obstacle row touches exactly two columns $(p_{x,k}, p_{y,k})$.

Combining the derivations above, we can see the Jacobian $\nabla_z g$ is block-banded in time.

- Initial block at x_0 is identity.
- Each dynamics row touches (x_k, u_k, x_{k+1}) with at most 4 nonzeros in the x- and y-rows and 3 nonzeros in the θ-row.
- Each obstacle row touches only $(p_{x,k}, p_{y,k})$.

Fig. 4.8 illustrates the sparsity structure of the constraint Jacobian.

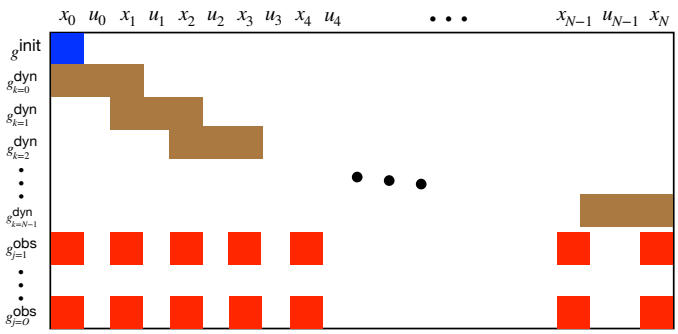

Figure 4.8: Sparsity structure of constraint Jacobian.

The following Python code snippet defines a problem class `UnicycleTO` with definitions of the objective, constraints, objective gradient, and constraints Jacobian. Note that the definition of the constraints Jacobian is highly involved because it defines the Jacobian as a sparse matrix with a predefined sparsity pattern. The solver can leverage this sparsity pattern to be highly efficient.

```
class UnicycleTO:
    def __init__(self, P: Params):
        self.P = P
        # --- Build bounds on variables ---
        lb = -np.inf * np.ones(P.Z_DIM)
        ub =  np.inf * np.ones(P.Z_DIM)
        for k in range(P.N):
```

```
            iu = idx_u(k, P)
            lb[iu.start + 0] = P.v_min
            ub[iu.start + 0] = P.v_max
            lb[iu.start + 1] = P.w_min
            ub[iu.start + 1] = P.w_max
        self.lb = lb
        self.ub = ub

        # --- Build bounds on constraints ---
        cl = np.zeros(P.M)
        cu = np.zeros(P.M)
        # equalities: exactly 0
        cl[:P.n_ceq] = 0.0
        cu[:P.n_ceq] = 0.0
        # inequalities: c <= 0
        cl[P.n_ceq:] = -np.inf
        cu[P.n_ceq:] = 0.0
        self.cl = cl
        self.cu = cu

        # --- Precompute Jacobian sparsity (row, col) ---
        self.jac_rows, self.jac_cols = self._build_jacobian_structure()

    # Objective
    def objective(self, z):
        P = self.P
        X, U = unpack(z, P)

        # Terminal goal tracking
        pos_err = X[-1, 0:2] - P.B[0:2]
        th_err  = X[-1, 2]   - P.B[2]
        J_goal = P.w_goal_pos * np.dot(pos_err, pos_err) + P.w_goal_theta * (th_err**2)

        # Control effort
        J_u = P.w_u * np.sum(U * U)

        # Control smoothness
        dU = U[1:, :] - U[:-1, :]
        J_du = P.w_du * np.sum(dU * dU)

        return 0.5 * (J_goal + J_u + J_du)

    # Gradient of objective
    def gradient(self, z):
```

```
        P = self.P
        X, U = unpack(z, P)
        grad = np.zeros(P.Z_DIM)

        # Terminal contributions (no 0.5 after derivative: cancels 2)
        pos_err = X[-1, 0:2] - P.B[0:2]
        th_err  = X[-1, 2]   - P.B[2]
        gN = np.array([P.w_goal_pos * pos_err[0],
                       P.w_goal_pos * pos_err[1],
                       P.w_goal_theta * th_err])
        grad[idx_x(P.N, P)] += gN

        # Control effort: 0.5 * 2 * w_u * u = w_u * u
        for k in range(P.N):
            iu = idx_u(k, P)
            grad[iu] += P.w_u * U[k, :]

        # Control smoothness: 0.5 * w_du * sum ||u_{k+1}-u_k||^2
        # d/d u_k:   -w_du*(u_{k+1}-u_k)  from (k,k+1)
        # d/d u_{k+1}: +w_du*(u_{k+1}-u_k)
        for k in range(P.N - 1):
            du = U[k + 1, :] - U[k, :]
            grad[idx_u(k, P)]     += -P.w_du * du
            grad[idx_u(k + 1, P)] +=  P.w_du * du

        return grad

    # Constraints g(z)
    def constraints(self, z):
        P = self.P
        X, U = unpack(z, P)
        g = np.zeros(P.M)
        r = 0

        # Initial equality: X0 - A = 0
        g[r:r+3] = X[0, :] - P.A
        r += 3

        # Dynamics equalities
        for k in range(P.N):
            xk = X[k, :]
            uk = U[k, :]
            xnext = f_disc(xk, uk, P.h)
            g[r:r+3] = X[k + 1, :] - xnext
```

```
            r += 3

        # Obstacle inequalities: (r+margin)^2 - ((px-cx)^2 + (py-cy)^2) <= 0
        for k in range(P.N + 1):
            px, py = X[k, 0], X[k, 1]
            for j in range(P.nObs):
                cx, cy, r0 = P.obstacles[j]
                r_eff = r0 + P.safety_margin
                g[r] = (r_eff ** 2) - ((px - cx) ** 2 + (py - cy) ** 2)
                r += 1

        return g

    # Jacobian sparsity
    def jacobianstructure(self):
        return (np.array(self.jac_rows, dtype=int),
                np.array(self.jac_cols, dtype=int))

    # Jacobian values (in the same order as jacobianstructure)
    def jacobian(self, z):
        P = self.P
        X, U = unpack(z, P)
        vals = []
        r = 0

        # Initial eq: d/dX0 is identity (one per row)
        # rows r..r+2 with columns X0(px,py,th)
        for i in range(3):
            vals.append(1.0)
        r += 3

        # Dynamics eqs
        for k in range(P.N):
            xk = X[k, :]
            uk = U[k, :]
            th = xk[2]
            v  = uk[0]

            # Row r: g1 = x_{k+1,px} - [px_k + h v cos(th_k)]
            # d wrt X_{k+1,px}
            vals.append(1.0)
            # d wrt X_k,px
            vals.append(-1.0)
            # d wrt X_k,theta ( + h v sin(th) )
```

```
            vals.append(P.h * v * np.sin(th))
            # d wrt U_k,v ( - h cos(th) )
            vals.append(-P.h * np.cos(th))

            # Row r+1: g2 = x_{k+1,py} - [py_k + h v sin(th_k)]
            vals.append(1.0)                    # d wrt X_{k+1,py}
            vals.append(-1.0)                   # d wrt X_k,py
            vals.append(-P.h * v * np.cos(th))# d wrt X_k,theta
            vals.append(-P.h * np.sin(th))    # d wrt U_k,v

            # Row r+2: g3 = x_{k+1,th} - [th_k + h w_k]
            vals.append(1.0)   # d wrt X_{k+1,theta}
            vals.append(-1.0)  # d wrt X_k,theta
            vals.append(-P.h)  # d wrt U_k,omega

            r += 3

        # Obstacle inequalities
        for k in range(P.N + 1):
            px, py = X[k, 0], X[k, 1]
            for j in range(P.nObs):
                cx, cy, _ = P.obstacles[j]
                # d/d px_k: -2(px - cx)
                vals.append(-2.0 * (px - cx))
                # d/d py_k: -2(py - cy)
                vals.append(-2.0 * (py - cy))
                r += 1

        return np.array(vals, dtype=float)

    # --- Internal: build Jacobian sparsity pattern once ---
    def _build_jacobian_structure(self):
        P = self.P
        rows = []
        cols = []
        r = 0

        # Initial equality: g0..g2 depend on X0(px,py,th) diagonally
        for i in range(3):
            rows.append(r + i)
            cols.append(idx_x(0, P).start + i)
        r += 3

        # Dynamics equalities
```

```
        for k in range(P.N):
            ixk   = idx_x(k, P)
            ixkp1 = idx_x(k + 1, P)
            iuk   = idx_u(k, P)

            # Row r (px)
            rows.extend([r, r, r, r])
            cols.extend([
                ixkp1.start + 0,   # X_{k+1,px}
                ixk.start   + 0,   # X_k,px
                ixk.start   + 2,   # X_k,theta
                iuk.start   + 0    # U_k,v
            ])
            # Row r+1 (py)
            rows.extend([r + 1, r + 1, r + 1, r + 1])
            cols.extend([
                ixkp1.start + 1,   # X_{k+1,py}
                ixk.start   + 1,   # X_k,py
                ixk.start   + 2,   # X_k,theta
                iuk.start   + 0    # U_k,v
            ])
            # Row r+2 (theta)
            rows.extend([r + 2, r + 2, r + 2])
            cols.extend([
                ixkp1.start + 2,   # X_{k+1,theta}
                ixk.start   + 2,   # X_k,theta
                iuk.start   + 1    # U_k,omega
            ])
            r += 3

        # Obstacle inequalities: each depends only on px_k, py_k
        for k in range(P.N + 1):
            ixk = idx_x(k, P)
            for _ in range(P.nObs):
                rows.extend([r, r])
                cols.extend([ixk.start + 0, ixk.start + 1])
                r += 1

        assert r == P.M, "Jacobian structure row count mismatch"
        return rows, cols
```

After defining the problem class, we pass it to the interface of IPOPT using the folowing snippet.

```
# Initial guess
z0 = initial_guess(P)

# Problem + IPOPT
problem = UnicycleTO(P)
nlp = cyipopt.Problem(
    n=P.Z_DIM, m=P.M,
    problem_obj=problem,
    lb=problem.lb, ub=problem.ub,
    cl=problem.cl, cu=problem.cu
)

# Options (tweak as desired)
nlp.add_option("tol", 1e-6)
nlp.add_option("dual_inf_tol", 1e-6)
nlp.add_option("constr_viol_tol", 1e-6)
nlp.add_option("compl_inf_tol", 1e-6)
nlp.add_option("max_iter", 2000)
nlp.add_option("hessian_approximation", "limited-memory")
nlp.add_option("print_level", 5)

z_star, info = nlp.solve(z0)
```

In the case of three obstacles ($\delta = 0$), we obtain the trajectory shown in Fig. 4.9 with an all-zero initialization; the trajectory shown in Fig. 4.10 with a straight-line initialization; and the trajectory shown in Fig. @ref(fig: unicycle-to-ipopt-random) with a random initialization.

You can play with the full code here.

4.4 Model Predictive Control

Trajectory optimization (TO) computes (locally or globally) optimal trajectories that are dynamically feasible with respect to a chosen discrete-time transcription and satisfy system constraints at the discretization grid. As a consequence, the resulting plan is open loop: it neither accounts for disturbances nor provides feedback. A common remedy is LQR trajectory tracking (Section 4.2), which linearizes the dynamics along the nominal trajectory to synthesize a local feedback controller. While effective near the nominal path, this approach can struggle in the presence of obstacles or whenever the reference trajectory itself must change over time.

Model predictive control (MPC) addresses these limitations by turning TO into a receding-horizon feedback policy. At each control step, MPC resolves a finite-

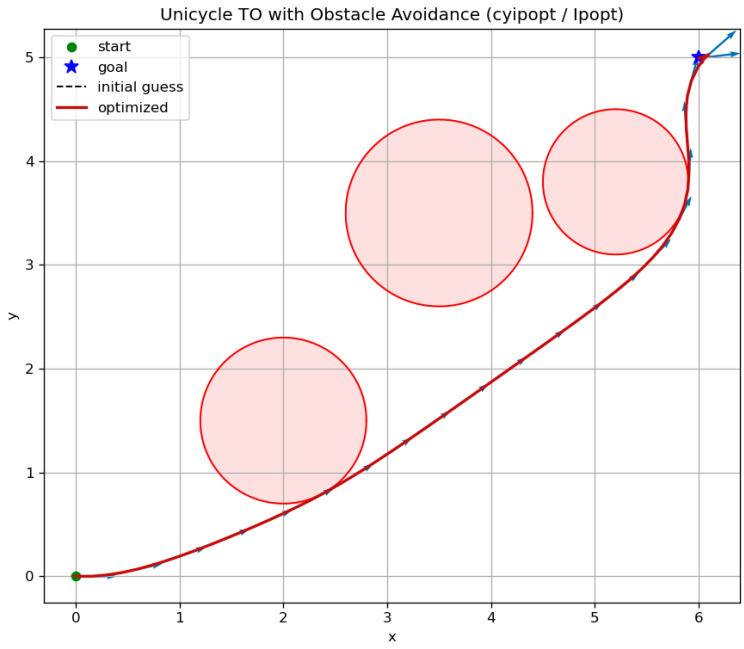

Figure 4.9: Trajectory optimization for unicyle using IPOPT (all-zero initialization).

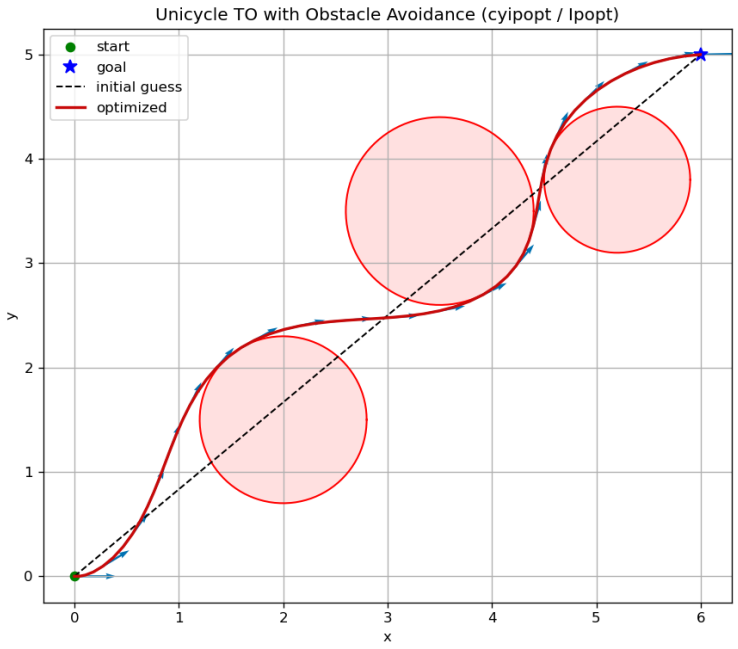

Figure 4.10: Trajectory optimization for unicyle using IPOPT (straight-line initialization).

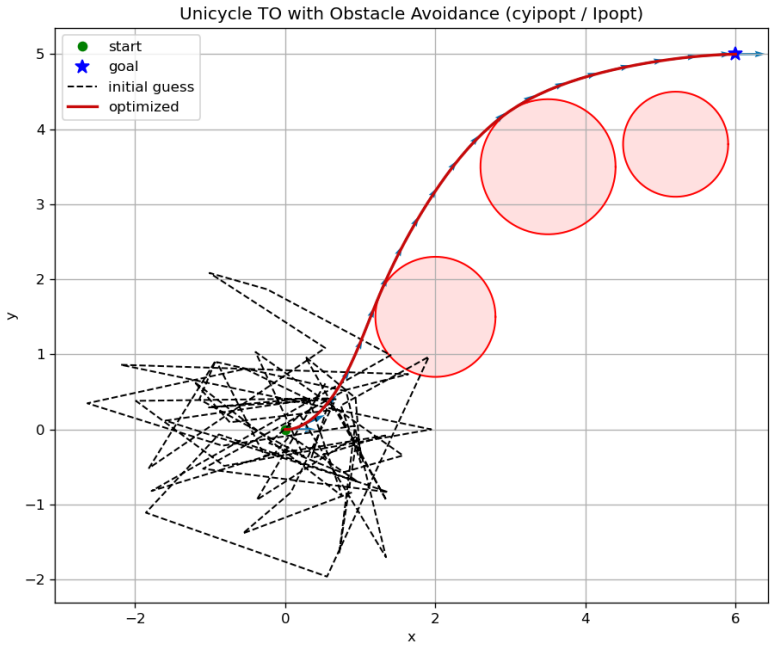

Figure 4.11: Trajectory optimization for unicyle using IPOPT (random initialization).

horizon optimal control problem with the current measured state as the initial condition, applies only the first control action, and then repeats the procedure at the next step. This closes the loop, providing robustness to model mismatch and disturbances while naturally adapting the trajectory as the environment or task evolves. In this section we introduce the core ideas behind MPC; for a comprehensive treatment, see (Rawlings et al., 2020).

Dynamics. Let us consider a discrete-time dynamical system

$$s_{t+1} = f(s_t, a_t, w_t), \tag{4.35}$$

where $s_t \in \mathbb{R}^n$ represents the state, $a_t \in \mathbb{R}^m$ represents the control/action, and $w_t \in \mathbb{R}^d$ represents a stochastic disturbance. With slight abuse of notation, let us use

$$s_{t+1} = f(s_t, a_t) \equiv f(s_t, a_t, 0)$$

to denote the system dynamics without the disturbance.

> Here we have used (s_t, a_t) to represent the state and control. This departs from the notation (4.1) used in the beginning. The reason for doing so will become clear soon.

Trajectory Optimization. At time step t, let the system's state be s_t as given (e.g., measured from sensor data). MPC solves the following open-loop TO problem, adapted from (4.31):

$$\begin{aligned}
\min_{\{x_k, u_k\}} \quad & \Phi(x_N) + \sum_{k=0}^{N-1} \ell_k(x_k, u_k) \\
\text{s.t.} \quad & x_{k+1} = f(x_k, u_k), \qquad t = 0, \dots, N-1, \\
& \boxed{x_0 = s_t} \quad \text{(given)}, \\
& x_k \in \mathcal{X}_k, \quad u_k \in \mathcal{U}_k \quad \text{(bounds)}, \\
& g_k(x_k, u_k) \le 0, \quad h_k(x_k, u_k) = 0 \quad \text{(path/terminal constraints)}.
\end{aligned} \tag{4.36}$$

Notice that in (4.36):

- We used k to denote the time step in the TO problem, in contrast to t in the system dynamics.
- We used (x_k, u_k) to denote the state and control in the TO problem, as opposed to (s_t, a_t) in the system dynamics.
- We enforce the TO problem starts at $x_0 = s_t$, i.e., the initial state in trajectory planning aligns with the current system state at time t.

Receding Horizon Control. Let

$$(x_0^\star, u_0^\star, x_1^\star, u_1^\star, \dots, x_{N-1}^\star, u_{N-1}^\star, x_N^\star) \tag{4.37}$$

be an optimal solution of the TO problem (4.36) (e.g., obtained from IPOPT). Instead of executing the entire sequence of optimal controls $(u_0^\star, u_1^\star, \dots, u_{N-1}^\star)$, MPC will only execute the first element. Formally, the actual control action applied to the system, denoted as a_t, is

$$a_t = u_0^\star. \tag{4.38}$$

Applying $a_t = u_0^\star$ to the real dynamics (4.35), the system will step into a new state s_{t+1} at time $t+1$:

$$s_{t+1} = f(s_t, u_0^\star, w_t).$$

Then, MPC solves a new TO problem that is exactly the same as the problem (4.36) at time t, except that the initial state is changed to s_{t+1}:

$$\begin{aligned} \min_{\{x_k, u_k\}} \quad & \Phi(x_N) + \sum_{k=0}^{N-1} \ell_k(x_k, u_k) \\ \text{s.t.} \quad & x_{k+1} = f(x_k, u_k), \qquad t = 0, \dots, N-1, \\ & \boxed{x_0 = s_{t+1}} \quad \text{(given)}, \\ & x_k \in \mathcal{X}_k, \quad u_k \in \mathcal{U}_k \quad \text{(bounds)}, \\ & g_k(x_k, u_k) \le 0, \quad h_k(x_k, u_k) = 0 \quad \text{(path/terminal constraints)}. \end{aligned} \tag{4.39}$$

Problems (4.36) and (4.39) are called *parametric optimization problems*, in the sense that the form of the optimization problem remains the same, but the value of the parameter s_t that defines the equality constraint has changed.

For this reason, we should restate the solution in (4.37) as

$$(x_0^\star(s_t), u_0^\star(s_t), x_1^\star(s_t), u_1^\star(s_t), \dots, x_{N-1}^\star(s_t), u_{N-1}^\star(s_t), x_N^\star(s_t)) \tag{4.40}$$

because they are all implicit functions of the parameter s_t.

Similarly, the solution to the TO problem (4.39) at time $t+1$ should be denoted as

$$(x_0^\star(s_{t+1}), u_0^\star(s_{t+1}), x_1^\star(s_{t+1}), u_1^\star(s_{t+1}), \dots, x_{N-1}^\star(s_{t+1}), u_{N-1}^\star(s_{t+1}), x_N^\star(s_{t+1})) \tag{4.41}$$

to indicate that they are functions of s_{t+1}.

After solving the TO problem at time $t+1$, MPC executes the first element:

$$a_{t+1} = u_0^\star(s_{t+1}),$$

and the system steps into a new state s_{t+2}, from which MPC solves a new TO problem with parameter s_{t+2}, and the process continues.

Implicit Feedback. Through receding horizon control, MPC creates an implicit feedback control policy. Let the system's state at time t be s_t, the feedback policy is:

$$a_t = u_0^\star(s_t) := \mu(s_t) = \arg\min \text{ (the TO problem with parameter } s_t). \quad (4.42)$$

Unlike the policy in the case of RL, which is an explicit neural network, the policy of MPC is implicit and comes from the optimal solution of the parametric TO problem.

Closed-Loop System. The closed-loop system under the MPC policy is therefore

$$s_{t+1} = f(s_t, \mu(s_t), w_t) := f_{\text{CL}}(s_t, w_t), \quad (4.43)$$

which becomes an uncontrolled system with disturbance w_t. The diagram of the closed-loop system is shown in Fig. 4.12. Lots of research have studied properties of the closed-loop system, such as stability and robustness to disturbance. One of the most important results states that, as long as the terminal loss function $\Phi(\cdot)$ satisfies a technical condition (being a control Lyapunov function), then the closed-loop system is stable. We do not go deep into these results for MPC analysis and refer the reader to (Rawlings et al., 2020).

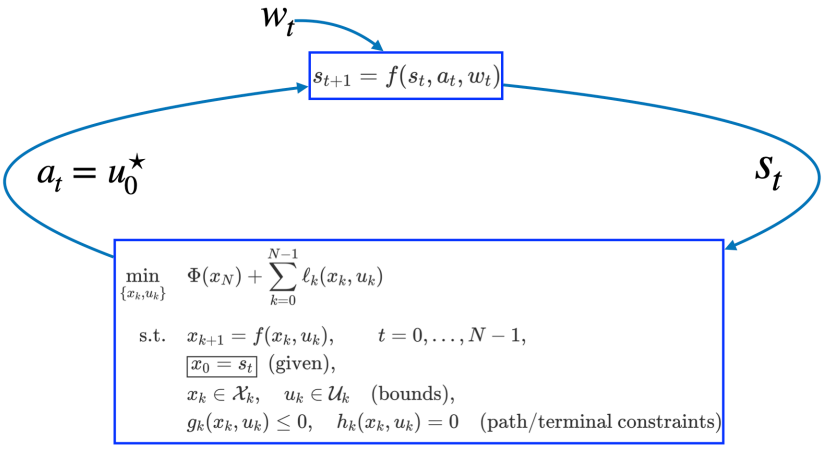

Figure 4.12: Model Predictive Control.

Warmstart for Parametric Optimization. The requirement that an MPC feedback policy solve an optimization problem online makes it computationally expensive. In some domains, e.g., chemical process control, control updates are infrequent (for example, every 10 minutes) and MPC is practical. In other domains such as robotics, control rates are much higher (e.g., 10,Hz), so reducing MPC's computational cost is critical. A common strategy is *warm-starting*: when solving the optimization for the new parameter s_{t+1} at time $t+1$, initialize the numerical solver with the previous solution (4.40) computed for s_t at time t.

If the change from s_t to s_{t+1} is small, the optimal solution at $t+1$ is likely close to that at t, and warm-starting typically reduces the solver's required iterations.

We now apply MPC to the stabilization of a double integrator with control constraints.

Example 4.6 (MPC for Double Integrator)**.** This example implements Model Predictive Control (MPC) for a 1D double-integrator (position–velocity) system using a quadratic program (QP).

Dynamics of the System. We model the kinematics of a point mass with (discrete-time) constant-acceleration physics. Let the state be

$$x_k = \begin{bmatrix} p_k \\ v_k \end{bmatrix} \in \mathbb{R}^2, \quad u_k \in \mathbb{R}$$

where p_k is position, v_k is velocity, and u_k is the commanded acceleration. With sampling time Δt, the discrete-time, *nominal* dynamics are

$$x_{k+1} = Ax_k + Bu_k, \quad A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix}.$$

Control Goal. We consider *regulation to a fixed target* $x_{\text{goal}} \in \mathbb{R}^2$, typically the origin:

$$x_{\text{goal}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

(Tracking a moving reference is immediate by updating x_{goal} online.)

Finite-Horizon Trajectory Optimization (QP). At each control time t, with measured state s_t, MPC solves a length-N open-loop trajectory optimization (TO) problem with decision variables $\{x_k, u_k\}_{k=0}^{N}$ (with u_N unused):

$$\min_{\{x_k,u_k\}} \quad \underbrace{(x_N - x_{\text{goal}})^\top Q_f (x_N - x_{\text{goal}})}_{\text{terminal cost}} + \sum_{k=0}^{N-1} \Big[\underbrace{(x_k - x_{\text{goal}})^\top Q (x_k - x_{\text{goal}})}_{\text{state cost}} + \underbrace{u_k^\top R u_k}_{\text{effort}} \Big]$$

$$\text{s.t.} \quad x_{k+1} = Ax_k + Bu_k, \quad k = 0, \dots, N-1,$$

$$\boxed{x_0 = s_t} \quad \text{(enforces the current initial condition)},$$

$$x_{\min} \le x_k \le x_{\max}, \quad k = 0, \dots, N,$$

$$u_{\min} \le u_k \le u_{\max}, \quad k = 0, \dots, N-1.$$

The problem in (4.6) is a convex QP (quadratic cost; linear dynamics and box constraints). We will write a Python code that builds this QP once with CVXPY and then only updates the Parameter for the measured state s_t and (optionally) x_{goal} at each MPC step.

Receding Horizon (Implicit Feedback). Let $\{x_k^\star, u_k^\star\}$ be the optimizer of (4.6). MPC applies only the first input

$$a_t = u_0^\star,$$

then measures the new state, shifts/warm-starts the QP, and resolves. This closes the loop and yields an implicit feedback policy $a_t = \mu(s_t) = u_0^\star(s_t)$.

True Dynamics with Disturbance. To test robustness, the closed-loop plant advances with additive disturbance:

$$s_{t+1} = A\, s_t + B\, a_t + w_t,$$

where w_t is zero-mean noise. In the code, w_t is a small Gaussian perturbation added to both position and velocity updates. This models sensor/actuator errors or unmodeled effects and illustrates how MPC corrects by re-solving at every step.

Experiment Setup.

- **Initialization.** Start from $x_0 = [4, 0]^\top$ (far from the goal). Fix $x_{\text{goal}} = [0, 0]^\top$.
- **Horizon and sampling.** $N = 20$, $\Delta t = 0.1$ s; run for $T = 120$ MPC steps (12 s).
- **QP solver and structure.** We solve with OSQP via CVXPY. The problem is constructed once; at each step we update the parameter $x_0 = s_t$. This preserves the QP structure (matrices P and A), enabling factorization reuse and faster solves.
- **Warm-start.** Shift the previous optimal sequence $u^\star$ left by one step and hold the last value, then roll out the dynamics from the new s_t to seed a feasible x-trajectory guess. These are passed to the solver via `warm_start=True`.

Fig. 4.13 shows the closed-loop position and velocity under the MPC controller in the case of small disturbancen (the standard deviations of the two elements of w_t are 0.002 and 0.005).

Fig. 4.14 shows the closed-loop position and velocity under the MPC controller in the case of large disturbancen (the standard deviations of the two elements of w_t are 0.02 and 0.05).

In both cases, the MPC policy successfully regulates the system around the origin, illustrating robustness of the MPC policy to disturbances.

You can play with the code here.

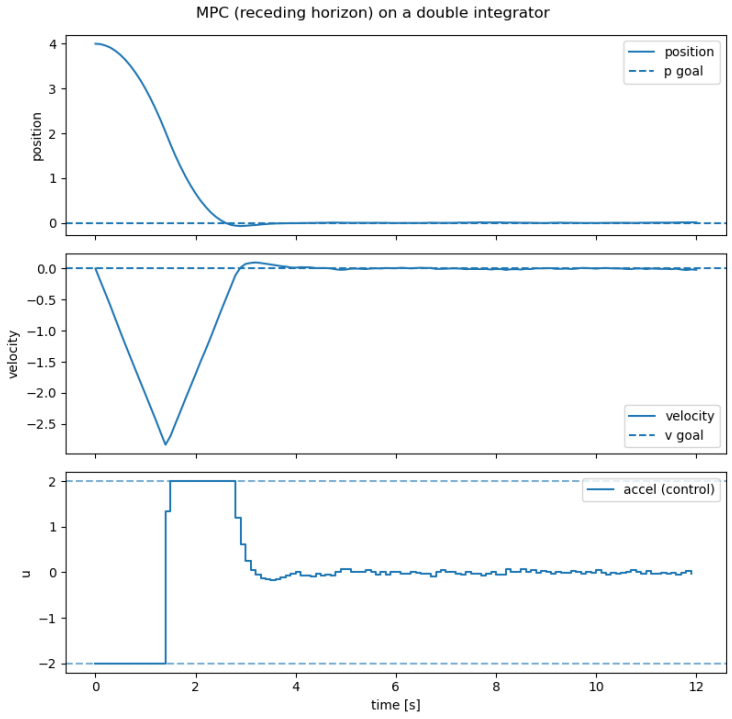

Figure 4.13: MPC for double integrator (small disturbance).

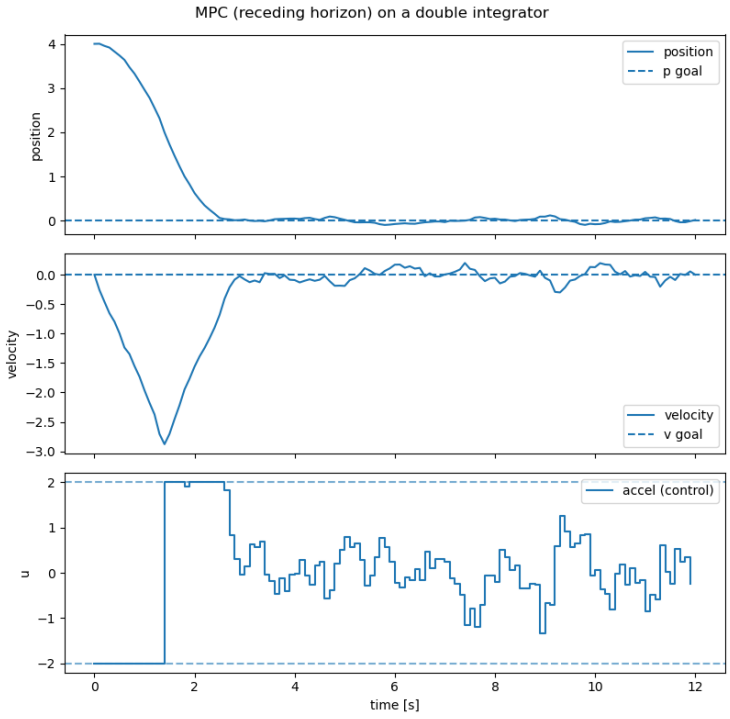

Figure 4.14: MPC for double integrator (large disturbance).

4.5 Model Predictive Path Integral Control

We have seen that numerical techniques–particularly gradient-based nonlinear programming methods–can be employed to solve trajectory optimization (TO) problems. Furthermore, solving TO problems repeatedly in an online fashion yields an implicit feedback policy, as in model predictive control (MPC).

The numerical methods discussed so far for TO rely on access to at least first-order derivatives of the objective and constraint functions. In this section, we introduce a different class of methods, known as model predictive path integral (MPPI) control (Williams et al., 2016). These methods do not require derivative information; instead, they rely solely on evaluating the objective and constraints through forward simulations of the system dynamics.

Problem Formulation. We consider a general nonlinear discrete-time dynamical system

$$x_{t+1} = f_t(x_t, u_t), \qquad t = 0, \dots, N-1,$$

with state $x_t \in \mathbb{R}^n$ and control $u_t \in \mathbb{R}^m$. Given an initial condition $x_0 = \hat{x}_0$ (known), the finite-horizon trajectory optimization problem is

$$\begin{aligned} \min_{\{x_t, u_t\}_{t=0}^{N-1}} \quad & \Phi(x_N) \; + \; \sum_{t=0}^{N-1} \ell_t(x_t, u_t) \\ \text{s.t.} \quad & x_{t+1} = f_t(x_t, u_t), \quad t = 0, \dots, N-1, \\ & x_0 = \hat{x}_0 \;\; \text{(given)}. \end{aligned}$$

Here, ℓ_t is the stage (running) cost and Φ is the terminal cost. A *dynamically feasible trajectory* $\{x_t, u_t\}_{t=0}^{N-1}$ satisfies the dynamics constraints for all t.

High-Level Idea of MPPI. MPPI is a *sampling-based*, *derivative-free*, receding-horizon controller derived from path-integral / information-theoretic control. Instead of solving the constrained optimization problem via gradients or linearizations, MPPI:

1. Maintains a nominal open-loop control sequence $U = \{u_0, \dots, u_{N-1}\}$.
2. At the current state x_0, it samples many noisy control sequences $u_t + \epsilon_t^{(k)}$ for rollouts $k = 1, \dots, K$.
3. For each rollout, it propagates the dynamics forward and scores the trajectory with the same task cost $\Phi + \sum \ell_t$.
4. It *soft-min aggregates* the sampled control perturbations with weights

$$w_k \;\propto\; \exp\Big(-\tfrac{1}{\lambda}\big(S^{(k)} - \min_j S^{(j)}\big)\Big),$$

where $S^{(k)}$ is the total cost of the k-th trajectory and $\lambda > 0$ is a temperature parameter.

5. It updates the nominal sequence by adding the weighted average of the sampled perturbations:

$$u_t \leftarrow u_t + \sum_{k=1}^{K} \bar{w}_k \, \epsilon_t^{(k)}, \qquad \bar{w}_k = \frac{w_k}{\sum_j w_j}.$$

   Optionally, if computational budget permits, one can repeat steps 1 to 5 for multiple iterations. If the control actions need to satisfy hard bounds, we can project u_t to the feasible set.

6. It applies only the *first* control u_0 to the real system, shifts the sequence forward, and repeats at the next time step (receding horizon).

Key properties.

- Works with general nonlinear dynamics and arbitrary costs; no gradients, no linearization required.
- Robustness arises from the soft-min over many trajectories.
- Parallelizable (GPU-friendly) because rollouts are independent.

Computational details of the MPPI algorithm are given as follows.

Rollouts and total cost. Given current state x_0 and nominal controls $U = \{u_t\}_{t=0}^{N-1}$, draw K noise sequences

$$\epsilon^{(k)} = \{\epsilon_0^{(k)}, \dots, \epsilon_{N-1}^{(k)}\}, \qquad \epsilon_t^{(k)} \sim \mathcal{N}(0, \Sigma_t)$$

(possibly time-correlated for smoothness). For each $k \in [1, K]$:

1. Simulate

$$\tilde{u}_t^{(k)} = u_t + \epsilon_t^{(k)}, \qquad x_{t+1}^{(k)} = f_t(x_t^{(k)}, \tilde{u}_t^{(k)}), \quad x_0^{(k)} = x_0.$$

2. Accumulate the total trajectory cost

$$S^{(k)} = \Phi(x_N^{(k)}) + \sum_{t=0}^{N-1} \ell_t(x_t^{(k)}, \tilde{u}_t^{(k)}).$$

Exponential weights (soft-min). Stabilize by subtracting the minimum cost:

$$w_k = \exp\Big(-\tfrac{1}{\lambda}(S^{(k)} - S_{\min}) \Big), \quad S_{\min} := \min_j S^{(j)}.$$

Normalize $\bar{w}_k = w_k / \sum_j w_j$.

Control update. For each time $t = 0, \dots, N-1$, update

$$u_t \leftarrow u_t + \sum_{k=1}^{K} \bar{w}_k \, \epsilon_t^{(k)}.$$

If control is bounded, project the controls onto the feasible sets.

Practical knobs.

- K: number of rollouts (compute–performance tradeoff).
- λ: temperature (smaller $\Rightarrow$ greedier toward the best rollouts).
- Σ_t: exploration covariance; can use AR(1) temporal correlation for smooth controls.

Example 4.7 (MPPI for Pendulum Swing Up)**.** We apply the MPPI controller, in an MPC receding-horizon control fashion to the pendulum swing-up problem.

We use $K = 16$ rollouts and set $\lambda = 1$.

When Σ_t is large enough, for example, when we set the standard deviation of the noise to be 2, the MPPI controller can successfully swing up the pendulum, demonstrating a highly sophisticated motion plan as shown in Fig. 4.15.

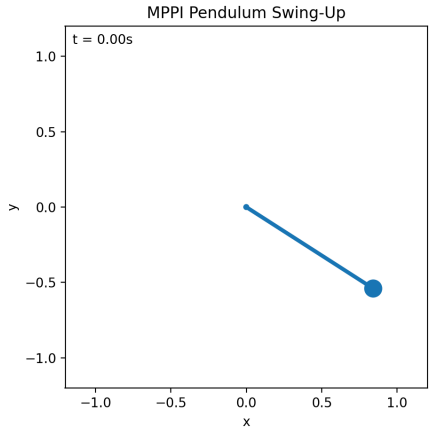

Figure 4.15: Pendulum swing-up using MPPI (large exploration).

However, when Σ_t is too small, when we set the standard deviation of the noise to be 0.1, the MPPI controller fails at the task, as shown in Fig. 4.16.

You can play with the code here.

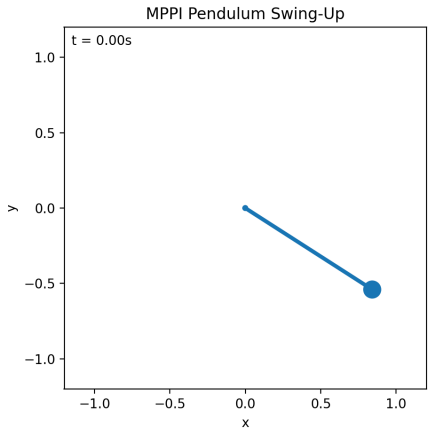

Figure 4.16: Pendulum swing-up using MPPI (small exploration).

4.5.1 Information-Theoretic Derivation of MPPI

We now derive Model Predictive Path Integral (MPPI) control from an information-theoretic (KL-control / control-as-inference) viewpoint. We proceed from a variational free-energy objective to the Gibbs (Boltzmann) optimal trajectory law, then obtain the practical MPPI update via moment projection and importance sampling.

Let a finite-horizon trajectory be $\tau = (x_0, u_0, \dots, u_{H-1}, x_H)$ generated by discrete-time dynamics

$$x_{t+1} = f_t(x_t, u_t), \quad t = 0, \dots, H-1.$$

Let $S(\tau)$ denote the total trajectory cost (e.g., terminal + running + control effort). Let $p(\tau)$ be a baseline (prior) trajectory distribution (e.g., nominal/uncontrolled dynamics plus a prior over control sequences).

We seek a controlled trajectory distribution $q(\tau)$ that trades performance against deviation from the prior via:

$$\mathcal{F}(q) \;=\; \mathbb{E}_q[S(\tau)] \;+\; \lambda\,\mathrm{KL}(q(\tau)\,\|\,p(\tau)), \qquad \lambda > 0.$$

This is convex in q (the first term is linear in q while the second term is convex in q). The scalar λ plays the role of a temperature (risk-sensitivity / exploration level).

Gibbs–Boltzmann Optimum. Our goal is now to minimize $\mathcal{F}(q)$ over all normalized densities $q(\tau)$. This is a constrained convex optimization in q. Therefore, we can formulate the Lagrangian and derive the KKT optimality conditions.

Exercise 4.2. Formulate the Lagrangian for problem (4.5.1) and find its KKT optimality conditions.

Solving the KKT optimality conditions, one can find the optimal solution

$$q^\star(\tau) \;\propto\; p(\tau)\,\exp(-S(\tau)/\lambda).$$

Thus, with proper normalization, we have

$$q^\star(\tau) \;=\; \frac{1}{Z}\,p(\tau)\,\exp(-S(\tau)/\lambda), \qquad Z \;=\; \int p(\tau)\,e^{-S(\tau)/\lambda}\,d\tau \;=\; \mathbb{E}_p\big[e^{-S/\lambda}\big].$$

Plugging $q^\star$ back into $\mathcal{F}$ gives the free-energy identity:

$$\min_q \Big\{\mathbb{E}_q[S] + \lambda\,\mathrm{KL}(q\|p)\Big\} \;=\; -\lambda \log \mathbb{E}_p\left[e^{-S/\lambda}\right].$$

From Gibbs–Boltzmann Optimum to MPPI Updates. Let a trajectory be $\tau = (x_0, u_0, \dots, u_{H-1}, x_H)$ with dynamics $x_{t+1} = f_t(x_t, u_t)$. We introduce a baseline (prior) trajectory distribution $p(\tau)$ as the law induced by:

$$\mathbf{v} = (v_0, \dots, v_{H-1}) \sim \mathcal{N}(\mathbf{U}, \Sigma), \qquad v_t = u_t + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \Sigma_t),$$

and rolling out the dynamics with controls $u_t = v_t$. Here $\mathbf{U} = (u_0, \dots, u_{H-1})$ is the current nominal control sequence.

Given a trajectory cost $S(\tau)$ and temperature $\lambda > 0$, the optimal controlled trajectory law is

$$q^\star(\tau) \;=\; \frac{1}{Z}\,p(\tau)\,\exp(-S(\tau)/\lambda), \qquad Z = \mathbb{E}_p\big[e^{-S/\lambda}\big].$$

Because of nonlinear dynamics and the exponential tilt by S, $q^\star$ **is generally not Gaussian**, even if p is.

MPPI *approximates* $q^\star$ by maintaining a *Gaussian over the control sequence* with the *same covariance* Σ and a tunable mean $\mathbf{U} = (u_0, \dots, u_{H-1})$:

$$\pi_{\mathbf{U}}(\mathbf{v}) \;=\; \mathcal{N}(\mathbf{v}; \mathbf{U}, \Sigma).$$

MPPI projects $q^\star$ onto this family by minimizing $\mathrm{KL}(\pi_{\mathbf{U}} \,\|\, q^\star)$ while *holding* Σ *fixed*. For Gaussians with fixed covariance, this yields **moment matching of the mean**:

$$\mathbf{U}_{\text{new}} \;=\; \mathbb{E}_{q^\star}[\mathbf{v}].$$

To perform this moment matching, we write $v_t = u_t + \epsilon_t$. Then

$$u_{t,\text{new}} \;=\; \mathbb{E}_{q^\star}[v_t] = u_t + \mathbb{E}_{q^\star}[\epsilon_t].$$

We cannot sample $q^\star$ directly, but by the Gibbs–Boltzmann form and importance sampling:

$$\mathbb{E}_{q^\star}[g(\tau)] = \frac{\mathbb{E}_p[\, g(\tau)\, e^{-S(\tau)/\lambda}\,]}{\mathbb{E}_p[\, e^{-S(\tau)/\lambda}\,]}.$$

Choosing $g(\tau) = \epsilon_t$ and using rollouts $\{\tau^{(k)}\}_{k=1}^K \sim p$ with costs $S^{(k)}$ and noises $\epsilon_t^{(k)}$ gives the MPPI update:

$$u_t \;\leftarrow\; u_t \;+\; \sum_{k=1}^{K} \bar{w}_k\, \epsilon_t^{(k)}, \qquad \bar{w}_k = \frac{\exp(-\frac{1}{\lambda}(S^{(k)} - S_{\min}))}{\sum_j \exp(-\frac{1}{\lambda}(S^{(j)} - S_{\min}))}.$$

Here $S_{\min} = \min_k S^{(k)}$ is a numerical stabilizer that cancels in the ratio.

Takeaway.

- $p(\tau)$: baseline trajectory law induced by Gaussian exploratory controls and the dynamics.
- $q^\star(\tau)$: Gibbs–Boltzmann tilt of p, **not** Gaussian in general.
- MPPI **approximates** $q^\star$ by a **Gaussian over controls with fixed covariance** Σ and updates the mean by **importance-weighted averaging of sampled noise**, yielding the standard MPPI updates used in practice.

4.6 Rapidly Exploring Random Tree

In Section 4.5, we examined the effectiveness of one particular sampling-based planning method—MPPI—for solving trajectory-planning problems, noting that it is often less susceptible to local minima than nonlinear-programming-based approaches. More broadly, sampling-based motion planning encompasses a wide family of algorithms, extensive enough to merit an entire semester for a comprehensive treatment. For readers interested in exploring this area further, the textbooks (LaValle, 2006) and (Choset et al., 2005) provide excellent overviews.

In this section, we introduce one of the most widely used sampling-based algorithms: the rapidly exploring random tree (RRT). We will begin with the case of *kinematic* motion planning, in which the system or robot is modeled as a point mass without dynamics, and then extend the discussion to *kinodynamic* motion planning, where the system's dynamics must be taken into account.

4.6.1 Kinematic Motion Planning

Problem Setup. We consider a robot with configuration space $\mathcal{C} \subset \mathbb{R}^d$. Obstacles occupy $\mathcal{C}_{\text{obs}}$, and the *free space* is

$$\mathcal{C}_{\text{free}} = \mathcal{C} \setminus \mathcal{C}_{\text{obs}}.$$

Given a start configuration $q_{\text{start}} \in \mathcal{C}_{\text{free}}$ and a *goal region* $\mathcal{C}_{\text{goal}} \subset \mathcal{C}_{\text{free}}$, the objective is to find a *collision-free path*

$$\sigma : [0,1] \to \mathcal{C}_{\text{free}}, \qquad \sigma(0) = q_{\text{start}},\ \sigma(1) \in \mathcal{C}_{\text{goal}}.$$

Here, $\sigma(\cdot)$ is a continuous curve in the configuration space $\mathcal{C}$ that connects the start position to the goal region. A path can be represented in several ways (e.g., piecewise-polynomial splines); in this section we adopt a **piecewise-linear** representation: the path is a sequence of waypoints $(q_0, q_1, \dots, q_M)$ joined by straight-line segments in $\mathcal{C}$, i.e.,

$$\sigma(s) = (1-s)\, q_i + s\, q_{i+1}, \quad s \in [0,1], \quad i = 0, \dots, M-1.$$

The RRT algorithm requires the following components.

Sampler. We require a mechanism to draw random configurations in free space. The simplest choice is uniform sampling:

$$q_{\text{rand}} \sim \mathcal{U}(\mathcal{C}_{\text{free}}),$$

optionally with *goal bias*: introduce $B \sim \text{Bernoulli}(p_{\text{goal}})$ and set

$$q_{\text{rand}} = \begin{cases} \text{a sample from } \mathcal{C}_{\text{goal}}, & \text{if } B = 1, \\ \text{a sample from } \mathcal{C}_{\text{free}}, & \text{if } B = 0. \end{cases}$$

To perform such sampling, we require that, given any configuration $q \in \mathcal{C}$, we can check its membership in $\mathcal{C}_{\text{free}}$, $\mathcal{C}_{\text{obs}}$, and $\mathcal{C}_{\text{goal}}$.

Metric (Distance Function). We need a distance $d : \mathcal{C} \times \mathcal{C} \to \mathbb{R}_{\geq 0}$ to compare configurations. Common choices:

$$d(q_i, q_j) = \|W(q_i - q_j)\|_2,$$

where W is a positive-definite weighting (e.g., to account for different joint ranges).

For SE(2) or SE(3) manifolds,

- handle angular wrap-around (e.g., modulo 2π),
- optionally combine translational and rotational parts with different weights,

- use a *geodesic* or a local chart; for rotations, a distance induced by the Lie group metric.

Sometimes d is defined in task space via forward kinematics $x = \Phi(q)$ (here q represents the joint angles of a robot arm and $\Phi(q)$ returns the position of the robot's end-effector):

$$d(q_i, q_j) = \|\Phi(q_i) - \Phi(q_j)\|,$$

useful when end-effector placement matters more than joint motion.

Nearest-Neighbor Query. Given the current tree T with vertex set $V(T)$, select the closest existing node to the new sample:

$$q_{\text{near}} \in \arg\min_{q \in V(T)} d(q,\, q_{\text{rand}}).$$

Implementation notes.

- Use spatial indices (kd-trees, ball trees, cover trees) for sublinear searches.
- Approximate nearest neighbor methods often suffice and substantially accelerate growth in higher dimensions.

Steering / Step Map. Move from q_{near} *toward* q_{rand} by at most a step size $\eta > 0$. In Euclidean $\mathcal{C}$,

$$\Delta = q_{\text{rand}} - q_{\text{near}}, \qquad q_{\text{new}} = q_{\text{near}} + \alpha\,\Delta, \quad \alpha = \min\left\{1,\ \frac{\eta}{\|\Delta\|_2}\right\}.$$

On manifolds (e.g., $SE(3)$), interpolate along a feasible local geodesic; for rotations, use group operations (e.g., $q_{\text{new}} = q_{\text{near}} \exp(\alpha \log(q_{\text{near}}^{-1} q_{\text{rand}}))$).

Tuning η.

- Smaller η improves collision fidelity and resolution in narrow passages but slows exploration.
- Larger η accelerates coverage but can skip through feasible corridors.

Collision Checking (Local Planner Validity). Validate the entire segment between q_{near} and q_{new}:

$$\text{ensure } \sigma(s) \in \mathcal{C}_{\text{free}} \quad \forall s \in [0,1], \quad \sigma(s) = (1-s)q_{\text{near}} + s q_{\text{new}}.$$

Resolution and robustness.

- Use a discretization step δ small enough relative to obstacle feature size and robot footprint.

- For articulated robots, perform swept-volume or continuous collision detection where possible; otherwise sample sufficiently along the edge (in joint and time parameter s).
- Inflate obstacles or deflate the robot model slightly to build clearance margins.

Stopping / Success Criteria. Let the goal be a set $\mathcal{C}_{\text{goal}}$ (e.g., a ball of radius ε around a target pose). Declare success when

$$q_{\text{new}} \in \mathcal{C}_{\text{goal}}.$$

Common termination rules.

- Maximum number of iterations or samples.
- Time budget exhausted (anytime behavior).
- Early stop after first solution (RRT) or continue to refine with rewiring (RRT*).

Post-processing (optional).

- **Shortcutting:** repeatedly replace subpaths by direct segments if collision-free.
- **Smoothing:** fit splines to waypoints while maintaining clearance, or re-optimize locally.

We are ready to present the RRT algorithm.

Algorithm: Rapidly-Exploring Random Tree (RRT) — Kinematic Case

Inputs: start configuration $q_{\text{start}} \in \mathcal{C}_{\text{free}}$; goal region $\mathcal{C}_{\text{goal}} \subset \mathcal{C}_{\text{free}}$; step size $\eta > 0$; goal-bias $p_{\text{goal}} \in [0,1]$; iteration budget N; metric $d : \mathcal{C} \times \mathcal{C} \to \mathbb{R}_{\geq 0}$; collision checker for segments in $\mathcal{C}$; nearest-neighbor data structure over the current vertex set.

Initialize the tree T with vertex set $V(T) \leftarrow \{q_{\text{start}}\}$ and no edges.

For $i = 0, 1, 2, \ldots, N-1$

1. **Goal-biased sampling.** Draw $B \sim \text{Bernoulli}(p_{\text{goal}})$ and set

$$q_{\text{rand}} \sim \begin{cases} \mathcal{U}(\mathcal{C}_{\text{goal}}), & \text{if } B = 1, \\ \mathcal{U}(\mathcal{C}_{\text{free}}), & \text{if } B = 0. \end{cases}$$

2. **Nearest neighbor.** Select

$$q_{\text{near}} \in \arg\min_{q \in V(T)} d(q,\, q_{\text{rand}})\,.$$

3. **Steering (bounded step).** Let $\Delta := q_{\text{rand}} - q_{\text{near}}$ in a local chart of $\mathcal{C}$ (with appropriate handling of angles/manifold coordinates). Define

$$\alpha := \min\left\{1,\ \frac{\eta}{\|\Delta\|}\right\}, \qquad q_{\text{new}} := q_{\text{near}} + \alpha\,\Delta.$$

   (On manifolds, replace the affine step with interpolation along a local geodesic.)

4. **Collision checking.** Verify the straight segment

$$\sigma(s) = (1-s)\,q_{\text{near}} + s\,q_{\text{new}}, \quad s \in [0,1],$$

   satisfies $\sigma(s) \in \mathcal{C}_{\text{free}}$ for all sampled s (or via continuous collision detection).

5. **Tree update.** If the segment is collision-free, append the vertex and edge:

$$V(T) \leftarrow V(T) \cup \{q_{\text{new}}\}, \qquad E(T) \leftarrow E(T) \cup \{(q_{\text{near}} \to q_{\text{new}})\}.$$

6. **Success test.** If $q_{\text{new}} \in \mathcal{C}_{\text{goal}}$, **terminate** and return the path formed by the unique tree route from q_{start} to q_{new}.

Stopping: If no success after N iterations (or upon time-budget exhaustion), **return failure** (or the best partial progress if maintained).

Properties.

- **Probabilistic completeness:** If a solution exists, the probability that RRT finds one approaches 1 as $N \to \infty$.
- **Not optimal:** The first-found solution can be far from shortest; RRT does not asymptotically improve path quality.

4.6.1.1 Make RRT Optimal: RRT*

The classical RRT is probabilistically complete but not optimal. To obtain asymptotic optimality (i.e., the best-path cost converges to the global optimum as the number of samples $N \to \infty$), we use **RRT***: a cost-aware variant that (i) selects parents by minimizing cost-to-come and (ii) rewires nearby nodes to improve their costs when possible.

Key Ideas Behind RRT*:

Cost functional. Let $c(\cdot)$ denote path cost (e.g., length or any additive, nonnegative, Lipschitz continuous cost). For an edge between two configurations q and q', write $c(q \to q')$ and define

$$c(q') \;=\; \min_{(q \to q') \in E(T)} \{c(q) + c(q \to q')\},$$

where $E(T)$ is the set of tree edges.

Near set. Rather than connecting to only the nearest node, examine a **ball of neighbors** around the new sample q_{new} whose radius shrinks with n:

$$r_n \;=\; \gamma \left(\frac{\log n}{n} \right)^{1/d},$$

where d is the dimension of $\mathcal{C}$ and $\gamma > \gamma^*$ is a constant depending on the measure of $\mathcal{C}_{\text{free}}$. (Alternative: k-nearest with $k_n \geq k_0 \log n$.)

Choose parent by cost. Among the near nodes that can connect to q_{new} collision-free, pick the parent that minimizes total cost-to-come.

Rewire. Attempt to improve each neighbor q by re-connecting it through q_{new} if this reduces $c(q)$ while preserving feasibility.

Asymptotic optimality. Under standard assumptions (e.g., δ-robustly feasible problem, absolutely continuous sampling over $\mathcal{C}_{\text{free}}$, consistent local planner, and the r_n or k_n schedules above), the best path cost in the tree converges almost surely to the global optimum.

The RRT* Algorithm is presented below.

Algorithm: RRT* (Asymptotically Optimal RRT)

Inputs: $q_{\text{start}} \in \mathcal{C}_{\text{free}}$, goal region $\mathcal{C}_{\text{goal}} \subset \mathcal{C}_{\text{free}}$; metric d; step size $\eta > 0$; goal-bias $p_{\text{goal}} \in [0, 1]$; iteration budget N; neighbor schedule $r_n = \gamma(\frac{\log n}{n})^{1/d}$ (or $k_n \geq k_0 \log n$).

Initialize T with $V(T) \leftarrow \{q_{\text{start}}\}$, $E(T) \leftarrow \emptyset$, $c(q_{\text{start}}) = 0$.

For $i = 0, 1, 2, \ldots, N-1$

1. **Goal-biased sampling.** Draw $B \sim \text{Bernoulli}(p_{\text{goal}})$ and set

$$q_{\text{rand}} \sim \begin{cases} \mathcal{U}(\mathcal{C}_{\text{goal}}), & B = 1, \\ \mathcal{U}(\mathcal{C}_{\text{free}}), & B = 0. \end{cases}$$

2. **Nearest and steer.** Choose

$$q_{\text{near}} \in \arg\min_{q \in V(T)} d(q, q_{\text{rand}}), \quad q_{\text{new}} := q_{\text{near}} + \alpha(q_{\text{rand}} - q_{\text{near}}),$$

with $\alpha = \min\{1, \eta / \|q_{\text{rand}} - q_{\text{near}}\|\}$. If the segment $[q_{\text{near}}, q_{\text{new}}]$ is not collision-free, **continue**.

3. **Near set.** Denote $n = |V(T)|$ as the number of nodes in the current tree, let

$$\mathcal{N}_n(q_{\text{new}}) \;=\; \{q \in V(T) : d(q, q_{\text{new}}) \leq r_n\}.$$

4. **Choose parent by cost.** Set provisional parent $\bar{q} \leftarrow q_{\text{near}}$ with provisional cost
$$\bar{c} \;=\; c(q_{\text{near}}) + c(q_{\text{near}} \to q_{\text{new}})\,.$$
For each $q \in \mathcal{N}_n(q_{\text{new}})$ such that the segment $[q, q_{\text{new}}]$ is collision-free, if
$$c(q) + c(q \to q_{\text{new}}) < \bar{c},$$
then update $\bar{q} \leftarrow q$, $\bar{c} \leftarrow c(q) + c(q \to q_{\text{new}})$.

5. **Insert.** Add q_{new} to $V(T)$, add edge $(\bar{q} \to q_{\text{new}})$ to $E(T)$, and set $c(q_{\text{new}}) \leftarrow \bar{c}$.

6. **Rewire.** For each $q \in \mathcal{N}_n(q_{\text{new}})$ such that segment $[q_{\text{new}}, q]$ is collision-free, if
$$c(q_{\text{new}}) + c(q_{\text{new}} \to q) < c(q),$$
then **change q's parent to** q_{new} and update $c(q)$ (and, if maintained, the costs of its descendants).

7. **Goal check.** If $q_{\text{new}} \in \mathcal{C}_{\text{goal}}$, record/update the current best path.

Stopping: Terminate upon time/iteration budget; return the best path found (its cost decreases and converges to the optimal value almost surely).

Intuition of why RRT* is optimal:

- The shrinking neighbor radius $r_n \propto (\log n/n)^{1/d}$ ensures that, as n grows, the graph locally approximates the geometry of $\mathcal{C}_{\text{free}}$ with just enough connectivity to avoid fragmentation.
- Parent selection and rewiring implement **dynamic programming** over this increasingly dense graph, continually lowering costs.
- Under regularity and δ-clearance, any optimal path can be covered by a sequence of balls of radius r_n; with high probability the tree includes nodes in each ball, enabling a piecewise improvement that converges to the optimal path cost.

Common Enhancements (Keep Optimality, Speed It Up):

- **Informed RRT***: after the first solution of cost $C^\star$, sample within the informed subset that can still achieve cost $< C^\star$.
- **Batch variants (BIT*)**: combine sampling with incremental graph search for faster convergence.
- **Pruning**: discard nodes whose best possible heuristic cannot beat the current best cost.

Example 4.8 (RRT* for 2D Kinematic Motion Planning)**.** We can apply RRT* to find collision-free paths from a start position to a goal region.

You can play with the code here.

The code provides an implementation of RRT* from scratch. For an off-the-shelf implementation of RRT* and other motion planning algorithms, we recommend the Open Motion Planning Library (OMPL).

4.6.2 Kinodynamic Motion Planning

Problem Setup. Let the *state space* be $\mathcal{X} \subset \mathbb{R}^d$ and the *control space* be $\mathcal{U} \subset \mathbb{R}^m$. The robot obeys differential constraints

$$\dot{x}(t) = f(x(t), u(t)), \qquad u(t) \in \mathcal{U},$$

with f locally Lipschitz in x and measurable in u. Obstacles occupy $\mathcal{X}_{\text{obs}} \subset \mathcal{X}$; the free space is

$$\mathcal{X}_{\text{free}} = \mathcal{X} \setminus \mathcal{X}_{\text{obs}}.$$

Given $x_{\text{start}} \in \mathcal{X}_{\text{free}}$ and a goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$, we seek a *feasible trajectory*

$$x : [0, T] \to \mathcal{X}_{\text{free}}, \quad u : [0, T] \to \mathcal{U}, \quad \dot{x} = f(x, u),\ x(0) = x_{\text{start}},\ x(T) \in \mathcal{X}_{\text{goal}}.$$

For optimal kinodynamic planning, we seek to minimize the additive cost

$$J[x, u] \;=\; \phi(x(T)) \;+\; \int_0^T \ell(x(t), u(t))\, dt,$$

e.g., minimum time ($\ell \equiv 1$), energy, or path-length surrogates.

As in the kinematic case, we grow a tree—now in *state space*—whose edges are *dynamically feasible rollouts.*

The RRT* algorithm for kinodynamic planning requires the following components.

State sampler. Draw states in free space with optional goal bias:

$$x_{\text{rand}} \sim \text{Uniform}(\mathcal{X}_{\text{goal}}) \text{ w.p. } p_{\text{goal}}, \quad x_{\text{rand}} \sim \text{Uniform}(\mathcal{X}_{\text{free}}) \text{ otherwise.}$$

Must support membership tests $x \in \mathcal{X}_{\text{free}}$ and $x \in \mathcal{X}_{\text{goal}}$.

Metric / heuristic. A distance $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ guides nearest-neighbor queries and "move-toward" decisions.

- Weighted Euclidean in a local chart (with angle wrap-around) is common:

$$d(x_i, x_j) = \|W(x_i - x_j)\|_2.$$

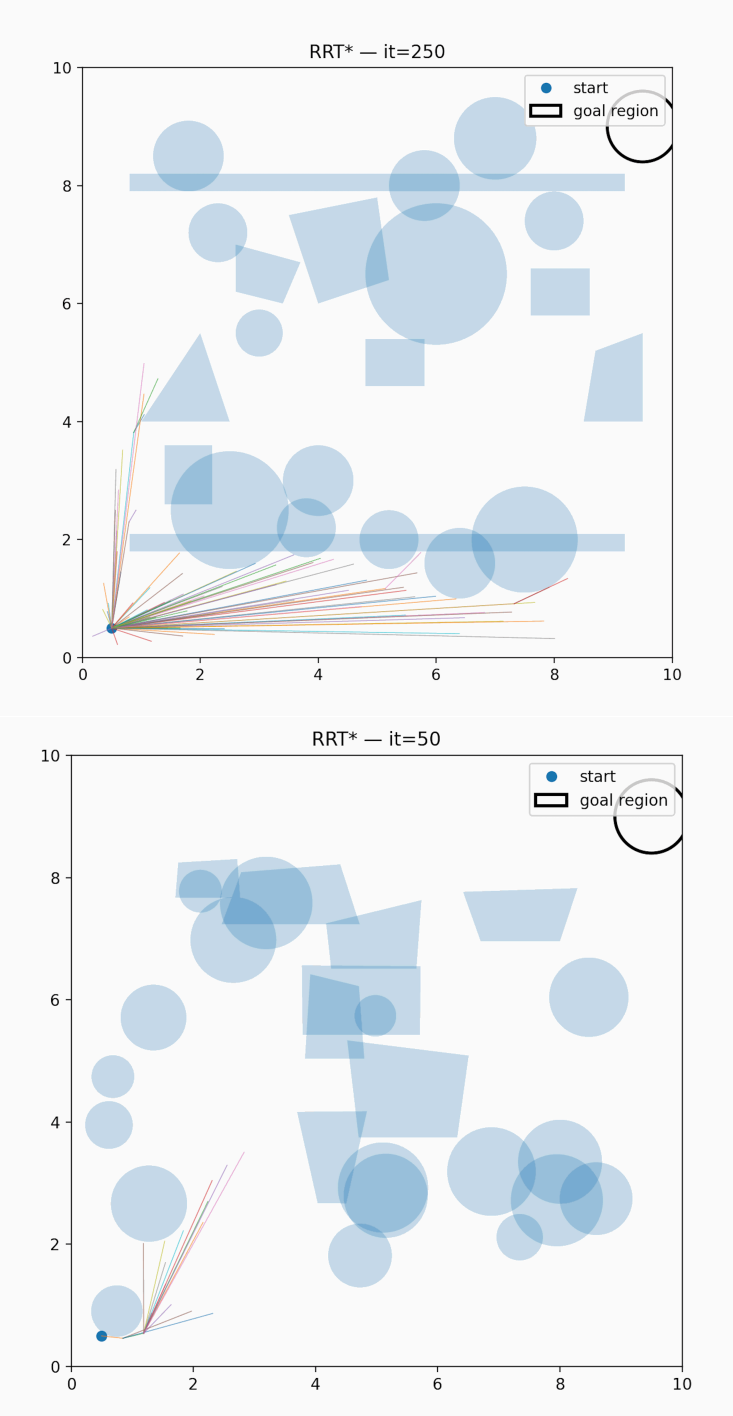

Figure 4.17: RRT* finding collision-free paths.

- Prefer system-aware local distances when available (e.g., Dubins/Reeds–Shepp, double integrator, LQR).

Example 4.9 (Distance Metric for Dubins Car)**.** In kinodynamic planning, two states that are close in Euclidean norm can still be far in terms of *actual time/effort to move between them under the robot's dynamics.* A system-aware local distance replaces a geometric norm

$$\|x_i - x_j\|$$

with a *control distance*

$$d_f(x_i, x_j) \;:=\; \inf_{u(\cdot),\,T} \Big\{ \int_0^T \ell(x(t), u(t))\, dt \;\Big|\; \dot{x} = f(x,u),\; x(0) = x_i,\; x(T) = x_j \Big\},$$

i.e., the **optimal local connection cost** (time, path length, energy) for your specific system f. This distance respects:

- **Nonholonomy** (e.g., car cannot sidestep).
- **Bounds** (e.g., max turn rate, min turning radius, velocity limits).
- **Asymmetry** (forward-only systems: going from x_i to x_j may cost more than reverse).

In practice we use closed-form local connectors when they exist or fast approximations (e.g., LQR quadratic form near an operating point). When nothing closed-form exists, we keep Euclidean for indexing but score candidates with a rollout-based cost.

For example, we consider the Dubins car system.

Model & constraints.

- State $q = (x, y, \theta) \in \mathbb{R}^2 \times \mathbb{S}^1$.
- Controls: constant forward speed $v > 0$; turn rate ω with $|\omega| \le \omega_{\max}$.
- Min turning radius $R = v/\omega_{\max}$.
- Kinematics:

$$\dot{x} = v\cos\theta, \quad \dot{y} = v\sin\theta, \quad \dot{\theta} = \omega, \quad |\omega| \le \omega_{\max}, \quad v \ge 0.$$

- **No reverse gear** (nonnegative speed).

Shortest feasible paths. Between two poses q_s, q_g (with no obstacles), the shortest path with curvature bound $1/R$ is a sequence of at most **three primitives**:

- **C**: constant-curvature arc (left L or right R) of radius R,
- **S**: straight segment.

Only six **word types** can be optimal:

$$\mathcal{W} = \{\mathbf{LSL}, \mathbf{RSR}, \mathbf{LSR}, \mathbf{RSL}, \mathbf{RLR}, \mathbf{LRL}\}.$$

Normalization. Translate/rotate so q_s is at the origin with heading 0. Scale coordinates by R (so the min radius becomes 1). Let the normalized goal be $(\bar{x}, \bar{y}, \bar{\theta})$.

Closed-form lengths. Each word has analytic formulas for the three segment lengths (t, p, q) (two arc angles and a straight length) derived from the circle tangency geometry; e.g., for **LSL**:

$$\begin{aligned}
&\text{Let } d = \sqrt{\bar{x}^2 + \bar{y}^2},\ \phi = \operatorname{atan2}(\bar{y}, \bar{x}),\\
&\quad t = \operatorname{mod}_{2\pi}(-\phi),\\
&\quad p = \sqrt{\max(0,\ d^2 - 2 + 2\cos(\bar{\theta} - \phi))}\ \ \text{(straight length)},\\
&\quad q = \operatorname{mod}_{2\pi}(\bar{\theta} - \phi),
\end{aligned}$$

The **Dubins length** is the minimum total length over all valid words:

$$L_{\text{Dubins}}(q_s, q_g) \ = \ R \min_{\text{word} \in \mathcal{W}} \ (t + p + q),$$

and with constant v, minimum time is $T_{\text{Dubins}} = L_{\text{Dubins}}/v$.

The Reeds–Shepp car model extends the Dubins car model to allow reverse gear.

Nearest neighbor. Given vertices $V(T) \subset \mathcal{X}$,

$$x_{\text{near}} \in \arg\min_{x \in V(T)} d(x,\, x_{\text{rand}}).$$

Use kd/ball trees or Approximate Nearest Neighbor (ANN) structures in a consistent local chart.

Forward "steering" (local planner). When no closed-form two-point connector exists, *forward propagate* from x_{near} by sampling controls and durations and integrating:

$$u \sim \mathcal{D}_u, \quad \tau \sim \mathcal{D}_\tau, \quad x^+(t) = \Phi(x_{\text{near}}, u, t),\ t \in [0, \tau],$$

where Φ is the numerical flow (e.g., RK4 with step h). Here $\mathcal{D}_u$ is the probability distribution over controls used to sample a candidate open-loop input (e.g., uniform over a bounded box, Gaussian around a heuristic, a discrete set of motion primitives, or biased by a policy); and $\mathcal{D}_\tau$ is the distribution over rollout durations (e.g., uniform on $[0, \tau_{\max}]$, a fixed shortlist of durations, or a geometric/exponential prior favoring short motions). Keep a rollout only if the *entire* trajectory stays in $\mathcal{X}_{\text{free}}$. The edge cost is

$$c(x_{\text{near}} \to x_{\text{new}}) = \int_0^\tau \ell(x^+(t), u)\, dt.$$

If a steering function exists for your system (e.g., Dubins, LQR-QR), use it in place of random propagation.

Collision checking (trajectory validity). Validate the continuous rollout (time-discretized sufficiently for geometry and dynamics).

Neighbor set for rewiring. With $n = |V(T)|$, use the RRT* radius schedule in **state dimension** d:

$$r_n = \gamma\Big(\tfrac{\log n}{n}\Big)^{1/d} \quad \text{or} \quad k\text{-nearest with } k_n \geq k_0 \log n.$$

Only neighbors reachable by a feasible local planner are considered connectable.

Costs. Each node stores cost-to-come $c(x)$ (integral of ℓ along the tree path). Parent selection and rewiring compare *integrated* edge costs from actual feasible rollouts (or optimal local connectors).

Stopping / success. Success when a node lies in $\mathcal{X}_{\text{goal}}$. As an anytime method, continue to improve cost by rewiring until time/iteration budget is exhausted.

The following pseudocode presents RRT* for kinodynamic motion planning.

Algorithm: RRT* — Kinodynamic Case

Inputs: initial state $x_{\text{start}} \in \mathcal{X}_{\text{free}}$; goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$; dynamics f; metric d; running cost ℓ, terminal cost ϕ; goal-bias $p_{\text{goal}} \in [0,1]$; budget N; neighbor schedule $r_n = \gamma(\frac{\log n}{n})^{1/d}$ (or $k_n \geq k_0 \log n$); forward-propagation integrator and control/duration samplers (or a system-specific steering method).

Initialize T: $V(T) \leftarrow \{x_{\text{start}}\}$, $E(T) \leftarrow \emptyset$, $c(x_{\text{start}}) = 0$.

For $i = 0, 1, 2, \ldots, N-1$

1. **Goal-biased sampling.** Draw x_{rand} from $\mathcal{X}_{\text{goal}}$ with probability p_{goal}, else from $\mathcal{X}_{\text{free}}$.
2. **Nearest.** Select $x_{\text{near}} \in \arg\min_{x \in V(T)} d(x,\, x_{\text{rand}})$.
3. **Forward "steer".** Generate a batch $\{(u_k, \tau_k)\}_{k=1}^{K}$, integrate $\dot{x} = f(x, u_k)$ from x_{near} up to τ_k, discard infeasible rollouts, and pick a candidate terminal x_{new} (e.g., closest to x_{rand} or lowest edge cost). If none feasible, **continue**.
4. **Near set.** With $n = |V(T)|$,
$$\mathcal{N}_n(x_{\text{new}}) = \{x \in V(T) : d(x, x_{\text{new}}) \leq r_n\}.$$
5. **Choose parent by cost (feasible-only).** Provisional parent $\bar{x} \leftarrow x_{\text{near}}$ with provisional cost
$$\bar{c} = c(x_{\text{near}}) + c(x_{\text{near}} \to x_{\text{new}}).$$

For each $x \in \mathcal{N}_n(x_{\text{new}})$, attempt a feasible local connection $x \to x_{\text{new}}$. If feasible and

$$c(x) + c(x \to x_{\text{new}}) < \bar{c},$$

set $\bar{x} \leftarrow x$ and $\bar{c}$ accordingly.

6. **Insert.** Add x_{new} to $V(T)$; add edge $(\bar{x} \to x_{\text{new}})$ (storing its control/duration); set $c(x_{\text{new}}) = \bar{c}$.

7. **Rewire (forward-only).** For each $x \in \mathcal{N}_n(x_{\text{new}})$, attempt a feasible connection $x_{\text{new}} \to x$. If

$$c(x_{\text{new}}) + c(x_{\text{new}} \to x) < c(x),$$

then **change x's parent to** x_{new} and update costs (and, if maintained, descendants).

8. **Goal check.** If $x_{\text{new}} \in \mathcal{X}_{\text{goal}}$, record/update the current best trajectory.

Stopping: On budget exhaustion, return the best feasible trajectory $(x(\cdot), u(\cdot))$ found.

Practical Notes.

- *Local planner.* Replace random forward propagation with **analytic/local-optimal connectors** when available (Dubins/Reeds–Shepp, LQR for linearized systems, double integrator), which dramatically accelerates convergence.
- *Hyperparameters.* Batch size K, duration bounds $\tau_{\max}$, and integrator step h balance exploration vs. fidelity; smaller h and moderate τ help in tight dynamics/geometry.
- *Heuristics.* Informed sampling (admissible lower bounds on remaining cost), pruning by lower-bound estimates, and k-nearest neighborhoods can maintain optimality while improving speed.

Asymptotic Optimality. Under standard assumptions—δ-robust feasibility, absolutely continuous sampling over $\mathcal{X}_{\text{free}}$, locally Lipschitz dynamics, nonnegative Lipschitz running cost, and a *consistent* local planner (feasible connections that approximate short motions)—the best trajectory cost produced by kinodynamic RRT* converges almost surely to the global optimum as $N \to \infty$. Intuitively, the shrinking neighborhood $r_n \propto (\log n/n)^{1/d}$ yields a graph that becomes dense in the reachable set, and parent selection plus rewiring implement dynamic programming over this increasingly rich local-connectivity structure.

Common Enhancements (keep optimality).

- **Informed RRT*** with kinodynamic admissible heuristics (e.g., minimum-time or minimum-energy lower bounds).
- **Batch planners (e.g., BIT*)** that merge sampling and search with system-aware heuristics.
- **Reachability and cost-based pruning** to focus computation where improvements are still possible.

Example 4.10 (RRT* for Dubins Car)**.** Continuning Example 4.9, we apply RRT* to collision-free kinodynamic path planning for the Dubins car model.

Fig. 4.18 shows the RRT* algorithm's running process.

You can play with the code here.

A similar YouTube demonstration video can be found here.

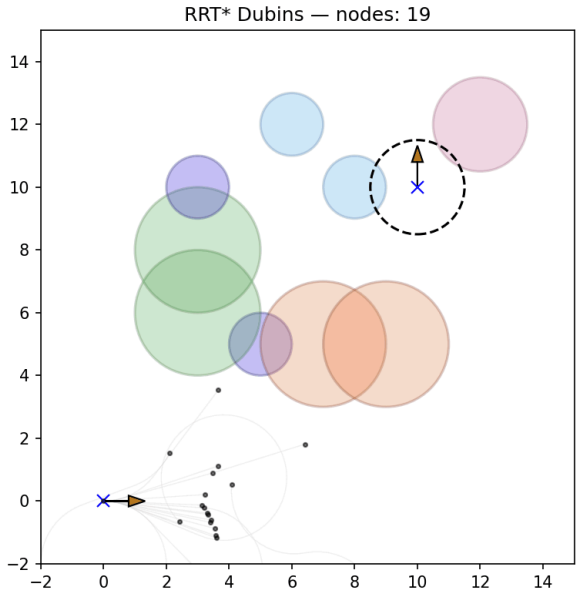

Figure 4.18: RRT* for Dubins Car.

Chapter 5

Advanced Materials

Appendix A

Convex Analysis and Optimization

A.1 Theory

A.1.1 Sets

Convex set is one of the most important concepts in convex optimization. Checking convexity of sets is crucial to determining whether a problem is a convex problem. Here we will present some definitions of some set notations in convex optimization.

Definition A.1 (Affine set)**.** A set $C \subset \mathbb{R}^n$ is affine if the line through any two distinct points in C lies in C, i.e., if for any $x_1, x_2 \in C$ and any $\theta \in \mathbb{R}$, we have $\theta x_1 + (1-\theta)x_2 \in C$.

Definition A.2 (Convex set)**.** A set $C \subset \mathbb{R}^n$ is convex if the line segment between any two distinct points in C lies in C, i.e., if for any $x_1, x_2 \in C$ and any $\theta \in [0,1]$, we have $\theta x_1 + (1-\theta)x_2 \in C$.

Definition A.3 (Cone)**.** A set $C \subset \mathbb{R}^n$ is a cone if for any $x \in C$ and any $\theta \geq 0$, we have $\theta x \in C$.

Definition A.4 (Convex Cone)**.** A set $C \subset \mathbb{R}^n$ is a convex cone if C is convex and a cone.

Below are some important examples of convex sets:

Definition A.5 (Hyperplane)**.** A hyperplane is a set of the form

$$\{x | a^T x = b\}$$

Definition A.6 (Halfspaces)**.** A (closed) halfspace is a set of the form

$$\{x|a^T x \leq b\}$$

Definition A.7 (Balls)**.** A ball is a set of the form

$$B(x,r) = \{y|\|y-x\|_2 \leq r\} = \{x+ru|\|u\|_2 \leq 1\}$$

where $r > 0$.

Definition A.8 (Ellipsoids)**.** A ellipsoid is a set of the form

$$\mathcal{E} = \{y|(y-x)^T P^{-1}(y-x) \leq 1\}$$

where P is symmetric and positive definite.

Definition A.9 (Polyhedra)**.** A polyhedra is defined as the solution set of a finite number of linear equalities and inequalities:

$$\mathcal{P} = \{x|a_j^T x \leq b_j, j = 1,...,m, c_k^T x = d_k, k = 1,...,p\}$$

Definition A.10 (Norm ball)**.** A norm ball B of radius r and a center x_c associated with the norm $\|\cdot\|$ is defined as:

$$B = \{x|\|x - x_c\| \leq r\}$$

Definition A.11 (Norm cone)**.** A norm cone C associated with the norm $\|\cdot\|$ is defined as:

$$C = \{(x,t)|\|x\| \leq t\} \subset \mathbb{R}^{n+1}$$

Simplexes are important family of polyhedra. Suppose the $k+1$ points $v_0,...,v_k \in \mathbb{R}^n$ are affinely independent, which means $v_1 - v_0,...,v_k - v_0$ are linearly independent.

Definition A.12 (Simplex)**.** A simplex C defined by points $v_0,...,v_k$ is:

$$C = \mathbf{conv}\{v_0,...,v_k\} = \{\theta_0 v_0 + ...\theta_k v_k|\theta \succeq 0, \mathbf{1}^T\theta = 1\}$$

Extremely important examples of convex sets are positive semidefinite cones:

Definition A.13 (Symmetric,positive semidefinite,positive definite matrices)**.**

1. Symmetric matrices: $\mathbf{S}^n = \{X \in \mathbb{R}^{n\times n}|X = X^T\}$
2. Symmetric Positive Semidefinite matrices: $\mathbf{S}_+^n = \{X \in \mathbf{S}^n|X \succeq 0\}$
3. Symmetric Positive definite matrices: $\mathbf{S}_{++}^n = \{X \in \mathbf{S}^n|X \succ 0\}$

In most scenarios, the set we encounter is more complicated. In general it is extermely hard to determine whether a set in convex or not. But if the set is 'generated' by some convex sets, we can easily determine its convexity. So let's focus on operations that preserve convexity:

Proposition A.1. *Assume S is convex, $S_\alpha, \alpha \in \mathcal{A}$ is a family of convex sets. Following operations on convex sets will preserve convexity:*

1. *Intersection: $\bigcap_{\alpha\in\mathcal{A}} S_\alpha$ is convex.*
2. *Image under affine function: A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is affine if it has the form $f(x) = Ax+b$. The image of S under affine function f is convex. I.e. $f(S) = \{f(x)|x \in S\}$ is convex*
3. *Image under perspective function: We define the perspective function $P : \mathbb{R}^{n+1}$, with domain $\mathbf{dom}P = \mathbb{R}^n \times \mathbb{R}_{++}$(where $\mathbb{R}_{++} = \{x \in \mathbb{R}|x > 0\}$) as $P(z,t) = z/t$. The image of S under perspective function is convex.*
4. *Image under linear-fractional function: We define linear fractional function $f : \mathbb{R}^n \to \mathbb{R}^m$ as:$f(x) = (Ax+b)/(c^Tx+d)$ with $\mathbf{dom}f = \{x|c^Tx+d > 0\|$. The image of S under linear fractional functions is convex.*

In some cases, the restrictions of **interior** is too strict. For example, imagine a plane in $\mathbb{R}^3$. The interior of the plane is $\emptyset$. But intuitively many property should be extended to this kind of situation. Because the points in the plane also lies 'inside' the convex set. Thus, we will define **relative interior**. First we will define **affine hull**.

Definition A.14 (Affine hull)**.** The affine hull of a set S is the smallest affine set that contains S, which can be written as:

$$\text{aff}(S) = \{\sum_{i=1}^{k} \alpha_i x_i | k > 0, x_i \in S, \alpha_i \in \mathbb{R}, \sum_{i=1}^{k} \alpha_i = 1\}$$

Definition A.15 (Relative Interior)**.** The relative interior of a set S (denoted relint(S)) is defined as its interior within the affine hull of S. I.e.

$$\text{relint}(S) := \{x \in S : \text{there exists } \epsilon > 0 \text{ such that } N_\epsilon \cap \text{aff}(S) \subset S\}$$

where $N_\epsilon(x)$ is a ball of radius ϵ centered on x.

A.1.2 Convex function

In this section, let's define convex functions:

Definition A.16 (Convex function)**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is **convex** if **dom** f is convex and $\forall x, y \in$ **dom** f and with $\theta \in [0,1]$, we have:

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$$

The function is **strictly convex** if the inequality holds whenever $x \neq y$ and $\theta \in (0,1)$.

If a function is differentiable, it will be easier for us to check its convexity:

Proposition A.2 (Conditions for Convex function)**.** *1.(First order condition) Suppose f is differentiable, then f is convex if and only if $\boldsymbol{domf}$ is convex and $\forall x, y \in \boldsymbol{domf}$,*

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

2.(Second order conditions) Suppose f is twice differentiable, then f is convex if and only if $\boldsymbol{domf}$ is convex and $\forall x \in \boldsymbol{domf}$,

$$\nabla^2 f(x) \succeq \boldsymbol{0}$$

For the same purpose, some operations that preserve the convexity of the convex functions are presented here:

Proposition A.3 (Operations that preserve convexity)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a convex function and $g_1, ..., g_n$ be convex functions. The following operations will preserve convexity of the function:*

1.(Nonnegative weighted sum): A nonnegative weighted sum of convex functions:

$$f = \omega_1 f_1 + ... + \omega_m f_m$$

2.(Composition with an affine mapping) Suppose $A \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$, then $g(x) = f(Ax + b)$ is convex.

3.(Pointwise maximum and supremum) $g(x) = \max\{g_1(x), ..., g_n(x)\}$ is convex. If $h(x, y)$ is convex in x for each $y \in \mathcal{A}$, then $\sup_{y \in \mathcal{A}} h(x, y)$ is also convex in x.

4.(Minimization) If $h(x, y)$ is convex in (x, y), and C is a convex nonempty set, then $\inf_{x \in C} h(x, y)$ is convex in x.

5.(Perspective of a function) The perspective of f is the function $h : \mathbb{R}^{n+1} \to \mathbb{R}$ defined by: $h(x, t) = tf(x/t)$ with domain $\boldsymbol{dom}\ h = \{(x, t) | x/t \in \boldsymbol{domf}, t > 0\}$. And h is convex.

A.1.3 Lagrange dual

We consider an optimization problem in the standard form (without assuming convexity of anything):

$$\begin{aligned} p^* = \quad & \min_x \quad f_0(x) \\ & \text{s.t.} \quad f_i(x) \leq 0 \quad i = 1..., m \\ & \qquad h_i(x) = 0 \quad i = 1, ..., p \end{aligned} \tag{A.1}$$

Definition A.17 (Lagrange dual function)**.** The Lagrangian related to the problem above is defined as:

$$L(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x)$$

The Lagrange dual function is defined as:

$$g(\lambda, \nu) = \inf_{x \in \mathcal{D}} L(x, \lambda, \nu)$$

When the Lagrangian is unbounded below in x, the dual function takes on the value $-\infty$. Note that since the Lagrange dual function is a pointwise infimum of a family of affine functions of (λ, ν), so it's concave. The Lagrange dual function will give us lower bounds of the optimal value of the original problem:

$$g(\lambda, \nu) \leq p^*$$

. We can see that, the dual function can give a nontrivial lower bound only when $\lambda \succeq 0$. Thus we can solve the following dual problem to get the best lower bound.

Definition A.18 (Lagrange dual problem)**.** The lagrangian dual problem is defined as follows:

$$\begin{aligned} d^* = \quad & \max_{\lambda, \nu} \quad g(\lambda, \nu) \\ & \text{s.t.} \quad \lambda \succeq 0 \end{aligned} \tag{A.2}$$

This is a convex optimization problem.

We can easily see that

$$d^* \leq p^*$$

always hold. This property is called **weak duality**. If

$$d^* = p^*$$

, it's called **strong duality**. Strong duality does not hold in general, but it usually holfs for convex problems. We can find conditions that guarantee strong duality in convex problems, which are called constrained qualifications. Slater's constraint qualification is a useful one.

Theorem A.1 (Slater's constraint qualification)**.** *Strong duality holds for a convex problem*

$$\begin{aligned} p^* = \quad & \min_{x} \quad f_0(x) \\ & s.t. \quad f_i(x) \leq 0 \quad i = 1..., m \\ & \qquad Ax = b \end{aligned} \tag{A.3}$$

if it is strictly feasible, i.e.

$$\exists x \in \textbf{\textit{relint}}\mathcal{D} : \quad f_i(x) < 0, \quad i = 1...m, \quad Ax = b$$

And the linear inequalities do not need to hold with strict inequality.

A.1.4 KKT condition

Note that if strong duality holds, denote x^* to be primal optimal, and (λ^*, ν^*) to be dual optimal. Then:

$$\begin{aligned} f_0(x^*) = g(\lambda^*, \nu^*) =& \inf_x (f_0(x) + \sum_{i=1}^m \lambda_i^* f_i(x) + \sum_{i=1}^p \nu_i^* h_i(x)) \\ \leq& f_0(x^*) + \sum_{i=1}^m \lambda_i^* f_i(x) + \sum_{i=1}^p \nu_i^* h_i(x) \\ \leq& f_0(x^*) \end{aligned} \tag{A.4}$$

from this, combining $\lambda^* \geq 0$ and $f_i(x^*) \leq 0$, we can know that: $\lambda_i^* f_i(x^*) = 0 \quad i = 1 \cdots m$. This means for λ_i^* and $f_i(x^*)$, one of them must be zero, which is known as complementary slackness).

Thus we arrived at the following four conditions, which are called KKT conditions.

Theorem A.2 (Karush-Kuhn-Tucker(KKT) Conditions)**.** *The following four conditions are called KKT conditions (for a problem with differentiable f_i, h_i)*

1. *Primal feasible:* $f_i(x) \leq 0, i, \cdots, m,\ h_i(x) = 0, i = 1, \cdots, p$
2. *Dual feasible:* $\lambda \succeq 0$
3. *Complementary slackness:* $\lambda_i f_i(x) = 0, i = 1, \cdots, m$
4. *Gradient of Lagrangian with respect to x vanishes:* $\nabla f_0(x) + \sum_{i=1}^m \lambda_i \nabla f_i(x) + \sum_{i=1}^p \nu_i \nabla h_i(x) = 0$

From the discussion above, we know that if strong duality holds and x, λ, ν are optimal, then they must satisfy the KKT conditions.

Also if x, λ, ν satisfy KKT for a convex problem, then they are optimal. However, the converse is not generally true, since KKT condition implies strong duality. If Slater's condition is satisfied, then x is optimal if and only if there exist λ, ν that satisfy KKT conditions. Sometimes, by solving the KKT system, we can derive the closed-form solution of a optimization directly. Also, sometimes we will use the residual of the KKT system as the termination condition.

In general, f_i, h_i may not be differentiable. There are also KKT conditions for them, which will include knowledge of subdifferential and will not be included here.

A.2 Practice

A.2.1 CVX Introduction

In the last section, we have learned basic concepts and theorems in convex optimization. In this section, on the other hand, we will introduce you how to model basic convex optimization problems with CVX, an easy-to-use MATLAB package. To install CVX, please refer to this page. Note that every time you what to use the CVX package, you should add it to your MATLAB path. For example, if I install CVX package in the parent directory of my current directory with default directory name `cvx`, the following line should be added before your CVX codes:

```
addpath(genpath("../cvx/"));
```

With CVX, it is incredibly easy for us to define and solve a convex optimization problem. You just need to:

1. define the variables.
2. define the objective function you want to minimize or maximize.
3. define the constraints.

After running your codes, the optimal objective value is stored in the variable `cvx_optval`, and the problem status is stored in the variable `cvx_status` (when your problem is well-defined, this variable's value will be `Solved`). The optimal solutions will be stored in the variables you define.

Throughout this section, we will study five types of convex optimization problems: linear programming (LP), quadratic programming (QP), (convex) quadratically constrained quadratic programming (QCQP), second-order cone programming (SOCP), and semidefinite programming (SDP). Given two types of optimization problems A and B, we say $A < B$ if A can always be converted to B while the inverse is not true. Under this notation, we have

$$\text{LP} < \text{QP} < \text{QCQP} < \text{SOCP} < \text{SDP}$$

A.2.2 Linear Programming (LP)

Definition. An LP has the following form:

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} \ c^T x \\ &\text{subject to } Ax \leq b \end{aligned} \tag{A.5}$$

where x is the variable, $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$ are the parameters. Note that the constraint $Ax \leq b$ already incorporates linear equality constraints. To see this, consider the constraint $A'x = b'$, we can reformulate it as $Ax \leq b$ by

$$\begin{bmatrix} A' \\ -A' \end{bmatrix} x \leq \begin{bmatrix} b' \\ -b' \end{bmatrix}$$

Example. Consider the problem of minimizing a linear function $c_1 x_1 + c_2 x_2$ over a rectangle $[-l_1, l_1] \times [-l_2, l_2]$. We can convert it to the standard LP form in (A.5) by simply setting c as $[c_1,\ c_2]^T$ and the linear inequality constraint as

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} l_1 \\ l_1 \\ l_2 \\ l_2 \end{bmatrix}$$

Corresponding CVX codes are shown below:

```
%% Define the LP example setting
c1 = 2;
c2 = -5;
l1 = 3;
l2 = 7;
% parameters: c, A, b
c = [c1; c2];
A = [1, 0; -1, 0; 0, 1; 0, -1];
b = [l1; l1; l2; l2];

%% solve LP
cvx_begin
    variable x(2); % define variables [x1, x2]
    minimize(c' * x); % define the objective
    subject to
        A * x <= b; % define the linear constraint
cvx_end
```

A.2.3 Quadratic Programming (QP)

Definition. A QP has the following form:

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} x^T P x + q^T x \tag{A.6}$$

$$\text{subject to } Gx \leq h \tag{A.7}$$

$$Ax = b \tag{A.8}$$

where $P \in \mathcal{S}_+^n, q \in \mathbb{R}^n, G \in \mathbb{R}^{m\times n}, h \in \mathbb{R}^m, A \in \mathbb{R}^{p\times n}, b \in \mathbb{R}^p$. Here $\mathcal{S}_+^n$ denotes the set of positive semidefinite matrices of size $n \times n$. Obviously, if we set P as zero, QP will degenerate to LP.

Example. Consider the problem of minimizing a quadratic function

$$f(x_1, x_2) = p_1 x_1^2 + 2p_2 x_1 x_2 + p_3 x_2^2 + q_1 x_1 + q_2 x_2$$

over a rectangle $[-l_1, l_1] \times [-l_2, l_2]$. Since $P = 2\begin{bmatrix} p_1 & p_2 \\ p_2 & p_3 \end{bmatrix} \succeq 0$, the following two conditions must hold:

$$\begin{cases} p_1 \geq 0 \\ p_1 p_3 - 4p_2^2 \geq 0 \end{cases}$$

Same as in the LP example, G and h can be expressed as:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} l_1 \\ l_1 \\ l_2 \\ l_2 \end{bmatrix}$$

Corresponding CVX codes are shown below:

```
%% Define the QP example setting
p1 = 2;
p2 = 0.5;
p3 = 4;
q1 = -3;
q2 = -6.5;
l1 = 2;
l2 = 2.5;
% check if the generated P is positive semidefinite
tmp1 = (p1 >= 0);
tmp2 = (p1*p3 - 4*p2^2 >= 0);
if ~(tmp1 && tmp2)
    error("P is not positve semidefinite!");
end
% parameters: P, q, G, h
P = 2 * [p1, p2; p2, p3];
q = [q1; q2];
G = [1, 0; -1, 0; 0, 1; 0, -1];
h = [l1; l1; l2; l2];

%% Solve the QP problem
cvx_begin
    variable x(2); % define variables [x1; x2]
```

```
    % define the objective, where quad_form(x, P) = x'*P*x
    obj = 0.5 * quad_form(x, P) + q' * x;
    minimize(obj);
    subject to
        G * x <= h; % define the linear constraint
cvx_end
```

A.2.4 Quadratically Constrained Quadratic Programming (QCQP)

Definition. An (convex) QCQP has the following form:

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}x^T P_0 x + q_0^T x \tag{A.9}$$

$$\text{subject to } \frac{1}{2}x^T P_i x + q_i^T x + r_i \leq 0,\ i = 1 \dots m \tag{A.10}$$

$$Ax = b \tag{A.11}$$

where $P_i \in \mathcal{S}_+^n, i = 0 \dots m$, $q_i \in \mathbb{R}^n, i = 0 \dots m$, $A \in \mathbb{R}^{p\times n}$, and $b \in \mathbb{R}^p$. Note that in other literature, you may find a more general form of QCQP: they don't require P_i's to be positive semidefinite. Yet in this case, the problem is non-convex and beyond our scope.

Example. We study the problem of getting the minimum distance between two ellipses. By convention, when the ellipses overlap, we set the minimum distance as 0. This problem can be exactly solved by (convex) QCQP. Consider two ellipses of the following form:

$$\begin{cases} \frac{1}{2}\begin{bmatrix} y_1 \\ z_1 \end{bmatrix}^T K_1 \begin{bmatrix} y_1 \\ z_1 \end{bmatrix} + k_1^T \begin{bmatrix} y_1 \\ z_1 \end{bmatrix} + c_1 \leq 0 \\ \frac{1}{2}\begin{bmatrix} y_2 \\ z_2 \end{bmatrix}^T K_2 \begin{bmatrix} y_2 \\ z_2 \end{bmatrix} + k_2^T \begin{bmatrix} y_2 \\ z_2 \end{bmatrix} + c_2 \leq 0 \end{cases}$$

where $[y_1, z_1]^T$ and $[y_2, z_2]^T$ are arbitrary points inside the two ellipses respectively. Also, two ensure the ellipses are well defined, we should enforce the following properties in $(K_i, k_i, c_i), i = 1, 2$: (1) $K_i \succ 0$; (2) Let $K_i = L_i L_i^T$ be the Cholesky decomposition of K_i. Then, ellipse i can be rewritten as:

$$\frac{1}{2} \parallel L_i^T \begin{bmatrix} y_i \\ z_i \end{bmatrix} - L_i^{-1} k_i \parallel^2 \leq \frac{1}{2} \parallel L_i^{-1} k_i \parallel^2 - c_i$$

Thus,

$$\frac{1}{2} \parallel L_i^{-1} k_i \parallel^2 - c_i > 0$$

With these two assumptions, we want to minimize:

$$\frac{1}{2}(y_1 - y_2)^2 + (z_1 - z_2)^2$$

Now, we construct P, q, r's in QCQP with the above parameters. Define the variable x as $[y_1, z_1, y_2, z_2]$.

(1) P_0 can be obtained from:

$$\frac{1}{2}(y_1 - y_2)^2 + (z_1 - z_2)^2 = \frac{1}{2}\begin{bmatrix} y_1 \\ z_1 \\ y_2 \\ z_2 \end{bmatrix}^T \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ z_1 \\ y_2 \\ z_2 \end{bmatrix}$$

(2) P_1, q_1, r_1 can be obtained from:

$$\frac{1}{2}\begin{bmatrix} y_1 \\ z_1 \end{bmatrix}^T K_1 \begin{bmatrix} y_1 \\ z_1 \end{bmatrix} + k_1^T \begin{bmatrix} y_1 \\ z_1 \end{bmatrix} + c_1 = \frac{1}{2}x^T \begin{bmatrix} K_1 & O \\ O & O \end{bmatrix} + \begin{bmatrix} k_1 \\ O \end{bmatrix}^T x + c_1 \leq 0$$

(3) P_2, q_2, r_2 can be obtained from:

$$\frac{1}{2}\begin{bmatrix} y_2 \\ z_2 \end{bmatrix}^T K_2 \begin{bmatrix} y_2 \\ z_2 \end{bmatrix} + k_2^T \begin{bmatrix} y_2 \\ z_2 \end{bmatrix} + c_2 = \frac{1}{2}x^T \begin{bmatrix} O & O \\ O & K_2 \end{bmatrix} + \begin{bmatrix} O \\ k_2 \end{bmatrix}^T x + c_2 \leq 0$$

The corresponding codes are shown below. In this example, we test the minimum distance between a circle $y_1^2 + z_1^2 \leq 1$ and another circle $(y_2 - 2)^2 + (z_2 - 2)^2 \leq 1$. You can check whether the result from QCQP aligns with your manual calculation.

```
%% Define the QCQP example setting
K1 = eye(2);
k1 = zeros(2, 1);
c1 = -0.5;
K2 = eye(2);
k2 = [2; 2];
c2 = 3.5;
if ~(if_ellipse(K1, k1, c1) && if_ellipse(K2, k2, c2))
    error("The example setting is not correct");
end
% define parameters P0, P1, P2, q1, q2, r1, r2
P0 = [1,0,-1,0; 0,1,0,-1; -1,0,1,0; 0,-1,0,1];
P1 = zeros(4, 4);
P1(1:2, 1:2) = K1;
```

```
P2 = zeros(4, 4);
P2(3:4, 3:4) = K2;
q1 = [k1; zeros(2, 1)];
q2 = [zeros(2, 1); k2];
r1 = c1;
r2 = c2;

%% Solve the QCQP problem
cvx_begin
    variable x(4); % define variables [y1; z1; y2; z2]
    % define the objective, where quad_form(x, P) = x'*P*x
    obj = 0.5 * quad_form(x, P0);
    minimize(obj);
    subject to
        0.5 * quad_form(x, P1) + q1' * x + r1 <= 0;
        0.5 * quad_form(x, P2) + q2' * x + r2 <= 0;
cvx_end

%% detect whether (K, k, c) generates a ellipse
function flag = if_ellipse(K, k, c)
    L = chol(K);
    radius_square = 0.5 * norm(L \ k)^2 - c; % L \ k = inv(L) * k
    flag = (radius_square > 0);
end
```

A.2.5 Second-Order Cone Programming (SOCP)

Definition. An SOCP has the following form:

$$\min_{x \in \mathbb{R}^n} f^T x \tag{A.12}$$

$$\text{subject to } ||A_i x + b_i||_2 \leq c_i^T x + d_i, \ i = 1 \dots m \tag{A.13}$$

$$Fx = g \tag{A.14}$$

where $f \in \mathbb{R}^n, A_i \in \mathbb{R}^{n_i \times n}, b_i \in \mathbb{R}^{n_i}, c_i \in \mathbb{R}^n, d_i \in \mathbb{R}, F \in \mathbb{R}^{p \times n}$, and $g \in \mathbb{R}^p$.

Example. We consider the problem of stochastic linear programming:

$$\min_x c^T x \tag{A.15}$$

$$\text{subject to } \mathbb{P}(a_i^T x \leq b_i) \geq p, \ i = 1 \dots m \tag{A.16}$$

$$a_i \sim \mathcal{N}(\bar{a}_i, \Sigma_i), \ i = 1 \dots m \tag{A.17}$$

Here p should be more than 0.5. We show that this problem can be converted to a SOCP:

Since $a_i \sim \mathcal{N}(\bar{a}_i, \Sigma_i)$, then $(a_i^T x - b_i) \sim \mathcal{N}(\bar{a}_i^T x - b_i, x^T \Sigma_i x)$. Standardize it:

$$t := ||\Sigma_i^{\frac{1}{2}} x||_2^{-1} \left\{(a_i^T x - b_i) - (\bar{a}_i^T x - b_i)\right\} \sim \mathcal{N}(0, 1)$$

Then,

$$\begin{aligned} \mathbb{P}(a_i^T x \leq b_i) &= \mathbb{P}(a_i^T x - b_i \leq 0) && \text{(A.18)} \\ &= \mathbb{P}(t \leq -||\Sigma_i^{\frac{1}{2}} x||_2^{-1}(\bar{a}_i^T x - b_i)) && \text{(A.19)} \\ &= \Phi(-||\Sigma_i^{\frac{1}{2}} x||_2^{-1}(\bar{a}_i^T x - b_i)) && \text{(A.20)} \end{aligned}$$

Here $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution:

$$\Phi(\xi) = \int_{-\infty}^{\xi} e^{-\frac{1}{2}t^2}\, dt$$

Thus,

$$\begin{aligned} & \mathbb{P}(a_i^T x \leq b_i) \geq p && \text{(A.21)} \\ \Longleftrightarrow & \Phi(-||\Sigma_i^{\frac{1}{2}} x||_2^{-1}(\bar{a}_i^T x - b_i)) \geq p && \text{(A.22)} \\ \Longleftrightarrow & -||\Sigma_i^{\frac{1}{2}} x||_2^{-1}(\bar{a}_i^T x - b_i) \geq \Phi^{-1}(p) && \text{(A.23)} \\ \Longleftrightarrow & \Phi^{-1}(p)||\Sigma_i^{\frac{1}{2}} x||_2 \leq b_i - \bar{a}_i^T x && \text{(A.24)} \end{aligned}$$

which is exactly the same as inequality constraints in SOCP formulation. (You can see why we enforce $p > 0.5$ here: otherwise $\Phi^{-1}(p)$ will be negative and the constraint with not be an second-order cone.)

In the following code example, we set up four inequality constraints and let $\bar{a}_i^T x \leq b_i,\ i = 1 \ldots 4$ form an square located at the origin of size 2. Then, for convenience, we set $\Sigma_i \equiv \sigma^2 I$.

```
%% Define the SOCP example setting
bar_a1 = [1; 0];
b1 = 1;
bar_a2 = [0; 1];
b2 = 1;
bar_a3 = [-1; 0];
b3 = 1;
bar_a4 = [0; -1];
b4 = 1;
sigma = 0.1;
c = [2; 3];
p = 0.9; % p should be more than 0.5
Phi_inv = norminv(p); % get Phi^{-1}(p)
```

```
%% Solve the SOCP problem
cvx_begin
    variable x(2); % define variables [x1; x2]
    minimize(c' * x);
    subject to
        sigma*Phi_inv * norm(x) <= b1 - bar_a1' * x;
        sigma*Phi_inv * norm(x) <= b2 - bar_a2' * x;
        sigma*Phi_inv * norm(x) <= b3 - bar_a3' * x;
        sigma*Phi_inv * norm(x) <= b4 - bar_a4' * x;
cvx_end
```

A.2.6 Semidefinite Programming (SDP)

Definition. An SDP has the following form:

$$\min_{X_i, x_i} \sum_{i=1}^{n_s} C_i \cdot X_i + \sum_{i=1}^{n_u} c_i \cdot x_i \tag{A.25}$$

$$\text{subject to } \sum_{i=1}^{n_s} A_{i,j} \cdot X_i + \sum_{i=1}^{n_u} a_{i,j} \cdot x_i = b_j, \quad j = 1 \dots m \tag{A.26}$$

$$X_i \in \mathcal{S}_+^{D_i}, \quad i = 1 \dots n_s \tag{A.27}$$

$$x_i \in \mathbb{R}^{d_i}, \quad i = 1 \dots n_u \tag{A.28}$$

where $C_i, A_{i,j} \in \mathbb{R}^{D_i \times D_i}$, $c_i, a_{i,j} \in \mathbb{R}^{d_i}$, and $\cdot$ means element-wise product. For two square matrices A, B, the dot product $A \cdot B$ is equal to $\text{tr}(AB)$; for two vectors a, b, the dot product $a \cdot b$ is the same as inner product $a^T b$.

Note that actually there are many "standard" forms of SDP. For example, in the convex optimization theory part, you may find an SDP that looks like:

$$\min_X C \cdot X \tag{A.29}$$

$$\text{subject to } A \cdot X = b \tag{A.30}$$

$$X \succeq 0 \tag{A.31}$$

It is convenient for us to analyze the theoretical properties of SDP with this form. Also, in SDP solvers' User Guide, you may see more complex SDP forms which involve more general convex cones. For example, see MOSEK's MATLAB API docs. Here we turn to use the form of (A.25) for two reasons: (1) it is general enough: our SDP example below can be converted to this form (also, SDPs from sum-of-squares programming in this book are exactly of the form (A.25)); (2) it is more readable than more complex forms.

Example. We consider the problem of finding the minimum eigenvalue for a positive semidefinite matrix S. We will show that this problem can be converted

to (A.25). Since S is positive semidefinite, the finding procedure can be cast as

$$\max_{\lambda} \lambda \tag{A.32}$$

$$\text{subject to } S - \lambda I \succeq 0 \tag{A.33}$$

Now define an auxiliary matrix $X := S - \lambda I$. We have

$$\min_{\lambda, X} \ -\lambda \tag{A.34}$$

$$\text{subject to } X + \lambda I = S \tag{A.35}$$

$$X \succeq 0 \tag{A.36}$$

It is obvious that the linear matrix equality constraint $X+\lambda I = S$ can be divided into several linear scalar equality constraints in (A.25). For example, we consider $S \in \mathbb{S}^3_+$. Thereby $X + \lambda I = S$ will lead to 6 linear equality constraints (We don't consider X is a symmetric matrix here, since most solvers will implicitly consider this. Thus, only the upper-triangular part of X and S are actually used in the equality construction.):

$$\begin{bmatrix}1&0&0\\0&0&0\\0&0&0\end{bmatrix} \cdot X + \lambda = S[0,0], \begin{bmatrix}0&1&0\\0&0&0\\0&0&0\end{bmatrix} \cdot X = S[0,1], \begin{bmatrix}0&0&1\\0&0&0\\0&0&0\end{bmatrix} \cdot X = S[0,2] \tag{A.37}$$

$$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix} \cdot X + \lambda = S[1,1], \begin{bmatrix}0&0&0\\0&0&1\\0&0&0\end{bmatrix} \cdot X = S[1,2], \begin{bmatrix}0&0&0\\0&0&0\\0&0&1\end{bmatrix} \cdot X + \lambda = S[2,2] \tag{A.38}$$

Seems tedious? Fortunately, CVX provides a high-level API to handle these linear equality constraints: you just need to write down

```
X + lam * eye(3) == S; % linear equality constraints: X + lam *I = S
```

CVX will autometically convert this high-level constraint to (A.25) and pass them to the underlying solver.

To generate a ramdom $S \in \mathcal{S}^3_+$, you just need to assign three nonnegative eigenvalues to the program. After that, an random S will be generated by $S = Q\ \mathrm{diag}(\lambda_1, \lambda_2, \lambda_3)\ Q^T$, where Q is random orthonormal matrix.

```
%% Define the SDP example setting
lam_list = [0.7; 2.4; 3.7];
S = generate_random_PD_matrix(lam_list); % get a PD matrix S
```

```
%% Solve the SDP problem
cvx_begin
    variable X(3, 3) symmetric;
    variable lam;
    maximize(lam);
    subject to
        % here "==" should be read as "is in"
        X == semidefinite(3);
        X + lam * eye(3) == S;
cvx_end

% this function help to generate PD matrix of size 3*3
% if you provide the eigenvalues [lam_1, lam_2, lam_3]
function S = generate_random_PD_matrix(lam_list)
    if ~all(lam_list >= 0) % all eigenvalues >= 0
        error("All eigenvalues must be nonnegative.");
    end
    D = diag(lam_list);
    % use QR factorization to generate a random orthonormal matrix Q
    [Q, ~] = qr(rand(3, 3));
    S =  Q * D * Q';
end
```

A.2.7 CVXPY Introduction and Examples

Apart from CVX MATLAB, we also have a Python package called CVXPY, which functions almost the same as CVX MATLAB. To define and solve a convex optimization problem CVXPY, basically, there are three steps (apart from importing necessary packages):

- Step 1: Define parameters and variables in a certain type of convex problem. Here variables are what you are trying to optimize or "learn". Parameters are the "coefficients" of variables in the objective and constraints.
- Step 2: Define the objective function and constraints.
- Step 3: Solve the problem and get the results.

Here we provide the CVXPY codes for the above five convex optimization examples.

A.2.7.1 LP

```
import cvxpy as cp
import numpy as np

## Define the LP example setting
c1 = 2
c2 = -5
l1 = 3
l2 = 7

## Step 1: define variables and parameters
x = cp.Variable(2) # variable: x = [x1, x2]^T
# parameters: c, A, b
c = np.array([c1, c2])
A = np.array([[1, 0], [-1, 0], [0, 1], [0, -1]])
b = np.array([l1, l1, l2, l2])

## Step 2: define objective and constraints
obj = cp.Minimize(c.T @ x)
constraints = [A @ x <= b]
prob = cp.Problem(obj, constraints) # form the problem

## Step 3: solve problem and get results
prob.solve()
print("status: ", prob.status) # check whether the status is "optimal"
print("optimal value: ", prob.value) # optimal objective
print("optimal solution: ", x.value) # optimal x
```

A.2.7.2 QP

```
import cvxpy as cp
import numpy as np

## Define the LP example setting
p1 = 2
p2 = 0.5
p3 = 4
q1 = -3
q2 = -6.5
l1 = 2
```

```
l2 = 2.5
# check if the generated P is positive semidefinite
tmp1 = (p1 >= 0)
tmp2 = (p1*p3 - 4*p2**2 >= 0)
assert(tmp1 and tmp2, "P is not positve semidefinite!")

## Step 1: define variables and parameters
x = cp.Variable(2) # variable: x = [x1, x2]^T
# parameters: P, q, G, h
P = 2*np.array([[p1, p2], [p2, p3]])
q = np.array([q1, q2])
G = np.array([[1, 0], [-1, 0], [0, 1], [0, -1]])
h = np.array([l1, l1, l2, l2])

## Step 2: define the objective and constraints
fx = 0.5 * cp.quad_form(x, P) + q.T @ x
obj = cp.Minimize(fx)
constraints = [G @ x <= h]
prob = cp.Problem(obj, constraints) # form the problem

## Step 3: solve the problem and get results
prob.solve()
print("status: ", prob.status) # check whether the status is "optimal"
print("optimal value: ", prob.value) # optimal objective
print("optimal solution: ", x.value) # optimal x
```

A.2.7.3 QCQP

```
import cvxpy as cp
import numpy as np
from numpy.linalg import cholesky, inv, norm

## Define the QCQP example setting
def if_ellipse(K, k, c):
    # examine whether 0.5*x^T K x + k^T x + c <= 0 is a ellipse
    # if K is not positive semidefinite, Cholesky will raise an error
    L = cholesky(K)
    radius_square = 0.5 * norm(inv(L) @ k)**2 - c
    return radius_square > 0
K1 = np.eye(2)
k1 = np.zeros(2)
c1 = -0.5
```

```
K2 = np.array([[1, 0], [0, 1]])
k2 = np.array([2, 2])
c2 = 3.5
if not (if_ellipse(K1, k1, c1) and if_ellipse(K2, k2, c2)):
    raise ValueError("The example setting is not correct")

## Step 1: define variables and parameters
P0 = np.array([[1,0,-1,0], [0,1,0,-1], [-1,0,1,0], [0,-1,0,1]])
P1 = np.zeros((4,4))
P1[:2, :2] = K1
P2 = np.zeros((4,4))
P2[2:, 2:] = K2
q1 = np.concatenate([k1, np.zeros(2)])
q2 = np.concatenate([np.zeros(2), k2])
r1 = c1
r2 = c2

## Step 2: define objective and constraints
x = cp.Variable(4) # variable: x = [y1, z1, y2, z2]^T
fx = 0.5 * cp.quad_form(x, P0)
obj = cp.Minimize(fx)
con1 = (0.5 * cp.quad_form(x, P1) + q1.T @ x + r1 <= 0) # ellipse 1
con2 = (0.5 * cp.quad_form(x, P2) + q2.T @ x + r2 <= 0) # ellipse 2
constraints = [con1, con2]
prob = cp.Problem(obj, constraints) # form the problem

## Step 3: solve problem and get results
prob.solve()
print("status: ", prob.status) # check whether the status is "optimal"
print("optimal value: ", prob.value) # optimal objective
print("optimal solution: ", x.value) # optimal x
```

A.2.7.4 SOCP

```
import cvxpy as cp
import numpy as np
from scipy.stats import norm

## Define the SOCP example setting
# define bar_ai, bi (i = 1, 2, 3, 4)
bar_a1 = np.array([1, 0])
b1 = 1
```

```
bar_a2 = np.array([0, 1])
b2 = 1
bar_a3 = np.array([-1, 0])
b3 = 1
bar_a4 = np.array([0, -1])
b4 = 1
sigma = 0.1
c = np.array([2, 3])
p = 0.9 # p should be more than 0.5

## Step 1: define variables and parameters
Phi_inv = norm.ppf(p) # get Phi^{-1}(p)

## Step 2: define objective and constraints
x = cp.Variable(2) # variable: x = [x1, x2]^T
obj = cp.Minimize(c.T @ x)
# use cp.SOC(t, x) to create the SOC constraint ||x||_2 <= t
constraints = [
    cp.SOC(b1 - bar_a1.T @ x, sigma*Phi_inv*x),
    cp.SOC(b2 - bar_a2.T @ x, sigma*Phi_inv*x),
    cp.SOC(b3 - bar_a3.T @ x, sigma*Phi_inv*x),
    cp.SOC(b4 - bar_a4.T @ x, sigma*Phi_inv*x),
]
prob = cp.Problem(obj, constraints) # form the problem

## Step 3: solve problem and get results
prob.solve()
print("status: ", prob.status) # check whether the status is "optimal"
print("optimal value: ", prob.value) # optimal objective
print("optimal solution: ", x.value) # optimal x
```

A.2.7.5 SDP

```
import cvxpy as cp
import numpy as np
from scipy.stats import ortho_group

## Define the SDP example setting
# this function help to generate PD matrix of size 3*3
# if you provide the eigenvalues [lam_1, lam_2, lam_3]
def generate_random_PD_matrix(lam_list):
    assert np.all(lam_list >= 0) # all eigenvalues >= 0
```

```
    # S = Q @ D @ Q.T
    D = np.diag(lam_list)
    Q = ortho_group.rvs(3)
    return Q @ D @ Q.T
lam_list = np.array([0.5, 2.4, 3.7])
S = generate_random_PD_matrix(lam_list) # get a PD matrix S

## Step 1: define variables and parameters
# get coefficients for equality constraints
A_00 = np.array([[1, 0, 0], [0, 0, 0], [0, 0, 0]]) # tr(A_00 @ X) + lam = S_00
A_01 = np.array([[0, 1, 0], [0, 0, 0], [0, 0, 0]]) # tr(A_01 @ X) = S_01
A_02 = np.array([[0, 0, 1], [0, 0, 0], [0, 0, 0]]) # tr(A_02 @ X) = S_02
A_11 = np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]]) # tr(A_11 @ X) + lam = S_11
A_12 = np.array([[0, 0, 0], [0, 0, 1], [0, 0, 0]]) # tr(A_12 @ X) = S_12
A_22 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 1]]) # tr(A_22 @ X) + lam = S_22

## Step 2: define objective and constraints
# define a PD matrix variable X of size 3*3
X = cp.Variable((3, 3), symmetric=True)
constraints = [X >> 0] # the operator >> denotes matrix inequality
lam = cp.Variable(1)
constraints += [
    cp.trace(A_00 @ X) + lam == S[0,0],
    cp.trace(A_01 @ X) == S[0,1],
    cp.trace(A_02 @ X) == S[0,2],
    cp.trace(A_11 @ X) + lam == S[1,1],
    cp.trace(A_12 @ X) == S[1,2],
    cp.trace(A_22 @ X) + lam == S[2,2],
]
obj = cp.Minimize(-lam)
prob = cp.Problem(obj, constraints) # form the problem

## Step 3: solve problem and get results
prob.solve()
print("status: ", prob.status) # check whether the status is "optimal"
print("optimal value: ", prob.value) # optimal objective
print("optimal solution: ", lam.value) # optimal lam
```

Appendix B

Linear System Theory

Thanks to Shucheng Kang for writing this Appendix.

B.1 Stability

B.1.1 Continuous-Time Stability

Consider the continuous-time linear time-invariant (LTI) system

$$\dot{x} = Ax. \tag{B.1}$$

the system is said to be "diagonalizable" if A is diagonalizable.

Definition B.1 (Asymptotic and Marginal Stability)**.** The diagonalizable, LTI system (B.1) is

1. "asymptotically stable" if $x(t) \to 0$ as $t \to \infty$ for every initial condition x_0
2. "marginally stable" if $x(t) \nrightarrow 0$ but remains bounded as $t \to \infty$ for every initial condition x_0
3. "stable" if it is either asymptotically or marginally stable
4. "unstable" if it is not stable

One can show that A's eigenvalues determine the LTI system's stability, as the following Theorem states:

Theorem B.1 (Stability of Continuous-Time LTI System)**.** *The diagonalizable*[1]*, LTI system* (B.1) *is*

1. *asymptotically stable if* $Re(\lambda_i) < 0$ *for all* i
2. *marginally stable if* $Re(\lambda_i) \leq 0$ *for all* i *and there exists at least one* i *for which* $Re(\lambda_i) = 0$
3. *stable if* $Re(\lambda_i) \leq 0$ *for all* i
4. *unstable if* $Re(\lambda_i) > 0$ *for at least one* i

Proof. Here we only represent the proof of (1). Similar procedure can be adopted for the proof of (2) - (4).

Since A is diagonalizable, there exists an similarity transformation matrix T, s.t. $A = T\Lambda T^{-1}$, where $\Lambda = \text{diag}(\lambda_1, ..., \lambda_n)$. Then, under the coordinate transformation $z = T^{-1}x$, $\dot{x} = Ax$ can be restated as $\dot{z} = \Lambda z$. Consider the i's component of z:

$$\dot{z}_i = \lambda_i z_i \Longrightarrow z_i(t) = e^{\lambda_i t} z_i(0)$$

Since $\text{Re}(\lambda_i) < 0$, $z_i(t)$ will go to 0 as $t \to 0$ regardless how we choose $z_i(0)$.

□

B.1.2 Discrete-Time Stability

Now consider the diagonalizable, discrete-time linear time-invariant (LTI) system

$$x_{t+1} = Ax_t. \tag{B.2}$$

Theorem B.2 (Stability of Discrete-Time LTI System)**.** *The diagonalizable, discrete-time LTI system* (B.2) *is*

1. *asymptotically stable if* $|\lambda_i| < 1$ *for all* i
2. *marginally stable if* $|\lambda_i| \leq 1$ *for all* i *and there exists at least one* i *for which* $|\lambda_i| = 1$
3. *stable if* $|\lambda_i| \leq 1$ *for all* i
4. *unstable if* $|\lambda_i| > 1$ *for at least one* i*.*

Note that $|\lambda_i| < 1$ *means the eigenvalue lies strictly inside the unit circle in the complex plane.*

[1] when A is not diagonalizable, similar results can be derived via Jordan decomposition.

Proof. Here we only represent the proof of (1). Similar procedure can be adopted for the proof of (2) - (4).

Since A is diagonalizable, there exists an similarity transformation matrix T, s.t. $A = T\Lambda T^{-1}$, where $\Lambda = \text{diag}(\lambda_1, ..., \lambda_n)$. Then, under the coordinate transformation $z = T^{-1}x$, $x_{t+1} = Ax$ can be restated as $z_{t+1} = \Lambda z_t$. Expanding the recursion, we have

$$z_t = \Lambda^{t-1} z_0 \Longrightarrow z_{t,i} = \lambda_i^{t-1} z_{0,i}$$

Since $|\lambda_i| < 1$, $z_{t,i}$ will go to 0 as $t \to 0$ regardless how we choose $z_{0,i}$. □

B.1.3 Lyapunov Analysis

Theorem B.3 (Lyapunov Equation)**.** *The following is equivalent for a linear time-invariant system $\dot{x} = Ax$*

1. *The system is globally asymptotically stable, i.e., A is Hurwitz and $\lim_{t\to\infty} x(t) = 0$ regardless of the initial condition;*
2. *For any positive definite matrix Q, the unique solution P to the Lyapunov equation*
$$A^T P + PA = -Q \tag{B.3}$$
is positive definite.

Proof. (a): $2 \Rightarrow 1$. Suppose we are given two positive definite matrices $P, Q \succ 0$ that satisfies the Lyapunov equation (B.3). Define a scalar function

$$V(x) = x^T P x.$$

It is clear that $V > 0$ for any $x \neq 0$ and $V(x) = 0$ (i.e., $V(x)$ is positive definite). We also see $V(x)$ is radially unbounded because:

$$V(x) \geq \lambda_{\min}(P)\|x\|^2 \Rightarrow \lim_{x\to\infty} V(x) \to \infty.$$

The time derivative of V reads

$$\dot{V} = 2x^T P\dot{x} = x^T(A^T P + PA)x = -x^T Q x.$$

Clearly, $\dot{V} < 0$ for any $x \neq 0$ and $\dot{V}(0) = 0$. According to Lyapunov's global stability theorem **??**, we conclude the linear system $\dot{x} = Ax$ is globally asymptotically stable at $x = 0$.

(b): $1 \Rightarrow 2$. Suppose A is Hurwitz, we want to show that, for any $Q \succ 0$, there exists a unique $P \succ 0$ satisfying the Lyapunov equation (B.3). In fact, consider the matrix

$$P = \int_{t=0}^{\infty} e^{A^T t} Q e^{At} dt.$$

Because A is Hurwitz, the integral exists, and clearly $P \succ 0$ due to $Q \succ 0$. To show this choice of P satisfies the Lyapunov equation, we write

$$A^T P + PA = \int_{t=0}^{\infty} \left(A^T e^{A^T t} Q e^{At} + e^{A^T t} Q e^{At} A \right) dt \tag{B.4}$$

$$= \int_{t=0}^{\infty} d\left(e^{A^T t} Q e^{At} \right) \tag{B.5}$$

$$= e^{A^T t} Q e^{At}|_{t=\infty} - e^{A^T t} Q e^{At}|_{t=0} = -Q, \tag{B.6}$$

where the last equality holds because $e^{A\infty} = 0$ (recall A is Hurwitz).

To show the uniqueness of P, we assume that there exists another matrix P' that also satisfies the Lyapunov equation. Therefore,

$$P' = e^{A^T t} P' e^{At}|_{t=0} - e^{A^T t} P' e^{At}|_{t=\infty} \tag{B.7}$$

$$= -\int_{t=0}^{\infty} d\left(e^{A^T t} P' e^{At} \right) \tag{B.8}$$

$$= -\int_{t=0}^{\infty} e^{A^T t} \left(A^T P' + P' A \right) e^{At} dt \tag{B.9}$$

$$= \int_{t=0}^{\infty} e^{A^T t} Q e^{At} dt = P, \tag{B.10}$$

leading to $P' = P$. Hence, the solution is unique. □

Convergence rate estimation. We now show that Theorem B.3 can allow us to quantify the convergence rate of a (stable) linear system towards zero.

For a Hurwitz linear system $\dot{x} = Ax$, let us pick a positive definite matrix Q. Theorem B.3 tells us we can find a unique $P \succ 0$ satisfying the Lyapunov equation (B.3). In this case, we can upper bound the scalar function $V = x^T P x$ as

$$V \leq \lambda_{\max}(P) \|x\|^2.$$

The time derivative of V is $\dot{V} = -x^T Q x$, which can be upper bounded by

$$\dot{V} \leq -\lambda_{\min}(Q) \|x\|^2 \tag{B.11}$$

$$= -\frac{\lambda_{\min}(Q)}{\lambda_{\max}(P)} \underbrace{\left(\lambda_{\max}(P) \|x\|^2 \right)}_{\geq V} \tag{B.12}$$

$$\leq -\frac{\lambda_{\min}(Q)}{\lambda_{\max}(P)} V. \tag{B.13}$$

Denoting $\gamma(Q) = \frac{\lambda_{\min}(Q)}{\lambda_{\max}(P)}$, the above inequality implies

$$V(0) e^{-\gamma(Q) t} \geq V(t) = x^T P x \geq \lambda_{\min}(P) \|x\|^2.$$

As a result, $\|x\|^2$ converges to zero exponentially with a rate at least $\gamma(Q)$, and $\|x\|$ converges to zero exponentially with a rate at least $\gamma(Q)/2$.

Best convergence rate estimation. I have used $\gamma(Q)$ to make it explict that the rate γ depends on the choice of Q, because P is computed from the Lyapunov equation as an implicit function of Q. Naturally, choosing different Q will lead to different $\gamma(Q)$. So what is the choice of Q that maximizes the convergence rate estimation?

Corollary B.1 (Maximum Convergence Rate Estimation)**.** *$Q = I$ maximizes the convergence rate estimation.*

Proof. let us denote P_0 as the solution to the Lyapunov equation with $Q = I$

$$A^T P_0 + P_0 A = -I.$$

Let P be the solution corresponding to a different choice of Q

$$A^T P + PA = -Q.$$

Without loss of generality, we can assume $\lambda_{\min}(Q) = 1$, because rescaling Q will recale P by the same factor, which does not affect $\gamma(Q)$. Subtracting the two Lyapunov equations above we get

$$A^T(P - P_0) + (P - P_0)A = -(Q - I).$$

Since $Q - I \succeq 0$ (due to $\lambda_{\min}(Q) = 1$), we know $P - P_0 \succeq 0$ and $\lambda_{\max}(P) \geq \lambda_{\max}(P_0)$. As a result,

$$\gamma(Q) = \frac{\lambda_{\min}(Q)}{\lambda_{\max}(P)} = \frac{\lambda_{\min}(I)}{\lambda_{\max}(P)} \leq \frac{\lambda_{\min}(I)}{\lambda_{\max}(P_0)} = \gamma(I),$$

and $Q = I$ maximizes the convergence rate estimation. □

B.2 Controllability and Observability

Consider the following linear time-invariant (LTI) system

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{B.14}$$

where $x \in \mathbb{R}^n$ the state, $u \in \mathbb{R}^m$ the control input, $y \in \mathbb{R}^p$ the output, and A, B, C, D are constant matrices with proper sizes. If we know the initial state

$x(0)$ and the control inputs $u(t)$ over a period of time $t \in [0, t_1]$, the system trajectory $(x(t), y(t))$ can be determined as

$$\begin{aligned} x(t) &= e^{At}x(0) + \int_0^t e^{A(t-\tau)}Bu(\tau)d\tau \\ y(t) &= Cx(t) + Du(t) \end{aligned} \tag{B.15}$$

To study the internal structure of linear systems, two important properties should be considered: controllability and observability. In the following analysis, we will see that they are actually dual concepts. Their definitions (Chen, 1984) are given below.

Definition B.2 (Controllability)**.** The LTI system (B.14), or the pair (A, B), is controllable, if for any initial state $x(0) = x_0$ and final state x_f, there exists a sequence of control inputs that transfer the system from x_0 to x_f in finite time.

Definition B.3 (Observability)**.** The LTI system (B.14), or the pair (C, A), is observable, if for any unknown initial state $x(0)$, there exists a finite time $t_1 > 0$, such that knowing y and u over $[0, t_1]$ suffices to determine $x(0)$.

Sometimes it will become more convenient for us to analyze the system (B.14) under another coordinate basis, i.e., $z = Tx$, where the coordinate transformation T is nonsingular (i.e., full-rank). Define $A' = TAT^{-1}, B' = PB, C' = CT^{-1}, D' = D$, we get

$$\begin{aligned} \dot{z} &= A'z + B'u \\ y &= C'z + D'u \end{aligned}$$

Since the coordinate transformation only changes the system's coordinate basis, physical properties like controllability and observability will not change.

B.2.1 Cayley-Hamilton Theorem

In the analysis of controllability and observability, Cayley Hamilton Theorem lays the foundation. The statement of the theory and its (elegant) proof are given blow. Some useful corollaries are also presented.

Theorem B.4 (Cayley-Hamilton)**.** *Let $A \in \mathbb{C}^{n\times n}$ and denote the characteristic polynomial of A as*

$$det(\lambda I - A) = \lambda^n + a_1\lambda^{n-1} + \cdots + a_n \in \mathbb{C}[\lambda],$$

which is a polynomial in a single variable λ with coefficients $a_1, \ldots, a_n$. Then

$$A^n + a_1A^{n-1} + \cdots + a_nI = 0$$

Proof. Define the adjugate of $\lambda I - A$ as

$$B = \text{adj}(\lambda I - A)$$

From B's definition, we have

$$(\lambda I - A)B = \det(\lambda I - A)I = (\lambda^n + a_1\lambda^{n-1} + \cdots + a_n)I \tag{B.16}$$

Also, B is a polynomial matrix over λ, whose maximum degree is no more than $n-1$. Therefore, we write B as follows:

$$B = \sum_{i=0}^{n-1} \lambda^i B_i$$

where B_i's are constant matrices. In this way, we unfold $(\lambda I - A)B$:

$$\begin{aligned}(\lambda I - A)B &= (\lambda I - A)\sum_{i=0}^{n-1} \lambda^i B_i \\ &= \lambda^n B_{n-1} + \sum_{i=1}^{n-1} \lambda^i(-AB_i + B_{i-1}) - AB_0\end{aligned} \tag{B.17}$$

Since λ can be arbitrarily set, matching the coefficients of (B.16) and (B.17), we have

$$\begin{aligned} B_{n-1} &= I \\ -AB_i + B_{i-1} &= a_{n-i}I, \quad i = 1 \dots n-1 \\ -AB_0 &= a_n I \end{aligned}$$

Thus, we have

$$\begin{aligned} &B_{n-1} \cdot A^n + \sum_{i=1}^{n-1}(-AB_i + B_{i-1}) \cdot A^i + (-AB_0) \cdot I \\ =&I \cdot A^n + \sum_{i=1}^{n-1}(a_{n-i}I) \cdot A^i + (a_n I) \cdot I \\ =&A^n + a_1 A^{n-1} + a_2 A^{n-2} + \cdots + a_n I \end{aligned}$$

On the other hand, one can easily check that

$$B_{n-1} \cdot A^n + \sum_{i=1}^{n-1}(-AB_i + B_{i-1}) \cdot A^i + (-AB_0) \cdot I = 0$$

since each term offsets completely. Therefore,

$$A^n + a_1 A^{n-1} + a_2 A^{n-2} + \cdots + a_n I = 0,$$

concluding the proof. □

Here are some corollaries of the Cayley-Hamilton Theorem.

Corollary B.2. *For any $A \in \mathbb{C}^{n\times n}, B \in \mathbb{C}^{n\times m}, k \geq n$, $A^k B$ is a linear combination of $B, AB, A^2B, \ldots, A^{n-1}B$.*

Proof. Directly from Cayley Hamilton Theorem, A^n can be expressed as a linear combination of $I, A, A^2, \ldots, A^{n-1}$. By recursion, it is easy to show that for all $m > n$, A^m is also a linear combination of $I, A, A^2, \ldots, A^{n-1}$. Post-multiply both sides with B, we get what we want. □

Corollary B.3. *For any $A \in \mathbb{C}^{n\times n}, B \in \mathbb{C}^{n\times m}, k > n$, the following equality always holds:*

$$\mathit{rank}(\begin{bmatrix} B & AB & \ldots & A^{n-1}B \end{bmatrix}) = \mathit{rank}(\begin{bmatrix} B & AB & \ldots & A^{k-1}B \end{bmatrix})$$

Proof. First prove LHS $\leq$ RHS. $\forall v \in \mathbb{C}^n$ such that

$$v^* \begin{bmatrix} B & AB & \ldots & A^{k-1}B \end{bmatrix} = v^* \begin{bmatrix} B & AB & \ldots & A^{n-1}B & \ldots A^{k-1}B \end{bmatrix} = 0$$

$v^* \begin{bmatrix} B & AB & \ldots & A^{n-1}B \end{bmatrix} = 0$ must hold.

Second prove LHS $\geq$ RHS. For any $v \in \mathbb{C}^n$ such that $v^* \begin{bmatrix} B & AB & \ldots & A^{n-1}B \end{bmatrix} = 0$ and any $k > n$, by Corollary B.2, there exists a sequence $c_i, i = 0 \ldots n-1$ satisfy the following:

$$v^* A^k B = v^* \sum_{i=0}^{n-1} c_i A^i B = 0$$

Therefore, $v^* \begin{bmatrix} B & AB & \ldots & A^{k-1}B \end{bmatrix} = 0$. □

Corollary B.4. *For any $A \in \mathbb{C}^{n\times n}, B \in \mathbb{C}^{n\times m}$, define*

$$\mathcal{C} = \begin{bmatrix} B & AB & \ldots & A^{n-1}B \end{bmatrix}$$

If $\mathit{rank}(\mathcal{C}) = k_1 < n$, there exist a similarity transformation T such that

$$TAT^{-1} = \begin{bmatrix} \bar{A}_c & \bar{A}_{12} \\ 0 & \bar{A}_{\bar{c}} \end{bmatrix}, TB = \begin{bmatrix} \bar{B}_c \\ 0 \end{bmatrix}$$

where $\bar{A}_c \in \mathbb{C}^{k_1\times k_1}, \bar{B}_c \in \mathbb{C}^{k_1\times m}$. Moreover, the matrix

$$\bar{\mathcal{C}} := \begin{bmatrix} \bar{B}_c & \bar{A}_c\bar{B}_c & \bar{A}_c^2\bar{B}_c & \ldots & \bar{A}_c^{k_1-1}\bar{B}_c \end{bmatrix}$$

has full row rank.

Proof. Since $\mathcal{C}$ is not full row rank, we pick k_1 linearly independent columns from $\mathcal{C}$. Denote them as $q_1 \ldots q_{k_1}, q_i \in \mathbb{C}^n$. Then, we arbitrarily set other $n-k_1$ vectors $q_{k_1+1} \ldots q_n$ as long as

$$Q = \begin{bmatrix} q_1 & \ldots & q_{k_1} & q_{k_1+1} & \ldots & q_n \end{bmatrix}$$

is invertible. Define the similarity transformation matrix by $T = Q^{-1}$. Note that Aq_i can be seen as a column picked from $A^k B, k \in \{1 \dots n\}$, which is guaranteed to be a linear combination of $B, AB, \dots, A^{n-1}B$ from Cayley Hamilton Theorem. Thus, Aq_i is bound to be a linear transformation of columns from $\begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix} = \mathcal{C}$. Since $q_1 \dots q_{k_1}$ is the largest linearly independent column vector set from $\mathcal{C}$, this implies Aq_i can be expressed as a linear combination of $q_1 \dots q_{k_1}$:

$$\begin{aligned} AQ = AT^{-1} &= A \begin{bmatrix} q_1 & \dots & q_{k_1} & q_{k_1+1} & \dots & q_n \end{bmatrix} \\ &= \begin{bmatrix} q_1 & \dots & q_{k_1} & q_{k_1+1} & \dots & q_n \end{bmatrix} \begin{bmatrix} \bar{A}_c & \bar{A}_{12} \\ 0 & \bar{A}_{\bar{c}} \end{bmatrix} = T^{-1} \begin{bmatrix} \bar{A}_c & \bar{A}_{12} \\ 0 & \bar{A}_{\bar{c}} \end{bmatrix} \end{aligned}$$

Similarly, B itself is part of $\mathcal{C}$. Therefore, each column of B is naturally a linear combination of $q_1 \dots q_{k_1}$:

$$B = \begin{bmatrix} q_1 & \dots & q_{k_1} & q_{k_1+1} & \dots & q_n \end{bmatrix} \begin{bmatrix} \bar{B}_c \\ 0 \end{bmatrix} = T^{-1} \begin{bmatrix} \bar{B}_c \\ 0 \end{bmatrix}$$

To see $\bar{\mathcal{C}}$ has full row rank, note that $\text{rank}\mathcal{C} = k_1$ and

$$\mathcal{C} = T^{-1} \begin{bmatrix} \bar{B}_c & \bar{A}_c\bar{B}_c & \bar{A}_c^2\bar{B}_c & \dots & \bar{A}_c^{k_1-1}\bar{B}_c & \dots & \bar{A}_c^{n-1}\bar{B}_c \\ 0 & 0 & 0 & \dots & 0 & \dots & 0 \end{bmatrix}$$

Thus,

$$\text{rank} \begin{bmatrix} \bar{B}_c & \bar{A}_c\bar{B}_c & \bar{A}_c^2\bar{B}_c & \dots & \bar{A}_c^{k_1-1}\bar{B}_c & \dots & \bar{A}_c^{n-1}\bar{B}_c \end{bmatrix} = k_1.$$

By Corollary B.3, $\text{rank}\bar{\mathcal{C}} = k_1$. □

The following Corollary is especially useful in the study of pole assignment in the single-input-multiple-output (SIMO) LTI system.

Corollary B.5. *For any $A \in \mathbb{C}^{n\times n}, b \in \mathbb{C}^n$, if*

$$\mathcal{C} = \begin{bmatrix} b & Ab & \dots & A^{n-1}b \end{bmatrix} \in \mathbb{C}^{n\times n}$$

has full rank, then there exists a similarity transformation T such that

$$TAT^{-1} = A_1 := \begin{bmatrix} -a_1 & -a_2 & \dots & -a_{n-1} & -a_n \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}, \quad Tb = b_1 := \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $a_1, \dots, a_n$ are the coefficients of A's characteristic polynomial:

$$\det(A - \lambda I) = \lambda^n + a_1\lambda^{n-1} + \dots + a_n\lambda$$

Proof. Since $\mathcal{C}$ is invertible, define its inverse

$$\mathcal{C}^{-1} = \begin{bmatrix} M_1 \\ M_2 \\ ... \\ M_n \end{bmatrix}$$

where $M_i \in \mathbb{C}^{1\times n}$. Then,

$$I = \mathcal{C}^{-1}\mathcal{C} = \begin{bmatrix} M_1 b & M_1 Ab & ... & M_1 A^{n-1}b \\ M_2 b & M_2 Ab & ... & M_2 A^{n-1}b \\ \vdots & \vdots & & \vdots \\ M_n b & M_n Ab & ... & M_n A^{n-1}b \end{bmatrix} \Longrightarrow \begin{cases} M_n A^{n-1}b = 1 \\ M_n A^i b = 0, \ i = 0, ..., n-2 \end{cases}$$

Now we claim that the transformation matrix T can be constructed as follows:

$$T = \begin{bmatrix} M_n A^{n-1} \\ M_n A^{n-2} \\ ... \\ M_n \end{bmatrix}$$

We first show T is invertible by calculating $T\mathcal{C}$:

$$T\mathcal{C} = \begin{bmatrix} M_n A^{n-1}b & \star & ... & \star \\ M_n A^{n-2}b & M_n A^{n-1}b & ... & \star \\ \vdots & \vdots & & \vdots \\ M_n b & M_n Ab & ... & M_n A^{n-1}b \end{bmatrix} = \begin{bmatrix} 1 & \star & ... & \star \\ 0 & 1 & ... & \star \\ \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 1 \end{bmatrix}$$

Then we calculate Tb and TA:

$$Tb = \begin{bmatrix} M_n A^{n-1}b \\ M_n A^{n-2}b \\ \vdots \\ M_n b \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$TA = \begin{bmatrix} M_n A^n \\ M_n A^{n-1} \\ \vdots \\ M_n A \end{bmatrix} = \begin{bmatrix} -M_n \cdot \sum_{i=0}^{n-1} a_{n-i} A^i \\ M_n A^{n-1} \\ \vdots \\ M_n A \end{bmatrix}$$

$$= \begin{bmatrix} -a_1 & -a_2 & ... & -a_{n-1} & -a_n \\ 1 & 0 & ... & 0 & 0 \\ 0 & 1 & ... & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & ... & 1 & 0 \end{bmatrix} \begin{bmatrix} M_n A^{n-1} \\ M_n A^{n-2} \\ \vdots \\ M_n A \\ M_n \end{bmatrix} = A_1 T$$

where the penultimate equality uses Cayley Hamilton Theorem. □

B.2.2 Equivalent Statements for Controllability

There are a few equivalent statements to express an LTI system's controllability that one should be familiar with:

Theorem B.5 (Equivalent Statements for Controllability)**.** *The following statements are equivalent (Chen, 1984), (Zhou et al., 1996):*

1. (A, B) *is controllable.*
2. *The matrix*
$$W_c(t) := \int_0^t e^{A\tau} BB^* e^{A^*\tau} d\tau$$
is positive definite for any $t > 0$.
3. *The controllability matrix*
$$\mathcal{C} = \begin{bmatrix} B & AB & A^2B & \dots & A^{n-1}B \end{bmatrix}$$
has full row rank.
4. *The matrix* $[A - \lambda I, B]$ *has full row rank for all* $\lambda \in \mathbb{C}$.
5. *Let* λ *and* x *be any eigenvalue and any corresponding left eigenvector* A*, i.e.,* $x^*A = x^*\lambda$*, then* $x^*B \neq 0$.
6. *The eigenvalues of* $A + BF$ *can be freely assigned (with the restriction that complex eigenvalues are in conjugate pairs) by a suitable choice of* F.
7. *If, in addition, all eigenvalues of* A *have negative real parts, then the unique solution of*
$$AW_c + W_cA^* = -BB^*$$
is positive definite. The solution is called the controllability Gramian *and can be expressed as*
$$W_c = \int_0^\infty e^{A\tau} BB^* e^{A^*\tau} d\tau$$

Proof. (1. $\Rightarrow$ 2.) Prove by contradiction. Assume that (A, B) is controllable but $W_c(t_1)$ is singular for some $t_1 > 0$. This implies there exists a real vector $v \neq 0 \in \mathbb{R}^n$, s.t.

$$v^*W_c(t_1)v = v^*(\int_0^{t_1} e^{At} BB^* e^{A^*t} dt)v = \int_0^{t_1} v^*(e^{At} BB^* e^{A^*t})v \; dt = 0$$

Since $e^{At} BB^* e^{A^*t} \succeq 0$ for all t, we must have

$$\begin{aligned} & v^*(e^{At} BB^* e^{A^*t})v = \| \, v^*Be^{At} \, \|^2 = 0, \quad \forall t \in [0, t_1] \\ \Longrightarrow & v^*Be^{At} = 0, \quad \forall t \in [0, t_1] \end{aligned}$$

Setting $x(t_1) = 0$, from (B.15), we have

$$0 = e^{At_1}x(0) + \int_0^{t_1} e^{A(t_1-\tau)}Bu(\tau)d\tau = 0$$

Pre-multiply the above equation by v^*, then

$$0 = v^* e^{At_1}x(0)$$

Since $x(0)$ can be chosen arbitrarily, we set $x(0) = ve^{-At_1}$, which results in $v = 0$. Contradiction!

(2. $\Rightarrow$ 1.) For any $x(0) = x_0, t_1 > 0, x(t_1) = x_1$, since $W_c(t_1) \succ 0$, we set the control inputs as

$$u(t) = -B^* e^{A^*(t_1-t)}W_c^{-1}(t_1)[e^{At_1}x_0 - x_1]$$

We claim that the picked $u(t)$ satisfies (B.15) by

$$\begin{aligned}
&e^{At}x_0 + \int_0^{t_1} e^{A(t_1-t)}Bu(t)dt \\
&= e^{At}x_0 - \int_0^{t_1} e^{A(t_1-t)}BB^*e^{A^*(t_1-t)}dt \cdot W_c^{-1}(t_1)[e^{At_1}x_0 - x_1] \\
&\overset{\tau=t_1-t}{=} e^{At}x_0 - \underbrace{\int_0^{t_1} e^{A\tau}BB^*e^{A^*\tau}d\tau}_{W_c(t_1)} \cdot W_c^{-1}(t_1)[e^{At_1}x_0 - x_1] \\
&= e^{At}x_0 - [e^{At_1}x_0 - x_1] = x_1
\end{aligned}$$

(2. $\Rightarrow$ 3.) Prove by contradiction. Suppose $W_c(t) \succ 0, \forall t > 0$ but $\mathcal{C}$ is not of full row rank. Then there exists $v \neq 0 \in \mathbb{C}^n$, s.t.

$$v^*A^kB = 0, \quad k = 0 \dots n-1$$

By Corollary B.2, we have

$$v^*A^kB = 0,\ \forall k \in \mathbb{N} \Longrightarrow v^*e^{At}B = 0,\ \forall t > 0$$

which implies

$$v^*W_c(t)v = v^*(\int_0^t e^{A\tau}BB^*e^{A^*\tau}d\tau)v = 0, \quad \forall t > 0$$

Contradiction!

(3. $\Rightarrow$ 2.) Prove by contradiction. Suppose $\mathcal{C}$ has full row rank but $W_c(t_1)$ is singular at some $t_1 > 0$. Then, similar to the proof in (1. $\Rightarrow$ 2.), there exists $v \neq 0 \in \mathbb{C}^n$, s.t. $F(t) := v^*e^{At}B \equiv 0, \forall t \in [0, t_1]$. Since $F(t)$ is infinitely

differentiable, we get its i's derivative at $t = 0$, where $i = 0, 1, ...\, n-1$. This results in

$$\left.\frac{d^i F}{dt^i}\right|_{t=0} = v^* A^i e^{At} B\big|_{t=0} = v^* A^i B = 0, \quad i = 0 \ldots n-1$$

Thus, $v^* \begin{bmatrix} B & AB & ... & A^{n-1}B \end{bmatrix} = 0$. Contradiction!

(3. $\Rightarrow$ 4.) Proof by contradiction. Suppose $[A - \lambda I, B]$ does not have full row rank for some $\lambda \in \mathbb{C}$. Then, there exists $v \neq 0 \in \mathbb{C}^n$, s.t. $v^*[A - \lambda I, B] = 0$. This implies $v^* A = v^* \lambda$ and $v^* B = 0$. On the other hand,

$$v^* \begin{bmatrix} B & AB & ... & A^{n-1}B \end{bmatrix} = v^* \begin{bmatrix} B & \lambda B & ... & \lambda^{n-1}B \end{bmatrix} = 0$$

Contradiction!

(4. $\Rightarrow$ 5.) Proof by contradiction. If there exists a left eigenvector and eigenvalue pair (x, λ), s.t. $x^* A = \lambda x^*$ while $x^* B = 0$, then $x^*[A - \lambda I, B] = 0$. Contradiction!

(5. $\Rightarrow$ 3.) Proof by contradiction. If the controllability matrix $\mathcal{C}$ does not have full row rank, i.e., $\text{rank}(\mathcal{C}) = k < n$. Then, from Corollary B.4, there exists a similarity transformation T, s.t.

$$TAT^{-1} = \begin{bmatrix} \bar{A}_c & \bar{A}_{12} \\ 0 & \bar{A}_{\bar{c}} \end{bmatrix}, \quad TB = \begin{bmatrix} \bar{B}_c \\ 0 \end{bmatrix}$$

where $\bar{A}_c \in \mathbb{R}^{k \times k}, \bar{A}_{\bar{c}} \in \mathbb{R}^{(n-k)\times(n-k)}$. Now arbitrarily pick one of $\bar{A}_{\bar{c}}$'s left eigenvector $x_{\bar{c}}$ and its corresponding eigenvalue λ_1. Define the vector $x = \begin{bmatrix} 0 \\ x_{\bar{c}} \end{bmatrix}$. Then,

$$x^*(TAT^{-1}) = \begin{bmatrix} 0 & x_{\bar{c}}^* \end{bmatrix} \begin{bmatrix} \bar{A}_c & \bar{A}_{12} \\ 0 & \bar{A}_{\bar{c}} \end{bmatrix} = \begin{bmatrix} 0 & x_{\bar{c}}^* \bar{A}_{\bar{c}} \end{bmatrix} = \begin{bmatrix} 0 & \lambda_1 x_{\bar{c}}^* \end{bmatrix} = \lambda_1 x^*$$

$$x^*(TB) = \begin{bmatrix} 0 & x_{\bar{x}} \end{bmatrix} \begin{bmatrix} B_{\bar{c}} \\ 0 \end{bmatrix} = 0$$

which implies (TAT^{-1}, TB) is not controllable. However, similarity transformation does not change controllability. Contradiction!

(6. $\Rightarrow$ 1.) Prove by contradiction. If (A, B) is not controllable, i.e., $\text{rank}(\mathcal{C}) = k < n$. Then from Corollary B.4, there exists a similarity transformation T s.t.

$$TAT^{-1} = \begin{bmatrix} \bar{A}_c & \bar{A}_{12} \\ 0 & \bar{A}_{\bar{c}} \end{bmatrix}, \quad TB = \begin{bmatrix} \bar{B}_c \\ 0 \end{bmatrix}$$

Now arbitrarily pick $F \in \mathbb{R}^{m \times n}$ and define $FT^{-1} = [F_1, F_2]$, where $F_1 \in$

$\mathbb{R}^{m\times k}, F_2 \in \mathbb{R}^{m\times(n-k)}$. Thus,

$$\begin{aligned}
\det(A+BF-\lambda I) &= \det\left(T^{-1}\begin{bmatrix}\bar{A}_c & \bar{A}_{12}\\ 0 & \bar{A}_{\bar{c}}\end{bmatrix}T + T^{-1}\begin{bmatrix}\bar{B}_c\\ 0\end{bmatrix}F - \lambda\begin{bmatrix}I_1 & 0\\ 0 & I_2\end{bmatrix}\right)\\
&= \det\left(T^{-1}\left\{\begin{bmatrix}\bar{A}_c & \bar{A}_{12}\\ 0 & \bar{A}_{\bar{c}}\end{bmatrix} + \begin{bmatrix}\bar{B}_c\\ 0\end{bmatrix}FT^{-1} - \lambda\begin{bmatrix}I_1 & 0\\ 0 & I_2\end{bmatrix}\right\}T\right)\\
&= \det\left(\begin{bmatrix}\bar{A}_c & \bar{A}_{12}\\ 0 & \bar{A}_{\bar{c}}\end{bmatrix} + \begin{bmatrix}\bar{B}_c\\ 0\end{bmatrix}\begin{bmatrix}F_1 & F_2\end{bmatrix} - \lambda\begin{bmatrix}I_1 & 0\\ 0 & I_2\end{bmatrix}\right)\\
&= \det\begin{bmatrix}\bar{A}_c + \bar{B}_cF_1 - \lambda I_1 & \bar{A}_{12} + \bar{B}_cF_2\\ 0 & \bar{A}_{\bar{c}} - \lambda I_2\end{bmatrix}\\
&= \det(\bar{A}_c + \bar{B}_cF_1 - \lambda I_1)\cdot\det(\bar{A}_{\bar{c}} - \lambda I_2)
\end{aligned}$$

where I_1 is the identity matrix of size k. Similarly, I_2 of size $n-k$. Thus, at least $n-k$ eigenvalues of $A+BF$ cannot be freely assigned by choosing F. Contradiction!

(1. $\Rightarrow$ 6.) Here we only represent the SIMO case. For the MIMO case, the proof is far more complex. Interesting readers can refer to (Davison and Wonham, 1968) (the shortest proof I can find). Since there is only one input, the matrix B degenerate to vector b. From Corollary B.5, there exist a similarity transformation matrix T, s.t.

$$TAT^{-1} = A_1 := \begin{bmatrix}-a_1 & -a_2 & \dots & -a_{n-1} & -a_n\\ 1 & 0 & \dots & 0 & 0\\ 0 & 1 & \dots & 0 & 0\\ \vdots & \vdots & & \vdots & \vdots\\ 0 & 0 & \dots & 1 & 0\end{bmatrix}, \quad Tb = b_1 := \begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}$$

For any $F \in \mathbb{C}^{1\times n}$, denote FT^{-1} as $[f_1, f_2, \dots, f_n]$. Calculating the characteristic polynomial of $A+bF$:

$$\begin{aligned}
\det(\lambda I - A - bF) &= \det(\lambda I - T^{-1}A_1T - T^{-1}b_1F)\\
&= \det(\lambda I - A_1 - b_1FT^{-1})\\
&= \det\begin{bmatrix}\lambda + a_1 - f_1 & \lambda + a_2 - f_2 & \dots & \lambda + a_{n-1} - f_{n-1} & \lambda + a_n - f_n\\ -1 & \lambda & \dots & 0 & 0\\ 0 & -1 & \dots & 0 & 0\\ \vdots & \vdots & & \vdots & \vdots\\ 0 & 0 & \dots & -1 & \lambda\end{bmatrix}\\
&= \lambda^n + (a_1 - f_1)\lambda^{n-1} + \cdots + (a_n - f_n)
\end{aligned}$$

By choosing $[f_1, f_2, \dots, f_n]$, $A+bF$'s eigenvalues can be arbitrarily set.

(7. $\Rightarrow$ 1.) Prove by contradiction. Assume that (A, B) is not controllable. Then from 2., there exists $v \neq 0 \in \mathbb{C}^n$ and $t_1 > 0$,

$$F(t) = v^*e^{At}B = 0, \quad \forall t \in [0, t_1]$$

Now consider $F(z) = v^* e^{Az} B, z \in \mathcal{C}$, which is a vector of analytic function in complex analysis. For a arbitrary $t_2 \in (0, t_1)$, we have $F^{(i)}(t_2) = 0, \forall i \in \mathbb{N}$. Then, by invoking the fact from complex analysis: "Let G a connected open set and $f : G \to \mathbb{C}$ be analytic, then $f \equiv 0$ on G, if and only if there is a point $a \in G$ such that $f^{(i)}(a) = 0, \forall n \in \mathbb{N}$", we have $f(z) \equiv 0, \forall z \in \mathbb{C}$.

On the other hand, however, $W_c \succ 0$ implies there exists $t_3 > 0$, such that for the above v, we have $v^* e^{At_3} B \neq 0$. Contradiction!

$(1. \Rightarrow 7.)$ Since (A, B) is controllable, from 2., $W_c(t) \succ 0, \forall t$. Therefore, $W_c \succ 0$. The existence and uniqueness of the solution for $AW_c + W_c A^* = -BB^*$ can be obtained directly from the proof of Theorem B.3, by setting Q there to be positive semidefinite. □

B.2.3 Duality

Although controllability and observability seemingly have no direct connections from their definitions B.2 and B.3, the following theorem (Chen, 1984) states their tight relations.

Theorem B.6 (Theorem of Duality)**.** *The pair (C, A) is observable if and only if (A^*, C^*) is controllable.*

Proof.

(1) We first show that (C, A) is observable if and only if the $n \times n$ matrix $W_o(t) = \int_0^t e^{A^*\tau} C^* C e^{A\tau}$ is positive definite (nonsingular) for any $t > 0$:

"$\Longleftarrow$": From (B.15), given initial state $x(0)$ and the inputs $u(t)$, $y(t)$ can be expressed as

$$y(t) = Ce^{At}x(0) + C\int_0^t e^{A(t-\tau)}Bu(\tau)d\tau + Du(t)$$

Define a known function $\bar{y}(t)$ as $y(t) - C\int_0^t e^{A(t-\tau)}Bu(\tau)d\tau - Du(t)$ and we will get

$$Ce^{At}x(0) = \bar{y}(t)$$

Pre-multiply the above equation by $e^{A^*t}C^*$ and integrate it over $[0, t_1]$ to yield

$$(\int_0^{t_1} e^{A^*t}C^*Ce^{At}dt)x(0) = W_o(t_1)x(0) = \int_0^{t_1} e^{A^*t}C^*\bar{y}(t)dt$$

Since $W_o(t_1) \succ 0$,

$$x(0) = W_o(t_1)^{-1}\int_0^{t_1} e^{A^*t}C^*\bar{y}(t)dt$$

can be observed.

"$\Longrightarrow$": Prove by contradiction. Suppose (C, A) is observable but there exists $t_1 > 0$, s.t. $W_o(t_1)$ is singular. This implies there exists $v \neq 0 \in \mathbb{C}^n$, s.t.

$$v^* W_o(t_1) v = 0 \Longrightarrow Ce^{At}v \equiv 0, \ \forall t \in [0, t_1]$$

Similar to the proof of Theorem B.5 (7. $\Rightarrow$ 1.), we can use conclusions from complex analysis to claim that $Ce^{At}v \equiv 0, \forall t > 0$. On the other hand, we set $u(t) \equiv 0$, which results in $y(t) = Ce^{At}x(0)$. In this case $x(0) = 0$ and $x(0) = v \neq 0$ will lead to the same output responses $y(t)$ over $t > 0$, which implies (C, A) is not observable. Contradiction!

(2) Next we show the duality of controllability and observability:

From (1) we know (C, A) is controllable if and only of

$$\int_0^t e^{A^*\tau} C^* C e^{A\tau} d\tau = \int_0^t e^{(A^*)\tau} (C^*)^* (C^*) e^{(A^*)^*\tau} d\tau$$

is nonsingular for all $t > 0$. The latter is exactly the definition of (A^*, C^*)'s controllability Gramian $W_c(t)$.

□

B.2.4 Equivalent Statements for Observability

With the Theorem of Duality B.6, we can directly write down the equivalent statements of observability without any additional proofs:

Theorem B.7 (Equivalent Statements for Observability)**.** *The following statements are equivalent (Chen, 1984), (Zhou et al., 1996):*

1. (C, A) *is observable.*
2. *The matrix*
$$W_o(t) := \int_0^t e^{A^*\tau} C^* C e^{A\tau} d\tau$$
is positive definite for any $t > 0$.
3. *The observability matrix*
$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \cdots \\ CA^{n-1} \end{bmatrix}$$
has full column rank.

4. *The matrix* $\begin{bmatrix} A-\lambda I \\ C \end{bmatrix}$ *has full column rank for all* $\lambda \in \mathbb{C}$.

5. *Let* λ *and* y *be any eigenvalue and any corresponding right eigenvector of* A*, i.e.,* $Ay = \lambda y$*, then* $Cy \neq 0$.

6. *The eigenvalues of* $A + LC$ *can be freely assigned (with the restriction that complex eigenvalues are in conjugate pairs) by a suitable choice of* L.

7. (A^*, C^*) *is controllable.*

8. *If, in addition, all eigenvalues of* A *have negative parts, then the unique solution of*

$$A^* W_o + W_o A = -C^* C$$

is positive definite. The solution is called the observability Gramian *and can be expressed as*

$$W_o = \int_0^\infty e^{A^*\tau} C^* C e^{A\tau} d\tau$$

B.3 Stabilizability And Detectability

To define stabilizability and detectability of an LTI system, we first introduce the concept of *system mode*, which can be naturally derived from the fifth definition of controllability B.5 (observability B.7).

Definition B.4 (System Mode)**.** λ is a mode of an LTI system, if it is an eigenvalue of A. The mode λ is said to be:

- stable, if $\mathrm{Re}\lambda < 0$,
- controllable, if $x^* B \neq 0$ for all left eigenvectors of A associated with λ,
- observable, if $Cx \neq 0$ for all right eigenvectors of A associated with λ.

Otherwise, the mode is said to be uncontrollable (unobservable).

With the concept of system mode, the fifth definition of controllability B.5 (observability B.7) can be restated as

> An LTI system is controllable (observable) if and only if all modes are controllable (observable).

Stabilizability (detectability) is defined similarly via loosening part of controllability (observability) conditions.

Definition B.5 (Stabilizability)**.** An LTI system is said to be stabilizable if all of its unstable modes are controllable.

Definition B.6 (Detectability)**.** An LTI system is said to be detectable if all of its unstable modes are observable.

Like in the case of controllability and observability, duality also holds in stabilizability and detectability. Moreover, similarity transformation will not influence an LTI system's stabilizability and detectability.

B.3.1 Equivalent Statements for Stabilizability

Theorem B.8 (Equivalent Statements for Stabilizability)**.** *The following statements are equivalent (Zhou et al., 1996):*

1. (A, B) *is stabilizable.*
2. *For all* λ *and* x *such that* $x^*A = \lambda x^*$ *and* $Re\lambda \geq 0$, $x^*B \neq 0$.
3. *The matrix* $[A - \lambda I, B]$ *has full rank for all* $Re\lambda \geq 0$.
4. *There exists a matrix* F *such that* $A + BF$ *are Hurwitz.*

Proof. (1. $\Leftrightarrow$ 2.) Directly from stabilizability's definition.

(2. $\Leftrightarrow$ 3.) If 2. holds but 3. not hold, then there exists $v \neq 0 \in \mathbb{C}^n$, s.t.

$$v^*[A - \lambda I, B] = 0 \Leftrightarrow v^*A = \lambda v^*, v^*B = 0, \mathrm{Re}\lambda \geq 0$$

Contradiction! Vice versa.

(4. $\Rightarrow$ 2.) Prove by contradiction. Suppose there $x \neq 0 \in \mathbb{C}^n$, s.t.

$$x^*[A - \lambda I, B] = 0 \Leftrightarrow x^*A = \lambda x^*, x^*B = 0, \mathrm{Re}\lambda \geq 0$$

Thus, for any F,

$$x^*(A + BF) = \lambda x^*, \mathrm{Re}\lambda \geq 0$$

On the other hand, suppose $A + BF$ has I Jordon blocks, with each equipped with an eigenvalue $\eta_i, i = 1 \dots I$ (note that η_α may be equal to η_β, i.e., they are equivalent eigenvalues with different Jordon blocks). Since $A + BF$'s eigenvalues all have negative real parts, $\mathrm{Re}(\eta_i) < 0, i = 1 \dots I$. For each $\eta_i, i \in \{1 \dots i\}$, denote its K_i generalized left eigenvectors as $v_{i,1}, v_{i,2}, \dots v_{i,K_i}$. By definition, $\sum_{i=1}^{I} K_i = n$ and

$$\begin{aligned} v_{i,1}^*(A + BF) &= v_{i,1}^* \cdot \eta_i \\ v_{i,2}^*(A + BF) &= v_{i,1}^* + v_{i,2}^* \cdot \eta_i \\ &\vdots \\ v_{i,K_i}^*(A + BF) &= v_{i,K_i-1}^* + v_{i,K_i}^* \cdot \eta_i \end{aligned}$$

for all $i \in \{1 \dots i\}$. Also, $v_{i,k}, i = 1 \dots I, k = 1 \dots K_i$ are linearly independent and spans $\mathbb{C}^n$. Therefore,

$$x^* = \sum_{i=1}^{I} \sum_{k=1}^{K_i} \xi_{i,k} \cdot v_{i,k}^*$$

which leads to

$$\sum_{i=1}^{I} \sum_{k=1}^{K_i} \xi_{i,k} \cdot v_{i,k}^* (A + BF) = \sum_{i=1}^{I} \sum_{k=1}^{K_i} \xi_{i,k} \cdot \lambda \cdot v_{i,k}^*$$

Since $v_{i,k}$'s are $A + BF$'s generalized eigenvectors, we have

$$\begin{aligned}
&\sum_{i=1}^{I} \sum_{k=1}^{K_i} \xi_{i,k} \cdot v_{i,k}^* \cdot (A + BF) \\
=&\sum_{i=1}^{I} \left\{ \xi_{i,1} \cdot \eta_i \cdot v_{i,1}^* + \sum_{k=2}^{K_i} \xi_{i,k} (v_{i,k-1}^* + \eta_i \cdot v_{i,k}^*) \right\} \\
=&\sum_{i=1}^{I} \left\{ \sum_{k=1}^{K_i - 1} (\xi_{i,k} \cdot \eta_i + \xi_{i,k+1}) v_{i,k}^* + \xi_{i,K_i} \cdot \eta_i \cdot v_{i,K_i}^* \right\}
\end{aligned}$$

Combining the above two equations:

$$\sum_{i=1}^{I} \left\{ \sum_{k=1}^{K_i - 1} \left[\xi_{i,k} \cdot (\eta_i - \lambda) + \xi_{i,k+1} \right] v_{i,k}^* + \xi_{i,K_i} \cdot (\eta_i - \lambda) \cdot v_{i,K_i}^* = 0 \right\}$$

Since $v_{i,k}$'s are linearly independent, for any $i \in \{i \dots I\}$:

$$\begin{aligned}
\xi_{i,1} \cdot (\eta_i - \lambda) + \xi_{i,2} = 0 &\Rightarrow \xi_{i,2} = (-1) \cdot \xi_{i,1} \cdot (\eta_i - \lambda) \\
\xi_{i,2} \cdot (\eta_i - \lambda) + \xi_{i,3} = 0 &\Rightarrow \xi_{i,3} = (-1)^2 \cdot \xi_{i,1} \cdot (\eta_i - \lambda)^2 \\
&\vdots \\
\xi_{i,K_i - 1} \cdot (\eta_i - \lambda) + \xi_{i,K_i} = 0 &\Rightarrow \xi_{i,K_i} = (-1)^{K_i - 1} \cdot \xi_{i,1} \cdot (\eta_i - \lambda)^{K_i - 1} \\
\xi_{i,K_i} \cdot (\eta_i - \lambda) = 0 &
\end{aligned}$$

Thus,

$$(-1)^{K_i - 1} \cdot \xi_{i,1} \cdot (\eta_i - \lambda)^{K_i} = 0$$

Denote $\xi_{i,1}$ as $r_1 e^{\theta_1}$, $(\eta_i - \lambda)$ as $r_2 e^{\theta_2}$. Since $\mathrm{Re}\lambda \geq 0, \mathrm{Re}(\eta_i) < 0$, $r_2 > 0$. On the other hand, the following equation suggests

$$r_1 r_2^{K_i - 1} e^{j[\theta_1 + \theta_2 (K_i - 1)]} = 0$$

Thus, r_1 has to be 0, which implies $\xi_{i,1} = 0$. By recursion, $\xi_{i,k} = 0, \forall k = 1 \dots K_i$. Contradiction!

($1. \Rightarrow 4.$) If (A, B) is controllable, then from Theorem **??**(thm:lticontrollable)'s sixth definition, we can freely assign the poles of $A+BF$ via choosing F properly.

Otherwise, if (A, B) is uncontrollable, then from Corollary B.4 and proof of Theorem B.5 ($6. \Rightarrow 1.$), there exists a similarity transformation T, s.t.

$$TAT^{-1} = \begin{bmatrix} \bar{A}_c & \bar{A}_{12} \\ 0 & \bar{A}_{\bar{c}} \end{bmatrix}, \quad TB = \begin{bmatrix} \bar{B}_c \\ 0 \end{bmatrix}$$

and

$$\det(A + BF - \lambda I) = \underbrace{\det(\bar{A}_c + \bar{B}_c F_1 - \lambda I_1)}_{\chi_c(\lambda)} \cdot \underbrace{\det(\bar{A}_{\bar{c}} - \lambda I_2)}_{\chi_{\bar{c}}(\lambda)}$$

where $\bar{A}_c \in \mathbb{C}^{k_1 \times k_1}$, I_1 identity matrix of size k_1, $[F_1, F_2] = FT^{-1}$, and $k_1 = \text{rank}\mathcal{C}$. Additionally, $(\bar{A}_c, \bar{B}_c)$ is controllable. Thus, $\chi_c(\lambda)$'s zeros can be freely assigned by choosing proper F, i.e., system modes with $\chi_c(\lambda)$ is controllable, regardless of its stability. On the other hand, system modes with $\chi_{\bar{c}}(\lambda)$ must be stable. Otherwise, we cannot affect it by assigning F, which is a contradiction to statement (1). Therefore, (TAT^{-1}, TB) is stabilizable. Since similarity transformation does not change stabilizability, (A, B) is stabilizable. □

B.3.2 Equivalent Statements for Detectability

Thanks to duality, we can directly write down the equivalent statements of observability without any additional proofs:

Theorem B.9 (Equivalent Statements for Detectability)**.** *The following statements are equivalent (Zhou et al., 1996):*

1. *(C, A) is detectable.*
2. *For all λ and x such that $Ax = \lambda x$ and $Re\lambda \geq 0$, $Cx \neq 0$.*
3. *The matrix $\begin{bmatrix} A - \lambda I \\ C \end{bmatrix}$ has full rank for all $Re\lambda \geq 0$.*
4. *There exists a matrix L such that $A + LC$ are Hurwitz.*
5. *(A^*, C^*) is stabilizable.*

Bibliography

Antos, A., Szepesvári, C., and Munos, R. (2007). Fitted q-iteration in continuous action-space mdps. *Advances in neural information processing systems*, 20.

Arnold, W. F. and Laub, A. J. (1984). Generalized eigenproblem algorithms and software for algebraic riccati equations. *Proceedings of the IEEE*, 72(12):1746–1754.

Baird, L. et al. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the twelfth international conference on machine learning*, pages 30–37.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (2012). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846.

Chen, C.-T. (1984). *Linear system theory and design*. Saunders college publishing.

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G. A., and Burgard, W. (2005). *Principles of robot motion: theory, algorithms, and implementations*. MIT press.

Davison, E. and Wonham, W. (1968). On pole assignment in multivariable linear systems. *IEEE Transactions on Automatic Control*, 13(6):747–748.

Fan, J., Wang, Z., Xie, Y., and Yang, Z. (2020). A theoretical analysis of deep q-learning. In *Learning for dynamics and control*, pages 486–489. PMLR.

Garrigos, G. and Gower, R. M. (2023). Handbook of convergence theorems for (stochastic) gradient methods. *arXiv preprint arXiv:2301.11235*.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. Pmlr.

Janner, M., Fu, J., Zhang, M., and Levine, S. (2019). When to trust your model: Model-based policy optimization. *Advances in neural information processing systems*, 32.

Kakade, S. M. (2001). A natural policy gradient. *Advances in neural information processing systems*, 14.

Kang, S., Xu, X., Sarva, J., Liang, L., and Yang, H. (2024). Fast and certifiable trajectory optimization. In *International Workshop on the Algorithmic Foundations of Robotics*.

Kearns, M. J. and Singh, S. (2000). Bias-variance error bounds for temporal difference updates. In *COLT*, pages 142–147.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Liu, D. C. and Nocedal, J. (1989). On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528.

Mahmood, A. R., Yu, H., White, M., and Sutton, R. S. (2015). Emphatic temporal-difference learning. *arXiv preprint arXiv:1507.01569*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Munos, R. and Szepesvári, C. (2008). Finite-time bounds for fitted value iteration. *Journal of Machine Learning Research*, 9(5).

Nesterov, Y. (2018). *Lectures on convex optimization*, volume 137. Springer.

Nocedal, J. and Wright, S. J. (1999). *Numerical optimization*. Springer.

Rawlings, J. B., Mayne, D. Q., and Diehl, M. (2020). *Model predictive control: theory, computation, and design*, volume 2. Nob Hill Publishing Madison, WI.

Riedmiller, M. (2005). Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European conference on machine learning*, pages 317–328. Springer.

Robbins, H. and Siegmund, D. (1971). A convergence theorem for non negative almost supermartingales and some applications. In *Optimizing methods in statistics*, pages 233–257. Elsevier.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015a). Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015b). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, C., and Wiewiora, E. (2009). Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th annual international conference on machine learning*, pages 993–1000.

Sutton, R. S., Szepesvári, C., and Maei, H. R. (2008). A convergent o(n) algorithm for off-policy temporal-difference learning with linear function approximation. *Advances in neural information processing systems*, 21(21):1609–1616.

Wächter, A. and Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57.

Williams, G., Drews, P., Goldfain, B., Rehg, J. M., and Theodorou, E. A. (2016). Aggressive driving with model predictive path integral control. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 1433–1440. IEEE.

Zhou, K., Doyle, J., and Glover, K. (1996). Robust and optimal control. *Control Engineering Practice*, 4(8):1189–1190.