Practical Reinforcement Learning: From Algorithms to Applications

Pierre-Luc Bacon

Monday 3rd November, 2025

Curvenote

1 Modeling

1.1 Why Build a Model? For Whom?

> "The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work."
> — John von Neumann

The word *model* means different things depending on who you ask.

In machine learning, it typically refers to a parameterized function (often a neural network) fit to data. When we say *"we trained a model,"* we usually mean adjusting parameters so it makes good predictions. But that's a narrow view.

In control, operations research, or structural economics, a model refers more broadly to a formal specification of a decision problem. It includes how a system evolves over time, what parts of the world we choose to represent, what decisions are available, what can be observed or measured, and how outcomes are evaluated. It also encodes assumptions about time (discrete or continuous, finite or infinite horizon), uncertainty, and information structure.

To clarify terminology, I'll use the term decision-making model to refer to this broader object: one that includes not just system dynamics, but also a specification of state, control, observations, objectives, time structure, and information assumptions. In this sense, the model defines the structure of the decision problem. It's the formal scaffold on which we build optimization or learning procedures.

Depending on the setting, we may ask different things from a decision-making model. Sometimes we want a model that supports counterfactual reasoning or policy evaluation, and are willing to bake in more assumptions to get there. Other times, we just need a model that supports prediction or simulation, even if it remains agnostic about internal mechanisms.

This mirrors a interesting distinction in econometrics between structural and reduced-form approaches. Structural models aim to capture the underlying process that generates behavior, enabling reasoning about what would happen under alternative policies or conditions. Reduced-form models, by contrast, focus on capturing statistical regularities (often to estimate causal effects) without necessarily modeling the mechanisms that generate them. Both are forms of modeling, just with different goals. The same applies in control and RL: some models are built to support simulation and optimization, while others serve more diagnostic or predictive roles, with fewer assumptions about how the system works internally.

This chapter steps back from algorithms to focus on the modeling side. What kinds of models do we need to support decision-making from data? What are their assumptions? What do they let us express or ignore? And how do they shape what learning and optimization can even mean?

1.2 Modeling, Realism, and Control

Realism is only one way to assess a model. When the purpose of modeling is to support control or decision making, accuracy in reproducing every detail of the system is not always necessary. What matters more is whether the model leads to decisions that perform well when applied in practice. A model may simplify the physics, ignore some variables, or group complex interactions into a disturbance term. As long as it retains the core feedback structure relevant to the control task, it can still be effective.

In some cases, high-fidelity models can be counterproductive. Their complexity makes them harder to understand, slower to simulate, and more difficult to tune. Worse, they may include uncertain parameters that do not affect the control decisions but still influence the outcome of optimization. The resulting decisions can become fragile or overfitted to details that are not stable across different operating conditions.

A useful model for control is one that focuses on the variables, dynamics, and constraints that shape the decisions to be made. It should capture the key trade-offs without trying to account for every effect. In traditional control design, this principle appears through model simplification: engineers reduce the system to a manageable form, then use feedback to absorb remaining uncertainty. Reinforcement learning adopts a similar mindset, though often implicitly. It allows for model error and evaluates success based on the quality of the policy when deployed, rather than on the accuracy of the model itself.

1.2.1 Example: A simple model that supports better decisions

Researchers at the U.S. National Renewable Energy Laboratory investigated how to reduce cooling costs in a typical home in Austin, Texas [Cole et al., 2014]. They had access to a detailed EnergyPlus simulation of the building, which included thousands of internal variables: layered wall models, HVAC cycling behavior, occupancy schedules, and detailed weather inputs.

Although this simulator could closely reproduce indoor temperatures, it was too slow and too complex to use as a planning tool. Instead, the researchers constructed a much simpler model using just two parameters: an effective thermal resistance and an effective thermal capacitance. This reduced model did not capture short-term temperature fluctuations and could be off by as much as two degrees on hot afternoons.

Despite these inaccuracies, the simplified model proved useful for testing different cooling strategies. One such strategy involved cooling the house early in the morning when electricity prices were low, letting the temperature rise slowly during the expensive late-afternoon period, and reheating only slightly overnight. When this strategy was simulated in the full EnergyPlus model, it reduced peak compressor power by approximately 70 percent and lowered total cooling cost by about 60 percent compared to a standard thermostat schedule.

The reason this worked is that the simple model captured the most important structural feature of the system: the thermal mass of the building acts as a

buffer that allows load shifting over time. That was enough to discover a control strategy that exploited this property. The many other effects present in the full simulation did not change the main conclusions and could be treated as part of the background variability.

This example shows that a model can be inaccurate in detail but still highly effective in guiding decisions. For control, what matters is not whether the model matches reality in every respect, but whether it helps identify actions that perform well under real-world conditions.

1.3 Dynamics Models for Decision Making

The kind of model we need here is a **dynamics model**. It does not just describe correlations. It tells us how a system **evolves in time** and, most importantly for control, how that evolution **responds to inputs** we choose.

A dynamics model earns its keep by answering counterfactuals of the form: *given an initial condition and an input schedule, what trajectory should I expect?* That ability to roll a trajectory forward under different candidate inputs is the backbone of planning, policy evaluation, and learning from interaction.

At this level, we can think of the model as a trajectory generator:

$$(\mathbf{x}_0,\ \{\mathbf{u}_t\},\ \{\mathbf{d}_t\}) \longmapsto \{\mathbf{x}_t,\ \mathbf{y}_t\}_{t=0:T}, \tag{1}$$

where $\mathbf{u}_t$ are **controls** we set, $\mathbf{d}_t$ are **exogenous drivers** we do not control (weather, inflow, demand), $\mathbf{x}_t$ are internal **system variables**, and $\mathbf{y}_t$ are **observations**. The split between $\mathbf{u}$ and $\mathbf{d}$ is practical: it separates what we can act on from what we must accommodate.

Two design pressures shape such models:

1. **Responsiveness to inputs.** The model must expose the levers that matter for the decision problem, even if everything underneath is approximate.

2. **Memory management.** To simulate step by step, we need a compact summary of the past that is sufficient to predict the next step once an input arrives. That summary is what we will call the **state**.

This brings us to a standard but powerful representation. Rather than carry the full history, we look for a variable $\mathbf{x}_t$ that captures "what matters so far" for predicting what comes next under a given input. With that variable in hand, the model advances in small increments and can be composed with estimators and controllers.

With this motivation in place, we can now introduce the formalism.

1.4 The State-Space Perspective

Most dynamics models, whether derived from physics or learned from data, can be cast into **state-space form**. The state $\mathbf{x}$ is the compact memory that summarizes the past for prediction and control. Inputs $\mathbf{u}$ perturb that state, exogenous drivers $\mathbf{d}$ push it around, and outputs $\mathbf{y}$ are what we can measure. The equations look the same whether time is treated in discrete steps or as a continuous variable.

1.4.1 Discrete versus continuous time

How we represent time is dictated by how we sense and actuate: digital controllers sample and apply inputs in steps; the underlying physics evolve continuously.

Time can be represented in two complementary ways, depending on how the system is sensed, actuated, or modelled.

In **discrete time**, we treat time as an integer counter, $t = 0, 1, 2, \ldots$, advancing in fixed steps. This matches how digital systems operate: sensors are sampled periodically, decisions are made at regular intervals, and most logged data takes this form.

Continuous time treats time as a real variable, $t \in R_{\geq 0}$. Many physical systems (mechanical, thermal, chemical) are most naturally expressed this way, using differential equations to describe how state changes.

The two views are interchangeable to some extent. A continuous-time model can be discretized through numerical integration, although this involves approximation. The degree of approximation depends on both the step size Δt and the integration algorithm used. Conversely, a discrete-time policy can be extended to continuous time by holding inputs constant over time intervals (a zeroth-order hold), or by interpolating between values.

In physical systems, this hybrid setup is almost always present. Control software sends discrete commands to hardware (say, the output of a PID controller) which are then processed by a DAC (digital-to-analog converter) and applied to the plant through analog signals. The hardware might hold a voltage constant, ramp it, or apply some analog shaping. On the sensing side, continuous signals are sampled via ADCs before reaching a digital controller. So in practice, even systems governed by continuous dynamics end up interfacing with the digital world through discrete-time approximations.

This raises a natural question: if everything eventually gets discretized anyway, why not just model everything in discrete time from the start?

In many cases, we do. But continuous-time models can still be useful, sometimes even necessary. They often make physical assumptions more explicit, connect more naturally to domain knowledge (e.g. differential equations in mechanics or thermodynamics), and expose invariances or conserved quantities that get obscured by time discretization. They also make it easier to model systems at different time scales, or to reason about how behaviors change as resolution increases. So while implementation happens in discrete time, thinking in continuous time can clarify the structure of the model.

Still, it's helpful to see how both representations look in mathematical form. The state-space equations are nearly identical with different notations depending on how time is represented.

Discrete time

Having defined state as the summary we carry forward, a step of prediction applies the chosen input and advances the state.

$\mathbf{x}_{t+1} = f_t(\mathbf{x}_t, \mathbf{u}_t), \qquad \mathbf{y}_t = h_t(\mathbf{x}_t, \mathbf{u}_t).$

Continuous time

$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)), \qquad \mathbf{y}(t) = h(\mathbf{x}(t), \mathbf{u}(t)).$

The dot denotes a derivative with respect to real time; everything else (state, control, observation) remains the same.

When the functions f and h are linear we obtain

Linearity is not a belief about the world, it is a modeling choice that trades fidelity for transparency and speed.

$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}, \qquad \mathbf{y} = C\mathbf{x} + D\mathbf{u}.$

The matrices A, B, C, D may vary with t. Readers with an ML background will recognise the parallel with recurrent neural networks: the state is the hidden vector, the control the input, and the output the read-out layer.

Classical control often moves to the frequency domain, using Laplace and Z-transforms to turn differential and difference equations into algebraic ones. That is invaluable for stability analysis of linear time-invariant systems, but the time-domain state-space view is more flexible for learning and simulation, so we will keep our primary focus there.

1.5 Examples of Deterministic Dynamics: HVAC Control

Imagine you're in Montréal, in the middle of February. Outside it's -20°C, but inside your home, a thermostat tries to keep things comfortable. When the indoor temperature drops below your setpoint, the heating system kicks in. That system (a small building, a heater, the surrounding weather) can be modeled mathematically.

We start with a very simple approximation: treat the entire room as a single "thermal mass," like a big air-filled box that heats up or cools down depending on how much heat flows in or out.

Let $\mathbf{x}(t)$ be the indoor air temperature at time t, and $\mathbf{u}(t)$ be the heating power supplied by the HVAC system. The outside air temperature, denoted $\mathbf{d}(t)$, affects the system too, acting as a known disturbance. Then the rate of change of indoor temperature is:

$$\dot{\mathbf{x}}(t) = -\frac{1}{RC}\mathbf{x}(t) + \frac{1}{RC}\mathbf{d}(t) + \frac{1}{C}\mathbf{u}(t). \tag{2}$$

Here:

- R is a thermal resistance: how well the walls insulate.
- C is a thermal capacitance: how much energy it takes to heat the air.

This is a **continuous-time linear system**, and we can write it in standard state-space form:

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) + \mathbf{E}\mathbf{d}(t), \quad \mathbf{y}(t) = \mathbf{C}\mathbf{x}(t), \tag{3}$$

with:

- $\mathbf{x}(t)$: indoor air temperature (the state)
- $\mathbf{u}(t)$: heater input (the control)
- $\mathbf{d}(t)$: outdoor temperature (disturbance)
- $\mathbf{y}(t)$: observed indoor temperature (output)

- $\mathbf{A} = -\frac{1}{RC}$
- $\mathbf{B} = \frac{1}{C}$
- $\mathbf{E} = \frac{1}{RC}$
- $\mathbf{C} = 1$

This model is simple, but too simplistic. It ignores the fact that the walls themselves store heat and release it slowly. This kind of delay is called **thermal inertia**: even if you turn the heater off, the walls might continue to warm the room for a while.

To capture this effect, we need to expand our state to include the wall temperature. We now model two coupled thermal masses: one for the air, and one for the wall. Heat can flow from the heater into the air, from the air into the wall, and from the wall out to the environment. This gives a more realistic description of how heat moves through a building envelope.

We write down an energy balance for each mass:

- For the air:

$$C_{\text{air}} \frac{dT_{\text{in}}}{dt} = \frac{T_{\text{wall}} - T_{\text{in}}}{R_{\text{ia}}} + u(t), \tag{4}$$

- For the wall:

$$C_{\text{wall}} \frac{dT_{\text{wall}}}{dt} = \frac{T_{\text{out}} - T_{\text{wall}}}{R_{\text{wo}}} - \frac{T_{\text{wall}} - T_{\text{in}}}{R_{\text{ia}}}. \tag{5}$$

Each term on the right-hand side corresponds to a flow of heat: the air gains heat from the wall and the heater, and the wall exchanges heat with both the air and the outside.

Now define the state vector:

$$\mathbf{x}(t) = \begin{bmatrix} T_{\text{in}}(t) \\ T_{\text{wall}}(t) \end{bmatrix}, \quad \mathbf{u}(t) = u(t), \quad \mathbf{d}(t) = T_{\text{out}}(t). \tag{6}$$

Dividing both equations by their respective capacitances and rearranging terms, we arrive at the coupled system:

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) + \mathbf{E}\mathbf{d}(t), \quad \mathbf{y}(t) = \mathbf{C}\mathbf{x}(t), \tag{7}$$

with:

$$\mathbf{A} = \begin{bmatrix} -\frac{1}{R_{\text{ia}} C_{\text{air}}} & \frac{1}{R_{\text{ia}} C_{\text{air}}} \\ \frac{1}{R_{\text{ia}} C_{\text{wall}}} & -\left(\frac{1}{R_{\text{ia}}} + \frac{1}{R_{\text{wo}}}\right) \frac{1}{C_{\text{wall}}} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \frac{1}{C_{\text{air}}} \\ 0 \end{bmatrix}, \quad \mathbf{E} = \begin{bmatrix} 0 \\ \frac{1}{R_{\text{wo}} C_{\text{wall}}} \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1 & 0 \end{bmatrix}. \tag{8}$$

Each entry in $\mathbf{A}$ has a physical interpretation:

- A_{11}: heat loss from the air to the wall
- A_{12}: heat gain by the air from the wall
- A_{21}: heat gain by the wall from the air
- A_{22}: net loss from the wall to both the air and the outside

The temperatures are now dynamically coupled: any change in one affects the other. The wall acts as a buffer that absorbs and releases heat over time.

This is still a linear system, just with a 2D state. But already it behaves differently. The walls absorb and release heat, smoothing out fluctuations and slowing down the system's response.

As we add more rooms, walls, or building elements, the system grows. Each new temperature adds a new state. The equations still have the same structure, and their sparsity follows the building layout. Nodes represent temperatures; edges encode how heat flows between them.

1.5.1 What Do We Control?

This network of states is what we control. What we mean by "control input" $\mathbf{u}(t)$ depends on both what we want to achieve and what we can implement in practice.

The most direct interpretation is to let $\mathbf{u}(t)$ represent the actual heating power delivered to the system, measured in watts. This makes sense when modeling from physical principles or simulating a system with fine-grained actuation.

In many real buildings, however, thermostats don't issue power commands. They activate a relay, turning the heater on or off based on whether the measured temperature crosses a setpoint. Some systems allow for modulated control—such as varying fan speed or partially opening a valve—but those details are often hidden behind firmware or closed controllers.

A common implementation involves a **PID control loop** that compares the measured temperature to a setpoint and adjusts the control signal accordingly. While the actual logic might be simple, the resulting behavior appears smoothed or delayed from the perspective of the building.

Depending on the abstraction level, we might:

- Treat $\mathbf{u}(t)$ as continuous power input, if designing the full control logic.
- Use it as a setpoint input, assuming a lower-level controller handles the rest.
- Or reduce it to a binary signal—heater on or off—when working with logged behavior from a smart thermostat.

Each perspective shapes the kind of model we build and the kind of control problem we pose. If we're aiming to design a controller from scratch, it may be

worth modeling the full closed-loop dynamics. If the goal is to tune setpoints or learn policies from data, a coarser abstraction might be not only sufficient, but more robust.

1.5.2 Why This Model?

At this point, you might wonder: why go through the trouble of building this kind of physics-based model at all? After all, if we can log indoor temperatures, thermostat actions, and weather data, isn't it easier to just learn a model from data? A neural ODE, for example, would let us define a parameterized function:

$$\dot{\mathbf{z}}(t) = f_{\boldsymbol{\theta}}(\mathbf{z}(t), \mathbf{u}(t), \mathbf{d}(t)), \quad \mathbf{y}(t) = g_{\boldsymbol{\theta}}(\mathbf{z}(t)), \tag{9}$$

with both $f_{\boldsymbol{\theta}}$ and $g_{\boldsymbol{\theta}}$ learned from data. The internal state $\mathbf{z}(t)$ is not tied to any physical quantity. It just needs to be expressive enough to explain the observations.

That flexibility can be useful, particularly when a large dataset is already available. But in building control and energy modeling, the constraints are usually different.

Often, the engineer or consultant on site is working under tight time and information budgets. A floor plan might be available, along with some basic specs on insulation or window types, and a few days of logged sensor data. The task might be to simulate load under different weather scenarios, tune a controller, or just help understand why a room is slow to heat up. The model has to be built quickly, adapted easily, and remain understandable to others working on the same system.

In that context, RC models are often the default choice: not because they are inherently better, but because they fit the workflow.

Interpretability. The parameters correspond to things you can reason about: thermal resistance, capacitance, heat transfer between zones. You can cross-check values against architectural plans, or adjust them manually when something doesn't line up. You can tell which wall or zone is contributing to slow recovery times.

Identifiability with limited data. RC models can often be calibrated from short data traces, even when not all state variables are directly observable. The structure already imposes constraints: heat flows from hot to cold, dynamics are passive, responses are smooth. Those properties help narrow the space of valid parameter settings. A neural ODE, in contrast, typically needs more data to settle into stable and plausible dynamics—especially if no additional constraints are enforced during training.

Simplicity and reuse. Once the model is built, it's straightforward to modify. If a window is replaced, or a wall gets insulated, you only need to update a few numbers. It's easy to pass along to another engineer or embed in a larger simulation. A model like

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{E}\mathbf{d} \tag{10}$$

is linear and low-dimensional. Simulating it is cheap, even if you do it many times. That may not matter now, but it will matter later, when we want to optimize over trajectories or learn from them.

This doesn't mean RC models are always sufficient. They simplify or ignore many effects: solar gains, occupancy, nonlinearities, humidity, equipment switching behavior. If those effects are significant, and you have enough data, a black-box model (neural ODE or otherwise) might achieve lower prediction error. In practice, though, it's common to combine the two: use the RC structure as a backbone, and learn a residual model to correct for unmodeled dynamics.

1.6 From Deterministic to Stochastic

The models we've seen so far were deterministic: given an initial state and input sequence, the system evolves in a fixed, predictable way. But real systems rarely behave so neatly. Sensors are noisy. Parameters drift. The world changes in ways we can't fully model.

To account for this uncertainty, we move from deterministic dynamics to **stochastic models**. There are two equivalent but conceptually distinct ways to do this.

1.6.1 Function plus Noise

The most direct extension adds a noise term to the dynamics:

$$\mathbf{x}_{t+1} = f_t(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w}_t), \quad \mathbf{w}_t \sim p_{\mathbf{w}}. \tag{11}$$

If the noise is additive and Gaussian, we recover the standard linear-Gaussian setup used in Kalman filtering:

$$\mathbf{x}_{t+1} = A\mathbf{x}_t + B\mathbf{u}_t + \mathbf{w}_t, \quad \mathbf{w}_t \sim \mathcal{N}(0, Q). \tag{12}$$

But we're not restricted to Gaussian or additive noise. For instance, if the noise distribution is non-Gaussian:

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t) + \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{Laplace, or Student-t}, \tag{13}$$

then $\mathbf{x}_{t+1}$ inherits those properties. This is known as a **convolution model**: the next-state distribution is a shifted version of the noise distribution, centered around the deterministic prediction. More formally, it's a special case of a **pushforward measure**: the randomness from $\mathbf{w}_t$ is "pushed forward" through the function f to yield a distribution over outcomes.

Or the noise might enter multiplicatively:

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t) + \Gamma(\mathbf{x}_t, \mathbf{u}_t)\mathbf{w}_t, \tag{14}$$

where Γ is a matrix that modulates the effect of the noise, potentially depending on state and control. If Γ is invertible, we can even write down an explicit density via a change-of-variables:

$$p(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \mathbf{u}_t) = p_{\mathbf{w}} \left(\Gamma^{-1}(\mathbf{x}_t, \mathbf{u}_t) \left[\mathbf{x}_{t+1} - f(\mathbf{x}_t, \mathbf{u}_t) \right] \right) \cdot \left| \det \Gamma^{-1} \right|. \tag{15}$$

This kind of structured noise is common in practice, for example, when disturbances are amplified at certain operating points.

The **function-plus-noise** view is natural when we have a physical or simulator-based model and want to account for uncertainty around it. It is **constructive**: we know how the system evolves and how the randomness enters. This means we can **track the source of variability along a trajectory**, which is particularly useful for techniques like **reparameterization** or **infinitesimal perturbation analysis (IPA)**. These methods rely on being able to differentiate through the noise injection mechanism, something that is much easier when the noise is explicit and structured.

1.6.2 Transition Kernel

The second perspective skips over the internal noise and defines the system directly in terms of the probability distribution over next states:

$$p(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \mathbf{u}_t). \tag{16}$$

This **transition kernel** encodes all the uncertainty in the system's evolution, without reference to any underlying noise source or functional form.

This view is strictly more general: it includes the function-plus-noise case as a special instance. If we do know the function f and the noise distribution $p_{\mathbf{w}}$ from the generative model, then the transition kernel is obtained by "pushing" the randomness through the function:

$$p(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \mathbf{u}_t) = \int \delta(\mathbf{x}_{t+1} - f(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w}))\, p_{\mathbf{w}}(\mathbf{w})\, d\mathbf{w}. \tag{17}$$

This might look abstract, but it's just marginalization: for each possible noise value $\mathbf{w}$, we compute the resulting next state, and then average over all possible $\mathbf{w}$, weighted by how likely each one is.

If the noise were discrete, this becomes a sum:

$$p(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \mathbf{u}_t) = \sum_{i=1}^{k} 1\{f(\mathbf{x}_t, \mathbf{u}_t, w_i) = \mathbf{x}_{t+1}\} \cdot p_i \tag{18}$$

This abstraction is especially useful when we don't know (or don't care about) the underlying function or noise distribution. All we need is the ability to sample transitions or estimate their likelihoods. This is the default formulation in reinforcement learning, econometrics, and other settings focused on behavior rather than mechanism.

1.6.3 Continuous-Time Analogue

In continuous time, the stochastic dynamics of a system are often described using a **stochastic differential equation (SDE)**:

$$d\mathbf{X}_t = f(\mathbf{X}_t, \mathbf{U}_t)\, dt + \sigma(\mathbf{X}_t, \mathbf{U}_t)\, d\mathbf{W}_t, \tag{19}$$

where $\mathbf{W}_t$ is Brownian motion. The first term, called the **drift**, describes the average motion of the system. The second, scaled by σ, models how random fluctuations (diffusion) enter over time. Just like in discrete time, this is a **function + noise** model: the state evolves through a deterministic path perturbed by stochastic input.

This generative view again induces a probability distribution over future states. At any future time $t + \Delta t$, the system doesn't land at a single state but is described by a distribution that depends on the initial condition and the noise along the way.

Mathematically, this distribution evolves according to what's called the **Fokker–Planck equation**—a partial differential equation that governs how probability density "flows" through time. It plays the same role here as the transition kernel did in discrete time: describing how likely the system is to be in any given state, without referring to the noise directly.

While the mathematical generalization is clean, working with continuous-time stochastic models can be more challenging. Simulating sample paths is often straightforward (eg. nowadays diffusion models in generative AI), but writing down or computing the exact transition distribution usually isn't. That's why many practical methods still rely on discrete-time approximations, even when the underlying system is continuous.

Example: Managing a Québec Hydroelectric Reservoir On the James Bay plateau, 1 400 km north of Montréal, the Robert-Bourassa reservoir stores roughly 62 km^3 of water, more than the volume of Lake Ontario above its minimum operating level. Sixteen giant turbines sit 140 m below the surface, converting that stored head into 5.6 GW of electricity, about a fifth of Hydro-Québec's total capacity. A steady share of that output feeds Québec's aluminium smelters, which depend on stable, uninterrupted power.

Water managers face competing objectives:

- **Flood safety.** Sudden snowmelt or storms can overfill the basin, forcing emergency spillways to open. These events are spectacular, but carry real downstream risk and economic cost.
- **Energy reliability.** If the level falls too low, turbines sit idle and contracts go unmet. Voltage dips at the smelters are measured in lost millions.

A basic deterministic model for the reservoir's mass balance is just bookkeeping:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{r}_t - \mathbf{u}_t, \tag{20}$$

where $\mathbf{x}_t$ is the current reservoir level, $\mathbf{u}_t$ is the controlled outflow through turbines, and $\mathbf{r}_t$ is the natural inflow from rainfall and upstream runoff.

But inflow is variable, and its statistical structure matters. Two hydrological regimes dominate:

- In spring, melting snow over days can produce a long-tailed inflow distribution, often modeled as log-normal or Gamma.
- In summer, convective storms yield a skewed mixture: a point mass at zero (no rain), and a thin but heavy tail capturing sudden bursts.

This motivates a simple stochastic extension:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \mathbf{u}_t + \mathbf{w}_t, \quad \mathbf{w}_t \sim \begin{cases} 0 & \text{with prob. } p_0, \\ \text{LogNormal}(\mu, \sigma^2) & \text{with prob. } 1 - p_0. \end{cases} \tag{21}$$

Here the physics is fixed, and all uncertainty sits in the inflow term $\mathbf{w}_t$. Rather than fitting a full transition model from $(\mathbf{x}_t, \mathbf{u}_t)$ to $\mathbf{x}_{t+1}$, we can isolate the inflow by rearranging the mass balance:

$$\hat{\mathbf{w}}_t = \mathbf{x}_{t+1} - \mathbf{x}_t + \mathbf{u}_t. \tag{22}$$

This gives a direct estimate of the realized inflow at each timestep. From there, the problem becomes one of density estimation: fit a probabilistic model to the residuals $\hat{\mathbf{w}}_t$. In spring, this might be a log-normal distribution. In summer, a two-part mixture: a point mass at zero, and an exponential tail. These distributions can be estimated by maximum likelihood, or adjusted using additional features (covariates) such as upstream snowpack or forecasted temperature.

This setup has practical benefits. Fixing the physical part of the model (how levels respond to inflow and outflow) helps focus the statistical modeling effort. Rather than fitting a full system model, we only need to estimate the variability in inflows. This reduces the number of degrees of freedom and makes the estimation problem easier to interpret. It also avoids conflating uncertainty in inflow with uncertainty in the system's response.

Compare this to a more generic approach, such as linear regression:

$$\mathbf{x}_{t+1} = a\mathbf{x}_t + b\mathbf{u}_t + \varepsilon_t. \tag{23}$$

This is straightforward to fit, but offers no guarantee that the result behaves sensibly. The model might violate conservation of mass, or compensate for inflow variation by adjusting coefficients a and b. This can lead to misleading conclusions, especially when extrapolating beyond the training data.

1.7 Partial Observability

So far, we've assumed that the full system state $\mathbf{x}_t$ is available. But in most real-world settings, only a partial or noisy observation is accessible. Sensors have limited coverage, measurements come with noise, and some variables aren't observable at all.

To model this, we introduce an **observation equation** alongside the system dynamics:

$$\begin{aligned} \mathbf{x}_{t+1} &= f_t(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w}_t), \quad \mathbf{w}_t \sim p_{\mathbf{w}}, \\ \mathbf{y}_t &= h_t(\mathbf{x}_t, \mathbf{v}_t), \quad \mathbf{v}_t \sim p_{\mathbf{v}}. \end{aligned} \tag{24}$$

The state $\mathbf{x}_t$ evolves under control inputs $\mathbf{u}_t$ and process noise $\mathbf{w}_t$, but we don't get to see $\mathbf{x}_t$ directly. Instead, we observe $\mathbf{y}_t$, which depends on $\mathbf{x}_t$ through some possibly nonlinear, noisy function h_t. The noise $\mathbf{v}_t$ captures measurement uncertainty.

This setup defines a partially observed system. Even if the underlying dynamics are known, we still face uncertainty due to limited visibility into the true state. The controller or estimator must rely on the observations $\mathbf{y}_{0\,t}$ to make sense of the hidden trajectory.

In the **deterministic** case, if the output map h_t is full-rank and invertible, we may be able to reconstruct the state directly from the output: no filtering required. But once noise is introduced, that invertibility becomes more subtle: even if h_t is bijective, the presence of $\mathbf{v}_t$ prevents us from recovering $\mathbf{x}_t$ exactly. In this case, we must shift from inversion to estimation, often via probabilistic inference.

In the **linear-Gaussian case**, the model simplifies to:

$$\begin{aligned} \mathbf{x}_{t+1} &= A\mathbf{x}_t + B\mathbf{u}_t + \mathbf{w}_t, \quad \mathbf{w}_t \sim \mathcal{N}(0, Q), \\ \mathbf{y}_t &= C\mathbf{x}_t + D\mathbf{u}_t + \mathbf{v}_t, \quad \mathbf{v}_t \sim \mathcal{N}(0, R). \end{aligned} \tag{25}$$

This is the classical state-space model used in signal processing and control. It's fully specified by the system matrices and the covariances Q and R. The state is no longer known, but under these assumptions it can be estimated recursively using the **Kalman filter**, which maintains a Gaussian belief over $\mathbf{x}_t$.

Even when the model is nonlinear or non-Gaussian, the structure remains the same: a dynamic state evolves, and a separate observation process links it to the data we see. Many modern estimation techniques, including extended and unscented Kalman filters, particle filters, and learned neural estimators, build on this core structure.

1.7.1 Observation Kernel View

Just as we moved from function-based dynamics to transition kernels, we can abstract away the noise source and define the **observation distribution** directly:

$$p(\mathbf{y}_t \mid \mathbf{x}_t). \tag{26}$$

This kernel summarizes what the sensors tell us about the hidden state. If we know the generative model—say, that $\mathbf{y}_t = h_t(\mathbf{x}_t) + \mathbf{v}_t$ with known $p_{\mathbf{v}}$—then this kernel is induced by marginalizing out $\mathbf{v}_t$:

$$p(\mathbf{y}_t \mid \mathbf{x}_t) = \int \delta\big(\mathbf{y}_t - h_t(\mathbf{x}_t, \mathbf{v})\big)\, p_{\mathbf{v}}(\mathbf{v})\, d\mathbf{v}. \tag{27}$$

But we don't have to start from the generative form. In practice, we might define or learn $p(\mathbf{y}_t \mid \mathbf{x}_t)$ directly, especially when dealing with black-box sensors, perception models, or abstract measurement processes.

1.7.2 Example – Stabilizing a Telescope's Vision with Adaptive Optics

On Earth, even the largest telescopes can't see perfectly. As starlight travels through the atmosphere, tiny air pockets with different temperatures bend the light in slightly different directions. The result is a distorted image: instead of a sharp point, a star looks like a flickering blob. The distortion happens fast, on the order of milliseconds, and changes continuously as wind moves the turbulent layers overhead.

This is where **adaptive optics (AO)** comes in. AO systems aim to cancel out these distortions in real time. They do this by measuring how the incoming wavefront of light is distorted and using a flexible mirror to apply a counter-distortion that straightens it back out. But there's a catch: you can't observe the wavefront directly. You only get noisy measurements of its **slopes** (the angles of tilt at various points), and you have to act fast, before the atmosphere changes again.

To design a controller here, we need a model of how the distortions evolve. And that means building a decision-making model: one that includes uncertainty, partial observability, and fast feedback.

State. The main object we're trying to track is the distortion of the incoming wavefront. We can't observe this phase field $\phi(\mathbf{r}, t)$ directly, but we can represent it approximately using a finite basis (e.g., Fourier or Zernike). The coefficients of this expansion form our internal state:

$$\mathbf{x}_t \in R^n \quad \text{(wavefront distortion at time } t\text{)}. \tag{28}$$

Dynamics. The atmosphere evolves in time. A simple but surprisingly effective model assumes the turbulence is "frozen" and just blown across the telescope by the wind. That gives us a **discrete-time linear model**:

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{w}_t, \tag{29}$$

where $\mathbf{A}$ shifts the distortion pattern in space, and $\mathbf{w}_t$ is a small random change from evolving turbulence. This noise is not arbitrary: its statistics follow

a power law derived from **Kolmogorov's turbulence model**. In particular, higher spatial frequencies (small-scale wiggles) have less energy than low ones. That lets us build a prior on how likely different distortions are.

Observations. We can't see the full wavefront. Instead, we use a **wavefront sensor**: a camera that captures how the light bends. What it actually measures are local slopes: the gradients of the wavefront, not the wavefront itself. So our observation model is:

$$\mathbf{y}_t = \mathbf{C}\mathbf{x}_t + \boldsymbol{\varepsilon}_t, \tag{30}$$

where $\mathbf{C}$ is a known matrix that maps wavefront distortion to measurable slope angles, and $\boldsymbol{\varepsilon}_t$ is measurement noise (e.g., due to photon limits).

Control. Our job is to flatten the wavefront using a deformable mirror. The mirror can apply a small counter-distortion $\mathbf{u}_t$ that subtracts from the atmospheric one:

$$\text{Residual state:} \quad \mathbf{x}_t^{\text{res}} = \mathbf{x}_t - \mathbf{B}\mathbf{u}_t. \tag{31}$$

The goal is to choose $\mathbf{u}_t$ to minimize the residual distortion by making the light flat again.

Why a model matters. Without a model, we'd just react to the current noisy measurements. But with a model, we can predict how the wavefront will evolve, filter out noise, and act preemptively. This is essential in AO, where decisions must be made every millisecond. Kalman filters are often used to track the hidden state $\mathbf{x}_t$, combining model predictions with noisy measurements, and linear-quadratic regulators (LQR) or other optimal controllers use those estimates to choose the best correction.

Time structure. This is a rare case where **continuous-time modeling** also plays a role. The true evolution of the turbulence is continuous, and we can model it using a **stochastic differential equation (SDE)**:

$$d\mathbf{x}(t) = \mathbf{F}\mathbf{x}(t)\,dt + \mathbf{G}\,d\mathbf{W}(t), \tag{32}$$

where $\mathbf{W}(t)$ is Brownian motion and the matrix $\mathbf{G}$ encodes the Kolmogorov spectrum. Discretizing this equation gives us the $\mathbf{A}$ and $\mathbf{Q}$ matrices for the discrete-time model above.

1.8 Programs as Models

Up to this point, we have described models using systems of equations—either differential or difference equations—that express how a system evolves over time. These **analytical models** define the transition structure explicitly. For instance, in discrete time, the evolution of the state is governed by a known function:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) \tag{33}$$

Given access to f, we can construct trajectories, analyze system behavior, and design control policies. The important feature here is not that the model evolves one step at a time, but that we are given the **local dynamics function** f itself.

In contrast, **simulation-based models** do not expose f directly. Instead, they define a procedure—implemented in code—that takes an initial state and input sequence and returns the resulting trajectory:

$$\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_T\} = \mathcal{S}(\mathbf{x}_0, \{\mathbf{u}_t\}_{t=0}^{T-1}) \tag{34}$$

Here, $\mathcal{S}$ represents the full simulator. Internally, it may apply numerical integration, scheduling logic, branching rules, or other computations. But these details are encapsulated. From the outside, we can only query the simulator by running it.

This distinction is subtle but important. Both types of models can generate trajectories. What matters is the **interface**: analytical models provide direct access to f; simulation models do not. They offer a trajectory-generation interface, but hide the internal structure that produces it.

Case Study: Robotics — *MuJoCo*

MuJoCo illustrates this distinction well. It simulates the dynamics of articulated rigid bodies under contact constraints. The equations it solves include:

$M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}}) = \boldsymbol{\tau} + J^\top \boldsymbol{\lambda}$

$\phi(\mathbf{q}) = 0, \quad \boldsymbol{\lambda} \geq 0, \quad \boldsymbol{\lambda}^\top \phi = 0$

Here $\mathbf{q}$ are joint positions, M is the mass matrix, and $\boldsymbol{\lambda}$ are contact forces enforcing non-penetration. But these physical equations are part of a larger simulator that also includes:

- collision detection,
- contact force models,
- sensor and actuator emulation,
- and visual rendering.

The full behavior of a robot interacting with its environment emerges only when the simulator is executed. While the underlying physics are well-understood, the complexity of contact dynamics, collision detection, and sensor modeling makes it impractical to expose the local dynamics function f directly.

1.8.1 Systems with Discrete Events

Many simulation models arise when a system's dynamics are driven not by time-continuous evolution, but by the occurrence of events. These **discrete-event systems** (DES) change state only at specific, often asynchronous points in time. Between events, the state remains fixed.

A discrete-event system can be described by:

- a set of discrete states $\mathcal{X}$,
- a set of events $\mathcal{E}$,
- a transition function $f : \mathcal{X} \times \mathcal{E} \to \mathcal{X}$,
- and a time-advance function $t_a : \mathcal{X} \to R_{\geq 0}$.

At each point, the system checks which events are enabled and advances to the next scheduled one.

Example: Network Traffic Control System

Consider a software-defined networking (SDN) controller managing traffic routing in a data center. The system must make real-time decisions about packet forwarding paths based on network conditions and service requirements.

The discrete states $\mathcal{X}$ represent the current network configuration: active routing tables, link utilization levels, and quality-of-service priority queues at each switch.

The events $\mathcal{E}$ include:

- New flow requests arriving (video streaming, database queries, file transfers)
- Link failures or congestion threshold violations
- Flow completion notifications
- Load balancing triggers when servers exceed capacity
- Network policy updates from administrators

The transition function f captures how routing decisions change the network state. When a high-priority video conference flow arrives while a link is congested, the controller might transition to a new state where low-priority background traffic is rerouted through alternative paths.

The time-advance function t_a determines when the next routing decision occurs. Flow arrivals follow traffic patterns (bursty during business hours), while link failures are rare but unpredictable events.

Between events, packets follow the established routing rules—the same forwarding tables remain active across all switches. The control problem here is to adapt routing decisions to discrete network events, balancing throughput, latency, and reliability constraints.

1.8.2 Hybrid Systems

Some systems evolve continuously most of the time but undergo discrete jumps in response to certain conditions. These **hybrid systems** are common in control applications.

The system consists of:

- a set of discrete modes $q \in \mathcal{Q}$,
- continuous dynamics in each mode: $\dot{\mathbf{x}} = f_q(\mathbf{x})$,
- guards that specify when transitions between modes occur,
- and reset maps that update the state during such transitions.

Example: Thermostat Control

An HVAC system can be in one of several modes: `heating`, `cooling`, or `off`. The temperature evolves continuously according to physical laws, but when it crosses certain thresholds, the system switches modes:

```
if x < setpoint - delta:
    mode = "heating"
elif x > setpoint + delta:
    mode = "cooling"
else:
    mode = "off"
```

Within each mode, a different differential equation applies. This results in a piecewise-smooth trajectory with mode-dependent dynamics.

Case Study: Building Energy — *EnergyPlus*

EnergyPlus provides a sophisticated example of hybrid systems in building energy simulation. At its core are physical equations describing heat flows:

$$C_i \frac{dT_i}{dt} = \sum_j h_{ij} A_{ij} (T_j - T_i) + Q_i \tag{35}$$

It also solves implicit equations representing HVAC component behavior:

$$0 = f(T, \dot{m}, P) \tag{36}$$

But the actual simulator includes hundreds of thousands of lines of code handling:

- interpolated weather data,
- occupancy schedules,
- equipment performance curves,
- and control logic implemented as finite-state machines.

The result is a program that emulates how a building behaves over time, given environmental inputs and schedules. The hybrid nature emerges from the interaction between continuous thermal dynamics and discrete control decisions made by thermostats, occupancy sensors, and HVAC equipment.

1.8.3 Agent-Based Models

Some simulation models do not describe systems via global state transitions, but instead simulate the behavior of many individual components or **agents**, each following local rules. These **agent-based models** (ABMs) are widely used in epidemiology, ecology, and social modeling.

Each agent maintains its own internal state and acts according to probabilistic or rule-based logic. The system's behavior arises from the interactions among agents.

Example: Residential Energy Consumption under Dynamic Pricing

Consider a neighborhood where each household is an agent making energy consumption decisions based on real-time electricity pricing and thermal comfort preferences. Each household agent has:

- **Internal state**: current temperature, HVAC settings, comfort preferences, price sensitivity
- **Local decision rules**: MPC algorithms that optimize the trade-off between energy cost and thermal comfort
- **Unique characteristics**: different utility functions, thermal mass, occupancy patterns

The simulator might execute something like:

```
for household in neighborhood:
    # Each household solves its own MPC optimization
    current_price = utility.get_current_price()
    comfort_weight = household.comfort_preference

    # Optimize over prediction horizon
    optimal_setpoint = household.mpc_controller.optimize(
        current_temp=household.temperature,
        price_forecast=utility.price_forecast,
        comfort_weight=comfort_weight
    )

    household.set_hvac_setpoint(optimal_setpoint)

    # Update shared grid load
    neighborhood.total_demand += household.power_consumption
```

The macro-level demand patterns—peak shifting, load leveling, rebound effects—emerge from individual household optimization decisions. No single equation describes the neighborhood's energy consumption; it arises from the collective behavior of autonomous agents each solving their own control problems.

Case Study: Traffic Simulation — *SUMO*

SUMO demonstrates agent-based modeling in transportation systems. Each vehicle is an agent with its own route, driving behavior, and decision-making logic. The Krauss car-following rule shows how individual vehicle agents behave:

```
def update_vehicle(v, v_leader, gap, dt):
    v_safe = v_leader + (gap - min_gap) / tau
    v_desired = min(v_max, v_safe)
    \epsilon = random.uniform(0, 1)
    v_new = max(0, v_desired - \epsilon * a_max * dt)
    x_new = x + v_new * dt
    return x_new, v_new
```

Beyond car-following, each vehicle agent also:

- plans routes through the network based on travel time estimates,
- responds to traffic signals and road conditions,
- makes lane-changing decisions based on utility functions,
- and exhibits individual driving characteristics (aggressiveness, reaction time).

The emergent traffic patterns—congestion formation, traffic waves, bottlenecks—arise from the collective behavior of thousands of individual vehicle agents, each following local rules and making autonomous decisions.

Case Study: Modeling Curbside Access at Montréal–Trudeau (YUL)

Afternoon traffic at Montréal–Trudeau airport regularly backs up along the two-lane ramp leading to the departures curb. As passenger volumes rebound, the mix of private drop-offs, taxis, and shuttles converging in a confined space produces frequent delays. When curb dwell times rise—especially around wide-body departures—queues can spill back onto the access road and interfere with other flows on the airport campus.

To manage the situation, the airport operator relies on a dense sensor network. Cameras and license plate readers track vehicle trajectories across virtual gates, generating a real-time stream of entry points, curb interactions, and exit times. According to public statements, AI-based forecasting solutions have been deployed to anticipate congestion and suggest alternative routing options for passengers and drivers. While no technical details have been disclosed, this is a typical instance of a traffic prediction and control problem that lends itself to **agent-based modeling**.

In such a model, each vehicle is treated as an individual agent with internal state:

$$s_t = \big(\text{lane},\ x_t,\ v_t,\ \text{intent}\big), \tag{37}$$

where x_t and v_t denote longitudinal position and speed, and lane and intent capture higher-level behavioural traits. The simulation proceeds in discrete time. At each step, agents update their acceleration based on local traffic density (e.g., via a car-following model like IDM), evaluate potential lane changes (e.g., using a utility or incentive rule), and advance position accordingly.

The layout of the ramp—its geometry, merges, and constraints—is fixed. What changes are the traffic patterns and driver behaviours. These can be estimated from historical trajectories and updated as new data arrives. In a real-time setting, a filtering step adjusts the simulation so that its predicted flows remain consistent with current observations.

While the behaviour of each individual agent is governed by program logic and heuristics—such as car-following rules, desired speeds, or gap acceptance—some parameters are identified offline from historical data, while others are estimated online. This adaptation helps the model track observed conditions. But even with such adjustments, not all effects are easily captured.

Construction activity, weather disturbances, and irregular flight scheduling can introduce sudden shifts in flow that lie outside the scope of the structural model. To account for these, one can overlay a data-driven correction on top of the simulation. Suppose the simulator produces a queue length forecast q^{sim}_{t+h} over horizon h. A statistical model can be trained to predict the residual between this forecast and the observed outcome:

$$r_{t+h} = q^{\text{obs}}_{t+h} - q^{\text{sim}}_{t+h}, \tag{38}$$

as a function of exogenous features z_t, such as weather, incident flags, or scheduled arrivals. The final forecast then becomes:

$$\widehat{q}_{t+h} = q^{\text{sim}}_{t+h} + \phi(z_t), \tag{39}$$

where ϕ is a learned mapping from external conditions to the expected correction. The result is a hybrid model: the simulation enforces physical structure and agent-level behaviour, while the residual model compensates for aspects of the system that are harder to express analytically.

1.9 Looking Ahead

There's no universally correct way to model a system. Your choice depends on what you know, what you can observe, what you care about, and what tools you have.

This chapter laid out a spectrum—from explicit, mechanistic models to black-box simulators and learned dynamics. In every case, modeling choices define the structure of the problem and the space of possible solutions. In the

next chapters, we'll see how they anchor learning, optimization, and decision-making.

2 Numerical Trajectory Optimization

2.1 Discrete-Time Trajectory Optimization

In the previous chapter, we examined different ways to represent dynamical systems: continuous versus discrete time, deterministic versus stochastic, fully versus partially observable, and even simulation-based views such as agent-based or programmatic models. Our focus was on the **structure of models**: how they capture evolution, uncertainty, and information.

In this chapter, we turn to what makes these models useful for **decision-making**. The goal is no longer just to describe how a system behaves, but to leverage that description to **compute actions over time**. This doesn't mean the model prescribes actions on its own. Rather, it provides the scaffolding for optimization: given a model and an objective, we can derive the control inputs that make the modeled system behave well according to a chosen criterion.

Our entry point will be trajectory optimization. By a **trajectory**, we mean the time-indexed sequence of states and controls that the system follows under a plan: the states $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ together with the controls $(\mathbf{u}_1, \ldots, \mathbf{u}_{T-1})$. In this chapter, we focus on an **open-loop** viewpoint: starting from a known initial state, we compute the entire sequence of controls in advance and then apply it as-is. This is appealing because, for discrete-time problems, it yields a finite-dimensional optimization over a vector of decisions and cleanly exposes the structure of the constraints. In continuous time, the base formulation is infinite-dimensional; in this course we will rely on direct methods—time discretization and parameterization—to transform it into a finite-dimensional nonlinear program.

Open loop also has a clear limitation: if reality deviates from the model—due to disturbances, model mismatch, or unanticipated events—the state you actually reach may differ from the predicted one. The precomputed controls that were optimal for the nominal trajectory can then lead you further off course, and errors can compound over time.

Later, we will study **closed-loop (feedback)** strategies, where the choice of action at time t can depend on the state observed at time t. Instead of a single sequence, we optimize a policy π_t mapping states to controls, $\mathbf{u}_t = \pi_t(\mathbf{x}_t)$. Feedback makes plans resilient to unforeseen situations by adapting on the fly, but it leads to a more challenging problem class. We start with open-loop trajectory optimization to build core concepts and tools before tackling feedback design.

Discrete-Time Optimal Control Problems (DOCPs) Consider a system described by a **state** $\mathbf{x}_t \in R^n$, summarizing everything needed to predict its evolution. At each stage t, we can influence the system through a **control input** $\mathbf{u}_t \in R^m$. The dynamics specify how the state evolves:

$$\mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t), \tag{40}$$

where $\mathbf{f}_t$ may be nonlinear or time-varying. We assume the initial state $\mathbf{x}_1$ is known.

The goal is to pick a sequence of controls $\mathbf{u}_1, \ldots, \mathbf{u}_{T-1}$ that makes the trajectory desirable. But desirable in what sense? That depends on an **objective function**, which often includes two components:

$$\text{(i) stage cost: } c_t(\mathbf{x}_t, \mathbf{u}_t), \qquad \text{(ii) terminal cost: } c_T(\mathbf{x}_T). \tag{41}$$

The stage cost reflects ongoing penalties—energy, delay, risk. The terminal cost measures the value (or cost) of ending in a particular state. **Together, these give a discrete-time Bolza problem with path constraints and bounds**:

$$\begin{aligned} \text{minimize} \quad & c_T(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) \\ \text{subject to} \quad & \mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) \\ & \mathbf{g}_t(\mathbf{x}_t, \mathbf{u}_t) \leq \mathbf{0} \\ & \mathbf{x}_{\min} \leq \mathbf{x}_t \leq \mathbf{x}_{\max} \\ & \mathbf{u}_{\min} \leq \mathbf{u}_t \leq \mathbf{u}_{\max} \\ \text{given} \quad & \mathbf{x}_1 = \mathbf{x}_0 \ . \end{aligned} \tag{42}$$

Written this way, it may seem obvious that the decision variables are the controls $\mathbf{u}_t$. After all, in most intuitive descriptions of control, we think of choosing inputs to influence the system. But notice that in the program above, the entire state trajectory also appears as a set of variables, linked to the controls by the dynamics constraints. This is intentional: it reflects one way of writing the problem that makes the constraints explicit.

Why introduce $\mathbf{x}_t$ as decision variables if they can be simulated forward from the controls? Many readers hesitate here, and the question is natural: *If the model is deterministic and* $\mathbf{x}_1$ *is known, why not pick* $\mathbf{u}_{1:T-1}$ *and compute* $\mathbf{x}_{2:T}$ *on the fly?* That instinct leads to **single shooting**, a method we will return to shortly.

Already in this formulation, though, we see an important theme: **the structure of the problem matters**. Ignoring it can make our life much harder. The reason is twofold:

- **Dimensionality grows with the horizon.** For a horizon of length T, the program has roughly $(T-1)(m+n)$ decision variables.
- **Temporal coupling.** Each control affects all future states and costs. The feasible set is not a simple box but a narrow manifold defined by the dynamics.

Together, these features explain why specialized methods exist and why the way we write the problem influences the algorithms we can use. Whether we keep states explicit or eliminate them through forward simulation determines not just the problem size, but also its conditioning and the trade-offs between robustness and computational effort.

Existence of Solutions and Optimality Conditions Now that we have the optimization problem written down, a natural question arises: **does this program always have a solution?** And if it does, how can we recognize one when we see it? These questions bring us into the territory of feasibility and optimality conditions.

When does a solution exist? Notice first that nothing in the problem statement required the dynamics

$$\mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) \tag{43}$$

to be stable. In fact, many problems of interest involve unstable systems; think of balancing a pole or steering a spacecraft. What matters is that the dynamics are **well defined**: given a state–control pair, the rule $\mathbf{f}_t$ produces a valid next state.

In continuous time, one usually requires $\mathbf{f}$ to be continuous (often Lipschitz continuous) in $\mathbf{x}$ so that the ODE has a unique solution on the horizon of interest. In discrete time, the requirement is lighter—we only need the update map to be well posed.

Existence also hinges on **feasibility**. A candidate control sequence must generate a trajectory that respects all constraints: the dynamics, any bounds on state and control, and any terminal requirements. If no such sequence exists, the feasible set is empty and the problem has no solution. This can happen if the constraints are overly strict, or if the system is uncontrollable from the given initial condition.

What does optimality look like? Assume the feasible set is nonempty. To characterize a point that is not only feasible but **locally optimal**, we use the Lagrange multiplier machinery from nonlinear programming. For a smooth problem

$$\begin{aligned} \min_{\mathbf{z}} \quad & F(\mathbf{z}) \\ \text{s.t.} \quad & G(\mathbf{z}) = \mathbf{0}, \\ & H(\mathbf{z}) \geq \mathbf{0}, \end{aligned} \tag{44}$$

define the **Lagrangian**

$$\mathcal{L}(\mathbf{z}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = F(\mathbf{z}) + \boldsymbol{\lambda}^\top G(\mathbf{z}) + \boldsymbol{\mu}^\top H(\mathbf{z}), \qquad \boldsymbol{\mu} \geq \mathbf{0}. \tag{45}$$

For an inequality system $H(\mathbf{z}) \geq \mathbf{0}$ and a candidate point $\mathbf{z}$, the **active set** is

$$\mathcal{A}(\mathbf{z}) \;=\; \{\, i \;:\; H_i(\mathbf{z}) = 0 \,\}, \tag{46}$$

while indices with $H_i(\mathbf{z}) > 0$ are **inactive**. Only active inequalities can carry positive multipliers.

We now make a **constraint qualification** assumption. In plain language, it says the constraints near the solution intersect in a regular way so that the feasible set has a well-defined tangent space and the multipliers exist. Algebraically, this amounts to a **full row rank** condition on the Jacobian of the equalities together with the active inequalities:

$$\text{rows of } \big[\nabla G(\mathbf{z}^\star);\ \nabla H_{\mathcal{A}}(\mathbf{z}^\star)\big] \text{ are linearly independent.} \tag{47}$$

This is the **LICQ** (Linear Independence Constraint Qualification). In convex problems, **Slater's condition** (existence of a strictly feasible point) plays a similar role. You can think of these as the assumptions that let the linearized KKT equations be solvable; we do not literally invert that Jacobian, but the full-rank property is the key ingredient that would make such an inversion possible in principle.

Under such a constraint qualification, any local minimizer $\mathbf{z}^\star$ admits multipliers $(\boldsymbol{\lambda}^\star, \boldsymbol{\mu}^\star)$ that satisfy the **Karush–Kuhn–Tucker (KKT) conditions**:

$$\begin{aligned}
&\text{stationarity:} && \nabla_{\mathbf{z}} \mathcal{L}(\mathbf{z}^\star, \boldsymbol{\lambda}^\star, \boldsymbol{\mu}^\star) = \mathbf{0}, \\
&\text{primal feasibility:} && G(\mathbf{z}^\star) = \mathbf{0}, \quad H(\mathbf{z}^\star) \geq \mathbf{0}, \\
&\text{dual feasibility:} && \boldsymbol{\mu}^\star \geq \mathbf{0}, \\
&\text{complementarity:} && \mu_i^\star \, H_i(\mathbf{z}^\star) = 0 \quad \text{for all } i.
\end{aligned} \tag{48}$$

Only constraints that are **active** at $\mathbf{z}^\star$ can have $\mu_i^\star > 0$; inactive ones have $\mu_i^\star = 0$. The multipliers quantify marginal costs: $\lambda_j^\star$ measures how the optimal value changes if the j-th equality is relaxed, and $\mu_i^\star$ does the same for the i-th inequality. (If you prefer $h(\mathbf{z}) \leq 0$, signs flip accordingly.)

In our trajectory problems, $\mathbf{z}$ stacks state and control trajectories, G enforces the dynamics, and H collects bounds and path constraints. The equalities' multipliers act as **costates** or **shadow prices** for the dynamics. Writing the KKT system stage by stage yields the discrete-time Pontryagin principle, derived next. For convex programs these conditions are also sufficient.

What fails without a CQ? If the active gradients are dependent (for example duplicated or nearly parallel), the Jacobian loses rank; multipliers may then be nonunique or fail to exist, and the linearized equations become ill-posed. In transcribed trajectory problems this shows up as dependent dynamic constraints or redundant path constraints, which leads to fragile solver behavior.

From KKT to algorithms The KKT system can be read as the first-order optimality conditions of a **saddle-point** problem. With equalities $G(\mathbf{z}) = \mathbf{0}$ and inequalities $H(\mathbf{z}) \geq \mathbf{0}$, define the Lagrangian

$$\mathcal{L}(\mathbf{z}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = F(\mathbf{z}) + \boldsymbol{\lambda}^\top G(\mathbf{z}) + \boldsymbol{\mu}^\top H(\mathbf{z}), \quad \boldsymbol{\mu} \geq \mathbf{0}. \tag{49}$$

Optimality corresponds to a saddle: minimize in $\mathbf{z}$, maximize in $(\boldsymbol{\lambda}, \boldsymbol{\mu})$ (with $\boldsymbol{\mu}$ constrained to the nonnegative orthant).

Primal–dual gradient dynamics (Arrow–Hurwicz) The simplest algorithm mirrors this saddle structure by descending in the primal variables and ascending in the dual variables, with a projection for the inequalities:

$$\begin{aligned}\mathbf{z}^{k+1} &= \mathbf{z}^k - \alpha_k\big(\nabla F(\mathbf{z}^k) + \nabla G(\mathbf{z}^k)^\top \boldsymbol{\lambda}^k + \nabla H(\mathbf{z}^k)^\top \boldsymbol{\mu}^k\big),\\ \boldsymbol{\lambda}^{k+1} &= \boldsymbol{\lambda}^k + \beta_k\, G(\mathbf{z}^k),\\ \boldsymbol{\mu}^{k+1} &= \Pi_{\geq 0}\big(\boldsymbol{\mu}^k + \beta_k\, H(\mathbf{z}^k)\big).\end{aligned} \tag{50}$$

Here $\Pi_{\geq 0}$ is the projection onto $\{\boldsymbol{\mu} \geq 0\}$. In convex settings and with suitable step sizes, these iterates converge to a saddle point. In nonconvex problems (our trajectory optimizations after transcription), these updates are often used inside **augmented Lagrangian** or **penalty** frameworks to improve robustness, for example by replacing $\mathcal{L}$ with

$$\mathcal{L}_\rho(\mathbf{z}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathcal{L}(\mathbf{z}, \boldsymbol{\lambda}, \boldsymbol{\mu}) + \tfrac{\rho}{2}\|G(\mathbf{z})\|^2 + \tfrac{\rho}{2}\|\min\{0, H(\mathbf{z})\}\|^2, \tag{51}$$

which stabilizes the dual ascent when constraints are not yet well satisfied.

SQP as Newton on the KKT system (equality case) With **only equality constraints** $G(\mathbf{z}) = \mathbf{0}$, write first-order conditions

$$\nabla_{\mathbf{z}}\mathcal{L}(\mathbf{z}, \boldsymbol{\lambda}) = \mathbf{0}, \qquad G(\mathbf{z}) = \mathbf{0}, \quad \text{where } \mathcal{L} = F + \boldsymbol{\lambda}^\top G. \tag{52}$$

Applying Newton's method to this system gives the linear KKT solve

$$\begin{bmatrix} \nabla^2_{\mathbf{zz}}\mathcal{L}(\mathbf{z}^k, \boldsymbol{\lambda}^k) & \nabla G(\mathbf{z}^k)^\top \\ \nabla G(\mathbf{z}^k) & 0 \end{bmatrix} \begin{bmatrix} \Delta\mathbf{z} \\ \Delta\boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} \nabla_{\mathbf{z}}\mathcal{L}(\mathbf{z}^k, \boldsymbol{\lambda}^k) \\ G(\mathbf{z}^k) \end{bmatrix}. \tag{53}$$

This is exactly the step computed by **Sequential Quadratic Programming (SQP)** in the equality-constrained case: it is Newton's method on the KKT equations. For general problems with inequalities, SQP forms a **quadratic subproblem** by quadratically modeling F with $\nabla^2_{\mathbf{zz}}\mathcal{L}$ and linearizing the constraints, then solves that QP with line search or trust region. In least-squares-like problems one often uses **Gauss–Newton** (or a Levenberg–Marquardt trust region) as a positive-definite approximation to the Lagrangian Hessian.

In trajectory optimization. After transcription, the KKT matrix inherits banded/sparse structure from the dynamics. Newton/SQP steps can be computed efficiently by exploiting this structure; in the special case of quadratic models and linearized dynamics, the QP reduces to an LQR solve along the horizon (this is the backbone of iLQR/DDP-style methods). Primal–dual updates provide simpler iterations and are easy to implement; augmented terms are typically needed to obtain stable progress when constraints couple stages.

When to use which. Primal–dual gradients give lightweight iterations and are good for warm starts or as inner loops with penalties. SQP/Newton gives rapid local convergence when you are close to a solution and LICQ holds; use trust regions or line search to globalize.

Examples of DOCPs To make things concrete, here are three problems that are naturally posed as discrete-time OCPs. In each case, we seek an optimal trajectory of states and controls over a finite horizon.

Periodic Inventory Control Decisions are made once per period: choose order quantity $u_k \geq 0$ to meet forecast demand d_k. The state x_k is on-hand inventory with dynamics $x_{k+1} = x_k + u_k - d_k$. A typical stage cost is $c_k(x_k, u_k) = h\,[x_k]_+ + p\,[-x_k]_+ + c\,u_k$, trading off holding, backorder, and ordering costs. The horizon objective is $\min \sum_{k=0}^{T-1} c_k(x_k, u_k)$ subject to bounds.

End-of-Day Portfolio Rebalancing At each trading day k, choose trades u_k to adjust holdings h_k before next-day returns r_k realize. Deterministic planning uses predicted returns μ_k, with dynamics $h_{k+1} = (h_k + u_k) \odot (\mathbf{1} + \mu_k)$ and budget/box constraints. The stage cost can capture transaction costs and risk, e.g., $c_k(h_k, u_k) = \tau \|u_k\|_1 + \frac{\lambda}{2}\, h_k^\top \Sigma_k h_k$, with a terminal utility or wealth objective.

Daily Ad-Budget Allocation with Carryover Allocate spend $u_k \in [0, U_{\max}]$ to build awareness s_k with carryover dynamics $s_{k+1} = \alpha s_k + \beta u_k$. Conversions/revenue at day k follow a response curve $g(s_k, u_k)$; the goal is $\max \sum_{k=0}^{T-1} g(s_k, u_k) - c\,u_k$ subject to spend limits. This is naturally discrete because decisions and measurements occur daily.

DOCPs Arising from the Discretization of Continuous-Time OCPs Although many applications are natively discrete-time, it is also common to obtain a DOCP by discretizing a continuous-time formulation. Consider a system on $[0, T_c]$ given by

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t), \mathbf{u}(t)), \qquad \mathbf{x}(0) = \mathbf{x}_0. \tag{54}$$

Choose a step size $\Delta > 0$ and grid $t_k = k\,\Delta$. A one-step integration scheme induces a discrete map $\mathbf{F}_\Delta$ so that

$$\mathbf{x}_{k+1} = \mathbf{F}_\Delta(\mathbf{x}_k, \mathbf{u}_k, t_k), \qquad k = 0, \ldots, T-1, \tag{55}$$

where, for example, explicit Euler gives $\mathbf{F}_\Delta(\mathbf{x}, \mathbf{u}, t) = \mathbf{x} + \Delta\,\mathbf{f}(t, \mathbf{x}, \mathbf{u})$. The resulting discrete-time optimal control problem takes the Bolza form with these induced dynamics:

$$\begin{aligned} \min_{\{\mathbf{x}_k, \mathbf{u}_k\}} \; & c_T(\mathbf{x}_T) + \sum_{k=0}^{T-1} c_k(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t. } & \mathbf{x}_{k+1} - \mathbf{F}_\Delta(\mathbf{x}_k, \mathbf{u}_k, t_k) = 0, \quad k = 0, \ldots, T-1, \\ & \mathbf{x}_0 = \mathbf{x}_{\text{init}}. \end{aligned} \tag{56}$$

Programs as DOCPs and Differentiable Programming It is often useful to view a computer program itself as a discrete-time dynamical system. Let the **program state** collect memory, buffers, and intermediate variables, and let the **control** represent inputs or tunable decisions at each step. A single execution step defines a transition map

$$\mathbf{x}_{k+1} = \Phi_k(\mathbf{x}_k, \mathbf{u}_k), \tag{57}$$

and a scalar objective (e.g., loss, error, runtime, energy) yields a DOCP:

$$\min_{\{\mathbf{u}_k\}} \; c_T(\mathbf{x}_T) + \sum_{k=0}^{T-1} c_k(\mathbf{x}_k, \mathbf{u}_k) \quad \text{s.t.} \quad \mathbf{x}_{k+1} = \Phi_k(\mathbf{x}_k, \mathbf{u}_k). \tag{58}$$

In differentiable programming (e.g., JAX, PyTorch), the composed map $\Phi_{T-1} \circ \cdots \circ \Phi_0$ is differentiable, enabling reverse-mode automatic differentiation and efficient gradient-based trajectory optimization. When parts of the program are non-differentiable (discrete branches, simulators with events), DOCPs can still be solved using derivative-free or weak-gradient methods (eg. finite differences, SPSA, Nelder–Mead, CMA-ES, or evolutionary strategies) optionally combined with smoothing, relaxations, or stochastic estimators to navigate non-smooth regions.

Example: HTTP Retrier Optimization As an example we cast the problem of optimizing a "HTTP retrier with backoff" as a DOCP where the state tracks wall-clock time, attempt index, success, last code, and jitter; the control is the chosen wait time before the next request (the backoff schedule); the transition encapsulates waiting and a probabilistic request outcome; and the objective penalizes latency and failure. We then optimize the schedule either directly (per-step SPSA) or via a two-parameter exponential policy using common random numbers for variance reduction.

Example: Gradient Descent with Momentum as DOCP To connect this lens to familiar practice—and to hyperparameter optimization—treat the learning rate and momentum (or their schedules) as controls. Rather than fixing them a priori, we can optimize them as part of a trajectory optimization. The optimizer itself becomes the dynamical system whose execution we shape to minimize final loss.

Program: gradient descent with momentum on a quadratic loss. We fit $\boldsymbol{\theta} \in R^p$ to data $(\mathbf{A}, \mathbf{b})$ by minimizing

$$\ell(\boldsymbol{\theta}) = \tfrac{1}{2} \left\| \mathbf{A}\boldsymbol{\theta} - \mathbf{b} \right\|_2^2. \tag{59}$$

The program maintains parameters $\boldsymbol{\theta}_k$ and momentum $\mathbf{m}_k$. Each iteration does:

1. compute gradient $\mathbf{g}_k = \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}_k) = \mathbf{A}^\top(\mathbf{A}\boldsymbol{\theta}_k - \mathbf{b})$

2. update momentum $\mathbf{m}_{k+1} = \beta_k\,\mathbf{m}_k + \mathbf{g}_k$

3. update parameters $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k\,\mathbf{m}_{k+1}$

State, control, and transition. Define the state $\mathbf{x}_k = \begin{bmatrix}\boldsymbol{\theta}_k \\ \mathbf{m}_k\end{bmatrix} \in R^{2p}$ and the control $\mathbf{u}_k = \begin{bmatrix}\alpha_k \\ \beta_k\end{bmatrix}$. One program step is

$$\Phi_k(\mathbf{x}_k, \mathbf{u}_k) = \begin{bmatrix}\boldsymbol{\theta}_k - \alpha_k\big(\beta_k\,\mathbf{m}_k + \mathbf{A}^\top(\mathbf{A}\boldsymbol{\theta}_k - \mathbf{b})\big) \\ \beta_k\,\mathbf{m}_k + \mathbf{A}^\top(\mathbf{A}\boldsymbol{\theta}_k - \mathbf{b})\end{bmatrix}. \tag{60}$$

Executing the program for T iterations gives the trajectory

$$\mathbf{x}_{k+1} = \Phi_k(\mathbf{x}_k, \mathbf{u}_k), \quad k = 0, \ldots, T-1, \qquad \mathbf{x}_0 = \begin{bmatrix}\boldsymbol{\theta}_0 \\ \mathbf{m}_0\end{bmatrix}. \tag{61}$$

Objective as a DOCP. Choose terminal cost $c_T(\mathbf{x}_T) = \ell(\boldsymbol{\theta}_T)$ and (optionally) stage costs $c_k(\mathbf{x}_k, \mathbf{u}_k) = \rho_\alpha\,\alpha_k^2 + \rho_\beta\,(\beta_k - \bar{\beta})^2$. The program-as-control problem is

$$\min_{\{\alpha_k, \beta_k\}} \ell(\boldsymbol{\theta}_T) + \sum_{k=0}^{T-1}\big(\rho_\alpha\,\alpha_k^2 + \rho_\beta\,(\beta_k - \bar{\beta})^2\big) \quad \text{s.t.} \quad \mathbf{x}_{k+1} = \Phi_k(\mathbf{x}_k, \mathbf{u}_k). \tag{62}$$

Backpropagation = reverse-time costate recursion. Because Φ_k is differentiable, reverse-mode AD computes $\nabla_{\mathbf{u}_{0:T-1}}\big(c_T + \sum c_k\big)$ by propagating a costate $\boldsymbol{\lambda}_k = \partial\mathcal{J}/\partial\mathbf{x}_k$ backward:

$$\boldsymbol{\lambda}_T = \nabla_{\mathbf{x}_T} c_T, \qquad \boldsymbol{\lambda}_k = \nabla_{\mathbf{x}_k} c_k + \left(\nabla_{\mathbf{x}_k}\Phi_k\right)^\top \boldsymbol{\lambda}_{k+1}, \tag{63}$$

and the gradients with respect to controls are

$$\nabla_{\mathbf{u}_k}\mathcal{J} = \nabla_{\mathbf{u}_k} c_k + \left(\nabla_{\mathbf{u}_k}\Phi_k\right)^\top \boldsymbol{\lambda}_{k+1}. \tag{64}$$

Unrolling a tiny horizon ($T = 3$) to see the composition:

$$\begin{aligned} &\mathbf{x}_1 = \Phi_0(\mathbf{x}_0, \mathbf{u}_0), \\ &\mathbf{x}_2 = \Phi_1(\mathbf{x}_1, \mathbf{u}_1), \\ &\mathbf{x}_3 = \Phi_2(\mathbf{x}_2, \mathbf{u}_2), \qquad \mathcal{J} = c_T(\mathbf{x}_3) + \sum_{k=0}^{2} c_k(\mathbf{x}_k, \mathbf{u}_k). \end{aligned} \tag{65}$$

What if the program branches? Suppose we insert a "skip-small-gradients" branch

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k\,\mathbf{m}_{k+1}\,\mathbf{1}\{\|\mathbf{g}_k\| > \tau\}, \tag{66}$$

which is non-differentiable because of the indicator. The DOCP view still applies, but gradients are unreliable. Two practical paths: smooth the branch

(e.g., replace $\mathbf{1}\{\cdot\}$ with $\sigma((\|\mathbf{g}_k\| - \tau)/\epsilon)$ for small ϵ) and use autodiff; or go derivative-free on $\{\alpha_k, \beta_k, \tau\}$ (e.g., SPSA or CMA-ES) while keeping the inner dynamics exact.

Variants: Lagrange and Mayer Problems The Bolza form is general enough to cover most situations, but two common special cases are worth noting:

- **Lagrange problem (no terminal cost)** If the objective only accumulates stage costs:

$$\min_{\mathbf{u}_{1:T-1}} \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t). \tag{67}$$

Example: *Energy minimization for a delivery drone.* The concern is total battery use, regardless of the final position.

- **Mayer problem (terminal cost only)** If the objective depends only on the final state:

$$\min_{\mathbf{u}_{1:T-1}} c_T(\mathbf{x}_T). \tag{68}$$

Example: *Satellite orbital transfer.* The only goal is to reach a specified orbit, no matter the fuel spent along the way.

These distinctions matter when deriving optimality conditions, but conceptually they fit in the same framework: the system evolves over time, and we choose controls to shape the trajectory.

Reducing to Mayer Form by State Augmentation Although Bolza, Lagrange, and Mayer problems look different, they are equivalent in expressive power. Any problem with running costs can be rewritten as a Mayer problem (one whose objective depends only on the final state) through a simple trick: **augment the state with a running sum of costs**.

The idea is straightforward. Introduce a new variable, y_t, that keeps track of the cumulative cost so far. At each step, we update this running sum along with the system state:

$$\tilde{\mathbf{x}}_{t+1} = \begin{pmatrix} \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) \\ y_t + c_t(\mathbf{x}_t, \mathbf{u}_t) \end{pmatrix}, \tag{69}$$

where $\tilde{\mathbf{x}}_t = (\mathbf{x}_t, y_t)$. The terminal cost then becomes:

$$\tilde{c}_T(\tilde{\mathbf{x}}_T) = c_T(\mathbf{x}_T) + y_T. \tag{70}$$

The overall effect is that the explicit sum $\sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t)$ disappears from the objective and is captured implicitly by the augmented state. This lets us write every optimal control problem in Mayer form.

Why do this? Two reasons. First, it often simplifies **mathematical derivations**, as we will see later when deriving necessary conditions. Second, it can **streamline algorithmic implementation**: instead of writing separate code paths for Mayer, Lagrange, and Bolza problems, we can reduce everything to one canonical form. That said, this "one size fits all" approach isn't always best in practice—specialized formulations can sometimes be more efficient computationally, especially when the running cost has simple structure.

The unifying theme is that a DOCP may look like a generic NLP on paper, but its structure matters. Ignoring that structure often leads to impractical solutions, whereas formulations that expose sparsity and respect temporal coupling allow modern solvers to scale effectively. In the following sections, we will examine how these choices play out in practice through single shooting, multiple shooting, and collocation methods, and why different formulations strike different trade-offs between robustness and computational effort.

2.1.1 Numerical Methods for Solving DOCPs

Before we discuss specific algorithms, it is useful to clarify the goal: we want to recast a discrete-time optimal control problem as a standard nonlinear program (NLP). Collect all decision variables—states, controls, and any auxiliary variables—into a single vector $\mathbf{z} \in R^{n_z}$ and write

$$\begin{aligned} \min_{\mathbf{z}\in R^{n_z}} \quad & F(\mathbf{z}) \\ \text{s.t.} \quad & G(\mathbf{z}) = 0, \\ & H(\mathbf{z}) \geq 0, \end{aligned} \tag{71}$$

with maps $F : R^{n_z} \to R$, $G : R^{n_z} \to R^{r_e}$, and $H : R^{n_z} \to R^{r_h}$. In optimal control, G typically encodes dynamics and boundary conditions, while H captures path and box constraints.

There are multiple ways to arrive at (and benefit from) this NLP:

- Simultaneous (direct transcription / full discretization): keep all states and controls as variables and impose the dynamics as equality constraints. This is straightforward and exposes sparsity, but the problem can be large unless solver-side techniques (e.g., condensing) are exploited.
- Sequential (recursive elimination / single shooting): eliminate states by forward propagation from the initial condition, leaving controls as the main decision variables. This reduces dimension and constraints, but can be sensitive to initialization and longer horizons.
- Multiple shooting: introduce state variables at segment boundaries and enforce continuity between simulated segments. This compromises between size and conditioning and is often more robust than pure single shooting.

The next sections work through these formulations—starting with simultaneous methods, then sequential methods, and finally multiple shooting—before discussing how generic NLP solvers and specialized algorithms leverage the resulting structure in practice.

Simultaneous Methods In the simultaneous (also called direct transcription or full discretization) approach, we keep the entire trajectory explicit and enforce the dynamics as equality constraints. Starting from the Bolza DOCP,

$$\min_{\{\mathbf{x}_t, \mathbf{u}_t\}} \; c_T(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) \quad \text{s.t.} \quad \mathbf{x}_{t+1} - \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) = 0, \; t = 1, \ldots, T-1, \tag{72}$$

collect all variables into a single vector

$$\mathbf{z} := \begin{bmatrix} \mathbf{x}_1^\top & \cdots & \mathbf{x}_T^\top & \mathbf{u}_1^\top & \cdots & \mathbf{u}_{T-1}^\top \end{bmatrix}^\top \in R^{n_z}. \tag{73}$$

Path constraints typically apply only at selected times. Let E index additional equality constraints g_i and I index inequality constraints h_i. For each constraint i, define the set of time indices $K_i \subseteq \{1, \ldots, T\}$ where it is enforced (e.g., terminal constraints use $K_i = \{T\}$). The simultaneous transcription is the NLP

$$\begin{aligned} \min_{\mathbf{z}} \quad & F(\mathbf{z}) := c_T(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) \\ \text{s.t.} \quad & G(\mathbf{z}) = \begin{bmatrix} \big[\, g_i(\mathbf{x}_k, \mathbf{u}_k) \big]_{i \in E,\, k \in K_i} \\ \big[\, \mathbf{x}_{t+1} - \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) \big]_{t = 1:T-1} \\ \mathbf{x}_1 - \mathbf{x}_{\text{init}} \end{bmatrix} = \mathbf{0}, \\ & H(\mathbf{z}) = \big[\, h_i(\mathbf{x}_k, \mathbf{u}_k) \big]_{i \in I,\, k \in K_i} \;\geq\; \mathbf{0}, \end{aligned} \tag{74}$$

optionally with simple bounds $\mathbf{x}_{\text{lb}} \leq \mathbf{x}_t \leq \mathbf{x}_{\text{ub}}$ and $\mathbf{u}_{\text{lb}} \leq \mathbf{u}_t \leq \mathbf{u}_{\text{ub}}$ folded into H or provided to the solver separately. For notational convenience, some constraints may not depend on $\mathbf{u}_k$ at times in K_i; the indexing still helps specify when each condition is active.

This direct transcription is attractive because it is faithful to the model and exposes sparsity. The Jacobian of G has a block bi-diagonal structure induced by the dynamics, and the KKT matrix is sparse and structured—properties exploited by interior-point and SQP methods. The trade-off is size: with state dimension n and control dimension m, the decision vector has $(T{\cdot}n){+}((T{-}1){\cdot}m)$ entries, and there are roughly $(T-1) \cdot n$ dynamic equalities plus any path and boundary conditions. Techniques such as partial or full condensing eliminate state variables to reduce the equality set (at the cost of denser matrices), while keeping states explicit preserves sparsity and often improves robustness on long horizons and in the presence of state constraints.

Compared to alternatives, simultaneous methods avoid the long nonlinear dependency chains of single shooting and make it easier to impose state/path constraints. They can, however, demand more memory and per-iteration linear algebra, so practical performance hinges on exploiting sparsity and good initialization.

The same logic applies when selecting an optimizer. For small-scale problems, it is common to rely on general-purpose routines such as those in `scipy.optimize.minimize`. Derivative-free methods like Nelder–Mead require no gradients but scale poorly as dimensionality increases. Quasi-Newton schemes such as BFGS work well for moderate dimensions and can approximate gradients by finite differences, while large-scale trajectory optimization often calls for gradient-based constrained solvers such as interior-point or sequential quadratic programming methods that can exploit sparse Jacobians and benefit from automatic differentiation. Stochastic techniques, including genetic algorithms, simulated annealing, or particle swarm optimization, occasionally appear when gradients are unavailable, but their cost grows rapidly with dimension and they are rarely competitive for structured optimal control problems.

Sequential Methods The previous section showed how a discrete-time optimal control problem can be solved by treating all states and controls as decision variables and enforcing the dynamics as equality constraints. This produces a nonlinear program that can be passed to solvers such as `scipy.optimize.minimize` with the SLSQP method. For short horizons, this approach is straightforward and works well; the code stays close to the mathematical formulation.

It also has a real advantage: by keeping the states explicit and imposing the dynamics through constraints, we anchor the trajectory at multiple points. This extra structure helps stabilize the optimization, especially for long horizons where small deviations in early steps can otherwise propagate and cause the optimizer to drift or diverge. In that sense, this formulation is better conditioned and more robust than approaches that treat the dynamics implicitly.

The drawback is scale. As the horizon grows, the number of variables and constraints grows with it, and all are coupled by the dynamics. Each iteration of a sequential quadratic programming (SQP) or interior-point method requires building and factorizing large Jacobians and Hessians. These methods have been embedded in reinforcement learning and differentiable programming pipelines—through implicit layers or differentiable convex solvers—but the cost is significant. They remain serial, rely on repeated linear algebra factorizations, and are difficult to parallelize efficiently. When thousands of such problems must be solved inside a learning loop, the overhead becomes prohibitive.

This motivates an alternative that aligns better with the computational model of machine learning. If the dynamics are deterministic and state constraints are absent (or reducible to simple bounds on controls), we can eliminate the equality constraints altogether by making the states implicit. Instead of solving for both states and controls, we fix the initial state and roll the system forward under a candidate control sequence. This is the essence of **single**

shooting.

The term "shooting" comes from the idea of *aiming and firing* a trajectory from the initial state: you pick a control sequence, integrate (or step) the system forward, and see where it lands. If the final state misses the target, you adjust the controls and try again: like adjusting the angle of a shot until it hits the mark. It is called **single** shooting because we compute the entire trajectory in one pass from the starting point, without breaking it into segments. Later, we will contrast this with **multiple shooting**, where the horizon is divided into smaller arcs that are optimized jointly to improve stability and conditioning.

The analogy with deep learning is also immediate: the control sequence plays the role of parameters, the rollout is a forward pass, and the cost is a scalar loss. Gradients can be obtained with reverse-mode automatic differentiation. In the single shooting formulation of the DOCP, the constrained program

$$\min_{\mathbf{x}_{1:T},\,\mathbf{u}_{1:T-1}} J(\mathbf{x}_{1:T}, \mathbf{u}_{1:T-1}) \quad \text{s.t.} \quad \mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) \tag{75}$$

collapses to

$$\min_{\mathbf{u}_{1:T-1}} c_T\big(\boldsymbol{\phi}_T(\mathbf{u}, \mathbf{x}_1)\big) + \sum_{t=1}^{T-1} c_t\big(\boldsymbol{\phi}_t(\mathbf{u}, \mathbf{x}_1), \mathbf{u}_t\big), \qquad \mathbf{u}_{\text{lb}} \le \mathbf{u}_t \le \mathbf{u}_{\text{ub}}. \tag{76}$$

Here $\boldsymbol{\phi}_t$ denotes the state reached at time t by recursively applying the dynamics to the previous state and current control. This recursion can be written as

$$\boldsymbol{\phi}_{t+1}(\mathbf{u}, \mathbf{x}_1) = \mathbf{f}_t\big(\boldsymbol{\phi}_t(\mathbf{u}, \mathbf{x}_1), \mathbf{u}_t\big), \qquad \boldsymbol{\phi}_1 = \mathbf{x}_1. \tag{77}$$

Concretely, here is JAX-style pseudocode for defining `phi(u, x_0, t)` using `jax.lax.scan` with a zero-based time index:

```
def phi(u_seq, x0, t):
    """Return \phi_t(u, x0) with 0 -based t (\phi_0 = x0).

    u_seq: controls of length T (or T -1); only first t entries are used
    x0: initial state at time 0
    t: integer >= 0
    """
    if t <= 0:
        return x0

    def step(carry, u):
        x, t_idx = carry
        x_next = f(x, u, t_idx)
        return (x_next, t_idx + 1), None

    (x_t, _), _ = lax.scan(step, (x0, 0), u_seq[:t])
    return x_t
```

The pattern mirrors an RNN unroll: starting from an initial state ($\mathbf{x}_1^\star$) and a sequence of controls ($\mathbf{u}_{1:T-1}^*$), we propagate forward through the dynamics, updating the state at each step and accumulating cost along the way. This structural similarity is why single shooting often feels natural to practitioners with a deep learning background: the rollout is a forward pass, and gradients propagate backward through time exactly as in backpropagation through an RNN.

Algorithmically:

In JAX or PyTorch, this loop can be JIT-compiled and differentiated automatically. Any gradient-based optimizer—L-BFGS, Adam, even SGD—can be applied, making the pipeline look very much like training a neural network. In effect, we are "backpropagating through the world model" when computing $\nabla J(\mathbf{u})$.

Single shooting is attractive for its simplicity and compatibility with differentiable programming, but it has limitations. The absence of intermediate constraints makes it sensitive to initialization and prone to numerical instability over long horizons. When state constraints or robustness matter, formulations that keep states explicit—such as multiple shooting or collocation—become preferable. These trade-offs are the focus of the next section.

In Between Sequential and Simultaneous The two formulations we have seen so far lie at opposite ends. The **full discretization** approach keeps every state explicit and enforces the dynamics through equality constraints, which makes the structure clear but leads to a large optimization problem. At the other end, **single shooting** removes these constraints by simulating forward from the initial state, leaving only the controls as decision variables. That makes the problem smaller, but it also introduces a long and highly nonlinear dependency from the first control to the last state.

Multiple shooting sits in between. Instead of simulating the entire horizon in one shot, we divide it into smaller segments. For each segment, we keep its starting state as a decision variable and propagate forward using the dynamics for that segment. At the end, we enforce continuity by requiring that the simulated end state of one segment matches the decision variable for the next.

Formally, suppose the horizon of T steps is divided into K segments of length L (with $T = K \cdot L$ for simplicity). We introduce:

- The controls for each step: $\mathbf{u}_{1:T-1}$.
- The state at the start of each segment: $\mathbf{x}_1, \ldots, \mathbf{x}_K$.

Given $\mathbf{x}_k$ and the controls in its segment, we compute the predicted terminal state by simulating forward:

$$\hat{\mathbf{x}}_{k+1} = \Phi(\mathbf{x}_k, \mathbf{u}_{\text{segment } k}), \tag{78}$$

where Φ represents L applications of the dynamics. Continuity constraints enforce:

$$\mathbf{x}_{k+1} - \hat{\mathbf{x}}_{k+1} = 0, \qquad k = 1, \ldots, K-1. \tag{79}$$

The resulting nonlinear program looks like this:

$$\begin{aligned} \min_{\{\mathbf{x}_k, \mathbf{u}_t\}} \quad & c_T(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) \\ \text{subject to} \quad & \mathbf{x}_{k+1} - \Phi(\mathbf{x}_k, \mathbf{u}_{\text{segment } k}) = 0, \quad k = 1, \ldots, K-1, \\ & \mathbf{u}_{\text{lb}} \le \mathbf{u}_t \le \mathbf{u}_{\text{ub}}, \\ & \text{boundary conditions on } \mathbf{x}_1 \text{ and } \mathbf{x}_K. \end{aligned} \tag{80}$$

Compared to the full NLP, we no longer introduce every intermediate state as a variable—only the anchors at segment boundaries. Inside each segment, states are reconstructed by simulation. Compared to single shooting, these anchors break the long dependency chain that makes optimization unstable: gradients only have to travel across L steps before they hit a decision variable, rather than the entire horizon. This is the same reason why exploding or vanishing gradients appear in deep recurrent networks: when the chain is too long, information either dies out or blows up. Multiple shooting shortens the chain and improves conditioning.

By adjusting the number of segments K, we can interpolate between the two extremes: $K = 1$ gives single shooting, while $K = T$ recovers the full direct NLP. In practice, a moderate number of segments often strikes a good balance between robustness and complexity.

2.1.2 The Discrete-Time Pontryagin Principle

If we take the Bolza formulation of the DOCP and apply the KKT conditions directly, we obtain an optimization system with many multipliers and constraints. Written in raw form, it looks like any other nonlinear program. But in control, this structure has a long history and a name of its own: the **Pontryagin principle**. In fact, the discrete-time version can be seen as the structured KKT system that emerges once we introduce multipliers for the dynamics and collect terms stage by stage.

We work with the Bolza program

$$\begin{aligned} \min_{\{\mathbf{x}_t, \mathbf{u}_t\}} \quad & c_T(\mathbf{x}_T) \; + \; \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) \\ \text{s.t.} \quad & \mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t), \quad t = 1, \ldots, T-1, \\ & \mathbf{g}_t(\mathbf{x}_t, \mathbf{u}_t) \le \mathbf{0}, \quad \mathbf{u}_t \in \mathcal{U}_t, \\ & \mathbf{h}(\mathbf{x}_T) = \mathbf{0} \quad \text{(optional terminal equalities)}. \end{aligned} \tag{81}$$

Introduce **costates** $\boldsymbol{\lambda}_{t+1} \in R^n$ for the dynamics, multipliers $\boldsymbol{\mu}_t \ge \mathbf{0}$ for path inequalities, and $\boldsymbol{\nu}$ for terminal equalities. The Lagrangian is

$$\mathcal{L} = c_T(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) + \sum_{t=1}^{T-1} \boldsymbol{\lambda}_{t+1}^\top \big(\mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) - \mathbf{x}_{t+1}\big) + \sum_{t=1}^{T-1} \boldsymbol{\mu}_t^\top \mathbf{g}_t(\mathbf{x}_t, \mathbf{u}_t) + \boldsymbol{\nu}^\top \mathbf{h}(\mathbf{x}_T). \tag{82}$$

It is convenient to package the stagewise terms in a **Hamiltonian**

$$H_t(\mathbf{x}_t, \mathbf{u}_t, \boldsymbol{\lambda}_{t+1}, \boldsymbol{\mu}_t) := c_t(\mathbf{x}_t, \mathbf{u}_t) + \boldsymbol{\lambda}_{t+1}^\top \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) + \boldsymbol{\mu}_t^\top \mathbf{g}_t(\mathbf{x}_t, \mathbf{u}_t). \tag{83}$$

Then

$$\mathcal{L} = c_T(\mathbf{x}_T) + \boldsymbol{\nu}^\top \mathbf{h}(\mathbf{x}_T) + \sum_{t=1}^{T-1} \Big[H_t(\mathbf{x}_t, \mathbf{u}_t, \boldsymbol{\lambda}_{t+1}, \boldsymbol{\mu}_t) - \boldsymbol{\lambda}_{t+1}^\top \mathbf{x}_{t+1} \Big]. \tag{84}$$

Gradient convention
Throughout this section, we use the **denominator layout** (gradient layout) convention:

- $\nabla_{\mathbf{x}} f(\mathbf{x})$ produces a **column vector** (gradient)
- $\frac{\partial f}{\partial \mathbf{x}}$ produces the Jacobian matrix
- For scalar functions: $\nabla_{\mathbf{x}} f = \left(\frac{\partial f}{\partial \mathbf{x}}\right)^\top$

This is the standard convention in optimization and control theory.

Necessary conditions Taking first-order variations and collecting terms gives the discrete-time adjoint system, control stationarity, and complementarity. At a local minimum $\{\mathbf{x}_t^\star, \mathbf{u}_t^\star\}$ with multipliers $\{\boldsymbol{\lambda}_t^\star, \boldsymbol{\mu}_t^\star, \boldsymbol{\nu}^\star\}$:

State dynamics (primal feasibility)

$$\mathbf{x}_{t+1}^\star = \mathbf{f}_t(\mathbf{x}_t^\star, \mathbf{u}_t^\star), \quad t = 1, \dots, T-1. \tag{85}$$

Costate recursion (backward "adjoint" equation)

$$\boldsymbol{\lambda}_t^\star = \nabla_{\mathbf{x}} H_t\big(\mathbf{x}_t^\star, \mathbf{u}_t^\star, \boldsymbol{\lambda}_{t+1}^\star, \boldsymbol{\mu}_t^\star\big) = \nabla_{\mathbf{x}} c_t(\mathbf{x}_t^\star, \mathbf{u}_t^\star) + \big[\nabla_{\mathbf{x}} \mathbf{f}_t(\mathbf{x}_t^\star, \mathbf{u}_t^\star)\big]^\top \boldsymbol{\lambda}_{t+1}^\star + \big[\nabla_{\mathbf{x}} \mathbf{g}_t(\mathbf{x}_t^\star, \mathbf{u}_t^\star)\big]^\top \boldsymbol{\mu}_t^\star, \tag{86}$$

with the **terminal condition**

$$\boldsymbol{\lambda}_T^\star = \nabla_{\mathbf{x}} c_T(\mathbf{x}_T^\star) + \big[\nabla_{\mathbf{x}} \mathbf{h}(\mathbf{x}_T^\star)\big]^\top \boldsymbol{\nu}^\star \quad \text{(and } \boldsymbol{\nu}^\star = \mathbf{0} \text{ if there are no terminal equalities).} \tag{87}$$

Control stationarity (first-order optimality in $\mathbf{u}_t$) If $\mathcal{U}_t = R^m$ (no explicit set constraint), then

$$\nabla_{\mathbf{u}} H_t\big(\mathbf{x}_t^\star, \mathbf{u}_t^\star, \boldsymbol{\lambda}_{t+1}^\star, \boldsymbol{\mu}_t^\star\big) = \mathbf{0}. \tag{88}$$

If $\mathcal{U}_t$ imposes bounds or a convex set, the condition becomes the **variational inequality**

$$\mathbf{0} \in \nabla_{\mathbf{u}} H_t(\cdot) \; + \; N_{\mathcal{U}_t}(\mathbf{u}_t^\star), \tag{89}$$

where $N_{\mathcal{U}_t}(\cdot)$ is the normal cone to $\mathcal{U}_t$. For simple box bounds, this reduces to standard KKT sign and complementarity conditions on the components of $\mathbf{u}_t^\star$.

Path-constraint multipliers (primal/dual feasibility and complementarity)

$$\mathbf{g}_t(\mathbf{x}_t^\star, \mathbf{u}_t^\star) \le \mathbf{0}, \quad \boldsymbol{\mu}_t^\star \ge \mathbf{0}, \quad \mu_{t,i}^\star \, g_{t,i}(\mathbf{x}_t^\star, \mathbf{u}_t^\star) = 0 \quad \text{for all } i, t. \tag{90}$$

Terminal equalities (if present)

$$\mathbf{h}(\mathbf{x}_T^\star) = \mathbf{0}. \tag{91}$$

The triplet "forward state, backward costate, control stationarity" is the discrete-time Euler–Lagrange system tailored to control with dynamics. It is the same KKT logic as before, but organized stagewise through the Hamiltonian.

Proposition 2.1 (Discrete-time Pontryagin necessary conditions (summary))**.** *At a local minimum of the DOCP*

$$\min_{\{\mathbf{x}_t, \mathbf{u}_t\}} \; c_T(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) \quad s.t. \quad \mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t), \; \mathbf{g}_t(\mathbf{x}_t, \mathbf{u}_t) \le \mathbf{0}, \; \mathbf{h}(\mathbf{x}_T) = \mathbf{0}, \tag{92}$$

there exist multipliers $\{\boldsymbol{\lambda}_{t+1}\}$, $\{\boldsymbol{\mu}_t \ge \mathbf{0}\}$, *and (if present)* $\boldsymbol{\nu}$ *such that, for* $t = 1, \ldots, T-1$*:*

- *State dynamics:* $\mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t)$.
- *Backward costate recursion:*

$$\boldsymbol{\lambda}_t = \nabla_{\mathbf{x}} c_t(\mathbf{x}_t, \mathbf{u}_t) + \big[\nabla_{\mathbf{x}} \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t)\big]^\top \boldsymbol{\lambda}_{t+1} + \big[\nabla_{\mathbf{x}} \mathbf{g}_t(\mathbf{x}_t, \mathbf{u}_t)\big]^\top \boldsymbol{\mu}_t. \tag{93}$$

- *Terminal condition:* $\boldsymbol{\lambda}_T = \nabla_{\mathbf{x}} c_T(\mathbf{x}_T) + \big[\nabla_{\mathbf{x}} \mathbf{h}(\mathbf{x}_T)\big]^\top \boldsymbol{\nu}$.
- *Control stationarity (unconstrained control):* $\nabla_{\mathbf{u}} H_t(\cdot) = \mathbf{0}$*; with a convex control set* $\mathcal{U}_t$, $\mathbf{0} \in \nabla_{\mathbf{u}} H_t(\cdot) + N_{\mathcal{U}_t}(\mathbf{u}_t)$.

- *Path inequalities:* $\mathbf{g}_t(\mathbf{x}_t, \mathbf{u}_t) \le \mathbf{0}$, $\boldsymbol{\mu}_t \ge \mathbf{0}$, *and complementarity* $\mu_{t,i}\, g_{t,i}(\mathbf{x}_t, \mathbf{u}_t) = 0$ *for all* i.
- *Terminal equalities (if present):* $\mathbf{h}(\mathbf{x}_T) = \mathbf{0}$.

Here $H_t(\mathbf{x}_t, \mathbf{u}_t, \boldsymbol{\lambda}_{t+1}, \boldsymbol{\mu}_t) := c_t(\mathbf{x}_t, \mathbf{u}_t) + \boldsymbol{\lambda}_{t+1}^\top \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t) + \boldsymbol{\mu}_t^\top \mathbf{g}_t(\mathbf{x}_t, \mathbf{u}_t)$ *is the stage Hamiltonian.*

The adjoint equation as reverse accumulation Optimization needs sensitivities. In trajectory problems we adjust decisions—controls or parameters—to reduce an objective while respecting dynamics and constraints. First-order methods in the unconstrained case (e.g., gradient descent, L-BFGS, Adam) require the gradient of the objective with respect to all controls, and constrained methods (SQP, interior-point) require gradients of the Lagrangian, i.e., of costs and constraints. The discrete-time adjoint equations provide these derivatives in a way that scales to long horizons and many decision variables.

Consider

$$J = c_T(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t), \qquad \mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t). \tag{94}$$

A single forward rollout computes and stores the trajectory $\mathbf{x}_{1:T}$. A single backward sweep then applies the reverse-mode chain rule stage by stage. Defining the costate by

$$\boldsymbol{\lambda}_T = \nabla_{\mathbf{x}} c_T(\mathbf{x}_T), \qquad \boldsymbol{\lambda}_t = \nabla_{\mathbf{x}} c_t(\mathbf{x}_t, \mathbf{u}_t) + \big[\nabla_{\mathbf{x}} \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t)\big]^\top \boldsymbol{\lambda}_{t+1}, \quad t = T-1, \dots, 1, \tag{95}$$

yields exactly the discrete-time adjoint (PMP) recursion. The gradient with respect to each control follows from the same reverse pass:

$$\nabla_{\mathbf{u}_t} J = \nabla_{\mathbf{u}} c_t(\mathbf{x}_t, \mathbf{u}_t) + \big[\nabla_{\mathbf{u}} \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t)\big]^\top \boldsymbol{\lambda}_{t+1}. \tag{96}$$

Two points are worth emphasizing. Computationally, this reverse accumulation produces all control gradients with one forward rollout and one backward adjoint pass; its cost is essentially a small constant multiple of simulating the system once. Conceptually, the costate $\boldsymbol{\lambda}_t$ is the marginal effect of perturbing the state at time t on the total objective; the control gradient combines a direct contribution from c_t and an indirect contribution through how $\mathbf{u}_t$ changes the next state. This is the same structure that underlies backpropagation, expressed for dynamical systems.

It is instructive to contrast this with alternatives. Black-box finite differences perturb one decision at a time and re-roll the system, requiring on the order of p rollouts for p decision variables and suffering from step-size and noise issues—prohibitive when $p = (T-1)m$ for an m-dimensional control over T steps. Forward-mode (tangent) sensitivities propagate Jacobian–vector products for each parameter direction; their work also scales with p. Reverse-mode

(the adjoint) instead propagates a single vector $\boldsymbol{\lambda}_t$ backward and then reads off all partial derivatives $\nabla_{\mathbf{u}_t} J$ at once. For a scalar objective, its cost is effectively independent of p, at the price of storing (or checkpointing) the forward trajectory. This scalability is why the adjoint is the method of choice for gradient-based trajectory optimization and for constrained transcriptions via the Hamiltonian.

2.2 Trajectory Optimization in Continuous Time

As in the discrete-time setting, we work with three continuous-time variants that differ only in how the objective is written while sharing the same dynamics, path constraints, and bounds. The path constraints $\mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \leq \mathbf{0}$ are pointwise in time, and the bounds $\mathbf{x}_{\min} \leq \mathbf{x}(t) \leq \mathbf{x}_{\max}$ and $\mathbf{u}_{\min} \leq \mathbf{u}(t) \leq \mathbf{u}_{\max}$ are understood in the same pointwise sense.

These three forms are different lenses on the same task. Bolza contains both terminal and running terms. Lagrange can be turned into Mayer by augmenting the state with an accumulator:

$$\dot{z}(t) = c(\mathbf{x}(t), \mathbf{u}(t)), \quad z(t_0) = 0, \quad \text{minimize } z(t_f), \tag{97}$$

with the original dynamics left unchanged. Mayer is a special case of Bolza with zero running cost. We will use these equivalences freely, since a numerical method cares only about what must be evaluated and where those evaluations are taken.

With this catalog in place, we now pass from functions to finite representations.

2.2.1 Direct Transcription Methods

The discrete-time problems of the previous chapter already suggested how to proceed: we convert a continuous problem into one over finitely many numbers by deciding where to look at the trajectories and how to interpolate between those looks. We place a mesh $t_0 < t_1 < \cdots < t_N = t_f$ and, inside each window $[t_k, t_{k+1}]$, select a small set of interior fractions $\{\tau_j\}$ on the reference interval $[0, 1]$. The running cost is additive over windows, so we write it as a sum of local integrals, map each window to $[0, 1]$, and approximate each local integral by a quadrature rule with nodes τ_j and weights w_j. This produces

$$\int_{t_0}^{t_f} c(\mathbf{x}(t), \mathbf{u}(t))\, dt \;\approx\; \sum_{k=0}^{N-1} h_k \sum_{j=1}^{q} w_j\, c\big(\mathbf{x}(t_k + h_k\tau_j),\, \mathbf{u}(t_k + h_k\tau_j)\big), \tag{98}$$

with $h_k = t_{k+1} - t_k$. The dynamics are treated in the same way by the fundamental theorem of calculus,

$$\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k) = \int_{t_k}^{t_{k+1}} \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))\, dt \;\approx\; h_k \sum_{j=1}^{q} b_j\, \mathbf{f}\big(\mathbf{x}(t_k + h_k\tau_j),\, \mathbf{u}(t_k + h_k\tau_j)\big), \tag{99}$$

so the places where we "pay" running cost are the same places where we "account" for state changes. Path constraints and bounds are then enforced at the same interior times. In the infinite-horizon discounted case, the same formulas apply with an extra factor $e^{-\rho(t_k + h_k\tau_j)}$ multiplying the weights in the cost.

The values $\mathbf{x}(t_k + h_k\tau_j)$ and $\mathbf{u}(t_k + h_k\tau_j)$ do not exist a priori. We create them by a finite representation. One option is shooting: parameterize $\mathbf{u}$ on the mesh, integrate the ODE across each window with a chosen numerical step, and read interior values from that step. Another is collocation: represent $\mathbf{x}$ inside each window by a local polynomial and choose its coefficients so that the ODE holds at the interior nodes. Both constructions lead to the same structure: a nonlinear program whose objective is a composite quadrature of the running term (plus any terminal term in the Bolza case) and whose constraints are algebraic relations that encode the ODE and the pointwise inequalities at the selected nodes.

Specific choices recover familiar schemes. If we use the left endpoint as the single interior node, we obtain the forward Euler transcription. If we use both endpoints with equal weights, we obtain the trapezoidal transcription. Higher-order rules arise when we include interior nodes and richer polynomials for $\mathbf{x}$. What matters here is the unifying picture: choose nodes, translate integrals into weighted sums, and couple those evaluations to a finite trajectory representation so that cost and physics are enforced at the same places. This is the organizing idea that will guide the rest of the chapter.

Discretizing cost and dynamics together In a continuous-time OCP, integrals appear twice: in the objective, which accumulates running cost over time, and implicitly in the dynamics, since state changes over any interval are the integral of the vector field. To compute, we must approximate both the integrals and the unknown functions $\mathbf{x}(t)$ and $\mathbf{u}(t)$ with finitely many numbers that an optimizer can manipulate.

A natural way to do this is to lay down a finite set of time points (a mesh) over the horizon. You can think of the mesh as a grid we overlay on the "true" trajectories that exist as mathematical objects but are not directly accessible. Our aim is to approximate those trajectories and their integrals using values and simple models tied to the mesh. Using the same mesh for both the cost and the dynamics keeps the representation coherent: we evaluate what we pay and how the state changes at consistent times.

Concretely, we begin by choosing a mesh

$$t_0 < t_1 < \cdots < t_N = t_f, \qquad h_k := t_{k+1} - t_k. \tag{100}$$

The running cost is additive over disjoint intervals. When the horizon $[t_0, t_f]$ is partitioned by the mesh, additivity (linearity) of the integral gives

$$\int_{t_0}^{t_f} c(\mathbf{x}(t), \mathbf{u}(t))\, dt \;=\; \sum_{k=0}^{N-1} \int_{t_k}^{t_{k+1}} c(\mathbf{x}(t), \mathbf{u}(t))\, dt. \tag{101}$$

This identity is exact: it is just the additivity (linearity) of the Lebesgue/Riemann integral over a partition. No approximation has been made yet. Approximations enter only when we later replace each window integral by a quadrature rule: a finite set of nodes and positive weights prescribing an integral approximation.

This sets the table for three important ingredients that we will use throughout the chapter.

First, it turns a global object into **local contributions** that live on each $[t_k, t_{k+1}]$. Numerical integration is most effective when it is composite: we approximate each small interval integral and then sum the results. Doing so controls error uniformly, because the global quadrature error is the accumulation of local errors that shrink with the step size. It also allows non-uniform steps $h_k = t_{k+1} - t_k$, which we will use later for mesh refinement.

Second, the split aligns the cost with the **local dynamics constraints**. On each interval the ODE can be written, by the fundamental theorem of calculus, as

$$\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k) = \int_{t_k}^{t_{k+1}} \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))\, dt. \tag{102}$$

When we approximate this integral, we introduce interior evaluation points $t_{k,j} \in [t_k, t_{k+1}]$. Using the **same points** in the cost and in the dynamics ties $\mathbf{x}$ and $\mathbf{u}$ together coherently: the places where we "pay" for running cost are also the places where we enforce the ODE. This avoids a mismatch between where we approximate the objective and where we impose feasibility.

Third, the decomposition yields a nonlinear program with **sparse structure**. Each interval contributes a small block to the objective and constraints that depends only on variables from that interval (and its endpoints). Modern solvers exploit this banded sparsity to scale to long horizons.

With the split justified, we standardize the approximation. Map each interval to a reference domain via $t = t_k + h_k\tau$ with $\tau \in [0, 1]$ and $dt = h_k\, d\tau$. A **quadrature rule on** $[0, 1]$ is specified by evaluation points $\{\tau_j\}_{j=1}^q \subset [0, 1]$ and positive weights $\{w_j\}_{j=1}^q$ such that, for a smooth ϕ,

$$\int_0^1 \phi(\tau)\, d\tau \;\approx\; \sum_{j=1}^{q} w_j\, \phi(\tau_j). \tag{103}$$

Applying it on each interval gives

$$\int_{t_k}^{t_{k+1}} c(\mathbf{x}(t), \mathbf{u}(t))\, dt \;\approx\; h_k \sum_{j=1}^{q} w_j\, c\big(\mathbf{x}(t_k + h_k\tau_j),\, \mathbf{u}(t_k + h_k\tau_j)\big). \tag{104}$$

Summing these window contributions gives a composite approximation of the integral over $[t_0, t_f]$:

$$\int_{t_0}^{t_f} c(\mathbf{x}(t), \mathbf{u}(t))\, dt \;\approx\; \sum_{k=0}^{N-1} h_k \sum_{j=1}^{q} w_j\, c\big(\mathbf{x}(t_{k,j}), \mathbf{u}(t_{k,j})\big). \tag{105}$$

The outer index k indicates the window; the inner index i indicates the samples within that window; the factor h_k appears from the change of variables.

The dynamics admit the same treatment. By the fundamental theorem of calculus,

$$\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k) = \int_{t_k}^{t_{k+1}} \dot{\mathbf{x}}(t)\, dt = \int_{t_k}^{t_{k+1}} \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))\, dt. \tag{106}$$

Replacing this integral by a quadrature rule that uses the **same** nodes produces the window defect relation

$$\mathbf{x}_{k+1} - \mathbf{x}_k \;\approx\; h_k \sum_{j=1}^{q} b_j\, \mathbf{f}\big(\mathbf{x}(t_{k,j}), \mathbf{u}(t_{k,j})\big), \tag{107}$$

where $\{b_j\}$ are the weights used for the ODE. Path constraints $\mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \leq 0$ are imposed at selected nodes $t_{k,j}$ in the same spirit. Using the same evaluation points for cost and dynamics keeps the representation coherent: we "pay" running cost and "account" for state changes at the same times.

On the choice of interior points Once we select a mesh $t_0 < \cdots < t_N$ and interior fractions $\{\tau_j\}_{j=1}^{q}$ per window $[t_k, t_{k+1}]$, we need $\mathbf{x}(t_{k,j})$ and $\mathbf{u}(t_{k,j})$ at the evaluation times $t_{k,j} := t_k + h_k\tau_j$. These values do not preexist. They come from one of two constructions that align with the standard quadrature taxonomy: **step-function based** and **interpolating-function based** rules.

Step-function based construction (piecewise constants; rectangle or midpoint) Here we approximate the relevant time functions by step functions on each window. For controls, a common choice is piecewise-constant:

$$\mathbf{u}(t) = \mathbf{u}_k \quad \text{for } t \in [t_k, t_{k+1}]. \tag{108}$$

For the running cost and the vector field, the corresponding quadrature is a rectangle rule on $[t_k, t_{k+1}]$. Using the left endpoint gives

$$\int_{t_k}^{t_{k+1}} c(\mathbf{x}(t), \mathbf{u}(t))\, dt \;\approx\; h_k\, c(\mathbf{x}_k, \mathbf{u}_k), \tag{109}$$

and replacing the dynamics integral by the same step-function idea yields the forward Euler relation

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h_k\, \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, t_k). \tag{110}$$

If we prefer the midpoint rectangle rule, we sample at $t_{k+\frac{1}{2}} = t_k + \frac{h_k}{2}$. In practice we then generate $\mathbf{x}_{k+\frac{1}{2}}$ by a half-step of the chosen integrator, and set $\mathbf{u}_{k+\frac{1}{2}} = \mathbf{u}_k$ (piecewise-constant) or an average if we allow a short linear segment. Either way, interior values come from **integrating forward** given a step-function model for $\mathbf{u}$ and a rectangle-rule view of the integrals. This is the shooting viewpoint. Single shooting keeps only control parameters as decision variables; multiple shooting adds the window-start states and enforces step consistency.

Interpolating-function based construction (low-order polynomials; trapezoid, Simpson, Gauss/Radau/Lobatto) Here we approximate time functions by **polynomials** on each window. If we interpolate $\mathbf{x}(t)$ linearly between endpoints, the cost naturally uses the trapezoidal rule

$$\int_{t_k}^{t_{k+1}} c\,dt \;\approx\; \tfrac{h_k}{2}\big[c(\mathbf{x}_k,\mathbf{u}_k)+c(\mathbf{x}_{k+1},\mathbf{u}_{k+1})\big], \tag{111}$$

and the dynamics use the matched trapezoidal defect

$$\mathbf{x}_{k+1}=\mathbf{x}_k+\tfrac{h_k}{2}\Big[\mathbf{f}(\mathbf{x}_k,\mathbf{u}_k,t_k)+\mathbf{f}(\mathbf{x}_{k+1},\mathbf{u}_{k+1},t_{k+1})\Big]. \tag{112}$$

With a quadratic interpolation that includes the midpoint, Simpson's rule appears in the cost and the Hermite–Simpson relations tie $\mathbf{x}_{k+\frac{1}{2}}$ to endpoint values and slopes. More generally, **collocation** chooses interior nodes on $[t_k,t_{k+1}]$ (equally spaced gives Newton–Cotes like trapezoid or Simpson; Gaussian points give Gauss, Radau, or Lobatto schemes) and enforces the ODE at those nodes:

$$\frac{d}{dt}\mathbf{x}(t_{k,j})=\mathbf{f}\big(\mathbf{x}(t_{k,j}),\mathbf{u}(t_{k,j}),t_{k,j}\big), \tag{113}$$

with continuity at endpoints. The interior values $\mathbf{x}(t_{k,j})$ are **evaluations of the decision polynomials**; $\mathbf{u}(t_{k,j})$ follows from the chosen control interpolation (constant, linear, or quadratic). The running cost is evaluated by the same interpolatory quadrature at the same nodes, which keeps "where we pay" aligned with "where we enforce."

2.2.2 Polynomial Interpolation

We often want to construct a function that passes through a given set of points. For example, suppose we know a function should satisfy:

$$f(x_0)=y_0,\quad f(x_1)=y_1,\quad \ldots,\quad f(x_m)=y_m. \tag{114}$$

These are called **interpolation constraints**. Our goal is to find a function $f(x)$ that satisfies all of them exactly.

To make the problem tractable, we restrict ourselves to a class of functions. In polynomial interpolation, we assume that $f(x)$ is a polynomial of degree at most N. That means we are trying to find coefficients $c_0,\ldots,c_N$ such that

$$f(x)=\sum_{n=0}^{N} c_n\,\phi_n(x), \tag{115}$$

where the functions $\phi_n(x)$ form a basis for the space of polynomials. The most common choice is the **monomial basis**, where $\phi_n(x)=x^n$. This gives:

$$f(x)=c_0+c_1x+c_2x^2+\cdots+c_Nx^N. \tag{116}$$

Other valid bases include Legendre, Chebyshev, and Lagrange polynomials, each chosen for specific numerical properties. But all span the same function space.

To find a unique solution, we need the number of unknowns (the c_n) to match the number of constraints. Since a degree-N polynomial has $N+1$ coefficients, we need:

$$N + 1 = m + 1 \quad \Rightarrow \quad N = m. \tag{117}$$

So if we want a function that passes through 4 points, we need a cubic polynomial ($N = 3$). Choosing a higher degree than necessary would give us infinitely many solutions; a lower degree may make the problem unsolvable.

Solving for the Coefficients (Monomial Basis) If we fix the basis functions to be monomials, we can build a system of equations by plugging in each x_i into $f(x)$. This gives:

$$\begin{aligned} f(x_0) &= c_0 + c_1 x_0 + c_2 x_0^2 + \cdots + c_N x_0^N = y_0 \\ f(x_1) &= c_0 + c_1 x_1 + c_2 x_1^2 + \cdots + c_N x_1^N = y_1 \\ &\vdots \\ f(x_m) &= c_0 + c_1 x_m + c_2 x_m^2 + \cdots + c_N x_m^N = y_m \end{aligned} \tag{118}$$

This system can be written in matrix form as:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^N \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{119}$$

The matrix on the left is called the **Vandermonde matrix**. Solving this system gives the coefficients c_n that define the interpolating polynomial.

Using a Different Basis We don't have to use monomials. We can pick any set of basis functions $\phi_n(x)$, such as Chebyshev or Fourier modes, and follow the same steps. The interpolating function becomes:

$$f(x) = \sum_{n=0}^{N} c_n \, \phi_n(x), \tag{120}$$

and each interpolation constraint becomes:

$$f(x_i) = \sum_{n=0}^{N} c_n \, \phi_n(x_i) = y_i. \tag{121}$$

Assembling these into a system gives:

$$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \dots & \phi_N(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \dots & \phi_N(x_1) \\ \vdots & \vdots & & \vdots \\ \phi_0(x_m) & \phi_1(x_m) & \dots & \phi_N(x_m) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{122}$$

From here, we solve as before and reconstruct $f(x)$.

Derivative Constraints Sometimes, instead of a value constraint $f(x_i) = y_i$, we want to impose a slope constraint $f'(x_i) = s_i$. This is common in applications like spline interpolation or collocation methods, where derivative information is available from an ODE.

Since

$$f(x) = \sum_{n=0}^{N} c_n \phi_n(x) \quad \Rightarrow \quad f'(x) = \sum_{n=0}^{N} c_n \phi_n'(x), \tag{123}$$

we can directly write the slope constraint:

$$f'(x_i) = \sum_{n=0}^{N} c_n \phi_n'(x_i) = s_i. \tag{124}$$

To enforce this, we replace one of the interpolation equations in our system with this slope constraint. The resulting system still has $m+1$ equations and $N+1 = m+1$ unknowns.

Concretely, suppose we have $k+1$ value constraints at nodes $X = \{x_0, \dots, x_k\}$ with values $Y = \{y_0, \dots, y_k\}$ and r slope constraints at nodes $Z = \{z_1, \dots, z_r\}$ with slopes $S = \{s_1, \dots, s_r\}$, with $k+1+r = N+1$. The linear system for the coefficients $\mathbf{c} = [c_0, \dots, c_N]^\top$ is

$$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_N(x_0) \\ \vdots & \vdots & & \vdots \\ \phi_0(x_k) & \phi_1(x_k) & \cdots & \phi_N(x_k) \\ \phi_0'(z_1) & \phi_1'(z_1) & \cdots & \phi_N'(z_1) \\ \vdots & \vdots & & \vdots \\ \phi_0'(z_r) & \phi_1'(z_r) & \cdots & \phi_N'(z_r) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_k \\ s_1 \\ \vdots \\ s_r \end{bmatrix}. \tag{125}$$

If a value and a slope are imposed at the same node, take $z_j = x_i$ and include both the value row and the derivative row; the system remains square. In the monomial basis, the top block is the Vandermonde matrix and the derivative block has entries $n\,x^{\,n-1}$. Once solved for $\mathbf{c}$, reconstruct

$$f(x) = \sum_{n=0}^{N} c_n\, \phi_n(x). \tag{126}$$

Interpolating ODE Trajectories (Collocation) Having established the general framework of interpolation, we now apply these concepts to the specific context of approximating trajectories governed by ordinary differential equations. The idea of applying polynomial interpolation with derivative constraints yields a method known as "collocation". More precisely, a **degree-s collocation method** is a way to discretize an ordinary differential equation (ODE) by approximating the solution on each time interval with a polynomial of degree s, and then enforcing that this polynomial satisfies the ODE exactly at s carefully chosen points (the *collocation nodes*).

Consider a dynamical system described by the ordinary differential equation:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{127}$$

where $\mathbf{x}(t)$ represents the state trajectory and $\mathbf{u}(t)$ denotes the control input. Let us focus on a single mesh interval $[t_k, t_{k+1}]$ with step size $h_k := t_{k+1} - t_k$. To work with a standardized domain, we introduce the transformation $t = t_k + h_k\tau$ that maps the physical interval $[t_k, t_{k+1}]$ to the reference interval $[0, 1]$. On this reference interval, we select a set of collocation nodes $\{\tau_j\}_{j=0}^{K} \subset [0, 1]$.

Our goal is now to approximate the unknown trajectory using a polynomial of degree K. Using a monomial basis, we represent (parameterize) our trajectory as:

$$\mathbf{x}_h(\tau) := \sum_{n=0}^{K} \mathbf{a}_n \, \tau^n \tag{128}$$

where $\mathbf{a}_n \in R^d$ are coefficient vectors to be determined. Collocation enforces the differential equation at a chosen set of nodes on $[0, 1]$. Depending on the node family, these nodes may be interior-only or may include one or both endpoints. With the polynomial state model, we can differentiate analytically. Using the change of variables $t = t_k + h_k\,\tau$, we obtain:

$$\dot{\mathbf{x}}_h(t_k + h_k\,\tau_j) = \frac{1}{h_k} \sum_{n=1}^{K} n\,\mathbf{a}_n\,\tau_j^{n-1} \tag{129}$$

The collocation condition requires that this polynomial derivative equals the right-hand side of the ODE at each collocation node τ_j:

$$\frac{1}{h_k} \sum_{n=1}^{K} n\,\mathbf{a}_n\,\tau_j^{n-1} = \mathbf{f}\left(\sum_{n=0}^{K} \mathbf{a}_n\,\tau_j^n,\ \mathbf{u}_j,\ t_k + h_k\,\tau_j\right), \quad \text{for each collocation node } \tau_j. \tag{130}$$

where $\mathbf{u}_j$ represents the control value at node τ_j.

Boundary Conditions and Node Families The choice of collocation nodes determines how boundary conditions are handled and affects the resulting

discretization properties. Three standard families are commonly used: Labatto, Randau and and Gauss.

Let's consider these three setup with more generality over any given basis. We start again by taking a mesh interval $[t_k, t_{k+1}]$ of length $h_k = t_{k+1} - t_k$ and reparametrize time by

$$t = t_k + h_k\,\tau, \qquad \tau \in [0,1]. \tag{131}$$

We then choose to represent the (unknown) state by a degree–K polynomial

$$\mathbf{x}_h(\tau) \;=\; \sum_{n=0}^{K} \mathbf{a}_n\,\phi_n(\tau), \tag{132}$$

where $\{\phi_n\}_{n=0}^{K}$ is any linearly independent basis of polynomials of degree $\le K$, and $\mathbf{a}_0, \ldots, \mathbf{a}_K$ are vector coefficients to be determined.

The **collocation condition** mean that we require for the chosen K collocation points that:

$$\frac{d}{dt}\mathbf{x}_h\big(\tau_j\big) \;=\; \mathbf{f}\big(\mathbf{x}_h(\tau_j), \mathbf{u}_h(\tau_j), t_k + h_k\tau_j\big), \qquad j = 0, 1, \ldots, K, \tag{133}$$

which, using $\frac{d}{dt} = (1/h_k)\frac{d}{d\tau}$, becomes

$$\underbrace{\tfrac{1}{h_k}\mathbf{x}_h'(\tau_j)}_{\text{polynomial slope at node}} \;=\; \underbrace{\mathbf{f}\big(\mathbf{x}_h(\tau_j), \mathbf{u}_h(\tau_j), t_k + h_k\tau_j\big)}_{\text{ODE slope at same point}}, \qquad j = 0, \ldots, K. \tag{134}$$

Put simply: choose some nodes inside the interval, and at each of those nodes force the slope of the polynomial approximation to match the slope prescribed by the ODE. What we mean by the expression "collocation conditions" is simply to say that we want to satisfy a set of "slope-matching equations" at the chosen nodes.

By **definition of the mesh variables**,

$$\mathbf{x}_k := \mathbf{x}_h(0), \qquad \mathbf{x}_{k+1} := \mathbf{x}_h(1), \tag{135}$$

and (if you sample the control at endpoints)

$$\mathbf{u}_k := \mathbf{u}_h(0), \qquad \mathbf{u}_{k+1} := \mathbf{u}_h(1). \tag{136}$$

With the monomial basis,

$$\phi_n(0) = \delta_{n0} \;\Rightarrow\; \mathbf{x}_h(0) = \sum_{n=0}^{K} \mathbf{a}_n\phi_n(0) = \mathbf{a}_0 = \mathbf{x}_k, \tag{137}$$

$$\phi_n(1) = 1 \;\Rightarrow\; \mathbf{x}_h(1) = \sum_{n=0}^{K} \mathbf{a}_n = \mathbf{x}_{k+1}. \tag{138}$$

For the derivative, $\phi_n'(\tau) = n\,\tau^{n-1}$, so

$$\mathbf{x}_h'(0) = \sum_{n=0}^{K} \mathbf{a}_n\,\phi_n'(0) = \mathbf{a}_1, \qquad \mathbf{x}_h'(1) = \sum_{n=1}^{K} n\,\mathbf{a}_n. \tag{139}$$

When chaining intervals into a global trajectory, **direct collocation enforces state continuity by construction**: the variable $\mathbf{x}_{k+1}$ at the end of one interval is the same as the starting variable of the next. What is *not* enforced automatically is **slope continuity**; the derivative at the end of one interval generally does not match the derivative at the start of the next. Different collocation methods may have different slope continuity properties depending on the chosen collocation nodes.

Lobatto Nodes (endpoints included):

The family of Labotto methods correspond to any set of so-called **Lobatto nodes** $\{\tau_j\}_{j=0}^{K}$ with the specificity that we require $\tau_0 = 0$ and $\tau_K = 1$. Let's assume that we work with the **power (monomial) basis** $\phi_n(\tau) = \tau^n$, so that

$$\mathbf{x}_h(\tau) = \sum_{n=0}^{K} \mathbf{a}_n\,\tau^n, \tag{140}$$

Differentiating $\mathbf{x}_h$ with respect to τ gives

$$\frac{d\mathbf{x}_h}{d\tau}(\tau) = \sum_{n=0}^{K} \mathbf{a}_n\,\phi_n'(\tau), \tag{141}$$

Since we have the chain rule $\frac{d}{dt} = \frac{1}{h_k}\frac{d}{d\tau}$ from the time transformation $t = t_k + h_k\tau$, the time derivative becomes

$$\frac{d\mathbf{x}_h}{dt}(t_k + h_k\tau) = \frac{1}{h_k}\frac{d\mathbf{x}_h}{d\tau}(\tau) = \frac{1}{h_k}\sum_{n=0}^{K} \mathbf{a}_n\,\phi_n'(\tau). \tag{142}$$

so the **collocation equations** at Lobatto nodes are

$$\frac{1}{h_k}\sum_{n=0}^{K} \mathbf{a}_n\,\phi_n'(\tau_j) \;=\; \mathbf{f}\Big(\sum_{n=0}^{K} \mathbf{a}_n\,\phi_n(\tau_j),\; \mathbf{u}_h(\tau_j),\; t_k + h_k\tau_j\Big), \qquad j = 0, 1, \ldots, K. \tag{143}$$

For $j = 0$ and $j = K$, these conditions become:

$$\frac{1}{h_k}\sum_{n=0}^{K} \mathbf{a}_n\,\phi_n'(0) \;=\; \mathbf{f}\Big(\sum_{n=0}^{K} \mathbf{a}_n\,\phi_n(0),\; \mathbf{u}_h(0),\; t_k\Big), \qquad \text{(left endpoint)} \tag{144}$$

$$\frac{1}{h_k}\sum_{n=0}^{K} \mathbf{a}_n\,\phi_n'(1) \;=\; \mathbf{f}\Big(\sum_{n=0}^{K} \mathbf{a}_n\,\phi_n(1),\; \mathbf{u}_h(1),\; t_{k+1}\Big), \qquad \text{(right endpoint)} \tag{145}$$

With the monomial basis $\phi_n(\tau) = \tau^n$, we have $\phi_n'(0) = n\delta_{n,1}$ (only $\phi_1' = 1$, others vanish) and $\phi_n'(1) = n$. Also, $\phi_n(0) = \delta_{n,0}$ and $\phi_n(1) = 1$ for all n. This simplifies the endpoint conditions to:

$$\frac{\mathbf{a}_1}{h_k} = \mathbf{f}(\mathbf{a}_0, \mathbf{u}_h(0), t_k) = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, t_k), \qquad \text{(left endpoint slope)} \tag{146}$$

$$\frac{1}{h_k}\sum_{n=1}^{K} n\,\mathbf{a}_n = \mathbf{f}\Big(\sum_{n=0}^{K}\mathbf{a}_n, \mathbf{u}_h(1), t_{k+1}\Big) = \mathbf{f}(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, t_{k+1}), \qquad \text{(right endpoint slope)} \tag{147}$$

These equations enforce that the polynomial's slope at both endpoints matches the ODE's prescribed slope, which is why the figure shows red tangent lines at both endpoints for Lobatto methods.

Radau Nodes (one endpoint included): Radau points include only one endpoint. Radau-I includes the left endpoint ($\tau_0 = 0$) while Radau-II includes the right endpoint ($\tau_K = 1$). This means that a radau collocation is defined by any set of collocation nodes such that $\tau_K = 1$. This translates into requireing that we match the ODE over the mesh only at the right endpoiunt in addition to the interior nodes. As a consequence, we leave the solution unconstrained to take any value on the left endpoint. When chaining up multiple intervals across a global solution, this may pose some complication as we will no longer be able to ensure continuity as the slope at one endpoitn need not match that of the next endpoint. (But could you have a situation where slopes match but the states don't line up?)

At the included endpoint the ODE is enforced (slope shown in the figure), while at the other endpoint continuity links adjacent intervals. For Radau-I with $K + 1$ points:

$$\mathbf{x}_k = \mathbf{x}_h(0) = \mathbf{a}_0 \tag{148}$$

The endpoint $\mathbf{x}_{k+1} = \mathbf{x}_h(1) = \sum_{n=0}^{K}\mathbf{a}_n$ is not directly constrained by a collocation condition, requiring separate continuity enforcement between intervals.

Gauss Nodes (endpoints excluded): Gauss points exclude both endpoints, using only interior points $\tau_j \in (0, 1)$ for $j = 1, \ldots, K$. The ODE is enforced only at interior nodes; both endpoints are handled through separate continuity constraints:

$$\mathbf{x}_k = \mathbf{x}_h(0) = \mathbf{a}_0 \tag{149}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_h(1) = \sum_{n=0}^{K}\mathbf{a}_n \tag{150}$$

Origins and Selection Criteria: These node families derive from orthogonal polynomial theory. Gauss nodes correspond to roots of Legendre polynomials and provide optimal quadrature accuracy for smooth integrands. Radau

nodes are roots of modified Legendre polynomials with one endpoint constraint, while Lobatto nodes include both endpoints and correspond to roots of derivatives of Legendre polynomials.

For optimal control applications, Radau-II nodes are often preferred because they provide implicit time-stepping behavior and good stability properties. Lobatto nodes simplify boundary condition handling but may require smaller time steps. Gauss nodes offer highest quadrature accuracy but complicate endpoint treatment.

Control Parameterization and Cost Integration The control inputs can be handled with similar polynomial approximations. We may use piecewise-constant controls, piecewise-linear controls, or higher-order polynomial parameterizations of the form:

$$\mathbf{u}_h(\tau) = \sum_{n=0}^{K_u} \mathbf{b}_n \, \tau^n \tag{151}$$

where $\mathbf{u}_j = \mathbf{u}_h(\tau_j)$ represents the control values at each collocation point. This polynomial framework extends to cost function evaluation, where running costs are integrated using the same quadrature nodes and weights:

$$\int_{t_k}^{t_{k+1}} c \, dt \approx h_k \sum_{j=0}^{K} w_j \, c\big(\mathbf{x}_h(\tau_j), \mathbf{u}_h(\tau_j), t_k + h_k \, \tau_j\big) \tag{152}$$

2.2.3 A Compendium of Direct Transcription Methods in Trajectory Optimization

The mesh and interior nodes are the common scaffold. What distinguishes one transcription from another is how we obtain values at those nodes and how we approximate the two integrals that appear implicitly and explicitly: the integral of the running cost and the integral that carries the state forward. In other words, we now commit to two design choices that mirror the previous section: a finite representation for $\mathbf{x}(t)$ and $\mathbf{u}(t)$ over each interval $[t_i, t_{i+1}]$, and a quadrature rule whose nodes and weights are used consistently for both cost and dynamics. The result is always a sparse nonlinear program; the differences are in where we sample and how we tie samples together.

Below, each transcription should be read as "same grid, same interior points, same evaluations for cost and physics," with only the local representation changing.

Euler Collocation Work on one interval $[t_k, t_{k+1}]$ of length h_k with the reparametrization $t = t_k + h_k \, \tau$, $\tau \in [0, 1]$. Assume a degree 1 polynomial:

$$\mathbf{x}_h(\tau) = \sum_{n=0}^{1} \mathbf{a}_n \, \phi_n(\tau), \tag{153}$$

for any basis $\{\phi_0, \phi_1\}$ of linear polynomials. Endpoint conditions give

$$\mathbf{x}_h(0) = \mathbf{x}_k \Rightarrow \mathbf{a}_0 = \mathbf{x}_k, \qquad \mathbf{x}_h(1) = \mathbf{x}_{k+1} \Rightarrow \mathbf{a}_1 = \mathbf{x}_{k+1} - \mathbf{x}_k. \tag{154}$$

by backsubstitution and because

$$\mathbf{x}_h(\tau) = \mathbf{a}_0 + \mathbf{a}_1\,\tau. \tag{155}$$

Furthermore, the derivative with respect to τ is:

$$\frac{d}{d\tau}\mathbf{x}_h(\tau) = \mathbf{a}_1 = \mathbf{x}_{k+1} - \mathbf{x}_k, \qquad \frac{d}{dt} = \frac{1}{h_k}\frac{d}{d\tau} \Rightarrow \frac{d}{dt}\mathbf{x}_h = \frac{1}{h_k}\,(\mathbf{x}_{k+1} - \mathbf{x}_k). \tag{156}$$

Because from the mapping $t(\tau) = t_k + h_k\tau$ we can invert: $\tau(t) = \frac{t-t_k}{h_k}$ and differentiating gives

$$\frac{d\tau}{dt} = \frac{1}{h_k}. \tag{157}$$

The **collocation condition** at a single **Radau-II node** $\tau = 1$:

$$\frac{1}{h_k}\,\mathbf{x}_h'(\tau)\Big|_{\tau=1} \;=\; \mathbf{f}\big(\mathbf{x}_h(1), \mathbf{u}_h(1), t_{k+1}\big) \;=\; \mathbf{f}\big(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, t_{k+1}\big). \tag{158}$$

Because $\mathbf{x}_h$ is linear, $\mathbf{x}_h'(\tau)$ is constant in τ, so $\mathbf{x}_h'(1) = \mathbf{x}_h'(\tau)$ for all τ. Moreover, linear interpolation between the two endpoints gives

$$\mathbf{x}_h'(\tau) = \mathbf{x}_{k+1} - \mathbf{x}_k. \tag{159}$$

Substitute this into the collocation condition:

$$\frac{1}{h_k}\big(\mathbf{x}_{k+1} - \mathbf{x}_k\big) \;=\; \mathbf{f}\big(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, t_{k+1}\big), \tag{160}$$

which is exactly the **implicit Euler** step

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h_k\,\mathbf{f}\big(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, t_{k+1}\big)\ . \tag{161}$$

The overall direct transcription is then:

Definition 2.1 (Implicit–Euler Collocation (Radau-II, degree 1))**.** *Let* $t_0 < \cdots < t_N$ *with* $h_i := t_{i+1} - t_i$*. Decision variables are* $\{\mathbf{x}_i\}_{i=0}^{N}$*,* $\{\mathbf{u}_i\}_{i=0}^{N}$*. Solve*

$$\begin{aligned}
\min\ & c_T(\mathbf{x}_N) \;+\; \sum_{i=0}^{N-1} h_i\, c(\mathbf{x}_{i+1}, \mathbf{u}_{i+1}) \\
s.t.\ & \mathbf{x}_{i+1} - \mathbf{x}_i - h_i\,\mathbf{f}(\mathbf{x}_{i+1}, \mathbf{u}_{i+1}) = \mathbf{0}, \quad i = 0, \ldots, N-1, \\
& \mathbf{g}(\mathbf{x}_{i+1}, \mathbf{u}_{i+1}) \le \mathbf{0}, \\
& \mathbf{x}_{\min} \le \mathbf{x}_i \le \mathbf{x}_{\max}, \quad \mathbf{u}_{\min} \le \mathbf{u}_i \le \mathbf{u}_{\max}, \\
& \mathbf{x}_0 = \mathbf{x}(t_0).
\end{aligned} \tag{162}$$

Note that:

- The running cost and path constraints are evaluated at the **same** right-endpoint where the dynamics are enforced, keeping "where we pay" aligned with "where we enforce."
- State continuity is automatic because $\mathbf{x}_{i+1}$ is a shared variable between adjacent intervals; slope continuity is not enforced unless you add it. Here's an updated subsection that explicitly says **what collocation nodes are chosen** and why the trapezoidal defect uses them the way it does.

  Side remark. If you instead collocate at the **left** endpoint (Radau-I with $\tau = 0$) with the same linear model, you obtain $\frac{1}{h_k}(\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, t_k)$, i.e., the **explicit Euler** step. In that very precise sense, explicit Euler can be viewed as a (left-endpoint) degree-1 collocation scheme.

Definition 2.2 (Explicit–Euler Collocation (Radau-I, degree 1))**.** *Let* $t_0 < \cdots < t_N$ *with* $h_i := t_{i+1} - t_i$*. Decision variables are* $\{\mathbf{x}_i\}_{i=0}^N$ *and* $\{\mathbf{u}_i\}_{i=0}^N$*. Solve*

$$\begin{aligned} \min\ & c_T(\mathbf{x}_N) \ + \ \sum_{i=0}^{N-1} h_i\, c(\mathbf{x}_i, \mathbf{u}_i) \\ s.t.\ & \mathbf{x}_{i+1} - \mathbf{x}_i - h_i\, \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) = \mathbf{0}, \quad i = 0, \ldots, N-1, \\ & \mathbf{g}(\mathbf{x}_i, \mathbf{u}_i) \le \mathbf{0}, \quad i = 0, \ldots, N-1, \\ & \mathbf{x}_{\min} \le \mathbf{x}_i \le \mathbf{x}_{\max}, \quad \mathbf{u}_{\min} \le \mathbf{u}_i \le \mathbf{u}_{\max}, \\ & \mathbf{x}_0 = \mathbf{x}(t_0). \end{aligned} \tag{163}$$

Trapezoidal collocation In this scheme we take the **two endpoints as the nodes** on each interval:

$$\tau_0 = 0, \qquad \tau_1 = 1 \quad (\text{"Lobatto with } K = 1\text{"}). \tag{164}$$

We approximate **x linearly** over $[t_i, t_{i+1}]$, and we evaluate both the running cost and the dynamics at these two nodes with **equal weights**. Because a linear polynomial has a **constant** derivative, we do **not** try to match the ODE's slope at both endpoints (that would overconstrain a linear function). Instead, we enforce the ODE in its **integral form** over the interval and approximate the integral of **f** by the **trapezoid rule** using those two nodes. This makes the cost quadrature and the state-update ("defect") use the **same nodes and weights**.

Definition 2.3 (Trapezoidal Collocation)**.** *Let* $t_0 < \cdots < t_N$ *with* $h_i := t_{i+1} - t_i$*. Decision variables are* $\{\mathbf{x}_i\}_{i=0}^N$*,* $\{\mathbf{u}_i\}_{i=0}^N$*. Solve*

$$
\begin{aligned}
\min_{\{\mathbf{x}_i,\mathbf{u}_i\}} \; & c_T(\mathbf{x}_N) \; + \; \sum_{i=0}^{N-1} \tfrac{h_i}{2} \Big[c(\mathbf{x}_i,\mathbf{u}_i) + c(\mathbf{x}_{i+1},\mathbf{u}_{i+1}) \Big] \\
s.t. \; & \mathbf{x}_{i+1} - \mathbf{x}_i \; - \; \tfrac{h_i}{2} \Big[\mathbf{f}(\mathbf{x}_i,\mathbf{u}_i) + \mathbf{f}(\mathbf{x}_{i+1},\mathbf{u}_{i+1}) \Big] \; = \; \mathbf{0}, \quad i = 0,\dots,N-1, \\
& \mathbf{g}(\mathbf{x}_i,\mathbf{u}_i) \le \mathbf{0}, \quad \mathbf{g}(\mathbf{x}_{i+1},\mathbf{u}_{i+1}) \le \mathbf{0}, \\
& \mathbf{x}_{\min} \le \mathbf{x}_i \le \mathbf{x}_{\max}, \quad \mathbf{u}_{\min} \le \mathbf{u}_i \le \mathbf{u}_{\max}, \\
& \mathbf{x}_0 = \mathbf{x}(t_0).
\end{aligned}
\tag{165}
$$

Summary: the **collocation nodes** for trapezoidal are the **two endpoints**; the state is **linear** on each interval; and the dynamics are enforced via the **integrated** ODE with the **trapezoid rule** at those two nodes, yielding the familiar trapezoidal defect.

Hermite–Simpson (quadratic interpolation; midpoint included) On each interval $[t_i, t_{i+1}]$ we pick **three collocation nodes** on the reference domain $\tau \in [0,1]$:

$$
\tau_0 = 0, \qquad \tau_{1/2} = \tfrac{1}{2}, \qquad \tau_1 = 1. \tag{166}
$$

So we evaluate at **left, midpoint, right**. These are the same three nodes used by Simpson's rule (weights 1:4:1) for numerical quadrature. We let $\mathbf{x}_h$ be **quadratic** in τ. Two things then happen:

1. **Integral (defect) enforcement with Simpson's rule.** We enforce the ODE in integral form over the interval and approximate the integral of $\mathbf{f}$ with Simpson's rule using the three nodes above. This yields the first constraint (the "Simpson defect"), which uses $\mathbf{f}$ evaluated at left, midpoint, and right.

2. **Slope matching at the midpoint (collocation).** Because a quadratic has limited shape, we don't try to match slopes at both endpoints. Instead, we **introduce midpoint variables** $(\mathbf{x}_{i+\frac{1}{2}}, \mathbf{u}_{i+\frac{1}{2}})$ and **match the ODE at the midpoint**. The second constraint below is exactly the midpoint collocation condition written in an equivalent Hermite form: it pins the midpoint state to the average of the endpoints plus a correction based on endpoint slopes, ensuring that the polynomial's derivative is consistent with the ODE **at** $\tau = \frac{1}{2}$.

This way, **where we pay** (Simpson quadrature) and **where we enforce** (midpoint collocation + Simpson defect) are aligned at the same three nodes, which is why the method is both accurate and well conditioned on smooth problems.

Definition 2.4 (Hermite–Simpson Transcription)**.** *Let* $t_0 < \cdots < t_N$ *with* $h_i := t_{i+1} - t_i$ *and midpoints* $t_{i+\frac{1}{2}}$*. Decision variables are* $\{\mathbf{x}_i\}_{i=0}^{N}$*,* $\{\mathbf{u}_i\}_{i=0}^{N}$*, plus midpoint variables* $\{\mathbf{x}_{i+\frac{1}{2}}, \mathbf{u}_{i+\frac{1}{2}}\}_{i=0}^{N-1}$*. Solve*

$$
\begin{aligned}
&\min\ c_T(\mathbf{x}_N) \ + \ \sum_{i=0}^{N-1} \tfrac{h_i}{6}\Big[c(\mathbf{x}_i,\mathbf{u}_i) + 4\,c(\mathbf{x}_{i+\frac{1}{2}},\mathbf{u}_{i+\frac{1}{2}}) + c(\mathbf{x}_{i+1},\mathbf{u}_{i+1})\Big] \\
&s.t.\ \underbrace{\mathbf{x}_{i+1} - \mathbf{x}_i - \tfrac{h_i}{6}\Big[\mathbf{f}(\mathbf{x}_i,\mathbf{u}_i) + 4\,\mathbf{f}(\mathbf{x}_{i+\frac{1}{2}},\mathbf{u}_{i+\frac{1}{2}}) + \mathbf{f}(\mathbf{x}_{i+1},\mathbf{u}_{i+1})\Big]}_{\textit{Simpson defect over } [t_i,t_{i+1}]} = \mathbf{0}, \\
&\quad \underbrace{\mathbf{x}_{i+\frac{1}{2}} - \tfrac{\mathbf{x}_i+\mathbf{x}_{i+1}}{2} - \tfrac{h_i}{8}\Big[\mathbf{f}(\mathbf{x}_i,\mathbf{u}_i) - \mathbf{f}(\mathbf{x}_{i+1},\mathbf{u}_{i+1})\Big]}_{\textit{midpoint collocation (slope matching at } t_{i+\frac{1}{2}})} = \mathbf{0}, \qquad (167) \\
&\quad \mathbf{g}(\mathbf{x}_i,\mathbf{u}_i) \le \mathbf{0}, \quad \mathbf{g}(\mathbf{x}_{i+\frac{1}{2}},\mathbf{u}_{i+\frac{1}{2}}) \le \mathbf{0}, \quad \mathbf{g}(\mathbf{x}_{i+1},\mathbf{u}_{i+1}) \le \mathbf{0}, \\
&\quad \mathbf{x}_{\min} \le \mathbf{x}_i, \mathbf{x}_{i+\frac{1}{2}} \le \mathbf{x}_{\max}, \quad \mathbf{u}_{\min} \le \mathbf{u}_i, \mathbf{u}_{i+\frac{1}{2}} \le \mathbf{u}_{\max}, \\
&\quad \mathbf{x}_0 = \mathbf{x}(t_0), \quad i = 0,\dots,N-1.
\end{aligned}
$$

Collocation nodes recap: $\tau = 0,\ \frac{1}{2},\ 1.$

- The **midpoint** is where we explicitly **match the ODE slope** (collocation).
- The **three nodes together** are used for the **Simpson integral** of $\mathbf{f}$ (state update) and of c (cost), keeping physics and objective synchronized.

2.2.4 Examples

Compressor Surge Problem A compressor is a machine that raises the pressure of a gas by squeezing it into a smaller volume. You find them in natural gas pipelines, jet engines, and factories. But compressors can run into trouble if the flow of gas becomes too small. In that case, the machine can "stall" much like an airplane wing at too high an angle. Instead of moving forward, the gas briefly pushes back, creating strong pressure oscillations that can damage the compressor and anything connected to it.

To prevent this, engineers often add a close-coupled valve (CCV) at the outlet. The valve can quickly adjust the flow to keep the compressor away from these unstable conditions. Our goal is to design a control strategy for operating this valve so that the compressor never enters surge.

Following [Gravdahl and Egeland, 1997] and Grancharova and Johansen [2012], we model the compressor using a simplified second-order representation:

$$
\begin{aligned}
\dot{x}_1 &= B(\Psi_e(x_1) - x_2 - u) \\
\dot{x}_2 &= \frac{1}{B}(x_1 - \Phi(x_2))
\end{aligned} \qquad (168)
$$

Here, $\mathbf{x} = [x_1, x_2]^T$ represents the state variables:

- x_1 is the normalized mass flow through the compressor.
- x_2 is the normalized pressure ratio across the compressor.

The control input u denotes the normalized mass flow through a CCV. The functions $\Psi_e(x_1)$ and $\Phi(x_2)$ represent the characteristics of the compressor and valve, respectively:

$$\begin{aligned} \Psi_e(x_1) &= \psi_{c0} + H\left(1 + 1.5\left(\frac{x_1}{W} - 1\right) - 0.5\left(\frac{x_1}{W} - 1\right)^3\right) \\ \Phi(x_2) &= \gamma \operatorname{sign}(x_2)\sqrt{|x_2|} \end{aligned} \tag{169}$$

The system parameters are given as $\gamma = 0.5$, $B = 1$, $H = 0.18$, $\psi_{c0} = 0.3$, and $W = 0.25$.

One possible way to pose the problem Grancharova and Johansen [2012] is by penalizing deviations from the setpoints using a quadratic penalty in the instantaneous cost function as well as in the terminal one. Furthermore, we also penalize taking large actions (which are energy hungry and potentially unsafe) within the integral term. The idea of penalizing deviations throughout is a natural way of posing the problem when solving it via single shooting. Another alternative, which we will explore below, is to set the desired setpoint as a hard terminal constraint.

The control objective is to stabilize the system and prevent surge, formulated as a continuous-time optimal control problem (COCP) in the Bolza form:

$$\begin{aligned} \text{minimize} \quad & \left[\int_0^T \alpha(\mathbf{x}(t) - \mathbf{x}^*)^T(\mathbf{x}(t) - \mathbf{x}^*) + \kappa u(t)^2\, dt\right] + \beta(\mathbf{x}(T) - \mathbf{x}^*)^T(\mathbf{x}(T) - \mathbf{x}^*) + Rv^2 \\ \text{subject to} \quad & \dot{x}_1(t) = B(\Psi_e(x_1(t)) - x_2(t) - u(t)) \\ & \dot{x}_2(t) = \frac{1}{B}(x_1(t) - \Phi(x_2(t))) \\ & u_{\min} \leq u(t) \leq u_{\max} \\ & -x_2(t) + 0.4 \leq v \\ & -v \leq 0 \\ & \mathbf{x}(0) = \mathbf{x}_0 \end{aligned} \tag{170}$$

The parameters α, β, κ, and R are non-negative weights that allow the designer to prioritize different aspects of performance (e.g., tight setpoint tracking vs. smooth control actions). We also constraint the control input to be within $0 \leq u(t) \leq 0.3$ due to the physical limitations of the valve.

The authors in Grancharova and Johansen [2012] also add a soft path constraint $x_2(t) \geq 0.4$ to ensure that we maintain a minimum pressure at all time. This is implemented as a soft constraint using slack variables. The reason that we have the term Rv^2 in the objective is to penalizes violations of the soft constraint: we allow for deviations, but don't want to do it too much.

In the experiment below, we choose the setpoint $\mathbf{x}^* = [0.40, 0.60]^T$ as it corresponds to an unstable equilibrium point. If we were to run the system without applying any control, we would see that the system starts to oscillate.

Solution by Trapezoidal Collocation Another way to pose the problem is by imposing a terminal state constraint on the system rather than through a penalty in the integral term. In the following experiment, we use a problem formulation of the form:

$$\begin{aligned} \text{minimize} \quad & \left[\int_0^T \kappa u(t)^2 \, dt\right] \\ \text{subject to} \quad & \dot{x}_1(t) = B(\Psi_e(x_1(t)) - x_2(t) - u(t)) \\ & \dot{x}_2(t) = \frac{1}{B}(x_1(t) - \Phi(x_2(t))) \\ & u_{\min} \leq u(t) \leq u_{\max} \\ & \mathbf{x}(0) = \mathbf{x}_0 \\ & \mathbf{x}(T) = \mathbf{x}^\star \end{aligned} \tag{171}$$

We then find a control function $u(t)$ and state trajectory $x(t)$ using the trapezoidal collocation method.

You can try to vary the number of collocation points in the code and observe how the state trajectory progressively matches the ground truth (the line denoted "integrated solution"). Note that this version of the code also lacks bound constraints on the variable x_2 to ensure a minimum pressure, as we did earlier. Consider this a good exercise to try on your own.

System Identification as Trajectory Optimization (Compressor Surge) We now turn the compressor surge model into a simple system identification task: estimate unknown parameters (here, the scalar B) from measured trajectories. This can be viewed as a trajectory optimization problem: choose parameters (and optionally states) to minimize reconstruction error while enforcing the dynamics.

Given time-aligned data $\{(\mathbf{u}_k, \mathbf{y}_k)\}_{k=0}^N$, model states $\mathbf{x}_k \in R^d$, outputs $\mathbf{y}_k \approx \mathbf{h}(\mathbf{x}_k; \boldsymbol{\theta})$, step Δt, and dynamics $\mathbf{f}(\mathbf{x}, \mathbf{u}; \boldsymbol{\theta})$, the simultaneous (full-discretization) viewpoint is

$$\begin{aligned} \min_{\boldsymbol{\theta},\, \{\mathbf{x}_k\}} \quad & \sum_{k \in K} \left\| \mathbf{y}_k - \mathbf{h}(\mathbf{x}_k; \boldsymbol{\theta}) \right\|_2^2 \\ \text{s.t.} \quad & \mathbf{x}_{k+1} - \mathbf{x}_k - \Delta t \, \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k; \boldsymbol{\theta}) = \mathbf{0}, \quad k = 0, \ldots, N-1, \\ & \mathbf{x}_0 \text{ given}, \end{aligned} \tag{172}$$

while the single-shooting (recursive elimination) variant eliminates the states by simulating forward from $\mathbf{x}_0$:

$$J(\boldsymbol{\theta}) := \sum_{k \in K} \left\| \mathbf{y}_k - \mathbf{h}(\boldsymbol{\phi}_k(\boldsymbol{\theta}; \mathbf{x}_0, \mathbf{u}_{0:N-1}) \right\|_2^2, \quad \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \tag{173}$$

where $\boldsymbol{\phi}_k$ denotes the state reached at step k by an RK4 rollout under parameter $\boldsymbol{\theta}$. In our demo the data grid and rollout grid coincide, so $\boldsymbol{\phi}_k = \mathbf{x}_k$ and no interpolation is required. We will identify B by fitting the model to data generated from the ground-truth $B = 1$ system under randomized initial conditions and small input perturbations.

Flight Trajectory Optimization We consider a concrete task: computing a fuel-optimal trajectory between Montréal–Trudeau (CYUL) and Toronto Pearson (CYYZ), taking into account both aircraft dynamics and wind conditions along the route. For this demo, we leverage the excellent library OpenAP.top which provides direct transcription methods and airplane dynamics models [Sun, 2022]. Furthermore, it allows us to import a a wind field comes from **ERA5** [C3S, 2018], a global atmospheric dataset. It combines historical observations from satellites, aircraft, and surface stations with a weather model to reconstruct the state of the atmosphere across space and time. In climate science, this is called a *reanalysis*.

ERA5 data is stored in **GRIB files**, a compact format widely used in meteorology. Each file contains a **gridded field**: values of wind and other variables arranged on a regular 4D lattice over longitude, latitude, altitude, and time. Since the aircraft rarely sits exactly on a grid point, we interpolate the wind components it sees as it moves.

The aircraft is modeled as a point mass with state

$$\mathbf{x}(t) = (x(t), y(t), h(t), m(t)), \tag{174}$$

where (x, y) is horizontal position, h is altitude, and m is remaining mass. Controls are Mach number $M(t)$, vertical speed $v_s(t)$, and heading angle $\psi(t)$. The equations of motion combine airspeed and wind:

$$\begin{aligned} \dot{x} &= v(M, h) \cos\psi \cos\gamma + u_w(x, y, h, t), \\ \dot{y} &= v(M, h) \sin\psi \cos\gamma + v_w(x, y, h, t), \\ \dot{h} &= v_s, \\ \dot{m} &= -\,\mathrm{FF}(T(h, M, v_s), h, M, v_s), \end{aligned} \tag{175}$$

where $\gamma = \arcsin(v_s/v)$ is the flight path angle and FF is the fuel flow rate based on current conditions. The wind terms u_w and v_w are taken from ERA5 and interpolated in space and time.

The optimization minimizes fuel burn over the CYUL–CYYZ leg. But the same setup could be used to minimize arrival time, or some weighted combination of time, cost, and emissions.

We use **OpenAP.top**, which solves the problem using direct collocation at **Legendre–Gauss–Lobatto (LGL)** points. Each trajectory segment is

mapped to the unit interval, the state is interpolated by Lagrange polynomials at nonuniform LGL nodes, and the dynamics are enforced at those points. Integration is done with matching quadrature weights.

This setup lets us optimize trajectories under realistic conditions by feeding in the appropriate ERA5 GRIB file (e.g., `era5_mtl_20230601_12.grib`). The result accounts for wind patterns (eg. headwinds, tailwinds, shear) along the corridor between Montréal and Toronto.

Hydro Cascade Scheduling with Physical Routing and Multiple Shooting Earlier in the book, we introduced a simplified view of hydro reservoir control, where the water level evolves in discrete time by accounting for inflow and outflow, with precipitation treated as a noisy input. While useful for learning and control design, this model abstracts away much of the physical behavior of actual rivers and dams.

In this chapter, we move toward a more realistic setup inspired by [Savorgnan et al., 2011]. We consider a series of dams arranged in a cascade, where the actions taken upstream influence downstream levels with a delay. The amount of power produced depends not only on how much water flows through the turbines, but also on the head (the vertical distance between the reservoir surface and the turbine outlet). The larger the head, the more potential energy is available for conversion into electricity, and the higher the power output.

To capture these effects, we follow a modeling approach inspired by the Saint-Venant equations, which describe how water levels and flows evolve in open channels. Instead of solving the full PDEs, we use a reduced model that approximates each dammed section of river (called a reach) as a lumped system governed by an ordinary differential equation. The main variable of interest is the water level $h_r(t)$, which changes over time depending on how much water enters, how much is discharged through the turbines $q_r(t)$, and how much is spilled $s_r(t)$. The mass balance for reach r is written as:

$$\frac{dh_r(t)}{dt} = \frac{1}{A_r}\left(z_r(t) - q_r(t) - s_r(t)\right), \tag{176}$$

where A_r is the surface area of the reservoir, assumed constant. The inflow $z_r(t)$ to a reach either comes from nature (for the first dam), or from the upstream turbine and spill discharge, delayed by a travel time τ_{r-1}:

$$z_1(t) = \text{inflow}(t), \qquad z_r(t) = q_{r-1}(t-\tau_{r-1}) + s_{r-1}(t-\tau_{r-1}), \quad \text{for } r > 1. \tag{177}$$

Power generation at each reach depends on how much water is discharged and the available head:

$$P_r(t) = \rho g \eta \, q_r(t) \, H_r(h_r(t)), \tag{178}$$

where ρ is water density, g is gravitational acceleration, η is turbine efficiency, and $H_r(h_r(t))$ denotes the head as a function of the water level. In some models,

the head is approximated as the difference between the current level and a fixed tailwater height (the water level downstream of the dam, after it has passed through the turbine).

The operator's goal is to meet a target generation profile $P^{\text{ref}}(t)$, such as one dictated by a market dispatch or load-following constraint. This leads to an objective that minimizes the deviation from the target over the full horizon:

$$\min_{\{q_r(t),s_r(t)\}} \int_0^T \left(\sum_{r=1}^{R} P_r(t) - P^{\text{ref}}(t) \right)^2 dt. \tag{179}$$

In practice, this is combined with operational constraints: turbine capacity $0 \le q_r(t) \le \bar{q}_r$, spillway limits $0 \le s_r(t) \le \bar{s}_r$, and safe level bounds $h_r^{\min} \le h_r(t) \le h_r^{\max}$. Depending on the use case, one may also penalize spill to encourage water conservation, or penalize fast changes in levels for ecological reasons.

What makes this problem particularly interesting is the coupling across space and time. An upstream reach cannot simply act in isolation: if the operator wants reach r to produce power at a specific time, the water must be released by reach $r-1$ sufficiently in advance. This coordination is further complicated by delays, nonlinearities in head-dependent power, and limited storage capacity.

We solve the problem using **multiple shooting**. Each reach is divided into local simulation segments over short time windows. Within each segment, the dynamics are integrated forward using the ODEs, and continuity constraints are added to ensure that the water levels match across segment boundaries. At the same time, the inflows passed from upstream reaches must arrive at the right time and be consistent with previous decisions. In discrete time, this gives rise to a set of state-update equations:

$$h_r^{k+1} = h_r^k + \Delta t \cdot \frac{1}{A_r}(z_r^k - q_r^k - s_r^k), \tag{180}$$

with delays handled by shifting z_r^k according to the appropriate travel time. These constraints are enforced as part of a nonlinear program, alongside the power tracking objective and control bounds.

Compared to the earlier inflow-outflow model, this richer setup introduces more structure, but also more opportunity. The cascade now behaves like a coordinated team: upstream reservoirs can store water in anticipation of future needs, while downstream dams adjust their output to match arrivals and avoid overflows. The optimization reveals not just a schedule, but a strategy for how the entire system should act together to meet demand.

The figure shows the result of a multiple-shooting optimization applied to a three-reach hydroelectric cascade. The time horizon is discretized into 16 intervals, and SciPy's `trust-constr` solver is used to find a feasible control sequence that satisfies mass balance, turbine and spillway limits, and Muskingum-style routing dynamics. Each reach integrates its own local ODE, with continuity constraints linking the flows between reaches.

The top-left panel shows the water levels in each reservoir. We observe that upstream reservoirs tend to increase their levels ahead of discharge events, building potential energy before releasing water downstream. The top-right panel shows turbine discharges for each reach. These vary smoothly and are temporally coordinated across the system. The bottom-right panel compares the total generation to a synthetic demand profile, which is generated by a sum of time-shifted sigmoids and normalized to be feasible given turbine capacities. The optimized schedule (orange) tracks this demand closely, while the initial guess (blue) lags behind. The bottom-left panel plots the routed inflows between reaches, which display the expected lag and smoothing effects from Muskingum routing. The interplay between these plots shows how the system anticipates, stores, and routes water to meet time-varying generation targets within physical and operational limits.

3 From Trajectories to Policies

3.1 Model Predictive Control

The trajectory optimization methods presented so far compute a complete control trajectory from an initial state to a final time or state. Once computed, this trajectory is executed without modification, making these methods fundamentally open-loop. The control function, $\mathbf{u}[k]$ in discrete time or $\mathbf{u}(t)$ in continuous time, depends only on the clock, reading off precomputed values from memory or interpolating between them. This approach assumes perfect models and no disturbances. Under these idealized conditions, repeating the same control sequence from the same initial state would always produce identical results.

Real systems face modeling errors, external disturbances, and measurement noise that accumulate over time. A precomputed trajectory becomes increasingly irrelevant as these perturbations push the actual system state away from the predicted path. The solution is to incorporate feedback, making control decisions that respond to the current state rather than blindly following a predetermined schedule. While dynamic programming provides the theoretical framework for deriving feedback policies through value functions and Bellman equations, there exists a more direct approach that leverages the trajectory optimization methods already developed.

Closing the Loop by Replanning Model Predictive Control creates a feedback controller by repeatedly solving trajectory optimization problems. Rather than computing a single trajectory for the entire task duration, MPC solves a finite-horizon problem at each time step, starting from the current measured state. The controller then applies only the first control action from this solution before repeating the entire process. This strategy transforms any trajectory optimization method into a feedback controller.

The Receding Horizon Principle The defining characteristic of MPC is its receding horizon strategy. At each time step, the controller solves an optimization problem looking a fixed duration into the future, but this prediction window constantly moves forward in time. The horizon "recedes" because it always starts from the current time and extends forward by the same amount.

Consider the discrete-time optimal control problem in Bolza form:

$$\begin{aligned} \text{minimize} \quad & c_T(\mathbf{x}_N) + \sum_{k=0}^{N-1} c(\mathbf{x}_k, \mathbf{u}_k) \\ \text{subject to} \quad & \mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{g}(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\ & \mathbf{u}_{\min} \leq \mathbf{u}_k \leq \mathbf{u}_{\max} \\ \text{given} \quad & \mathbf{x}_0 = \mathbf{x}_{\text{current}} \end{aligned} \tag{181}$$

At time step t, this problem optimizes over the interval $[t, t+N]$. At the next time step $t+1$, the horizon shifts to $[t+1, t+N+1]$. What makes this work is that only the first control $\mathbf{u}_0^*$ from each optimization is applied. The

remaining controls $\mathbf{u}_1^*, \ldots, \mathbf{u}_{N-1}^*$ are discarded, though they may initialize the next optimization through warm-starting.

This receding horizon principle enables feedback without computing an explicit policy. By constantly updating predictions based on current measurements, MPC naturally corrects for disturbances and model errors. The apparent waste of computing but not using most of the trajectory is actually the mechanism that provides robustness.

Horizon Selection and Problem Formulation The choice of prediction horizon depends on the control objective. We distinguish between three cases, each requiring different mathematical formulations.

Infinite-Horizon Regulation For stabilization problems where the system must operate indefinitely around an equilibrium, the true objective is:

$$J_\infty = \sum_{k=0}^{\infty} c(\mathbf{x}_k, \mathbf{u}_k) \tag{182}$$

Since this cannot be solved directly, MPC approximates it with:

$$\begin{aligned} \text{minimize} \quad & V_f(\mathbf{x}_N) + \sum_{k=0}^{N-1} c(\mathbf{x}_k, \mathbf{u}_k) \\ \text{subject to} \quad & \mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{x}_N \in \mathcal{X}_f \\ & \text{other constraints} \end{aligned} \tag{183}$$

The terminal cost $V_f(\mathbf{x}_N)$ approximates $\sum_{k=N}^{\infty} c(\mathbf{x}_k, \mathbf{u}_k)$, the cost-to-go beyond the horizon. The terminal constraint $\mathbf{x}_N \in \mathcal{X}_f$ ensures the state reaches a region where a known stabilizing controller exists. Without these terminal ingredients, the finite-horizon approximation may produce unstable behavior, as the controller ignores consequences beyond the horizon.

Finite-Duration Tasks For tasks ending at time t_f, the true objective spans from current time t to t_f:

$$J_{[t,t_f]} = c_f(\mathbf{x}(t_f)) + \sum_{k=t}^{t_f-1} c(\mathbf{x}_k, \mathbf{u}_k) \tag{184}$$

The MPC formulation must adapt as time progresses:

$$\begin{aligned} \text{minimize} \quad & c_{T,k}(\mathbf{x}_{N_k}) + \sum_{j=0}^{N_k-1} c(\mathbf{x}_j, \mathbf{u}_j) \\ \text{where} \quad & N_k = \min(N, t_f - t_k) \\ & c_{T,k} = \begin{cases} c_f & \text{if } t_k + N_k = t_f \\ c_T & \text{otherwise} \end{cases} \end{aligned} \tag{185}$$

As the task approaches completion, the horizon shrinks and the terminal cost switches from the approximation c_T to the true final cost c_f. This prevents the controller from optimizing beyond task completion, which would produce meaningless or aggressive control actions.

Periodic Tasks Some systems operate on repeating cycles where the optimal behavior depends on the time of day, week, or season. Consider a commercial building where heating costs are higher at night, electricity prices vary hourly, and occupancy patterns repeat daily. The MPC controller must account for these periodic patterns while planning over a finite horizon.

For tasks with period T_p, such as daily building operations, the formulation accounts for transitions across period boundaries:

$$\begin{aligned} \text{minimize} \quad & \sum_{k=0}^{N-1} c_k(\mathbf{x}_k, \mathbf{u}_k, \phi_k) \\ \text{where} \quad & \phi_k = (t + k) \mod T_p \\ & c_k(\cdot, \cdot, \phi) = \begin{cases} c_{\text{day}}(\cdot, \cdot) & \text{if } \phi \in [6\text{am}, 6\text{pm}] \\ c_{\text{night}}(\cdot, \cdot) & \text{otherwise} \end{cases} \end{aligned} \tag{186}$$

The cost function changes based on the phase ϕ within the period. Constraints may similarly depend on the phase, reflecting different operational requirements at different times.

The MPC Algorithm The complete MPC procedure implements the receding horizon principle through repeated optimization:

Successive Linearization and Quadratic Approximations For many regulation and tracking problems, the nonlinear dynamics and costs we encounter can be approximated locally by linear and quadratic functions. The basic idea is to linearize the system around the current operating point and approximate the cost with a quadratic form. This reduces each MPC subproblem to a **quadratic program (QP)**, which can be solved reliably and very quickly using standard solvers.

Suppose the true dynamics are nonlinear,

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k). \tag{187}$$

Around a nominal trajectory $(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k)$, we take a first-order expansion:

$$\mathbf{x}_{k+1} \approx f(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k) + \mathbf{A}_k(\mathbf{x}_k - \bar{\mathbf{x}}_k) + \mathbf{B}_k(\mathbf{u}_k - \bar{\mathbf{u}}_k), \tag{188}$$

with Jacobians

$$\mathbf{A}_k = \frac{\partial f}{\partial \mathbf{x}}(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k), \qquad \mathbf{B}_k = \frac{\partial f}{\partial \mathbf{u}}(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k). \tag{189}$$

Similarly, if the stage cost is nonlinear,

$$c(\mathbf{x}_k, \mathbf{u}_k), \tag{190}$$

we approximate it quadratically near the nominal point:

$$c(\mathbf{x}_k, \mathbf{u}_k) \;\approx\; \|\mathbf{x}_k - \mathbf{x}_k^{\text{ref}}\|_{\mathbf{Q}_k}^2 + \|\mathbf{u}_k - \mathbf{u}_k^{\text{ref}}\|_{\mathbf{R}_k}^2, \tag{191}$$

with positive semidefinite weighting matrices $\mathbf{Q}_k$ and $\mathbf{R}_k$.
The resulting MPC subproblem has the form

$$\begin{aligned} \min_{\mathbf{x}_{0:N}, \mathbf{u}_{0:N-1}} \quad & \|\mathbf{x}_N - \mathbf{x}_N^{\text{ref}}\|_{\mathbf{P}}^2 + \sum_{k=0}^{N-1} \left(\|\mathbf{x}_k - \mathbf{x}_k^{\text{ref}}\|_{\mathbf{Q}_k}^2 + \|\mathbf{u}_k - \mathbf{u}_k^{\text{ref}}\|_{\mathbf{R}_k}^2 \right) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k + \mathbf{d}_k, \\ & \mathbf{u}_{\min} \le \mathbf{u}_k \le \mathbf{u}_{\max}, \\ & \mathbf{x}_{\min} \le \mathbf{x}_k \le \mathbf{x}_{\max}, \\ & \mathbf{x}_0 = \mathbf{x}_{\text{current}}, \end{aligned} \tag{192}$$

where $\mathbf{d}_k = f(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k) - \mathbf{A}_k \bar{\mathbf{x}}_k - \mathbf{B}_k \bar{\mathbf{u}}_k$ captures the local affine offset.

Because the dynamics are now linear and the cost quadratic, this optimization problem is a convex quadratic program. Quadratic programs are attractive in practice: they can be solved at kilohertz rates with mature numerical methods, making them the backbone of many real-time MPC implementations.

At each MPC step, the controller updates its linearization around the new operating point, constructs the local QP, and solves it. The process repeats, with the linear model and quadratic cost refreshed at every reoptimization. Despite the approximation, this yields a closed-loop controller that inherits the fast computation of QPs while retaining the ability to track trajectories of the underlying nonlinear system.

Theoretical Guarantees The finite-horizon approximation in MPC brings a new challenge: the controller cannot see consequences beyond the horizon. Without proper design, this myopia can destabilize even simple systems. The solution is to carefully encode information about the infinite-horizon problem into the finite-horizon optimization through its terminal conditions.

Before diving into the mathematics, we should first establish what "stability" means and which tasks these theoretical guarantees address, as the notion of stability varies significantly across different control objectives.

Stability Notions Across Control Tasks The terminal conditions provide different types of guarantees depending on the control objective. For regulation problems, where the task is to drive the state to a fixed equilibrium $(\mathbf{x}_{\text{eq}}, \mathbf{u}_{\text{eq}})$ (often shifted to the origin), the stability guarantee is **asymptotic stability**: starting sufficiently close to the equilibrium, we have $\mathbf{x}_k \to \mathbf{x}_{\text{eq}}$ while constraints remain satisfied throughout the trajectory (**recursive feasibility**). This requires the stage cost $\ell(\mathbf{x}, \mathbf{u})$ to be positive definite in the deviation from equilibrium.

When tracking a constant setpoint, the task becomes following a constant reference $(\mathbf{x}_{\text{ref}}, \mathbf{u}_{\text{ref}})$ that solves the steady-state equations. This problem is handled by working in **error coordinates** $\tilde{\mathbf{x}} = \mathbf{x} - \mathbf{x}_{\text{ref}}$ and $\tilde{\mathbf{u}} = \mathbf{u} - \mathbf{u}_{\text{ref}}$, transforming the tracking problem into a regulation problem for the error system. The stability guarantee becomes asymptotic **tracking**, meaning $\tilde{\mathbf{x}}_k \to 0$, again with recursive feasibility.

The terminal conditions we discuss below primarily address regulation and constant reference tracking. Time-varying tracking and economic MPC require additional techniques such as tube MPC and dissipativity theory.

MPC with Stability Guarantees To provide theoretical guarantees, the finite-horizon MPC problem is augmented with three interconnected components. The **terminal cost** $V_f(\mathbf{x})$ approximates the cost-to-go beyond the horizon, providing a surrogate for the infinite-horizon tail that cannot be explicitly optimized. The **terminal constraint set** $\mathcal{X}_f$ defines a region where we have local knowledge of how to stabilize the system. Finally, the **terminal controller** $\kappa_f(\mathbf{x})$ provides a local stabilizing control law that remains valid within $\mathcal{X}_f$.

These components must satisfy specific compatibility conditions to provide theoretical guarantees:

Theorem 3.1 (Recursive Feasibility and Asymptotic Stability)**.** *Consider the MPC problem with terminal cost V_f, terminal set $\mathcal{X}_f$, and local controller κ_f. If the following conditions hold:*

***Control invariance**: For all $\mathbf{x} \in \mathcal{X}_f$, we have $\mathbf{f}(\mathbf{x}, \kappa_f(\mathbf{x})) \in \mathcal{X}_f$ (the set is invariant) and $\mathbf{g}(\mathbf{x}, \kappa_f(\mathbf{x})) \leq \mathbf{0}$ (constraints remain satisfied).*

***Lyapunov decrease**: For all $\mathbf{x} \in \mathcal{X}_f$:*

$$V_f(\mathbf{f}(\mathbf{x}, \kappa_f(\mathbf{x}))) - V_f(\mathbf{x}) \leq -\ell(\mathbf{x}, \kappa_f(\mathbf{x})) \tag{193}$$

where ℓ is the stage cost.

Then the MPC controller achieves recursive feasibility (if the problem is feasible at time k, it remains feasible at time $k+1$), asymptotic stability to the target equilibrium for regulation problems, and monotonic cost decrease along trajectories until the target is reached.

Suboptimality Bounds The finite-horizon MPC value $V_N(\mathbf{x})$ provides an upper bound approximation of the true infinite-horizon value $V_\infty(\mathbf{x})$. Un-

derstanding how close this approximation can be tells us about the effectiveness of short-horizon MPC.

The upper bound $V_N(\mathbf{x}) \geq V_\infty(\mathbf{x})$ follows immediately from the fact that MPC considers fewer control choices. The infinite-horizon controller can choose any sequence $(\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \ldots)$, while the N-horizon controller is restricted to sequences of the form $(\mathbf{u}_0, \ldots, \mathbf{u}_{N-1}, \kappa_f(\mathbf{x}_N), \kappa_f(\mathbf{x}_{N+1}), \ldots)$ where the tail follows the fixed terminal controller. Since the infinite-horizon problem optimizes over a larger feasible set, its optimal value cannot exceed that of the finite-horizon problem.

Deriving the Approximation Error The interesting question is bounding the approximation error $\varepsilon_N = V_N(\mathbf{x}) - V_\infty(\mathbf{x})$. This error represents the cost of being forced to use κ_f beyond the horizon rather than continuing to optimize.

Let $(\mathbf{u}_0^*, \mathbf{u}_1^*, \ldots)$ denote the infinite-horizon optimal control sequence with corresponding state trajectory $(\mathbf{x}_0^*, \mathbf{x}_1^*, \ldots)$ where $\mathbf{x}_0^* = \mathbf{x}$. The infinite-horizon cost is:

$$V_\infty(\mathbf{x}) = \sum_{k=0}^{\infty} \ell(\mathbf{x}_k^*, \mathbf{u}_k^*) \tag{194}$$

Now consider what happens when we truncate this optimal sequence at horizon N and continue with the terminal controller. The cost becomes:

$$\tilde{V}_N(\mathbf{x}) = \sum_{k=0}^{N-1} \ell(\mathbf{x}_k^*, \mathbf{u}_k^*) + V_f(\mathbf{x}_N^*) \tag{195}$$

where $V_f(\mathbf{x}_N^*)$ approximates the tail cost $\sum_{k=N}^{\infty} \ell(\mathbf{x}_k^*, \mathbf{u}_k^*)$.

Since $V_N(\mathbf{x})$ is the optimal N-horizon cost (which may do better than this particular truncated sequence), we have $V_N(\mathbf{x}) \leq \tilde{V}_N(\mathbf{x})$. The approximation error therefore satisfies:

$$\varepsilon_N \leq \tilde{V}_N(\mathbf{x}) - V_\infty(\mathbf{x}) = V_f(\mathbf{x}_N^*) - \sum_{k=N}^{\infty} \ell(\mathbf{x}_k^*, \mathbf{u}_k^*) \tag{196}$$

This bound shows that the approximation error depends on how well the terminal cost V_f approximates the true tail cost along the infinite-horizon optimal trajectory.

3.1.1 The Landscape of MPC Variants

Once the basic idea of receding-horizon control is clear, it is helpful to see how the same backbone accommodates many variations. In every case, we transcribe the continuous-time optimal control problem into a nonlinear program of the form

$$
\begin{aligned}
\text{minimize} \quad & c(\mathbf{x}_N) + \sum_{k=0}^{N-1} w_k \, c(\mathbf{x}_k, \mathbf{u}_k) \\
\text{subject to} \quad & \mathbf{x}_{k+1} = \mathbf{F}_k(\mathbf{x}_k, \mathbf{u}_k) \\
& \mathbf{g}(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0} \\
& \mathbf{x}_{\min} \leq \mathbf{x}_k \leq \mathbf{x}_{\max} \\
& \mathbf{u}_{\min} \leq \mathbf{u}_k \leq \mathbf{u}_{\max} \\
\text{given} \quad & \mathbf{x}_0 = \hat{\mathbf{x}}(t) \ .
\end{aligned}
\tag{197}
$$

The components in this NLP come from discretizing the continuous-time problem with a fixed horizon $[t, t+T]$ and step size Δt. The stage weights w_k and discrete dynamics $\mathbf{F}_k$ are determined by the choice of quadrature and integration scheme. With this blueprint in place, the rest is a matter of interpretation: how we define the cost, how we handle uncertainty, how we treat constraints, and what structure we exploit.

Tracking MPC The most common setup is reference tracking. Here, we are given time-varying target trajectories $(\mathbf{x}_k^{\text{ref}}, \mathbf{u}_k^{\text{ref}})$, and the controller's job is to keep the system close to these. The cost is typically quadratic:

$$
\begin{aligned}
c(\mathbf{x}_k, \mathbf{u}_k) &= \|\mathbf{x}_k - \mathbf{x}_k^{\text{ref}}\|_{\mathbf{Q}}^2 + \|\mathbf{u}_k - \mathbf{u}_k^{\text{ref}}\|_{\mathbf{R}}^2 \\
c(\mathbf{x}_N) &= \|\mathbf{x}_N - \mathbf{x}_N^{\text{ref}}\|_{\mathbf{P}}^2 \ .
\end{aligned}
\tag{198}
$$

When dynamics are linear and constraints are polyhedral, this yields a convex quadratic program at each time step.

Regulatory MPC In regulation tasks, we aim to bring the system back to an equilibrium point $(\mathbf{x}^e, \mathbf{u}^e)$, typically in the presence of disturbances. This is simply tracking MPC with constant references:

$$
\begin{aligned}
c(\mathbf{x}_k, \mathbf{u}_k) &= \|\mathbf{x}_k - \mathbf{x}^e\|_{\mathbf{Q}}^2 + \|\mathbf{u}_k - \mathbf{u}^e\|_{\mathbf{R}}^2 \\
c(\mathbf{x}_N) &= \|\mathbf{x}_N - \mathbf{x}^e\|_{\mathbf{P}}^2 \ .
\end{aligned}
\tag{199}
$$

To guarantee stability, it is common to include a terminal constraint $\mathbf{x}_N \in \mathcal{X}_f$, where $\mathcal{X}_f$ is a control-invariant set under a known feedback law.

Economic MPC Not all systems operate around a reference. Sometimes the goal is to optimize a true economic objective (eg. energy cost, revenue, efficiency) directly. This gives rise to **economic MPC**, where the cost functions reflect real operational performance:

$$
c(\mathbf{x}_k, \mathbf{u}_k) = c_{\text{op}}(\mathbf{x}_k, \mathbf{u}_k), \qquad c(\mathbf{x}_N) = c_{\text{op},T}(\mathbf{x}_N) \ . \tag{200}
$$

There is no reference trajectory here. The optimal behavior emerges from the cost itself. In this setting, standard stability arguments no longer apply

automatically, and one must be careful to add terminal penalties or constraints that ensure the closed-loop system remains well-behaved.

Robust MPC Some systems are exposed to external disturbances or small errors in the model. In those cases, we want the controller to make decisions that will still work no matter what happens, as long as the disturbances stay within some known bounds. This is the idea behind **robust MPC**.

Instead of planning a single trajectory, the controller plans a "nominal" path (what would happen in the absence of any disturbance) and then adds a feedback correction to react to whatever disturbances actually occur. This looks like:

$$\mathbf{u}_k = \bar{\mathbf{u}}_k + \mathbf{K}(\mathbf{x}_k - \bar{\mathbf{x}}_k) \ , \tag{201}$$

where $\bar{\mathbf{u}}_k$ is the planned input and $\mathbf{K}$ is a feedback gain that pulls the system back toward the nominal path if it deviates.

Because we know the worst-case size of the disturbance, we can estimate how far the real state might drift from the plan, and "shrink" the constraints accordingly. The result is that the nominal plan is kept safely away from constraint boundaries, so even if the system gets pushed around, it stays inside limits. This is often called **tube MPC** because the true trajectory stays inside a tube around the nominal one.

The main benefit is that we can handle uncertainty without solving a complicated worst-case optimization at every time step. All the uncertainty is accounted for in the design of the feedback $\mathbf{K}$ and the tightened constraints.

Stochastic MPC If disturbances are random rather than adversarial, a natural goal is to optimize expected cost while enforcing constraints probabilistically. This gives rise to **stochastic MPC**, in which:

- The cost becomes an expectation:

$$E\left[c(\mathbf{x}_N) + \sum_{k=0}^{N-1} w_k \, c(\mathbf{x}_k, \mathbf{u}_k)\right] \tag{202}$$

- Constraints are allowed to be violated with small probability:

$$P[\mathbf{g}(\mathbf{x}_k, \mathbf{u}_k) \leq \mathbf{0}] \geq 1 - \varepsilon \tag{203}$$

In practice, expectations are approximated using a finite set of disturbance scenarios drawn ahead of time. For each scenario, the system dynamics are simulated forward using the same control inputs $\mathbf{u}_k$, which are shared across all scenarios to respect non-anticipativity. The result is a single deterministic optimization problem with multiple parallel copies of the dynamics, one per sampled future. This retains the standard MPC structure, with only moderate growth in problem size.

Despite appearances, this is not dynamic programming. There is no value function or tree of all possible paths. There is only a finite set of futures chosen a priori, and optimized over directly. This scenario-based approach is common in energy systems such as hydro scheduling, where inflows are uncertain but sample trajectories can be generated from forecasts.

Risk constraints are typically enforced across all scenarios or encoded using risk measures like CVaR. For example, one might penalize violations that occur in the worst $(1-\alpha)\%$ of samples, while still optimizing expected performance overall.

Hybrid and Mixed-Integer MPC When systems involve discrete switches (eg. on/off valves, mode selection, or combinatorial logic) the MPC problem must include integer or binary variables. These show up in constraints like

$$\boldsymbol{\delta}_k \in \{0,1\}^m, \qquad \mathbf{u}_k \in \mathcal{U}(\boldsymbol{\delta}_k) \tag{204}$$

along with mode-dependent dynamics and costs. The resulting formulation is a **mixed-integer nonlinear program** (MINLP). The receding-horizon idea is the same, but each solve is more expensive due to the combinatorial nature of the decision space.

Distributed and Decentralized MPC Large-scale systems often consist of interacting subsystems. Distributed MPC decomposes the global NLP into smaller ones that run in parallel, with coordination constraints enforcing consistency across shared variables:

$$\sum_i \mathbf{H}^i \mathbf{z}_k^i = \mathbf{0} \qquad \text{(coupling constraint)} \tag{205}$$

Each subsystem solves a local problem over its own state and input variables, then exchanges information with neighbors. Coordination can be done via primal–dual methods, ADMM, or consensus schemes, but each local block looks like a standard MPC problem.

Adaptive and Learning-Based MPC In practice, we may not know the true model $\mathbf{F}_k$ or cost function c precisely. In **adaptive MPC**, these are updated online from data:

$$\mathbf{x}_{k+1} = \mathbf{F}_k(\mathbf{x}_k, \mathbf{u}_k; \boldsymbol{\theta}_t), \qquad c(\mathbf{x}_k, \mathbf{u}_k) = c(\mathbf{x}_k, \mathbf{u}_k; \boldsymbol{\phi}_t) \tag{206}$$

The parameters $\boldsymbol{\theta}_t$ and $\boldsymbol{\phi}_t$ are learned in real time. When combined with policy distillation, value approximation, or trajectory imitation, this leads to overlaps with reinforcement learning where the MPC solutions act as supervision for a reactive policy.

3.1.2 Robustness in Real-Time MPC

The trajectory optimization methods we have studied assume perfect models and deterministic dynamics. In practice, however, MPC controllers must operate in environments where models are approximate, disturbances are unpredictable, and computational resources are limited. The mathematical elegance of optimal control must always yield to the engineering reality of robust operation as **perfect optimality is less important than reliable operation**. This philosophy permeates industrial MPC applications. A controller that achieves 95% performance 100% of the time is superior to one that achieves 100% performance 95% of the time and fails catastrophically the remaining 5%. Airlines accept suboptimal fuel consumption over missed approaches, power grids tolerate efficiency losses to prevent blackouts, and chemical plants sacrifice yield for safety. By designing for failure, we want to to create MPC systems that degrade gracefully rather than fail catastrophically, maintaining safety and stability even when the impossible is asked of them.

Example: Wind Farm Yield Optimization Consider a wind farm where MPC controllers coordinate individual turbines to maximize overall power production while minimizing wake interference. Each turbine can adjust both its thrust coefficient (through blade pitch) and yaw angle to redirect its wake away from downstream turbines. At time t_k, the MPC controller solves the optimization problem:

$$\begin{aligned} \min_{\mathbf{u}_{0:N-1}} \quad & \sum_{i=0}^{N-1} \|\mathbf{x}_i - \mathbf{x}_i^{\text{ref}}\|_{\mathbf{Q}}^2 + \|\mathbf{u}_i\|_{\mathbf{R}}^2 \\ \text{s.t.} \quad & \mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) \\ & \mathbf{x}_i \in \mathcal{X}_{\text{safe}} \\ & \|\mathbf{u}_i\|_\infty \leq u_{\max} \\ & \mathbf{x}_0 = \mathbf{x}_{\text{current}} \end{aligned} \tag{207}$$

Now suppose an unexpected wind direction change occurs, shifting the incoming wind vector by 30 degrees. The current state $\mathbf{x}_{\text{current}}$ reflects wake patterns that no longer align with the new wind direction, and the optimizer discovers that no feasible trajectory exists that can redirect all wakes appropriately within the physical limits of yaw rate and thrust adjustment. The solver reports infeasibility.

This scenario reveals the fundamental challenge of real-time MPC: **constraint incompatibility**. When disturbances push the system into states from which recovery appears impossible, or when reference trajectories demand physically impossible maneuvers, the intersection of all constraint sets becomes empty. Model mismatch compounds this problem as prediction errors accumulate over the horizon.

Even when feasible solutions exist, **computational constraints** can prevent their discovery. A control loop running at 100 Hz allows only 10 millisec-

onds per iteration. If the solver requires 15 milliseconds to converge, we face an impossible choice: delay the control action and risk destabilizing the system, or apply an unconverged iterate that may violate critical constraints.

A third failure mode involves **numerical instabilities**: ill-conditioned matrices, rank deficiency, or division by zero in the linear algebra routines. These failures are particularly problematic because they occur sporadically, triggered by specific state configurations that create near-singular conditions in the optimization problem.

Softening Constraints Through Slack Variables The first approach to handling infeasibility recognizes that not all constraints carry equal importance. A chemical reactor's temperature must never exceed the runaway threshold: this is a hard constraint that cannot be violated. However, maintaining temperature within an optimal efficiency band is merely desirable. This can be treated as a soft constraint that we prefer to satisfy but can relax when necessary.

This hierarchy motivates reformulating the optimization problem using **slack variables**:

$$\begin{aligned} \min_{\mathbf{u},\boldsymbol{\epsilon}} \quad & \sum_{i=0}^{N-1} \|\mathbf{x}_i - \mathbf{x}_i^{\text{ref}}\|_{\mathbf{Q}}^2 + \|\mathbf{u}_i\|_{\mathbf{R}}^2 + \boldsymbol{\rho}^T \boldsymbol{\epsilon}_i \\ \text{s.t.} \quad & \mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) \\ & \mathbf{g}_{\text{hard}}(\mathbf{x}_i, \mathbf{u}_i) \leq \mathbf{0} \\ & \mathbf{g}_{\text{soft}}(\mathbf{x}_i, \mathbf{u}_i) \leq \boldsymbol{\epsilon}_i \\ & \boldsymbol{\epsilon}_i \geq \mathbf{0} \end{aligned} \tag{208}$$

The penalty weights $\boldsymbol{\rho}$ encode our priorities. Safety constraints might use $\rho_j = 10^6$, while comfort constraints use $\rho_j = 1$. This reformulated problem is always feasible as long as the hard constraints alone admit a solution. That is: we can always make the slack variables $\boldsymbol{\epsilon}$ sufficiently large to satisfy the soft constraints.

Rather than treating constraints as binary hard/soft categories, we can establish a **constraint hierarchy** that enables graceful degradation:

$$\begin{aligned} \text{Safety:} \quad & T_{\text{reactor}} \leq T_{\text{runaway}} - 10 && \rho = \infty \text{ (hard)} \\ \text{Equipment:} \quad & 0 \leq u_{\text{valve}} \leq 100 && \rho = 10^4 \\ \text{Efficiency:} \quad & T_{\text{optimal}} - 5 \leq T \leq T_{\text{optimal}} + 5 && \rho = 10^2 \\ \text{Comfort:} \quad & |T - T_{\text{setpoint}}| \leq 1 && \rho = 1 \end{aligned} \tag{209}$$

As conditions deteriorate, the controller abandons objectives in reverse priority order, maintaining safety even when optimality becomes impossible.

Feasibility Restoration When even soft constraints prove insufficient (perhaps due to catastrophic solver failure or corrupted problem structure) we need **feasibility restoration** that finds any feasible point regardless of optimality:

$$\begin{aligned} \min_{\mathbf{u},\mathbf{s}} \quad & \|\mathbf{s}\|_1 \\ \text{s.t.} \quad & \mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) + \mathbf{s}_i \\ & \mathbf{x}_{\min} - \mathbf{s}_{x,i} \leq \mathbf{x}_i \leq \mathbf{x}_{\max} + \mathbf{s}_{x,i} \\ & \mathbf{u}_{\min} \leq \mathbf{u}_i \leq \mathbf{u}_{\max} \\ & \mathbf{s} \geq \mathbf{0} \end{aligned} \tag{210}$$

This formulation temporarily relaxes even the dynamics constraints, finding the "least infeasible" solution. It answers the question: if we must violate something, what is the minimal violation required? Once feasibility is restored, we can warm-start the original problem from this point.

Reference Governors Rather than reacting to infeasibility after it occurs, we can prevent it by filtering references through a **reference governor**. Consider an aircraft following waypoints. Instead of passing waypoints directly to the MPC, the governor asks: what is the closest approachable reference from our current state?

$$\mathbf{r}_{\text{filtered}} = \arg\max_{\kappa \in [0,1]} \kappa \quad \text{s.t. MPC}(\mathbf{x}_{\text{current}}, \kappa \mathbf{r}_{\text{desired}} + (1-\kappa)\mathbf{x}_{\text{current}}) \text{ is feasible} \tag{211}$$

The governor performs a line search between the current state (always feasible since staying put requires no action) and the desired reference (potentially infeasible). This guarantees the MPC always receives feasible problems while making maximum progress toward the goal.

For computational efficiency, we can pre-compute the **maximal output admissible set**:

$$\mathcal{O}_\infty = \{\mathbf{r} : \exists \text{ feasible trajectory from } \mathbf{x} \text{ to } \mathbf{r} \text{ respecting all constraints}\} \tag{212}$$

Online, the governor simply projects the desired reference onto $\mathcal{O}_\infty$.

Backup Controllers When MPC fails entirely (due to solver crashes, timeouts, or numerical failures) we need backup controllers that require minimal computation while guaranteeing stability and keeping the system away from dangerous regions.

The standard approach uses a pre-computed **local LQR controller** around the equilibrium:

$$\mathbf{K}_{\text{LQR}}, \mathbf{P} = \text{LQR}(\mathbf{A}, \mathbf{B}, \mathbf{Q}, \mathbf{R}) \tag{213}$$

where $(\mathbf{A}, \mathbf{B})$ are the linearized dynamics at equilibrium. When MPC fails:

$$\mathbf{u}_{\text{backup}} = \begin{cases} \mathbf{K}_{\text{LQR}}(\mathbf{x} - \mathbf{x}_{\text{eq}}) & \text{if } \mathbf{x} \in \mathcal{X}_{\text{LQR}} \\ \mathbf{u}_{\text{safe}} & \text{otherwise} \end{cases} \tag{214}$$

The region $\mathcal{X}_{\text{LQR}} = \{\mathbf{x} : (\mathbf{x} - \mathbf{x}_{\text{eq}})^T \mathbf{P} (\mathbf{x} - \mathbf{x}_{\text{eq}}) \leq \alpha\}$ represents the largest invariant set where LQR is guaranteed to work.

Cascade Architectures Production MPC systems rarely rely on a single solver. Instead, they implement a **cascade of increasingly conservative controllers** that trade optimality for reliability:

```
def get_control(self, x, time_budget):
    """
    Multi -level cascade for robust real -time control
    """
    time_remaining = time_budget

    # Level 1: Full nonlinear MPC
    if time_remaining > 5e -3:  # 5ms minimum
        try:
            u, solve_time = self.solve_nmpc(x, time_remaining)
            if converged:
                return u
        except:
            pass
        time_remaining -= solve_time

    # Level 2: Simplified linear MPC
    if time_remaining > 1e -3:  # 1ms minimum
        try:
            # Linearize around current state
            A, B = self.linearize_dynamics(x)
            u, solve_time = self.solve_lmpc(x, A, B, time_remaining)
            return u
        except:
            pass
        time_remaining -= solve_time

    # Level 3: Explicit MPC lookup
    if time_remaining > 1e -4:  # 0.1ms minimum
        region = self.find_critical_region(x)
        if region is not None:
            return self.explicit_control_law[region](x)

    # Level 4: LQR backup
    if self.in_lqr_region(x):
        return self.K_lqr @ (x - self.x_eq)

    # Level 5: Emergency safe mode
    return self.emergency_stop(x)
```

Each level trades optimality for reliability: Level 1 provides optimal but computationally expensive control, Level 2 offers suboptimal but faster solutions, Level 3 provides pre-computed instant evaluation, Level 4 ensures stabilizing control without tracking, and Level 5 implements safe shutdown.

Even when using backup controllers, we can maintain solution continuity through **persistent warm-starting**:

$$\mathbf{z}_{\text{warm}}^{(k+1)} = \begin{cases} \text{shift}(\mathbf{z}^{(k)}) & \text{if MPC succeeded at time } k \\ \text{lift}(\mathbf{u}_{\text{backup}}^{(k)}) & \text{if backup controller used} \\ \text{propagate}(\mathbf{z}_{\text{warm}}^{(k)}) & \text{if maintaining virtual solution} \end{cases} \tag{215}$$

The **shift** operation takes a successful MPC solution and moves it forward by one time step, appending a terminal action: $[\mathbf{u}_1^{(k)}, \mathbf{u}_2^{(k)}, \ldots, \mathbf{u}_{N-1}^{(k)}, \kappa_f(\mathbf{x}_N^{(k)})]$. This shifted sequence provides natural temporal continuity for the next optimization.

When MPC fails and backup control is applied, the **lift** operation extends the single backup action $\mathbf{u}_{\text{backup}}^{(k)}$ into a full horizon-length sequence, either by repetition or by simulating the backup controller forward. This creates a reasonable warm-start guess from limited information.

The **propagate** operation maintains a "virtual" trajectory by continuing to evolve the previous solution as if it were still being executed, even when the actual system follows backup control. This forward simulation keeps the warm-start temporally aligned and relevant for when MPC recovers.

Example: Chemical Reactor Control Under Failure Consider a continuous stirred tank reactor (CSTR) where an exothermic reaction must be controlled:

$$\begin{aligned} \dot{C}_A &= \frac{q}{V}(C_{A,in} - C_A) - k_0 e^{-E/RT} C_A \\ \dot{T} &= \frac{q}{V}(T_{in} - T) + \frac{\Delta H}{\rho c_p} k_0 e^{-E/RT} C_A - \frac{UA}{\rho c_p V}(T - T_c) \end{aligned} \tag{216}$$

The MPC must maintain temperature below the runaway threshold T_{runaway} while maximizing conversion. Under normal operation, it solves:

$$\begin{aligned} \min \quad & -C_A(t_f) + \int_0^{t_f} \|T - T_{\text{optimal}}\|^2 dt \\ \text{s.t.} \quad & T \le T_{\text{runaway}} - \Delta T_{\text{safety}} \\ & q_{\min} \le q \le q_{\max} \end{aligned} \tag{217}$$

When the cooling system partially fails, T_c suddenly increases. The MPC cannot maintain T_{optimal} within safety limits. The cascade activates: soft constraints allow T to exceed T_{optimal} with penalty, the reference governor reduces the production target $C_{A,\text{target}}$, and if still infeasible, the backup controller switches to maximum cooling $q = q_{\max}$. If temperature approaches runaway, emergency shutdown stops the feed with $q = 0$.

3.1.3 Computational Efficiency via Parametric Programming

Real-time model predictive control places strict limits on computation. In applications such as adaptive optics, the controller must run at kilohertz rates. A sampling frequency of 1000 Hz allows only one millisecond per step to compute and apply a control input. This makes efficiency a first-class concern.

The structure of MPC lends itself naturally to optimization reuse. Each time step requires solving a problem with the same dynamics and constraints. Only the initial state, forecasts, or reference signals change. Instead of treating each instance as a new problem, we can frame MPC as a *parametric optimization problem* and focus on how the solution evolves with the parameter.

General Framework: Parametric Optimization We begin with a general optimization problem indexed by a parameter $\boldsymbol{\theta} \in \Theta \subset R^p$:

$$\begin{aligned} \min_{\mathbf{x} \in R^n} \quad & f(\mathbf{x}; \boldsymbol{\theta}) \\ \text{s.t.} \quad & \mathbf{g}(\mathbf{x}; \boldsymbol{\theta}) \leq \mathbf{0}, \\ & \mathbf{h}(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{0}. \end{aligned} \tag{218}$$

For each value of $\boldsymbol{\theta}$, we obtain a concrete optimization problem. The goal is to understand how the optimizer $\mathbf{x}^\star(\boldsymbol{\theta})$ and value function

$$v(\boldsymbol{\theta}) := \inf\{\, f(\mathbf{x}; \boldsymbol{\theta}) : \mathbf{x} \text{ feasible at } \boldsymbol{\theta} \,\} \tag{219}$$

depend on $\boldsymbol{\theta}$.

When the problem is smooth and regular, the Karush–Kuhn–Tucker (KKT) conditions characterize optimality:

$$\begin{aligned} \nabla_{\mathbf{x}} f(\mathbf{x}; \boldsymbol{\theta}) + \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}; \boldsymbol{\theta})^\top \boldsymbol{\lambda} + \nabla_{\mathbf{x}} \mathbf{h}(\mathbf{x}; \boldsymbol{\theta})^\top \boldsymbol{\nu} = 0, \\ \mathbf{g}(\mathbf{x}; \boldsymbol{\theta}) \leq 0, \quad \boldsymbol{\lambda} \geq 0, \quad \lambda_i g_i(\mathbf{x}; \boldsymbol{\theta}) = 0, \\ \mathbf{h}(\mathbf{x}; \boldsymbol{\theta}) = 0. \end{aligned} \tag{220}$$

If the active set remains fixed over changes in $\boldsymbol{\theta}$, the implicit function theorem ensures that the mappings

$$\boldsymbol{\theta} \mapsto \mathbf{x}^\star(\boldsymbol{\theta}), \quad \boldsymbol{\theta} \mapsto \boldsymbol{\lambda}^\star(\boldsymbol{\theta}), \quad \boldsymbol{\theta} \mapsto \boldsymbol{\nu}^\star(\boldsymbol{\theta}) \tag{221}$$

are differentiable.

In linear and quadratic programming, this structure becomes even more tractable. Consider a linear program with affine dependence on $\boldsymbol{\theta}$:

$$\min_{\mathbf{x}} \ \mathbf{c}(\boldsymbol{\theta})^\top \mathbf{x} \quad \text{s.t.} \quad \mathbf{A}(\boldsymbol{\theta}) \mathbf{x} \leq \mathbf{b}(\boldsymbol{\theta}). \tag{222}$$

Each active set determines a basis and thus a region in Θ where the solution is affine in $\boldsymbol{\theta}$. The feasible parameter space is partitioned into polyhedral regions, each with its own affine law.

Similarly, in strictly convex quadratic programs

$$\min_{\mathbf{x}} \ \tfrac{1}{2}\mathbf{x}^\top \mathbf{H}\mathbf{x} + \mathbf{q}(\boldsymbol{\theta})^\top \mathbf{x} \quad \text{s.t.} \quad \mathbf{A}\mathbf{x} \le \mathbf{b}(\boldsymbol{\theta}), \qquad \mathbf{H} \succ 0, \tag{223}$$

each active set again leads to an affine optimizer, with piecewise-affine global structure and a piecewise-quadratic value function.

Parametric programming focuses on the structure of the map $\boldsymbol{\theta} \mapsto \mathbf{x}^\star(\boldsymbol{\theta})$, and the regions over which this map takes a simple form.

Solution Sensitivity via the Implicit Function Theorem We often meet equations of the form

$$F(y, \boldsymbol{\theta}) = 0, \tag{224}$$

where $y \in R^m$ are unknowns and $\boldsymbol{\theta} \in R^p$ are parameters. The **implicit function theorem** says that, if F is smooth and the Jacobian with respect to y,

$$\frac{\partial F}{\partial y}(y^\star, \boldsymbol{\theta}^\star), \tag{225}$$

is invertible at a solution $(y^\star, \boldsymbol{\theta}^\star)$, then in a neighborhood of $\boldsymbol{\theta}^\star$ there exists a unique smooth mapping $y(\boldsymbol{\theta})$ with $F(y(\boldsymbol{\theta}), \boldsymbol{\theta}) = 0$ and $y(\boldsymbol{\theta}^\star) = y^\star$. Moreover, its derivative is

$$\frac{dy}{d\boldsymbol{\theta}}(\boldsymbol{\theta}^\star) \ = \ -\Big(\tfrac{\partial F}{\partial y}(y^\star, \boldsymbol{\theta}^\star)\Big)^{-1} \ \tfrac{\partial F}{\partial \boldsymbol{\theta}}(y^\star, \boldsymbol{\theta}^\star). \tag{226}$$

In words: if the square Jacobian in y is nonsingular, the solution varies smoothly with the parameter, and we can differentiate it by solving one linear system.

Return to (P_θ) and its KKT system. Collect the primal and dual variables into

$$y \ := \ (\mathbf{x},\, \boldsymbol{\lambda},\, \boldsymbol{\nu}), \tag{227}$$

and write the KKT equations as a single residual

$$F(y, \boldsymbol{\theta}) \ = \ \begin{bmatrix} \nabla_{\mathbf{x}} f(\mathbf{x}; \boldsymbol{\theta}) + \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}; \boldsymbol{\theta})^\top \boldsymbol{\lambda} + \nabla_{\mathbf{x}} \mathbf{h}(\mathbf{x}; \boldsymbol{\theta})^\top \boldsymbol{\nu} \\ \mathbf{h}(\mathbf{x}; \boldsymbol{\theta}) \\ \mathbf{g}_{\mathcal{A}}(\mathbf{x}; \boldsymbol{\theta}) \end{bmatrix} \ = \ \mathbf{0}. \tag{228}$$

Here $\mathcal{A}$ denotes the set of inequality constraints active at the solution (the complementarity part is encoded by keeping $\mathcal{A}$ fixed; see below).

To invoke IFT, we need the Jacobian $\partial F / \partial y$ to be invertible at $(y^\star, \boldsymbol{\theta}^\star)$. Standard regularity conditions that ensure this are:

- **LICQ (Linear Independence Constraint Qualification)** at $(\mathbf{x}^\star, \boldsymbol{\theta}^\star)$: the gradients of all active constraints are linearly independent.

- **Second-order sufficiency** on the critical cone (the Lagrangian Hessian is positive definite on feasible directions).
- **Strict complementarity** (optional but convenient): each active inequality has strictly positive multiplier.

Under these, the **KKT matrix**,

$$K \;=\; \frac{\partial F}{\partial y}(y^\star,\boldsymbol{\theta}^\star) \;=\; \begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L}(\mathbf{x}^\star,\boldsymbol{\lambda}^\star,\boldsymbol{\nu}^\star;\boldsymbol{\theta}^\star) & \nabla_{\mathbf{x}}\mathbf{g}_{\mathcal{A}}(\mathbf{x}^\star;\boldsymbol{\theta}^\star)^\top & \nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x}^\star;\boldsymbol{\theta}^\star)^\top \\ \nabla_{\mathbf{x}}\mathbf{g}_{\mathcal{A}}(\mathbf{x}^\star;\boldsymbol{\theta}^\star) & 0 & 0 \\ \nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x}^\star;\boldsymbol{\theta}^\star) & 0 & 0 \end{bmatrix}, \tag{229}$$

is nonsingular. Here $\mathcal{L} = f + \boldsymbol{\lambda}^\top \mathbf{g} + \boldsymbol{\nu}^\top \mathbf{h}$.

The right-hand side sensitivity to parameters is

$$G \;=\; \frac{\partial F}{\partial \boldsymbol{\theta}}(y^\star,\boldsymbol{\theta}^\star) \;=\; \begin{bmatrix} \nabla_{\boldsymbol{\theta}}\nabla_{\mathbf{x}} f + \sum_{i\in\mathcal{A}} \lambda_i^\star \nabla_{\boldsymbol{\theta}}\nabla_{\mathbf{x}} g_i + \sum_j \nu_j^\star \nabla_{\boldsymbol{\theta}}\nabla_{\mathbf{x}} h_j \\ \nabla_{\boldsymbol{\theta}}\mathbf{h} \\ \nabla_{\boldsymbol{\theta}}\mathbf{g}_{\mathcal{A}} \end{bmatrix}_{(\mathbf{x}^\star,\boldsymbol{\theta}^\star)}. \tag{230}$$

IFT then gives **local differentiability of the optimizer and multipliers**:

$$\frac{dy^\star}{d\boldsymbol{\theta}}(\boldsymbol{\theta}^\star) \;=\; -K^{-1}G. \tag{231}$$

The formula above is valid **as long as the active set $\mathcal{A}$ does not change**. If a constraint switches between active/inactive, the mapping remains piecewise smooth, but the derivative may jump. In MPC, this is exactly why warm-starts are very effective most of the time and occasionally require a refactorization when the active set flips.

In parametric MPC, $\boldsymbol{\theta}$ gathers the current state, references, and forecasts. The IFT tells us that, under regularity and a stable active set, the optimal trajectory and first input vary smoothly with $\boldsymbol{\theta}$. The linear map $-K^{-1}G$ is exactly the object used in sensitivity-based warm starts and real-time iterations: small changes in $\boldsymbol{\theta}$ can be propagated through a single KKT solve to update the primal–dual guess before taking one or two Newton/SQP steps.

Predictor-Corrector MPC We start with a smooth root-finding problem

$$F(y) = 0, \qquad F : R^m \to R^m. \tag{232}$$

Newton's method iterates

$$y^{(t+1)} \;=\; y^{(t)} - \big[\nabla F(y^{(t)})\big]^{-1} F\big(y^{(t)}\big), \tag{233}$$

or equivalently solves the linearized system

$$\nabla F(y^{(t)})\,\Delta y^{(t)} = -F\big(y^{(t)}\big), \qquad y^{(t+1)} = y^{(t)} + \Delta y^{(t)}. \tag{234}$$

Convergence is local and fast when the Jacobian is nonsingular and the initial guess is close.

Now suppose the root depends on a parameter:

$$F\big(y, \theta\big) = 0, \qquad \theta \in R. \tag{235}$$

We want the solution path $\theta \mapsto y^\star(\theta)$. **Numerical continuation** advances θ in small steps and uses the previous solution as a warm start for the next Newton solve. This is the simplest and most effective way to "track" solutions of parametric systems.

At a known solution $(y^\star, \theta^\star)$, differentiate $F(y^\star(\theta), \theta) = 0$ with respect to θ:

$$\nabla_y F(y^\star, \theta^\star)\,\frac{dy^\star}{d\theta}(\theta^\star) \;+\; \nabla_\theta F(y^\star, \theta^\star) \;=\; 0. \tag{236}$$

If $\nabla_y F$ is invertible (IFT conditions), the **tangent** is

$$\frac{dy^\star}{d\theta}(\theta^\star) \;=\; -\big[\nabla_y F(y^\star, \theta^\star)\big]^{-1}\,\nabla_\theta F(y^\star, \theta^\star). \tag{237}$$

This is exactly the **implicit differentiation** formula. Continuation uses it as a **predictor**:

$$y_{\text{pred}} \;=\; y^\star(\theta^\star) \;+\; \Delta\theta\,\frac{dy^\star}{d\theta}(\theta^\star). \tag{238}$$

Then a few **corrector** steps apply Newton to $F(\,\cdot\,, \theta^\star + \Delta\theta) = 0$ starting from y_{pred}. If Newton converges quickly, the step $\Delta\theta$ was appropriate; otherwise reduce $\Delta\theta$ and retry.

For parametric KKT systems, set $y = (\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu})$ where $\mathbf{x}$ stacks the primal decision variables (states and inputs), and $F(y, \theta) = 0$ the KKT residual with θ collecting state, references, forecasts. The **KKT matrix** $K = \partial F/\partial y$ and **parameter sensitivity** $G = \partial F/\partial\theta$ give the tangent

$$\frac{dy^\star}{d\theta} \;=\; -\,K^{-1}G. \tag{239}$$

Continuation then becomes:

1. **Predictor**: $y_{\text{pred}} = y^\star + (\Delta\theta)\,(-K^{-1}G)$.
2. **Corrector**: a few Newton/SQP steps on the KKT equations at the new θ.

In MPC, this yields efficient **warm starts** across time: as the parameter θ_t (current state, references) changes slightly, we predict the new primal–dual point and correct with 1–2 iterations—often enough to hit tolerance in real time.

Amortized Optimization and Neural Approximation of Controllers The idea of reusing structure across similar optimization problems is not exclusive to parametric programming. In machine learning, a related concept known as **amortized optimization** aims to reduce the cost of repeated inference by replacing explicit optimization with a function that has been *learned* to approximate the solution map. This approach shifts the computational burden from online solving to offline training.

The goal is to construct a function $\hat{\pi}_\phi(\boldsymbol{\theta})$, typically parameterized by a neural network, that maps the input $\boldsymbol{\theta}$ to an approximate solution $\hat{z}^\star(\boldsymbol{\theta})$ or control action $\hat{\mathbf{u}}_0^\star(\boldsymbol{\theta})$. Once trained, this map can be evaluated quickly at runtime, with no need to solve an optimization problem explicitly.

Amortized optimization has emerged in several contexts:

- In **probabilistic inference**, where variational autoencoders (VAEs) amortize the computation of posterior distributions across a dataset.
- In **meta-learning**, where the objective is to learn a model that generalizes across tasks by internalizing how to adapt.
- In **hyperparameter optimization**, where learning a surrogate model can guide the search over configuration space efficiently.

This perspective has also begun to influence control. Recent work investigates how to **amortize nonlinear MPC (NMPC)** policies into neural networks. The training data come from solving many instances of the underlying optimal control problem offline. The resulting neural policy $\hat{\pi}_\phi$ acts as a differentiable, low-latency controller that can generalize to new situations within the training distribution.

Compared to explicit MPC, which partitions the parameter space and stores exact solutions region by region, amortized control smooths over the domain by learning an approximate policy globally. It is less precise, but scalable to high-dimensional problems where enumeration of regions is impossible.

Neural network amortization is advantageous due to the expressivity of these models. However, the challenge is ensuring **constraint satisfaction and safety**, which are hard to guarantee with unconstrained neural approximators. Hybrid approaches attempt to address this by combining a neural warm-start policy with a final projection step, or by embedding the network within a constrained optimization layer. Other strategies include learning structured architectures that respect known physics or control symmetries.

Imitation Learning Framework Consider a fixed horizon N and parameter vector $\boldsymbol{\theta}$ encoding the current state, references, and forecasts. The oracle MPC controller solves

$$z^\star(\boldsymbol{\theta}) \in \arg \min_{z=(\mathbf{x}_{0:N}, \mathbf{u}_{0:N-1})} J(z;\boldsymbol{\theta}) \\ \text{s.t. } \mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k; \boldsymbol{\theta}), \quad k = 0..N-1, \\ g(\mathbf{x}_k, \mathbf{u}_k; \boldsymbol{\theta}) \leq 0,\ h(\mathbf{x}_N; \boldsymbol{\theta}) = 0. \tag{240}$$

The applied action is $\pi^\star(\boldsymbol{\theta}) := \mathbf{u}_0^\star(\boldsymbol{\theta})$. Our goal is to learn a fast surrogate mapping $\hat{\pi}_\phi : \boldsymbol{\theta} \mapsto \hat{\mathbf{u}}_0 \approx \pi^\star(\boldsymbol{\theta})$ that can be evaluated in microseconds, optionally followed by a safety projection layer.

Supervised learning from oracle solutions. One first samples parameters $\boldsymbol{\theta}^{(i)}$ from the operational domain and solves the corresponding NMPC problems offline. The resulting dataset

$$\mathcal{D} = \{(\boldsymbol{\theta}^{(i)},\, \mathbf{u}_0^\star(\boldsymbol{\theta}^{(i)}))\}_{i=1}^M \tag{241}$$

is then used to train a neural network $\hat{\pi}_\phi$ by minimizing

$$\min_\phi \ \frac{1}{M} \sum_{i=1}^{M} \left\| \hat{\pi}_\phi(\boldsymbol{\theta}^{(i)}) - \mathbf{u}_0^\star(\boldsymbol{\theta}^{(i)}) \right\|^2. \tag{242}$$

Once trained, the network acts as a surrogate for the optimizer, providing instantaneous evaluations that approximate the MPC law.

3.1.4 Example: Propofol Infusion Control

This problem explores the control of propofol infusion in total intravenous anesthesia (TIVA). Our presentation follows the problem formulation developped by Sawaguchi et al. [2008]. The primary objective is to maintain the desired level of unconsciousness while minimizing adverse reactions and ensuring quick recovery after surgery.

The level of unconsciousness is measured by the Bispectral Index (BIS), which is obtained using an electroencephalography (EEG) device. The BIS ranges from 0 (complete suppression of brain activity) to 100 (fully awake), with the target range for general anesthesia typically between 40 and 60.

The goal is to design a control system that regulates the infusion rate of propofol to maintain the BIS within the target range. This can be formulated as an optimal control problem:

$$\min_{u(t)} \int_0^T \left(BIS(t) - BIS_{\text{target}}\right)^2 + \lambda\, u(t)^2\, dt$$

subject to:

$$\begin{aligned}
\dot{x}_1 &= -(k_{10} + k_{12} + k_{13})x_1 + k_{21}x_2 + k_{31}x_3 + \frac{u(t)}{V_1} \\
\dot{x}_2 &= k_{12}x_1 - k_{21}x_2 \\
\dot{x}_3 &= k_{13}x_1 - k_{31}x_3 \\
\dot{x}_e &= k_{e0}(x_1 - x_e) \\
BIS(t) &= E_0 - E_{\max}\frac{x_e^\gamma}{x_e^\gamma + EC_{50}^\gamma}
\end{aligned}$$

Where:

- $u(t)$ is the propofol infusion rate (mg/kg/h)
- x_1, x_2, and x_3 are the drug concentrations in different body compartments
- x_e is the effect-site concentration
- k_{ij} are rate constants for drug transfer between compartments
- $BIS(t)$ is the Bispectral Index
- λ is a regularization parameter penalizing excessive drug use
- E_0, $E_{\max}$, EC_{50}, and γ are parameters of the pharmacodynamic model

The specific dynamics model used in this problem is so-called "Pharmacokinetic-Pharmacodynamic Model" and consists of three main components:

1. **Pharmacokinetic Model**, which describes how the drug distributes through the body over time. It's based on a three-compartment model:
    - Central compartment (blood and well-perfused organs)
    - Shallow peripheral compartment (muscle and other tissues)
    - Deep peripheral compartment (fat)
2. **Effect Site Model**, which represents the delay between drug concentration in the blood and its effect on the brain.
3. **Pharmacodynamic Model** that relates the effect-site concentration to the observed BIS.

The propofol infusion control problem presents several interesting challenges from a research perspective. First, there is a delay in how fast the drug can reach a different compartments in addition to the BIS measurements which can lag. This could lead to instability if not properly addressed in the control design.

Furthermore, every patient is different from another. Hence, we cannot simply learn a single controller offline and hope that it will generalize to an entire patient population. We will account for this variability through Model Predictive Control (MPC) and dynamically adapt to the model mismatch through replanning. How a patient will react to a given dose of drug also varies and must be carefully controlled to avoid overdoses. This adds an additional layer of complexity since we have to incorporate safety constraints. Finally, the patient might suddenly change state, for example due to surgical stimuli, and the controller must be able to adapt quickly to compensate for the disturbance to the system.

3.2 Dynamic Programming

Unlike the methods we've discussed so far, dynamic programming takes a step back and considers an entire family of related problems rather than a single optimization problem. This approach, while seemingly more complex at first glance, can often lead to efficient solutions.

Dynamic programming leverage the solution structure underlying many control problems that allows for a decomposition it into smaller, more manageable subproblems. Each subproblem is itself an optimization problem, embedded within the larger whole. This recursive structure is the foundation upon which dynamic programming constructs its solutions.

To ground our discussion, let us return to the domain of discrete-time optimal control problems (DOCPs). These problems frequently arise from the discretization of continuous-time optimal control problems. While the focus here will be on deterministic problems, these concepts extend naturally to stochastic problems by taking the expectation over the random quantities.

Consider a typical DOCP of Bolza type:

$$\begin{aligned} \text{minimize} \quad & Jc_{\mathrm{T}}(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) \\ \text{subject to} \quad & \mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t), \quad t = 1, \ldots, T-1, \\ & \mathbf{u}_{lb} \leq \mathbf{u}_t \leq \mathbf{u}_{ub}, \quad t = 1, \ldots, T, \\ & \mathbf{x}_{lb} \leq \mathbf{x}_t \leq \mathbf{x}_{ub}, \quad t = 1, \ldots, T, \\ \text{given} \quad & \mathbf{x}_1 \end{aligned}$$

Rather than considering only the total cost from the initial time to the final time, dynamic programming introduces the concept of cost from an arbitrary point in time to the end. This leads to the definition of the "cost-to-go" or "value function" $J_k(\mathbf{x}_k)$:

$$J_k(\mathbf{x}_k) c_{\mathrm{T}}(\mathbf{x}_T) + \sum_{t=k}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) \tag{243}$$

This function represents the total cost incurred from stage k onwards to the end of the time horizon, given that the system is initialized in state $\mathbf{x}_k$ at stage k. Suppose the problem has been solved from stage $k+1$ to the end, yielding the optimal cost-to-go $J^\star_{k+1}(\mathbf{x}_{k+1})$ for any state $\mathbf{x}_{k+1}$ at stage $k+1$. The question then becomes: how does this information inform the decision at stage k?

Given knowledge of the optimal behavior from $k+1$ onwards, the task reduces to determining the optimal action $\mathbf{u}_k$ at stage k. This control should minimize the sum of the immediate cost $c_k(\mathbf{x}_k, \mathbf{u}_k)$ and the optimal future cost $J^\star_{k+1}(\mathbf{x}_{k+1})$, where $\mathbf{x}_{k+1}$ is the resulting state after applying action $\mathbf{u}_k$. Mathematically, this is expressed as:

$$J^\star_k(\mathbf{x}_k) = \min_{\mathbf{u}_k} \left[c_k(\mathbf{x}_k, \mathbf{u}_k) + J^\star_{k+1}(\mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k)) \right] \tag{244}$$

This equation is known as Bellman's equation, named after Richard Bellman, who formulated the principle of optimality:

> An optimal policy has the property that whatever the previous state and decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the previous decision.

In other words, any sub-path of an optimal path, from any intermediate point to the end, must itself be optimal. This principle is the basis for the backward induction procedure which computes the optimal value function and provides closed-loop control capabilities without having to use an explicit NLP solver.

Dynamic programming can handle nonlinear systems and non-quadratic cost functions naturally. It provides a global optimal solution, when one exists, and can incorporate state and control constraints with relative ease. However, as the dimension of the state space increases, this approach suffers from what Bellman termed the "curse of dimensionality." The computational complexity and memory requirements grow exponentially with the state dimension, rendering direct application of dynamic programming intractable for high-dimensional problems.

Fortunately, learning-based methods offer efficient tools to combat the curse of dimensionality on two fronts: by using function approximation (e.g., neural networks) to avoid explicit discretization, and by leveraging randomization through Monte Carlo methods inherent in the learning paradigm. Most of this course is dedicated to those ideas.

Backward Recursion The principle of optimality provides a methodology for solving optimal control problems. Beginning at the final time horizon and working backwards, at each stage the local optimization problem given by Bellman's equation is solved. This process, termed backward recursion or backward induction, constructs the optimal value function stage by stage.

Upon completion of this backward pass, we now have access to the optimal control to take at any stage and in any state. Furthermore, we can simulate optimal trajectories from any initial state and applying the optimal policy at each stage to generate the optimal trajectory.

Theorem 3.2 (Backward induction solves deterministic Bolza DOCP)**.** ***Setting.*** *Let* $\mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t)$ *for* $t = 1, \dots, T-1$*, with admissible action sets* $\mathcal{U}_t(\mathbf{x}) \neq$*. Let stage costs* $c_t(\mathbf{x}, \mathbf{u})$ *and terminal cost* $c_{\mathrm{T}}(\mathbf{x})$ *be real-valued and bounded below. Assume for every* $(t, \mathbf{x})$ *the one-step problem*

$$\min_{\mathbf{u} \in \mathcal{U}_t(\mathbf{x})} \left\{ c_t(\mathbf{x}, \mathbf{u}) + J^\star_{t+1}(\mathbf{f}_t(\mathbf{x}, \mathbf{u})) \right\} \tag{245}$$

admits a minimizer (e.g., compact $\mathcal{U}_t(\mathbf{x})$ *and continuity suffice). Define* $J^\star_T(\mathbf{x}) \equiv c_{\mathrm{T}}(\mathbf{x})$ *and for* $t = T-1, \dots, 1$

$$J^\star_t(\mathbf{x}) \quad \min_{\mathbf{u} \in \mathcal{U}_t(\mathbf{x})} \Big[c_t(\mathbf{x}, \mathbf{u}) + J^\star_{t+1}\big(\mathbf{f}_t(\mathbf{x}, \mathbf{u})\big) \Big], \tag{246}$$

and select any minimizer $\boldsymbol{\mu}_t^\star(\mathbf{x}) \in \arg\min(\cdot)$.

Claim. *For every initial state* $\mathbf{x}_1$*, the control sequence* $\boldsymbol{\mu}_1^\star(\mathbf{x}_1), \dots, \boldsymbol{\mu}_{T-1}^\star(\mathbf{x}_{T-1})$ *generated by these selectors is optimal for the Bolza problem, and* $J_1^\star(\mathbf{x}_1)$ *equals the optimal cost. Moreover,* $J_t^\star(\cdot)$ *is the optimal cost-to-go from stage* t *for every state, i.e., backward induction recovers the entire value function.*

Proof. We give a direct proof by backward induction. The general idea is that any feasible sequence can be improved by replacing its tail with an optimal continuation, so optimal solutions can be built stage by stage. This is sometimes called a "cut-and-paste" argument.

Step 1 (verification of the recursion at a fixed stage).
Fix $t \in \{1, \dots, T-1\}$ and $\mathbf{x} \in X$. Consider any admissible control sequence $\mathbf{u}_t, \dots, \mathbf{u}_{T-1}$ starting from $\mathbf{x}_t = \mathbf{x}$ and define the induced states $\mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k)$. Its total cost from t is

$$c_t(\mathbf{x}_t, \mathbf{u}_t) + \sum_{k=t+1}^{T-1} c_k(\mathbf{x}_k, \mathbf{u}_k) + c_{\mathrm{T}}(\mathbf{x}_T). \tag{247}$$

By definition of $J_{t+1}^\star$, the tail cost satisfies

$$\sum_{k=t+1}^{T-1} c_k(\mathbf{x}_k, \mathbf{u}_k) + c_{\mathrm{T}}(\mathbf{x}_T) \;\geq\; J_{t+1}^\star(\mathbf{x}_{t+1}) \;=\; J_{t+1}^\star\big(\mathbf{f}_t(\mathbf{x}, \mathbf{u}_t)\big). \tag{248}$$

Hence the total cost is bounded below by

$$c_t(\mathbf{x}, \mathbf{u}_t) + J_{t+1}^\star\big(\mathbf{f}_t(\mathbf{x}, \mathbf{u}_t)\big). \tag{249}$$

Taking the minimum over $\mathbf{u}_t \in \mathcal{U}_t(\mathbf{x})$ yields

$$(\text{any admissible cost from } t) \;\geq\; J_t^\star(\mathbf{x}). \tag{$*$}$$

Step 2 (existence of an optimal prefix at stage t).
By assumption, there exists $\boldsymbol{\mu}_t^\star(\mathbf{x})$ attaining the minimum in the definition of $J_t^\star(\mathbf{x})$. If we now **paste** to $\boldsymbol{\mu}_t^\star(\mathbf{x})$ an optimal tail policy from $t+1$ (whose existence we will establish inductively), the resulting sequence attains cost exactly

$$c_t\big(\mathbf{x}, \boldsymbol{\mu}_t^\star(\mathbf{x})\big) + J_{t+1}^\star\Big(\mathbf{f}_t\big(\mathbf{x}, \boldsymbol{\mu}_t^\star(\mathbf{x})\big)\Big) = J_t^\star(\mathbf{x}), \tag{250}$$

which matches the lower bound $(*)$; hence it is optimal from t.

Step 3 (backward induction over time).
Base case $t = T$. The statement holds because $J_T^\star(\mathbf{x}) = c_{\mathrm{T}}(\mathbf{x})$ and there is no control to choose.

Inductive step. Assume the tail statement holds for $t+1$: from any state $\mathbf{x}_{t+1}$ there exists an optimal control sequence realizing $J_{t+1}^\star(\mathbf{x}_{t+1})$. Then by Steps 1–2, selecting $\boldsymbol{\mu}_t^\star(\mathbf{x}_t)$ at stage t and concatenating the optimal tail from $t+1$ yields an optimal sequence from t with value $J_t^\star(\mathbf{x}_t)$.

By backward induction, the claim holds for all t, in particular for $t = 1$ and any initial $\mathbf{x}_1$. Therefore the backward recursion both **certifies** optimality (verification) and **constructs** an optimal policy (synthesis), while recovering the full family $\{J_t^\star\}_{t=1}^T$.

□

Remark 3.1 (No "big NLP" required)**.** *The Bolza DOCP over the whole horizon couples all controls through the dynamics and is typically posed as a single large nonlinear program. The proof shows you can solve $T-1$* ***sequences of one-step problems*** *instead: at each $(t, \mathbf{x})$ minimize*

$$\mathbf{u} \mapsto c_t(\mathbf{x}, \mathbf{u}) + J_{t+1}^\star(\mathbf{f}_t(\mathbf{x}, \mathbf{u})). \tag{251}$$

In finite state–action spaces this becomes pure table lookup and argmin. In continuous spaces you still solve local one-step minimizations, but you avoid a monolithic horizon-coupled NLP.

Remark 3.2 (Graph interpretation (optional intuition))**.** *Unroll time to form a DAG whose nodes are $(t, \mathbf{x})$ and whose edges correspond to feasible controls with edge weight $c_t(\mathbf{x}, \mathbf{u})$. The terminal node cost is $c_{\mathrm{T}}(\cdot)$. The Bolza problem is a shortest-path problem on this DAG. The equation*

$$J_t^\star(\mathbf{x}) = \min_{\mathbf{u}} \{c_t(\mathbf{x}, \mathbf{u}) + J_{t+1}^\star(\mathbf{f}_t(\mathbf{x}, \mathbf{u}))\} \tag{252}$$

is exactly the dynamic programming recursion for shortest paths on acyclic graphs, hence backward induction is optimal.

Example: Optimal Harvest in Resource Management Dynamic programming is often used in resource management and conservation biology to devise policies to be implemented by decision makers and stakeholders : for eg. in fishereries, or timber harvesting. Per Conroy and Peterson [2013], we consider a population of a particular species, whose abundance we denote by x_t, where t represents discrete time steps. Our objective is to maximize the cumulative harvest over a finite time horizon, while also considering the long-term sustainability of the population. This optimization problem can be formulated as:

$$\text{maximize} \quad \sum_{t=t_0}^{t_f} F(x_t \cdot h_t) + F_{\mathrm{T}}(x_{t_f}) \tag{253}$$

Here, $F(\cdot)$ represents the immediate reward function associated with harvesting, h_t is the harvest rate at time t, and $F_{\mathrm{T}}(\cdot)$ denotes a terminal value function that could potentially assign value to the final population state. In this particular problem, we assign no terminal value to the final population state, setting $F_{\mathrm{T}}(x_{t_f}) = 0$ and allowing us to focus solely on the cumulative harvest over the time horizon.

In our model population model, the abundance of a specicy x ranges from 1 to 100 individuals. The decision variable is the harvest rate h, which can take values from the set $D = \{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$. The population dynamics are governed by a modified logistic growth model:

$$x_{t+1} = x_t + 0.3x_t(1 - x_t/125) - h_t x_t \tag{254}$$

where the 0.3 represents the growth rate and 125 is the carrying capacity (the maximum population size given the available resources). The logistic growth model returns continuous values; however our DP formulation uses a discrete state space. Therefore, we also round the the outcomes to the nearest integer.

Applying the principle of optimality, we can express the optimal value function $J^\star(x_t, t)$ recursively:

$$J^\star(x_t, t) = \max_{h_t \in D}(F(x, h, t) + J^*(x_{t+1}, t + 1)) \tag{255}$$

with the boundary condition $J^*(x_{t_f}) = 0$.

It's worth noting that while this example uses a relatively simple model, the same principles can be applied to more complex scenarios involving stochasticity, multiple species interactions, or spatial heterogeneity.

Handling Continuous Spaces with Interpolation In many real-world problems, such as our resource management example, the state space is inherently continuous. Dynamic programming, however, is usually defined on discrete state spaces. To reconcile this, we approximate the value function on a finite grid of points and use interpolation to estimate its value elsewhere.

In our earlier example, we acted as if population sizes could only be whole numbers: 1 fish, 2 fish, 3 fish. But real measurements don't fit neatly. What do you do with a survey that reports 42.7 fish? Our reflex in the code example was to round to the nearest integer, effectively saying "let's just call it 43." This corresponds to **nearest-neighbor interpolation**, also known as discretization. It's the zeroth-order case: you assume the value between grid points is constant and equal to the closest one. In practice, this amounts to overlaying a grid on the continuous landscape and forcing yourself to stand at the intersections. In our demo code, this step was carried out with `numpy.searchsorted`.

While easy to implement, nearest-neighbor interpolation can introduce artifacts:

1. Decisions may change abruptly, even if the state only shifts slightly.
2. Precision is lost, especially in regimes where small variations matter.
3. The curse of dimensionality forces an impractically fine grid if many state variables are added.

To address these issues, we can use **higher-order interpolation**. Instead of taking the nearest neighbor, we estimate the value at off-grid points by leveraging multiple nearby values.

Backward Recursion with Interpolation Suppose we have computed $J^\star_{k+1}(\mathbf{x})$ only at grid points $\mathbf{x} \in \mathcal{X}_{\text{grid}}$. To evaluate Bellman's equation at an arbitrary $\mathbf{x}_{k+1}$, we interpolate. Formally, let $I_{k+1}(\mathbf{x})$ be the interpolation operator that extends the value function from $\mathcal{X}_{\text{grid}}$ to the continuous space. Then:

$$J^\star_k(\mathbf{x}_k) = \min_{\mathbf{u}_k} \Big[c_k(\mathbf{x}_k, \mathbf{u}_k) + I_{k+1}\big(\mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k)\big) \Big]. \tag{256}$$

For instance, in one dimension, linear interpolation gives:

$$I_{k+1}(x) = J^\star_{k+1}(x_l) + \frac{x - x_l}{x_u - x_l}\big(J^\star_{k+1}(x_u) - J^\star_{k+1}(x_l)\big), \tag{257}$$

where x_l and x_u are the nearest grid points bracketing x. Linear interpolation is often sufficient, but higher-order methods (cubic splines, radial basis functions) can yield smoother and more accurate estimates. The choice of interpolation scheme and grid layout both affect accuracy and efficiency. A finer grid improves resolution but increases computational cost, motivating strategies like adaptive grid refinement or replacing interpolation altogether with parametric function approximation which we are going to see later in this book.

In higher-dimensional spaces, naive interpolation becomes prohibitively expensive due to the curse of dimensionality. Several approaches such as tensorized multilinear interpolation, radial basis functions, and machine learning models address this challenge by extending a common principle: they approximate the value function at unobserved points using information from a finite set of evaluations. However, as dimensionality continues to grow, even tensor methods face scalability limits, which is why flexible parametric models like neural networks have become essential tools for high-dimensional function approximation.

Example: Optimal Harvest with Linear Interpolation Here is a demonstration of the backward recursion procedure using linear interpolation.

Due to pedagogical considerations, this example is using our own implementation of the linear interpolation procedure. However, a more general and practical approach would be to use a built-in interpolation procedure in Numpy. Because our state space has a single dimension, we can simply use scipy.interpolate.interp1d which offers various interpolation methods through its `kind` argument, which can take values in 'linear', 'nearest', 'nearest-up', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', or 'next'. 'zero', 'slinear', 'quadratic' and 'cubic'.

Here's a more general implementation which here uses cubic interpolation through the `scipy.interpolate.interp1d` function:

3.2.1 Stochastic Dynamic Programming and Markov Decision Processes

While our previous discussion centered on deterministic systems, many real-world problems involve uncertainty. Stochastic Dynamic Programming (SDP) extends our framework to handle stochasticity in both the objective function and

system dynamics. This extension naturally leads us to consider more general policy classes and to formalize when simpler policies suffice.

Decision Rules and Policies Before diving into stochastic systems, we need to establish terminology for the different types of strategies a decision maker might employ. In the deterministic setting, we implicitly used feedback controllers of the form $u(\mathbf{x}, t)$. In the stochastic setting, we must be more precise about what information policies can use and how they select actions.

A **decision rule** is a prescription for action selection in each state at a specified decision epoch. These rules can vary in their complexity based on two main criteria:

1. **Dependence on history**: Markovian or History-dependent
2. **Action selection method**: Deterministic or Randomized

Markovian decision rules depend only on the current state, while **history-dependent rules** consider the entire sequence of past states and actions. Formally, a history h_t at time t is:

$$h_t = (s_1, a_1, \ldots, s_{t-1}, a_{t-1}, s_t) \tag{258}$$

The set of all possible histories at time t, denoted H_t, grows exponentially with t:

- $H_1 = \mathcal{S}$ (just the initial state)
- $H_2 = \mathcal{S} \times \mathcal{A} \times \mathcal{S}$
- $H_t = \mathcal{S} \times (\mathcal{A} \times \mathcal{S})^{t-1}$

Deterministic rules select an action with certainty, while **randomized rules** specify a probability distribution over the action space.

These classifications lead to four types of decision rules:

1. **Markovian Deterministic (MD)**: $\pi_t : \mathcal{S} \to \mathcal{A}_s$
2. **Markovian Randomized (MR)**: $\pi_t : \mathcal{S} \to \mathcal{P}(\mathcal{A}_s)$
3. **History-dependent Deterministic (HD)**: $\pi_t : H_t \to \mathcal{A}_s$
4. **History-dependent Randomized (HR)**: $\pi_t : H_t \to \mathcal{P}(\mathcal{A}_s)$

where $\mathcal{P}(\mathcal{A}_s)$ denotes the set of probability distributions over $\mathcal{A}_s$.

A **policy** $\boldsymbol{\pi}$ is a sequence of decision rules, one for each decision epoch:

$$\boldsymbol{\pi} = (\pi_1, \pi_2, ..., \pi_{N-1}) \tag{259}$$

The set of all policies of class K (where $K \in \{HR, HD, MR, MD\}$) is denoted as Π^K. These policy classes form a hierarchy:

$$\Pi^{MD} \subset \Pi^{MR} \subset \Pi^{HR}, \quad \Pi^{MD} \subset \Pi^{HD} \subset \Pi^{HR} \tag{260}$$

The largest set Π^{HR} contains all possible policies. We ask: under what conditions can we restrict attention to the much simpler set Π^{MD} without loss of optimality?

Notation: rules vs. policies

- **Decision rule (kernel).** A map from information to action distributions:
  - Markov, deterministic: $\pi_t : \mathcal{S} \to \mathcal{A}_s$
  - Markov, randomized: $\pi_t(\cdot \mid s) \in \Delta(\mathcal{A}_s)$
  - History-dependent: $\pi_t(\cdot \mid h_t) \in \Delta(\mathcal{A}_{s_t})$
- **Policy (sequence).** $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots)$.
- **Stationary policy.** $\boldsymbol{\pi} = \text{const}(\pi)$ with $\pi_t \equiv \pi \ \forall t$.
  By convention, we identify π with its stationary policy $\text{const}(\pi)$ when no confusion arises.

Stochastic System Dynamics In the stochastic setting, our system evolution takes the form:

$$\mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w}_t) \tag{261}$$

Here, $\mathbf{w}_t$ represents a random disturbance or noise term at time t due to the inherent uncertainty in the system's behavior. The stage cost function may also incorporate stochastic influences:

$$c_t(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w}_t) \tag{262}$$

In this context, our objective shifts from minimizing a deterministic cost to minimizing the expected total cost:

$$E\left[c_{\mathrm{T}}(\mathbf{x}_T) + \sum_{t=1}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w}_t)\right] \tag{263}$$

where the expectation is taken over the distributions of the random variables $\mathbf{w}_t$. The principle of optimality still holds in the stochastic case, but Bellman's optimality equation now involves an expectation:

$$J_k^\star(\mathbf{x}_k) = \min_{\mathbf{u}_k} E_{\mathbf{w}_k}\left[c_k(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}_k) + J_{k+1}^\star(\mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}_k))\right] \tag{264}$$

In practice, this expectation is often computed by discretizing the distribution of $\mathbf{w}_k$ when the set of possible disturbances is very large or even continuous. Let's say we approximate the distribution with K discrete values $\mathbf{w}_k^i$, each occurring with probability p_k^i. Then our Bellman equation becomes:

$$J_k^\star(\mathbf{x}_k) = \min_{\mathbf{u}_k} \sum_{i=1}^{K} p_k^i \left(c_k(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}_k^i) + J_{k+1}^\star(\mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}_k^i)) \right) \tag{265}$$

Optimality Equations in the Stochastic Setting When dealing with stochastic systems, a central question arises: what information should our control policy use? In the most general case, a policy might use the entire history of observations and actions. However, as we'll see, the Markovian structure of our problems allows for dramatic simplifications.

Let $h_t = (s_1, a_1, s_2, a_2, \ldots, s_{t-1}, a_{t-1}, s_t)$ denote the complete history up to time t. In the stochastic setting, the history-based optimality equations become:

$$u_t(h_t) = \sup_{a \in A_{s_t}} \left\{ r_t(s_t, a) + \sum_{j \in S} p_t(j \mid s_t, a)\, u_{t+1}(h_t, a, j) \right\}, \quad u_N(h_N) = r_N(s_N) \tag{266}$$

where we now explicitly use the transition probabilities $p_t(j|s_t, a)$ rather than a deterministic dynamics function.

Theorem 3.3 (Principle of optimality for stochastic systems)**.** *Let u_t^* be the optimal expected return from epoch t onward. Then:*

a. *u_t^* satisfies the optimality equations:*

$$u_t^*(h_t) = \sup_{a \in A_{s_t}} \left\{ r_t(s_t, a) + \sum_{j \in S} p_t(j|s_t, a) u_{t+1}^*(h_t, a, j) \right\} \tag{267}$$

with boundary condition $u_N^*(h_N) = r_N(s_N)$.

b. *Any policy π^* that selects actions attaining the supremum (or maximum) in the above equation at each history is optimal.*

Intuition: This formalizes Bellman's principle of optimality: "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." The recursive structure means that optimal local decisions (choosing the best action at each step) lead to global optimality, even with uncertainty captured by the transition probabilities.

A simplification occurs when we examine these history-based equations more closely. The Markov property of our system dynamics and rewards means that the optimal return actually depends on the history only through the current state:

Proposition 3.1 (State sufficiency for stochastic MDPs)**.** *In finite-horizon stochastic MDPs with Markovian dynamics and rewards, the optimal return $u_t^*(h_t)$ depends on the history only through the current state s_t. Thus we can write $u_t^*(h_t) = v_t^*(s_t)$ for some function v_t^* that depends only on state and time.*

Proof. Following Puterman [1994] Theorem 4.4.2. We proceed by backward induction.

Base case: At the terminal time N, we have $u_N^*(h_N) = r_N(s_N)$ by the boundary condition. Since the terminal reward depends only on the final state s_N and not on how we arrived there, $u_N^*(h_N) = u_N^*(s_N)$.

Inductive step: Assume $u_{t+1}^*(h_{t+1})$ depends on h_{t+1} only through s_{t+1} for all $t+1, \ldots, N$. Then from the optimality equation:

$$u_t^*(h_t) = \sup_{a \in A_{s_t}} \left\{ r_t(s_t, a) + \sum_{j \in S} p_t(j|s_t, a) u_{t+1}^*(h_t, a, j) \right\} \tag{268}$$

By the induction hypothesis, $u_{t+1}^*(h_t, a, j)$ depends only on the next state j, so:

$$u_t^*(h_t) = \sup_{a \in A_{s_t}} \left\{ r_t(s_t, a) + \sum_{j \in S} p_t(j|s_t, a) u_{t+1}^*(j) \right\} \tag{269}$$

Since the expression in brackets depends on h_t only through the current state s_t (the rewards and transition probabilities are Markovian), we conclude that $u_t^*(h_t) = u_t^*(s_t)$.

□

Intuition: The Markov property means that the current state contains all information needed to predict future evolution. The past provides no additional value for decision-making. This result allows us to work with value functions $v_t^*(s)$ indexed only by state and time, dramatically simplifying both theory and computation.

This state-sufficiency result, combined with the fact that randomization never helps when maximizing expected returns, leads to a dramatic simplification of the policy space:

Theorem 3.4 (Policy reduction for stochastic MDPs)**.** *For finite-horizon stochastic MDPs with finite state and action sets:*

$$\sup_{\pi \in \Pi^{\text{HR}}} v_\pi(s, t) = \max_{\pi \in \Pi^{\text{MD}}} v_\pi(s, t) \tag{270}$$

That is, there exists an optimal policy that is both deterministic and Markovian.

Proof. Sketch following Puterman [1994] Lemma 4.3.1 and Theorem 4.4.2. First, Lemma 4.3.1 shows that for any function w and any distribution q over actions, $\sup_a w(a) \geq \sum_a q(a) w(a)$. Thus randomization cannot improve the expected

value over choosing a single maximizing action. Second, by state sufficiency (Proposition Proposition 3.1 and Puterman [1994] Thm. 4.4.2(a)), the optimal return depends on the history only through (s_t, t). Therefore, selecting at each (s_t, t) an action that attains the maximum yields a deterministic Markov decision rule which is optimal whenever the maximum is attained. If only a supremum exists, ε-optimal selectors exist by choosing actions within ε of the supremum (see Puterman [1994] Thm. 4.3.4).

□

Intuition: Even in stochastic systems, randomization in the policy doesn't help when maximizing expected returns: you should always choose the action with the highest expected value. Combined with state sufficiency, this means simple state-to-action mappings are optimal.

These results justify focusing on deterministic Markov policies and lead to the backward recursion algorithm for stochastic systems:

While SDP provides us with a framework to for handling uncertainty, it makes the curse of dimensionality even more difficult to handle in practice. Both the state space and the disturbance space must be discretized. This can lead to a combinatorial explosion in the number of scenarios to be evaluated at each stage.

However, just as we tackled the challenges of continuous state spaces with discretization and interpolation, we can devise efficient methods to handle the additional complexity of evaluating expectations. This problem essentially becomes one of numerical integration. When the set of disturbances is continuous (as is often the case with continuous state spaces), we enter a domain where numerical quadrature methods could be applied. But these methods tend to scale poorly as the number of dimensions grows. This is where more efficient techniques, often rooted in Monte Carlo methods, come into play. The combination of two key ingredients emerges to tackle the curse of dimensionality:

1. Function approximation (through discretization, interpolation, neural networks, etc.)
2. Monte Carlo integration (simulation)

These two elements essentially distill the key ingredients of machine learning, which is the direction we'll be exploring in this course.

Example: Stochastic Optimal Harvest in Resource Management Building upon our previous deterministic model, we now introduce stochasticity to more accurately reflect the uncertainties inherent in real-world resource management scenarios [Conroy and Peterson, 2013]. As before, we consider a population of a particular species, whose abundance we denote by x_t, where t represents discrete time steps. Our objective remains to maximize the cumulative harvest over a finite time horizon, while also considering the long-term sustainability of the population. However, we now account for two sources of stochasticity:

partial controllability of harvest and environmental variability affecting growth rates. The optimization problem can be formulated as:

$$\text{maximize} \quad E\left[\sum_{t=t_0}^{t_f} F(x_t \cdot h_t)\right] \tag{271}$$

Here, $F(\cdot)$ represents the immediate reward function associated with harvesting, and h_t is the realized harvest rate at time t. The expectation $E[\cdot]$ over both harvest and growth rates, which we view as random variables. In our stochastic model, the abundance x still ranges from 1 to 100 individuals. The decision variable is now the desired harvest rate d_t, which can take values from the set $D = 0, 0.1, 0.2, 0.3, 0.4, 0.5$. However, the realized harvest rate h_t is stochastic and follows a discrete distribution:

$$h_t = \begin{cases} 0.75d_t & \text{with probability } 0.25 \\ d_t & \text{with probability } 0.5 \\ 1.25d_t & \text{with probability } 0.25 \end{cases} \tag{272}$$

By expressing the harvest rate as a random variable, we mean to capture the fact that harvesting is a not completely under our control: we might obtain more or less what we had intended to. Furthermore, we generalize the population dynamics to the stochastic case via:

$$

x_{t+1} = x_t + r_tx_t(1 - x_t/K) - h_tx_t $$

where $K = 125$ is the carrying capacity. The growth rate r_t is now stochastic and follows a discrete distribution:

$$r_t = \begin{cases} 0.85r_{\max} & \text{with probability } 0.25 \\ 1.05r_{\max} & \text{with probability } 0.5 \\ 1.15r_{\max} & \text{with probability } 0.25 \end{cases} \tag{273}$$

where $r_{\max} = 0.3$ is the maximum growth rate. Applying the principle of optimality, we can express the optimal value function $J^\star(x_t, t)$ recursively:

$$J^\star(x_t, t) = \max_{d(t) \in D} E\left[F(x_t \cdot h_t) + J^\star(x_{t+1}, t+1)\right] \tag{274}$$

where the expectation is taken over the harvest and growth rate random variables. The boundary condition remains $J^*(x_{t_f}) = 0$. We can now adapt our previous code to account for the stochasticity in our model. One important difference is that simulating a solution in this context requires multiple realizations of our process. This is an important consideration when evaluating reinforcement learning methods in practice, as success cannot be claimed based on a single successful trajectory.

Linear Quadratic Regulator via Dynamic Programming We now examine a special case where the backward recursion admits a closed-form solution. When the system dynamics are linear and the cost function is quadratic, the optimization at each stage can be solved analytically. Moreover, the value function itself maintains a quadratic structure throughout the recursion, and the optimal policy reduces to a simple linear feedback law. This result eliminates the need for discretization, interpolation, or any function approximation. The infinite-dimensional problem collapses to tracking a finite set of matrices.

Consider a discrete-time linear system:

$$\mathbf{x}_{t+1} = A_t \mathbf{x}_t + B_t \mathbf{u}_t \tag{275}$$

where $\mathbf{x}_t \in R^n$ is the state and $\mathbf{u}_t \in R^m$ is the control input. The matrices $A_t \in R^{n \times n}$ and $B_t \in R^{n \times m}$ describe the system dynamics at time t.

The cost function to be minimized is quadratic:

$$J = \frac{1}{2}\mathbf{x}_T^\top Q_T \mathbf{x}_T + \frac{1}{2}\sum_{t=0}^{T-1} \left(\mathbf{x}_t^\top Q_t \mathbf{x}_t + \mathbf{u}_t^\top R_t \mathbf{u}_t\right) \tag{276}$$

where $Q_T \succeq 0$ (positive semidefinite), $Q_t \succeq 0$, and $R_t \succ 0$ (positive definite) are symmetric matrices of appropriate dimensions. The positive definiteness of R_t ensures the minimization problem is well-posed.

What we now have to observe is that if the terminal cost is quadratic, then the value function at every earlier stage remains quadratic. This is not immediately obvious, but it follows from the structure of Bellman's equation combined with the linearity of the dynamics.

We claim that the optimal cost-to-go from any stage t takes the form:

$$J_t^\star(\mathbf{x}_t) = \frac{1}{2}\mathbf{x}_t^\top P_t \mathbf{x}_t \tag{277}$$

for some positive semidefinite matrix P_t. At the terminal time, this is true by definition: $P_T = Q_T$.

Let's verify this structure and derive the recursion for P_t using backward induction. Suppose we've established that $J_{t+1}^\star(\mathbf{x}_{t+1}) = \frac{1}{2}\mathbf{x}_{t+1}^\top P_{t+1}\mathbf{x}_{t+1}$. Bellman's equation at stage t states:

$$J_t^\star(\mathbf{x}_t) = \min_{\mathbf{u}_t}\left[\frac{1}{2}\mathbf{x}_t^\top Q_t \mathbf{x}_t + \frac{1}{2}\mathbf{u}_t^\top R_t \mathbf{u}_t + J_{t+1}^\star(\mathbf{x}_{t+1})\right] \tag{278}$$

Substituting the dynamics $\mathbf{x}_{t+1} = A_t\mathbf{x}_t + B_t\mathbf{u}_t$ and the quadratic form for $J_{t+1}^\star$:

$$J_t^\star(\mathbf{x}_t) = \min_{\mathbf{u}_t}\left[\frac{1}{2}\mathbf{x}_t^\top Q_t \mathbf{x}_t + \frac{1}{2}\mathbf{u}_t^\top R_t \mathbf{u}_t + \frac{1}{2}(A_t\mathbf{x}_t + B_t\mathbf{u}_t)^\top P_{t+1}(A_t\mathbf{x}_t + B_t\mathbf{u}_t)\right] \tag{279}$$

Expanding the last term:

$$(A_t\mathbf{x}_t+B_t\mathbf{u}_t)^\top P_{t+1}(A_t\mathbf{x}_t+B_t\mathbf{u}_t) = \mathbf{x}_t^\top A_t^\top P_{t+1}A_t\mathbf{x}_t+2\mathbf{x}_t^\top A_t^\top P_{t+1}B_t\mathbf{u}_t+\mathbf{u}_t^\top B_t^\top P_{t+1}B_t\mathbf{u}_t \tag{280}$$

The expression inside the minimization becomes:

$$\frac{1}{2}\mathbf{x}_t^\top Q_t\mathbf{x}_t+\frac{1}{2}\mathbf{u}_t^\top R_t\mathbf{u}_t+\frac{1}{2}\mathbf{x}_t^\top A_t^\top P_{t+1}A_t\mathbf{x}_t+\mathbf{x}_t^\top A_t^\top P_{t+1}B_t\mathbf{u}_t+\frac{1}{2}\mathbf{u}_t^\top B_t^\top P_{t+1}B_t\mathbf{u}_t \tag{281}$$

Collecting terms involving $\mathbf{u}_t$:

$$=\frac{1}{2}\mathbf{x}_t^\top(Q_t+A_t^\top P_{t+1}A_t)\mathbf{x}_t+\mathbf{x}_t^\top A_t^\top P_{t+1}B_t\mathbf{u}_t+\frac{1}{2}\mathbf{u}_t^\top(R_t+B_t^\top P_{t+1}B_t)\mathbf{u}_t \tag{282}$$

This is a quadratic function of $\mathbf{u}_t$. To find the minimizer, we take the gradient with respect to $\mathbf{u}_t$ and set it to zero:

$$\frac{\partial}{\partial\mathbf{u}_t} = (R_t+B_t^\top P_{t+1}B_t)\mathbf{u}_t+B_t^\top P_{t+1}A_t\mathbf{x}_t = 0 \tag{283}$$

Since $R_t+B_t^\top P_{t+1}B_t$ is positive definite (both R_t and P_{t+1} are positive semidefinite with R_t strictly positive), we can solve for the optimal control:

$$\mathbf{u}_t^\star = -(R_t+B_t^\top P_{t+1}B_t)^{-1}B_t^\top P_{t+1}A_t\mathbf{x}_t \tag{284}$$

Define the gain matrix:

$$K_t = (R_t+B_t^\top P_{t+1}B_t)^{-1}B_t^\top P_{t+1}A_t \tag{285}$$

so that $\mathbf{u}_t^\star = -K_t\mathbf{x}_t$. This is a **linear feedback policy**: the optimal control is simply a linear function of the current state.

Substituting $\mathbf{u}_t^\star$ back into the cost-to-go expression and simplifying (by completing the square), we obtain:

$$J_t^\star(\mathbf{x}_t) = \frac{1}{2}\mathbf{x}_t^\top P_t\mathbf{x}_t \tag{286}$$

where P_t satisfies the **discrete-time Riccati equation**:

$$P_t = Q_t+A_t^\top P_{t+1}A_t-A_t^\top P_{t+1}B_t(R_t+B_t^\top P_{t+1}B_t)^{-1}B_t^\top P_{t+1}A_t \tag{287}$$

Putting everything together, the backward induction procedure under the LQR setting then becomes:

3.2.2 Markov Decision Process Formulation

Rather than expressing the stochasticity in our system through a disturbance term as a parameter to a deterministic difference equation, we often work with an alternative representation (more common in operations research) which uses the Markov Decision Process formulation. The idea is that when we model our system in this way with the disturbance term being drawn indepently of the previous stages, the induced trajectory are those of a Markov chain. Hence, we can re-cast our control problem in that language, leading to the so-called Markov Decision Process framework in which we express the system dynamics in terms of transition probabilities rather than explicit state equations. In this framework, we express the probability that the system is in a given state using the transition probability function:

$$p_t(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) \tag{288}$$

This function gives the probability of transitioning to state $\mathbf{x}_{t+1}$ at time $t+1$, given that the system is in state $\mathbf{x}_t$ and action $\mathbf{u}_t$ is taken at time t. Therefore, p_t specifies a conditional probability distribution over the next states: namely, the sum (for discrete state spaces) or integral over the next state should be 1.

Given the control theory formulation of our problem via a deterministic dynamics function and a noise term, we can derive the corresponding transition probability function through the following relationship:

$$\begin{aligned} p_t(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) &= P(\mathbf{W}_t \in \{\mathbf{w} \in \mathbf{W} : \mathbf{x}_{t+1} = f_t(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w})\}) \\ &= \sum_{\{\mathbf{w} \in \mathbf{W} : \mathbf{x}_{t+1} = f_t(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w})\}} q_t(\mathbf{w}) \end{aligned} \tag{289}$$

Here, $q_t(\mathbf{w})$ represents the probability density or mass function of the disturbance $\mathbf{W}_t$ (assuming discrete state spaces). When dealing with continuous spaces, the above expression simply contains an integral rather than a summation.

For a system with deterministic dynamics and no disturbance, the transition probabilities become much simpler and be expressed using the indicator function. Given a deterministic system with dynamics:

$$\mathbf{x}_{t+1} = f_t(\mathbf{x}_t, \mathbf{u}_t) \tag{290}$$

The transition probability function can be expressed as:

$$p_t(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) = \begin{cases} 1 & \text{if } \mathbf{x}_{t+1} = f_t(\mathbf{x}_t, \mathbf{u}_t) \\ 0 & \text{otherwise} \end{cases} \tag{291}$$

With this transition probability function, we can recast our Bellman optimality equation:

$$J_t^\star(\mathbf{x}_t) = \max_{\mathbf{u}_t \in \mathbf{U}} \left\{ c_t(\mathbf{x}_t, \mathbf{u}_t) + \sum_{\mathbf{x}_{t+1}} p_t(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) J_{t+1}^\star(\mathbf{x}_{t+1}) \right\} \tag{292}$$

Here, $c(\mathbf{x}_t, \mathbf{u}_t)$ represents the expected immediate reward (or negative cost) when in state $\mathbf{x}_t$ and taking action $\mathbf{u}_t$ at time t. The summation term computes the expected optimal value for the future states, weighted by their transition probabilities.

This formulation offers several advantages:

1. It makes the Markovian nature of the problem explicit: the future state depends only on the current state and action, not on the history of states and actions.

2. For discrete-state problems, the entire system dynamics can be specified by a set of transition matrices, one for each possible action.

3. It allows us to bridge the gap with the wealth of methods in the field of probabilistic graphical models and statistical machine learning techniques for modelling and analysis.

Notation in Operations Reseach The presentation above was intended to bridge the gap between the control-theoretic perspective and the world of closed-loop control through the idea of determining the value function of a parametric optimal control problem. We then saw how the backward induction procedure was applicable to both the deterministic and stochastic cases by taking the expectation over the disturbance variable. We then said that we can alternatively work with a representation of our system where instead of writing our model as a deterministic dynamics function taking a disturbance as an input, we would rather work directly via its transition probability function, which gives rise to the Markov chain interpretation of our system in simulation.

Note that the notation used in control theory tends to differ from that found in operations research communities, in which the field of dynamic programming flourished. We summarize those (purely notational) differences in this section.

In operations research, the system state at each decision epoch is typically denoted by $s \in \mathcal{S}$, where S is the set of possible system states. When the system is in state s, the decision maker may choose an action a from the set of allowable actions $\mathcal{A}_s$. The union of all action sets is denoted as $\mathcal{A} = \bigcup_{s \in \mathcal{S}} \mathcal{A}_s$.

The dynamics of the system are described by a transition probability function $p_t(j|s, a)$, which represents the probability of transitioning to state $j \in \mathcal{S}$ at time $t + 1$, given that the system is in state s at time t and action $a \in \mathcal{A}_s$ is chosen. This transition probability function satisfies:

$$\sum_{j \in \mathcal{S}} p_t(j|s, a) = 1 \tag{293}$$

It's worth noting that in operations research, we typically work with reward maximization rather than cost minimization, which is more common in control theory. However, we can easily switch between these perspectives by simply negating the quantity. That is, maximizing a reward function is equivalent to minimizing its negative, which we would then call a cost function.

The reward function is denoted by $r_t(s, a)$, representing the reward received at time t when the system is in state s and action a is taken. In some cases, the reward may also depend on the next state, in which case it is denoted as $r_t(s, a, j)$. The expected reward can then be computed as:

$$r_t(s, a) = \sum_{j \in \mathcal{S}} r_t(s, a, j) p_t(j|s, a) \tag{294}$$

Combined together, these elemetns specify a Markov decision process, which is fully described by the tuple:

$$\{T, S, \mathcal{A}_s, p_t(\cdot|s, a), r_t(s, a)\} \tag{295}$$

where T represents the set of decision epochs (the horizon).

What is an Optimal Policy? Let's go back to the starting point and define what it means for a policy to be optimal in a Markov Decision Problem. For this, we will be considering different possible search spaces (policy classes) and compare policies based on the ordering of their value from any possible start state. The value of a policy $\boldsymbol{\pi}$ (optimal or not) is defined as the expected total reward obtained by following that policy from a given starting state. Formally, for a finite-horizon MDP with N decision epochs, we define the value function $v^{\boldsymbol{\pi}}(s, t)$ as:

$$v^{\boldsymbol{\pi}}(s, t) E\left[\sum_{k=t}^{N-1} r_t(S_k, A_k) + r_N(S_N) \mid S_t = s\right] \tag{296}$$

where S_t is the state at time t, A_t is the action taken at time t, and r_t is the reward function. For simplicity, we write $v^{\boldsymbol{\pi}}(s)$ to denote $v^{\boldsymbol{\pi}}(s, 1)$, the value of following policy $\boldsymbol{\pi}$ from state s at the first stage over the entire horizon N.

In finite-horizon MDPs, our goal is to identify an optimal policy, denoted by $\boldsymbol{\pi}^*$, that maximizes total expected reward over the horizon N. Specifically:

$$v^{\boldsymbol{\pi}^*}(s) \geq v^{\boldsymbol{\pi}}(s), \quad \forall s \in \mathcal{S}, \quad \forall \boldsymbol{\pi} \in \Pi^{\mathrm{HR}} \tag{297}$$

We call $\boldsymbol{\pi}^*$ an **optimal policy** because it yields the highest possible value across all states and all policies within the policy class Π^{HR}. We denote by v^* the maximum value achievable by any policy:

$$v^*(s) = \max_{\boldsymbol{\pi} \in \Pi^{\mathrm{HR}}} v^{\boldsymbol{\pi}}(s), \quad \forall s \in \mathcal{S} \tag{298}$$

In reinforcement learning literature, v^* is typically referred to as the "optimal value function," while in some operations research references, it might be called

the "value of an MDP." An optimal policy $\boldsymbol{\pi}^*$ is one for which its value function equals the optimal value function:

$$v^{\boldsymbol{\pi}^*}(s) = v^*(s), \quad \forall s \in \mathcal{S} \tag{299}$$

This notion of optimality applies to every state. Policies optimal in this sense are sometimes called "uniformly optimal policies." A weaker notion of optimality, often encountered in reinforcement learning practice, is optimality with respect to an initial distribution of states. In this case, we seek a policy $\boldsymbol{\pi} \in \Pi^{\text{HR}}$ that maximizes:

$$\sum_{s \in \mathcal{S}} v^{\boldsymbol{\pi}}(s) P_1(S_1 = s) \tag{300}$$

where $P_1(S_1 = s)$ is the probability of starting in state s.

The maximum value can be achieved by searching over the space of deterministic Markovian Policies. Consequently:

$$v^*(s) = \max_{\boldsymbol{\pi} \in \Pi^{\text{HR}}} v^{\boldsymbol{\pi}}(s) = \max_{\boldsymbol{\pi} \in \Pi^{MD}} v^{\boldsymbol{\pi}}(s), \quad s \in S \tag{301}$$

This equality significantly simplifies the computational complexity of our algorithms, as the search problem can now be decomposed into N sub-problems in which we only have to search over the set of possible actions. This is the backward induction algorithm, which we present a second time, but departing this time from the control-theoretic notation and using the MDP formalism:

Note that the same procedure can also be used for finding the value of a policy with minor changes;

This code could also finally be adapted to support randomized policies using:

$$v^{\boldsymbol{\pi}}(s_t, t) = \sum_{a_t \in \mathcal{A}_{s_t}} \pi_t(a_t \mid s_t) \left(r_t(s_t, a_t) + \sum_{j \in S} p_t(j | s_t, a_t) v^{\boldsymbol{\pi}}(j, t+1) \right) \tag{302}$$

Example: Sample Size Determination in Pharmaceutical Development Pharmaceutical development is the process of bringing a new drug from initial discovery to market availability. This process is lengthy, expensive, and risky, typically involving several stages:

1. **Drug Discovery**: Identifying a compound that could potentially treat a disease.

2. **Preclinical Testing**: Laboratory and animal testing to assess safety and efficacy. . **Clinical Trials**: Testing the drug in humans, divided into phases:

    - Phase I: Testing for safety in a small group of healthy volunteers.

- Phase II: Testing for efficacy and side effects in a larger group with the target condition.
- Phase III: Large-scale testing to confirm efficacy and monitor side effects.

3. **Regulatory Review**: Submitting a New Drug Application (NDA) for approval.
4. **Post-Market Safety Monitoring**: Continuing to monitor the drug's effects after market release.

This process can take 10-15 years and cost over $1 billion Adams and Brantner [2009]. The high costs and risks involved call for a principled approach to decision making. We'll focus on the clinical trial phases and NDA approval, per the MDP model presented by Chang [2010]:

1. **States** (S): Our state space is $S = \{s_1, s_2, s_3, s_4\}$, where:
   - s_1: Phase I clinical trial
   - s_2: Phase II clinical trial
   - s_3: Phase III clinical trial
   - s_4: NDA approval
2. **Actions** (A): At each state, the action is choosing the sample size n_i for the corresponding clinical trial. The action space is $A = \{10, 11, ..., 1000\}$, representing possible sample sizes.
3. **Transition Probabilities** (P): The probability of moving from one state to the next depends on the chosen sample size and the inherent properties of the drug. We define:
   - $P(s_2|s_1, n_1) = p_{12}(n_1) = \sum_{i=0}^{\lfloor \eta_1 n_1 \rfloor} \binom{n_1}{i} p_0^i (1-p_0)^{n_1-i}$ where p_0 is the true toxicity rate and η_1 is the toxicity threshold for Phase I.

- Of particular interest is the transition from Phase II to Phase III which we model as:
  $P(s_3|s_2, n_2) = p_{23}(n_2) = \Phi\left(\frac{\sqrt{n_2}}{2}\delta - z_{1-\eta_2}\right)$
  where Φ is the cumulative distribution function (CDF) of the standard normal distribution:
  $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$
  This is giving us the probability that we would observe a treatment effect this large or larger if the null hypothesis (no treatment effect) were true. A higher probability indicates stronger evidence of a treatment effect, making it more likely that the drug will progress to Phase III.

In this expression, δ is called the "normalized treatment effect". In clinical trials, we're often interested in the difference between the treatment and control groups. The "normalized" part means we've adjusted this difference for the variability in the data. Specifically $\delta = \frac{\mu_t - \mu_c}{\sigma}$ where μ_t is the mean outcome in the treatment group, μ_c is the mean outcome in the control group, and σ is the standard deviation of the outcome. A larger δ indicates a stronger treatment effect.

Furthermore, the term $z_{1-\eta_2}$ is the $(1-\eta_2)$-quantile of the standard normal distribution. In other words, it's the value where the probability of a standard normal random variable being greater than this value is η_2. For example, if $\eta_2 = 0.05$, then $z_{1-\eta_2} \approx 1.645$. A smaller η_2 makes the trial more conservative, requiring stronger evidence to proceed to Phase III.

Finally, n_2 is the sample size for Phase II. The $\sqrt{n_2}$ term reflects that the precision of our estimate of the treatment effect improves with the square root of the sample size.

- $P(s_4|s_3, n_3) = p_{34}(n_3) = \Phi\left(\frac{\sqrt{n_3}}{2}\delta - z_{1-\eta_3}\right)$ where η_3 is the significance level for Phase III.

[resume]**Rewards** (R): The reward function captures the costs of running trials and the potential profit from a successful drug:

1.
   - $r(s_i, n_i) = -c_i(n_i)$ for $i = 1, 2, 3$, where $c_i(n_i)$ is the cost of running a trial with sample size n_i.
   - $r(s_4) = g_4$, where g_4 is the expected profit from a successful drug.
2. **Discount Factor** (γ): We use a discount factor $0 < \gamma \leq 1$ to account for the time value of money and risk preferences.

3.2.3 Infinite-Horizon MDPs

It often makes sense to model control problems over infinite horizons. We extend the previous setting and define the expected total reward of policy $\boldsymbol{\pi} \in \Pi^{\text{HR}}$, $v^{\boldsymbol{\pi}}$ as:

$$v^{\boldsymbol{\pi}}(s) = E\left[\sum_{t=1}^{\infty} r(S_t, A_t)\right] \tag{303}$$

One drawback of this model is that we could easily encounter values that are $+\infty$ or $-\infty$, even in a setting as simple as a single-state MDP which loops back into itself and where the accrued reward is nonzero.

Therefore, it is often more convenient to work with an alternative formulation which guarantees the existence of a limit: the expected total discounted reward of policy $\boldsymbol{\pi} \in \Pi^{\text{HR}}$ is defined to be:

$$v_\gamma^{\boldsymbol{\pi}}(s) \equiv \lim_{N\to\infty} E\left[\sum_{t=1}^{N} \gamma^{t-1} r(S_t, A_t)\right] \tag{304}$$

for $0 \leq \gamma < 1$ and when $\max_{s \in \mathcal{S}} \max_{a \in \mathcal{A}_s} |r(s,a)| = R_{\max} < \infty$, in which case, $|v_\gamma^{\boldsymbol{\pi}}(s)| \leq (1-\gamma)^{-1} R_{\max}$.

Finally, another possibility for the infinite-horizon setting is the so-called average reward or gain of policy $\boldsymbol{\pi} \in \Pi^{\text{HR}}$ defined as:

$$g^{\boldsymbol{\pi}}(s) \equiv \lim_{N \to \infty} \frac{1}{N} E\left[\sum_{t=1}^{N} r(S_t, A_t)\right] \tag{305}$$

We won't be working with this formulation in this course due to its inherent practical and theoretical complexities.

Extending the previous notion of optimality from finite-horizon models, a policy $\boldsymbol{\pi}^*$ is said to be discount optimal for a given γ if:

$$v_\gamma^{\boldsymbol{\pi}^*}(s) \geq v_\gamma^{\boldsymbol{\pi}}(s) \quad \text{for each } s \in S \text{ and all } \boldsymbol{\pi} \in \Pi^{\text{HR}} \tag{306}$$

Furthermore, the value of a discounted MDP $v_\gamma^*(s)$, is defined by:

$$v_\gamma^*(s) \equiv \max_{\boldsymbol{\pi} \in \Pi^{\text{HR}}} v_\gamma^{\boldsymbol{\pi}}(s) \tag{307}$$

More often, we refer to v_γ by simply calling it the optimal value function.

As for the finite-horizon setting, the infinite horizon discounted model does not require history-dependent policies, since for any $\boldsymbol{\pi} \in \Pi^{HR}$ there exists a $\boldsymbol{\pi}' \in \Pi^{MR}$ with identical total discounted reward:

$$v_\gamma^*(s) \equiv \max_{\boldsymbol{\pi} \in \Pi^{HR}} v_\gamma^{\boldsymbol{\pi}}(s) = \max_{\boldsymbol{\pi} \in \Pi^{MR}} v_\gamma^{\boldsymbol{\pi}}(s). \tag{308}$$

Random Horizon Interpretation of Discounting The use of discounting can be motivated both from a modeling perspective and as a means to ensure that the total reward remains bounded. From the modeling perspective, we can view discounting as a way to weight more or less importance on the immediate rewards vs. the long-term consequences. There is also another interpretation which stems from that of a finite horizon model but with an uncertain end time. More precisely:

Let $v_\nu^{\boldsymbol{\pi}}(s)$ denote the expected total reward obtained by using policy $\boldsymbol{\pi}$ when the horizon length ν is random. We define it by:

$$v_\nu^{\boldsymbol{\pi}}(s) \equiv E_s^{\boldsymbol{\pi}}\left[E_\nu\left\{\sum_{t=1}^{\nu} r(S_t, A_t)\right\}\right] \tag{309}$$

Theorem 3.5 (Random horizon interpretation of discounting)**.** *Suppose that the horizon ν follows a geometric distribution with parameter γ, $0 \leq \gamma < 1$, independent of the policy such that $P(\nu = n) = (1-\gamma)\gamma^{n-1}$, $n = 1, 2, \ldots$, then $v_\nu^{\boldsymbol{\pi}}(s) = v_\gamma^{\boldsymbol{\pi}}(s)$ for all $s \in \mathcal{S}$.*

Proof. See proposition 5.3.1 in Puterman [1994].

By definition of the finite-horizon value function and the law of total expectation:

$$v_\nu^{\boldsymbol{\pi}}(s) = \sum_{n=1}^{\infty} P(\nu = n) \cdot v_n^{\boldsymbol{\pi}}(s) = \sum_{n=1}^{\infty} (1-\gamma)\gamma^{n-1} \cdot E_s^{\boldsymbol{\pi}} \left\{ \sum_{t=1}^{n} r(S_t, A_t) \right\}. \quad (310)$$

Combining the expectation with the sum over n:

$$v_\nu^{\boldsymbol{\pi}}(s) = E_s^{\boldsymbol{\pi}} \left\{ \sum_{n=1}^{\infty} (1-\gamma)\gamma^{n-1} \sum_{t=1}^{n} r(S_t, A_t) \right\}. \quad (311)$$

Reordering the summations: Under the bounded reward assumption $|r(s,a)| \leq R_{\max}$ and $\gamma < 1$, we have

$$E_s^{\boldsymbol{\pi}} \left\{ \sum_{n=1}^{\infty} \sum_{t=1}^{n} |r(S_t, A_t)| \cdot (1-\gamma)\gamma^{n-1} \right\} \leq R_{\max} \sum_{n=1}^{\infty} n(1-\gamma)\gamma^{n-1} = \frac{R_{\max}}{1-\gamma} < \infty, \quad (312)$$

which justifies exchanging the order of summation by Fubini's theorem.

To reverse the order, note that the pair (n, t) with $1 \leq t \leq n$ can be reindexed by fixing t first and letting n range from t to ∞:

$$\sum_{n=1}^{\infty} \sum_{t=1}^{n} = \sum_{t=1}^{\infty} \sum_{n=t}^{\infty}. \quad (313)$$

Therefore:

$$v_\nu^{\boldsymbol{\pi}}(s) = E_s^{\boldsymbol{\pi}} \left\{ \sum_{t=1}^{\infty} r(S_t, A_t) \sum_{n=t}^{\infty} (1-\gamma)\gamma^{n-1} \right\}.$$

Evaluating the inner sum: Using the substitution $m = n - t + 1$ (so $n = m + t - 1$):

$$\begin{aligned} \sum_{n=t}^{\infty} (1-\gamma)\gamma^{n-1} &= \sum_{m=1}^{\infty} (1-\gamma)\gamma^{m+t-2} \\ &= \gamma^{t-1}(1-\gamma) \sum_{m=1}^{\infty} \gamma^{m-1} \\ &= \gamma^{t-1}(1-\gamma) \cdot \frac{1}{1-\gamma} = \gamma^{t-1}. \end{aligned}$$

Substituting back:

$$v_\nu^{\boldsymbol{\pi}}(s) = E_s^{\boldsymbol{\pi}} \left\{ \sum_{t=1}^{\infty} \gamma^{t-1} r(S_t, A_t) \right\} = v_\gamma^{\boldsymbol{\pi}}(s). \quad (314)$$

□

Vector Representation in Markov Decision Processes Let V be the set of bounded real-valued functions on a discrete state space S. This means any function $f \in V$ satisfies the condition:

$$\|f\| = \max_{s \in S} |f(s)| < \infty. \tag{315}$$

where notation $\|f\|$ represents the sup-norm (or ℓ_∞-norm) of the function f.

When working with discrete state spaces, we can interpret elements of V as vectors and linear operators on V as matrices, allowing us to leverage tools from linear algebra. The sup-norm (ℓ_∞ norm) of matrix $\mathbf{H}$ is defined as:

$$\|\mathbf{H}\| \equiv \max_{s \in S} \sum_{j \in S} |\mathbf{H}_{s,j}| \tag{316}$$

where $\mathbf{H}_{s,j}$ represents the (s, j)-th component of the matrix $\mathbf{H}$.

For a Markovian decision rule $\pi \in \Pi^{MD}$, we define:

$$\begin{aligned} \mathbf{r}_\pi(s) &\equiv r(s, \pi(s)), \quad \mathbf{r}_\pi \in R^{|S|}, \\ [\mathbf{P}_\pi]_{s,j} &\equiv p(j \mid s, \pi(s)), \quad \mathbf{P}_\pi \in R^{|S| \times |S|}. \end{aligned}$$

For a randomized decision rule $\pi \in \Pi^{MR}$, these definitions extend to:

$$\begin{aligned} \mathbf{r}_\pi(s) &\equiv \sum_{a \in A_s} \pi(a \mid s)\, r(s, a), \\ [\mathbf{P}_\pi]_{s,j} &\equiv \sum_{a \in A_s} \pi(a \mid s)\, p(j \mid s, a). \end{aligned}$$

In both cases, $\mathbf{r}_\pi$ denotes a reward vector in $R^{|S|}$, with each component $\mathbf{r}_\pi(s)$ representing the reward associated with state s. Similarly, $\mathbf{P}_\pi$ is a transition probability matrix in $R^{|S| \times |S|}$, capturing the transition probabilities under decision rule π.

For a nonstationary Markovian policy $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots) \in \Pi^{MR}$, the expected total discounted reward is given by:

$$\mathbf{v}_\gamma^{\boldsymbol{\pi}}(s) = E\left[\sum_{t=1}^{\infty} \gamma^{t-1} r\left(S_t, A_t\right) \,\middle|\, S_1 = s \right]. \tag{317}$$

Using vector notation, this can be expressed as:

$$\begin{aligned} \mathbf{v}_\gamma^{\boldsymbol{\pi}} &= \sum_{t=1}^{\infty} \gamma^{t-1} \mathbf{P}_{\boldsymbol{\pi}}^{t-1} \mathbf{r}_{\pi_1} \\ &= \mathbf{r}_{\pi_1} + \gamma \mathbf{P}_{\pi_1} \mathbf{r}_{\pi_2} + \gamma^2 \mathbf{P}_{\pi_1} \mathbf{P}_{\pi_2} \mathbf{r}_{\pi_3} + \cdots \\ &= \mathbf{r}_{\pi_1} + \gamma \mathbf{P}_{\pi_1} \left(\mathbf{r}_{\pi_2} + \gamma \mathbf{P}_{\pi_2} \mathbf{r}_{\pi_3} + \gamma^2 \mathbf{P}_{\pi_2} \mathbf{P}_{\pi_3} \mathbf{r}_{\pi_4} + \cdots \right). \end{aligned} \tag{318}$$

This formulation leads to a recursive relationship:

$$\begin{aligned}\mathbf{v}_\gamma^{\boldsymbol{\pi}} &= \mathbf{r}_{\pi_1} + \gamma \mathbf{P}_{\pi_1} \mathbf{v}_\gamma^{\boldsymbol{\pi}'} \\ &= \sum_{t=1}^{\infty} \gamma^{t-1} \mathbf{P}_{\boldsymbol{\pi}}^{t-1} \mathbf{r}_{\pi_t}\end{aligned}$$

where $\boldsymbol{\pi}' = (\pi_2, \pi_3, \ldots)$.

For a stationary policy $\boldsymbol{\pi} = \text{const}(\pi)$ with constant decision rule π, the total expected reward simplifies to:

$$\begin{aligned}\mathbf{v}_\gamma^{\pi} &= \mathbf{r}_{\pi} + \gamma \mathbf{P}_{\pi} \mathbf{v}_\gamma^{\pi} \\ &= \sum_{t=1}^{\infty} \gamma^{t-1} \mathbf{P}_{\pi}^{t-1} \mathbf{r}_{\pi}\end{aligned}$$

This last expression is called a Neumann series expansion, and it's guaranteed to exists under the assumptions of bounded reward and discount factor strictly less than one.

Theorem 3.6 (Neumann Series and Invertibility)**.** *The* ***spectral radius*** *of a matrix* $\mathbf{H}$ *is defined as:*

$$\rho(\mathbf{H}) \equiv \max_i |\lambda_i(\mathbf{H})| \tag{319}$$

where $\lambda_i(\mathbf{H})$ *are the eigenvalues of* $\mathbf{H}$*.*

Neumann Series Existence: *For any matrix* $\mathbf{H}$*, the Neumann series*

$$\sum_{t=0}^{\infty} \mathbf{H}^t = \mathbf{I} + \mathbf{H} + \mathbf{H}^2 + \cdots \tag{320}$$

converges if and only if $\rho(\mathbf{H}) < 1$*. When this condition holds, the matrix* $(\mathbf{I} - \mathbf{H})$ *is invertible and*

$$(\mathbf{I} - \mathbf{H})^{-1} = \sum_{t=0}^{\infty} \mathbf{H}^t. \tag{321}$$

Note that for any induced matrix norm $\|\cdot\|$ (i.e., a norm satisfying $\|\mathbf{Hv}\| \leq \|\mathbf{H}\| \cdot \|\mathbf{v}\|$ for all vectors $\mathbf{v}$) and any matrix $\mathbf{H}$, the spectral radius is bounded by:

$$\rho(\mathbf{H}) \leq \|\mathbf{H}\|. \tag{322}$$

This inequality provides a practical way to verify the convergence condition $\rho(\mathbf{H}) < 1$ by checking the simpler condition $\|\mathbf{H}\| < 1$ rather than trying to compute the eigenvalues directly.

We can now verify that $(\mathbf{I} - \gamma \mathbf{P}_d)$ is invertible and the Neumann series converges.

1. **Norm of the transition matrix:** Since $\mathbf{P}_d$ is a stochastic matrix (each row sums to 1 and all entries are non-negative), its ℓ_∞-norm is:

$$\|\mathbf{P}_d\| = \max_{s \in S} \sum_{j \in S} [\mathbf{P}_d]_{s,j} = \max_{s \in S} 1 = 1. \tag{323}$$

2. **Norm of the scaled matrix:** Using the homogeneity property of norms, we have:

$$\|\gamma \mathbf{P}_d\| = |\gamma| \cdot \|\mathbf{P}_d\| = |\gamma| \cdot 1 = |\gamma|. \tag{324}$$

3. **Bounding the spectral radius:** Since the spectral radius is bounded by the matrix norm:

$$\rho(\gamma \mathbf{P}_d) \leq \|\gamma \mathbf{P}_d\| = |\gamma|. \tag{325}$$

4. **Verifying convergence:** Since $0 \leq \gamma < 1$ by assumption, we have:

$$\rho(\gamma \mathbf{P}_d) \leq |\gamma| < 1. \tag{326}$$

   This strict inequality guarantees that $(\mathbf{I} - \gamma \mathbf{P}_d)$ is invertible and the Neumann series converges.

Therefore, the Neumann series expansion converges and yields:

$$\mathbf{v}_\gamma^{d^\infty} = (\mathbf{I} - \gamma \mathbf{P}_d)^{-1} \mathbf{r}_d = \sum_{t=0}^{\infty} (\gamma \mathbf{P}_d)^t \mathbf{r}_d = \sum_{t=1}^{\infty} \gamma^{t-1} \mathbf{P}_d^{t-1} \mathbf{r}_d. \tag{327}$$

Consequently, for a stationary policy, $\mathbf{v}_\gamma^{d^\infty}$ can be determined as the solution to the linear equation:

$$\mathbf{v} = \mathbf{r}_d + \gamma \mathbf{P}_d \mathbf{v}, \tag{328}$$

which can be rearranged to:

$$(\mathbf{I} - \gamma \mathbf{P}_d)\mathbf{v} = \mathbf{r}_d. \tag{329}$$

We can also characterize $\mathbf{v}_\gamma^{d^\infty}$ as the solution to an operator equation. More specifically, define the transformation L_d by

$$\mathrm{L}_d \mathbf{v} \equiv \mathbf{r}_d + \gamma \mathbf{P}_d \mathbf{v} \tag{330}$$

for any $\mathbf{v} \in V$. Intuitively, L_d takes a value function $\mathbf{v}$ as input and returns a new value function that combines immediate rewards ($\mathbf{r}_d$) with discounted future values ($\gamma \mathbf{P}_d \mathbf{v}$).

Note

While we often refer to L_d as a "linear operator" in the RL literature, it is technically an **affine operator** (or affine transformation), not a linear operator in the strict sense. To see why, recall that a linear operator $\mathcal{T}$ must satisfy:

1. **Additivity:** $\mathcal{T}(\mathbf{v}_1 + \mathbf{v}_2) = \mathcal{T}(\mathbf{v}_1) + \mathcal{T}(\mathbf{v}_2)$
2. **Homogeneity:** $\mathcal{T}(\alpha\mathbf{v}) = \alpha\mathcal{T}(\mathbf{v})$ for all scalars α

However, L_d fails the additivity test:

$$\mathrm{L}_d(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{r}_d + \gamma\mathbf{P}_d(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{r}_d + \gamma\mathbf{P}_d\mathbf{v}_1 + \gamma\mathbf{P}_d\mathbf{v}_2 \tag{331}$$

while

$$\mathrm{L}_d(\mathbf{v}_1) + \mathrm{L}_d(\mathbf{v}_2) = (\mathbf{r}_d + \gamma\mathbf{P}_d\mathbf{v}_1) + (\mathbf{r}_d + \gamma\mathbf{P}_d\mathbf{v}_2) = 2\mathbf{r}_d + \gamma\mathbf{P}_d\mathbf{v}_1 + \gamma\mathbf{P}_d\mathbf{v}_2. \tag{332}$$

The presence of the constant term $\mathbf{r}_d$ makes L_d affine rather than linear. An affine operator has the form $\mathcal{A}(\mathbf{v}) = \mathbf{b} + \mathcal{T}(\mathbf{v})$, where $\mathbf{b}$ is a constant vector and $\mathcal{T}$ is a linear operator. In our case, $\mathbf{b} = \mathbf{r}_d$ and $\mathcal{T}(\mathbf{v}) = \gamma\mathbf{P}_d\mathbf{v}$.

Despite this technical distinction, the term "linear operator" is commonly used in the reinforcement learning literature when referring to L_d, following a slight abuse of terminology.

Therefore, we view L_d as an operator mapping elements of V to V: i.e., $\mathrm{L}_d : V \to V$. The fact that the value function of a policy is the solution to a fixed-point equation can then be expressed with the statement:

$$\mathbf{v}_\gamma^{d^\infty} = \mathrm{L}_d\mathbf{v}_\gamma^{d^\infty}. \tag{333}$$

This is a **fixed-point equation**: the value function $\mathbf{v}_\gamma^{d^\infty}$ is a fixed point of the operator L_d.

Solving Operator Equations The operator equation we encountered in MDPs, $\mathbf{v}_\gamma^{d^\infty} = \mathrm{L}_d\mathbf{v}_\gamma^{d^\infty}$, is a specific instance of a more general class of problems known as operator equations. These equations appear in various fields of mathematics and applied sciences, ranging from differential equations to functional analysis.

Operator equations can take several forms, each with its own characteristics and solution methods:

1. **Fixed Point Form**: $x = \mathrm{T}(x)$, where $\mathrm{T} : X \to X$. Common in fixed-point problems, such as our MDP equation, we seek a fixed point x^* such that $x^* = \mathrm{T}(x^*)$.

2. **General Operator Equation**: $\mathrm{T}(x) = y$, where $\mathrm{T} : X \to Y$. Here, X and Y can be different spaces. We seek an $x \in X$ that satisfies the equation for a given $y \in Y$.

3. **Nonlinear Equation**: $\mathrm{T}(x) = 0$, where $\mathrm{T} : X \to Y$. A special case of the general operator equation where we seek roots or zeros of the operator.

4. **Variational Inequality**: Find $x^* \in K$ such that $\langle \mathrm{T}(x^*), x - x^* \rangle \geq 0$ for all $x \in K$. Here, K is a closed convex subset of X, and $\mathrm{T} : K \to X^*$ (the dual space of X). These problems often arise in optimization, game theory, and partial differential equations.

Successive Approximation Method For equations in fixed point form, a common numerical solution method is successive approximation, also known as fixed-point iteration:

The convergence of successive approximation depends on the properties of the operator T. In the simplest and most common setting, we assume T is a contraction mapping. The Banach Fixed-Point Theorem then guarantees that T has a unique fixed point, and the successive approximation method will converge to this fixed point from any starting point. Specifically, T is a contraction if there exists a constant $q \in [0, 1)$ such that for all $x, y \in X$:

$$d(\mathrm{T}(x), \mathrm{T}(y)) \leq q \cdot d(x, y) \tag{334}$$

where d is the metric on X. In this case, the rate of convergence is linear, with error bound:

$$d(x_n, x^*) \leq \frac{q^n}{1 - q} d(x_1, x_0) \tag{335}$$

However, the contraction mapping condition is not the only one that can lead to convergence. For instance, if T is nonexpansive (i.e., Lipschitz continuous with Lipschitz constant 1) and X is a Banach space with certain geometrical properties (e.g., uniformly convex), then under additional conditions (e.g., T has at least one fixed point), the successive approximation method can still converge, albeit potentially more slowly than in the contraction case.

In practice, when dealing with specific problems like MDPs or differential equations, the properties of the operator often naturally align with one of these convergence conditions. For example, in discounted MDPs, the Bellman operator is a contraction in the supremum norm, which guarantees the convergence of value iteration.

Newton-Kantorovich Method The Newton-Kantorovich method is a generalization of Newton's method from finite dimensional vector spaces to infinite dimensional function spaces: rather than iterating in the space of vectors, we are iterating in the space of functions.

Newton's method is often written as the familiar update:

$$x_{k+1} = x_k - [DF(x_k)]^{-1} F(x_k), \tag{336}$$

which makes it look as though the essence of the method is "take a derivative and invert it." But the real workhorse behind Newton's method (both in finite and infinite dimensions) is **linearization**.

At each step, the idea is to replace the nonlinear operator $F : X \to Y$ by a local surrogate model of the form

$$F(x + h) \approx F(x) + Lh, \tag{337}$$

where L is a linear map capturing how small perturbations in the input propagate to changes in the output. This is a Taylor-like expansion in Banach spaces: the role of the derivative is precisely to provide the correct notion of such a linear operator.

To find a root of F, we impose the condition that the surrogate vanishes at the next iterate:

$$0 = F(x + h) \approx F(x) + Lh. \tag{338}$$

Solving this linear equation gives the increment h. In finite dimensions, L is the Jacobian matrix; in Banach spaces, it must be the **Fréchet derivative**.

But what exactly is a Fréchet derivative in infinite dimensions? To understand this, we need to generalize the concept of derivative from finite-dimensional calculus. In infinite-dimensional spaces, there are several notions of differentiability, each with different strengths and requirements:

1. Gâteaux (Directional) Derivative

We say that the Gâteaux derivative of F at x in a specific direction h is:

$$F'(x; h) = \lim_{t \to 0} \frac{F(x + th) - F(x)}{t} \tag{339}$$

This quantity measures how the function F changes along the ray $x + th$. While this limit may exist for each direction h separately, it doesn't guarantee that the derivative is linear in h. This is a key limitation: the Gâteaux derivative can exist in all directions but still fail to provide a good linear approximation.

2. Hadamard Directional Derivative

Rather than considering a single direction of perturbation, we now consider a bundle of perturbations around h. We ask how the function changes as we approach the target direction from nearby directions. We say that F has a Hadamard directional derivative if:

$$F'(x; h) = \lim_{\substack{t \downarrow 0 \\ h' \to h}} \frac{F(x + th') - F(x)}{t} \tag{340}$$

This is a stronger condition than Gâteaux differentiability because it requires the limit to be uniform over nearby directions. However, it still doesn't guarantee linearity in h.

3. Fréchet Derivative

The strongest and most natural notion: F is Fréchet differentiable at x if there exists a bounded linear operator L such that:

$$\lim_{h \to 0} \frac{\|F(x+h) - F(x) - Lh\|}{\|h\|} = 0 \tag{341}$$

This definition directly addresses the inadequacy of the previous notions. Unlike Gâteaux and Hadamard derivatives, the Fréchet derivative explicitly requires the existence of a linear operator L that provides a good approximation. Key properties:

- L must be **linear** in h (unlike the directional derivatives above)
- The approximation error is $o(\|h\|)$, uniform in all directions
- This is the "true" derivative: it generalizes the Jacobian matrix to infinite dimensions
- Notation: $L = F'(x)$ or $DF(x)$

Relationship:

Fréchet differentiable ⇒ Hadamard directionally diff. ⇒ Gâteaux directionally diff. (342)

In the context of the Newton-Kantorovich method, we work with an operator $F : X \to Y$ where both X and Y are Banach spaces. The Fréchet derivative $F'(x)$ is the best linear approximation of F near x, and it's exactly this linear operator L that we use in our linearization $F(x+h) \approx F(x) + F'(x)h$.

Now apart from those mathematical technicalities, Newton-Kantorovich has in essence the same structure as that of the original Newton's method. That is, it applies the following sequence of steps:

1. **Linearize the Operator**: Given an approximation x_n, we consider the Fréchet derivative of F, denoted by $F'(x_n)$. This derivative is a linear operator that provides a local approximation of F near x_n.
2. **Set Up the Newton Step**: The method then solves the linearized equation for a correction h_n:

   $$F'(x_n)h_n = -F(x_n). \tag{343}$$

   This equation represents a linear system where h_n is chosen so that the linearized operator $F(x_n) + F'(x_n)h_n$ equals zero.
3. **Update the Solution**: The new approximation x_{n+1} is then given by:

   $$x_{n+1} = x_n + h_n. \tag{344}$$

   This correction step refines x_n, bringing it closer to the true solution.

4. **Repeat Until Convergence**: We repeat the linearization and update steps until the solution x_n converges to the desired tolerance, which can be verified by checking that $\|F(x_n)\|$ is sufficiently small, or by monitoring the norm $\|x_{n+1} - x_n\|$.

The convergence of Newton-Kantorovich does not hinge on F being a contraction over the entire domain (as it could be the case for successive approximation). The convergence properties of the Newton-Kantorovich method are as follows:

1. **Local Convergence**: Under mild conditions (e.g., F is Fréchet differentiable and $F'(x)$ is invertible near the solution), the method converges locally. This means that if the initial guess is sufficiently close to the true solution, the method will converge.

2. **Global Convergence**: Global convergence is not guaranteed in general. However, under stronger conditions (e.g., F is analytic and satisfies certain bounds), the method can converge globally.

3. **Rate of Convergence**: When the method converges, it typically exhibits quadratic convergence. This means that the error at each step is proportional to the square of the error at the previous step:

$$\|x_{n+1} - x^*\| \leq C\|x_n - x^*\|^2 \tag{345}$$

   where x^* is the true solution and C is some constant. This quadratic convergence is significantly faster than the linear convergence typically seen in methods like successive approximation.

Optimality Equations for Infinite-Horizon MDPs Recall that in the finite-horizon setting, the optimality equations are:

$$v_n(s) = \max_{a \in A_s} \left\{ r(s,a) + \gamma \sum_{j \in S} p(j|s,a) v_{n+1}(j) \right\} \tag{346}$$

where $v_n(s)$ is the value function at time step n for state s, A_s is the set of actions available in state s, $r(s,a)$ is the reward function, γ is the discount factor, and $p(j|s,a)$ is the transition probability from state s to state j given action a.

Intuitively, we would expect that by taking the limit of n to infinity, we might get the nonlinear equations:

$$v(s) = \max_{a \in A_s} \left\{ r(s,a) + \gamma \sum_{j \in S} p(j|s,a) v(j) \right\} \tag{347}$$

which are called the optimality equations or Bellman equations for infinite-horizon MDPs.

We can adopt an operator-theoretic perspective by defining operators on the space V of bounded real-valued functions on the state space S. For a deterministic Markov rule $\pi \in \Pi^{MD}$, define the **policy-evaluation operator**:

$$(\mathrm{L}_\pi v)(s) = r(s, \pi(s)) + \gamma \sum_{j \in \mathcal{S}} p(j|s, \pi(s)) v(j) \tag{348}$$

The **Bellman optimality operator** is then:

$$\mathrm{L}\mathbf{v} \equiv \max_{\pi \in \Pi^{MD}} \{\mathbf{r}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}\} \tag{349}$$

where Π^{MD} is the set of Markov deterministic decision rules, $\mathbf{r}_\pi$ is the reward vector under decision rule π, and $\mathbf{P}_\pi$ is the transition probability matrix under decision rule π.

Note that while we write $\max_{\pi \in \Pi^{MD}}$, we do not implement the above operator by enumerating all decision rules. Rather, the fact that we compare policies based on their value functions in a componentwise fashion means that maximizing over the space of Markovian deterministic rules reduces to the following update in component form:

$$(\mathrm{L}\mathbf{v})(s) = \max_{a \in \mathcal{A}_s} \left\{ r(s, a) + \gamma \sum_{j \in \mathcal{S}} p(j|s, a) v(j) \right\} \tag{350}$$

For convenience, we define the **greedy selector** $\mathrm{Greedy}(v) \in \Pi^{MD}$ that extracts an optimal decision rule from a value function:

$$\mathrm{Greedy}(v)(s) \in \arg\max_{a \in \mathcal{A}_s} \left\{ r(s, a) + \gamma \sum_{j \in \mathcal{S}} p(j|s, a) v(j) \right\} \tag{351}$$

In Puterman's terminology, such a greedy selector is called **v-improving** (or **conserving** when it achieves the maximum). This operator will be useful for expressing algorithms succinctly:

- **Value iteration:** $v_{k+1} = \mathrm{L} v_k$, then extract $\pi = \mathrm{Greedy}(v^*)$
- **Policy iteration:** $\pi_{k+1} = \mathrm{Greedy}(v^{\pi_k})$ with v^{π_k} solving $v = \mathrm{L}_{\pi_k} v$

The equivalence between these two forms can be shown mathematically, as demonstrated in the following proposition and proof.

Proposition 3.2. *The operator* L *defined as a maximization over Markov deterministic decision rules:*

$$(\mathrm{L}\mathbf{v})(s) = \max_{\pi \in \Pi^{MD}} \left\{ r(s, \pi(s)) + \gamma \sum_{j \in \mathcal{S}} p(j|s, \pi(s)) v(j) \right\} \tag{352}$$

is equivalent to the componentwise maximization over actions:

$$(\mathrm{L}\mathbf{v})(s) = \max_{a \in \mathcal{A}_s} \left\{ r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v(j) \right\} \tag{353}$$

Proof. Fix s. Let

$$Q_v(s,a) r(s,a) + \gamma \sum_j p(j \mid s,a)\, v(j). \tag{354}$$

For any rule $\pi \in \Pi^{MD}$, we have $(\mathrm{L}_\pi v)(s) = Q_v(s, \pi(s)) \leq \max_{a \in \mathcal{A}_s} Q_v(s,a)$. Taking the maximum over π gives

$$\max_{\pi \in \Pi^{MD}} (\mathrm{L}_\pi v)(s) \leq \max_{a \in \mathcal{A}_s} Q_v(s,a). \tag{355}$$

Conversely, choose a **greedy selector** $\pi^v \in \Pi^{MD}$ such that for each s,

$$\pi^v(s) \in \arg\max_{a \in \mathcal{A}_s} Q_v(s,a) \tag{356}$$

(possible since $\mathcal{A}_s$ is finite; otherwise use a measurable ε-greedy selector). Then

$$(\mathrm{L}_{\pi^v} v)(s) = Q_v(s, \pi^v(s)) = \max_{a \in \mathcal{A}_s} Q_v(s,a), \tag{357}$$

so $\max_\pi (\mathrm{L}_\pi v)(s) \geq \max_a Q_v(s,a)$. Combining both inequalities yields equality.

□

Algorithms for Solving the Optimality Equations The optimality equations are operator equations. Therefore, we can apply general numerical methods to solve them. Applying the successive approximation method to the Bellman optimality equation yields a method known as "value iteration" in dynamic programming. A direct application of the blueprint for successive approximation yields the following algorithm:

The termination criterion in this algorithm is based on a specific bound that provides guarantees on the quality of the solution. This is in contrast to supervised learning, where we often use arbitrary termination criteria based on computational budget or early stopping when the learning curve flattens. This is because establishing implementable generalization bounds in supervised learning is challenging.

However, in the dynamic programming context, we can derive various bounds that can be implemented in practice. These bounds help us terminate our procedure with a guarantee on the precision of our value function and, correspondingly, on the optimality of the resulting policy.

Proposition 3.3 (Convergence of Value Iteration)**.** *(Adapted from Puterman [1994] theorem 6.3.1)*

Let v_0 be any initial value function, $\varepsilon > 0$ a desired accuracy, and let $\{v_n\}$ be the sequence of value functions generated by value iteration, i.e., $v_{n+1} = \mathrm{L}v_n$ for $n \geq 0$, where L is the Bellman optimality operator. Then:

1. *v_n converges to the optimal value function v_γ^*,*
2. *The algorithm terminates in finite time,*
3. *The resulting policy π_ε is ε-optimal, and*
4. *When the algorithm terminates, v_{n+1} is within $\varepsilon/2$ of v_γ^*.*

Proof. Parts 1 and 2 follow directly from the fact that L is a contraction mapping. Hence, by Banach's fixed-point theorem, it has a unique fixed point (which is v_γ^*), and repeated application of L will converge to this fixed point. Moreover, this convergence happens at a geometric rate, which ensures that we reach the termination condition in finite time.

To show that the Bellman optimality operator L is a contraction mapping, we need to prove that for any two value functions v and u:

$$\|\mathrm{L}v - \mathrm{L}u\|_\infty \leq \gamma \|v - u\|_\infty \tag{358}$$

where $\gamma \in [0, 1)$ is the discount factor and $\|\cdot\|_\infty$ is the supremum norm.

Let's start by writing out the definition of $\mathrm{L}v$ and $\mathrm{L}u$:

$$(\mathrm{L}v)(s) = \max_{a \in A} \left\{ r(s, a) + \gamma \sum_{j \in \mathcal{S}} p(j|s, a) v(j) \right\}$$

$$(\mathrm{L}u)(s) = \max_{a \in A} \left\{ r(s, a) + \gamma \sum_{j \in \mathcal{S}} p(j|s, a) u(j) \right\}$$

For any state s, let a_v be the action that achieves the maximum for $(\mathrm{L}v)(s)$, and a_u be the action that achieves the maximum for $(\mathrm{L}u)(s)$. By the definition of these maximizers:

$$(\mathrm{L}v)(s) \geq r(s, a_u) + \gamma \sum_{j \in \mathcal{S}} p(j|s, a_u) v(j)$$

$$(\mathrm{L}u)(s) \geq r(s, a_v) + \gamma \sum_{j \in \mathcal{S}} p(j|s, a_v) u(j)$$

Subtracting these inequalities:

$$(\mathrm{L}v)(s) - (\mathrm{L}u)(s) \leq \gamma \sum_{j \in \mathcal{S}} p(j|s, a_v)(v(j) - u(j))$$
$$(\mathrm{L}u)(s) - (\mathrm{L}v)(s) \leq \gamma \sum_{j \in \mathcal{S}} p(j|s, a_u)(u(j) - v(j))$$

Taking the absolute value and using the fact that $\sum_{j \in \mathcal{S}} p(j|s, a) = 1$:

$$|(\mathrm{L}v)(s) - (\mathrm{L}u)(s)| \leq \gamma \max_{j \in \mathcal{S}} |v(j) - u(j)| = \gamma \|v - u\|_\infty \tag{359}$$

Since this holds for all $s \in \mathcal{S}$, taking the supremum over s gives:

$$\|\mathrm{L}v - \mathrm{L}u\|_\infty \leq \gamma \|v - u\|_\infty \tag{360}$$

Thus, L is a contraction mapping with contraction factor γ.

Now, let's prove parts 3 and 4. Suppose the algorithm has just terminated, i.e., $\|v_{n+1} - v_n\|_\infty < \frac{\varepsilon(1-\gamma)}{2\gamma}$ for some n. We want to show that our current value function v_{n+1} and the policy π_ε derived from it are close to optimal.

By the triangle inequality:

$$\|v_\gamma^{\pi_\varepsilon} - v_\gamma^*\|_\infty \leq \|v_\gamma^{\pi_\varepsilon} - v_{n+1}\|_\infty + \|v_{n+1} - v_\gamma^*\|_\infty \tag{361}$$

For the first term, since $v_\gamma^{\pi_\varepsilon}$ is the fixed point of $\mathrm{L}_{\pi_\varepsilon}$ and π_ε is greedy with respect to v_{n+1} (i.e., $\mathrm{L}_{\pi_\varepsilon} v_{n+1} = \mathrm{L}v_{n+1}$):

$$\begin{aligned} \|v_\gamma^{\pi_\varepsilon} - v_{n+1}\|_\infty &= \|\mathrm{L}_{\pi_\varepsilon} v_\gamma^{\pi_\varepsilon} - v_{n+1}\|_\infty \\ &\leq \|\mathrm{L}_{\pi_\varepsilon} v_\gamma^{\pi_\varepsilon} - \mathrm{L}_{\pi_\varepsilon} v_{n+1}\|_\infty + \|\mathrm{L}_{\pi_\varepsilon} v_{n+1} - v_{n+1}\|_\infty \\ &= \|\mathrm{L}_{\pi_\varepsilon} v_\gamma^{\pi_\varepsilon} - \mathrm{L}_{\pi_\varepsilon} v_{n+1}\|_\infty + \|\mathrm{L}v_{n+1} - v_{n+1}\|_\infty \\ &\leq \gamma \|v_\gamma^{\pi_\varepsilon} - v_{n+1}\|_\infty + \gamma \|v_{n+1} - v_n\|_\infty \end{aligned} \tag{362}$$

where we used that both L and $\mathrm{L}_{\pi_\varepsilon}$ are contractions with factor γ, and that $v_{n+1} = \mathrm{L}v_n$.

Rearranging:

$$\|v_\gamma^{\pi_\varepsilon} - v_{n+1}\|_\infty \leq \frac{\gamma}{1-\gamma} \|v_{n+1} - v_n\|_\infty \tag{363}$$

Similarly, since v_γ^* is the fixed point of L:

$$\|v_{n+1} - v_\gamma^*\|_\infty = \|\mathrm{L}v_n - \mathrm{L}v_\gamma^*\|_\infty \leq \gamma \|v_n - v_\gamma^*\|_\infty \leq \frac{\gamma}{1-\gamma} \|v_{n+1} - v_n\|_\infty \tag{364}$$

Since $\|v_{n+1} - v_n\|_\infty < \frac{\varepsilon(1-\gamma)}{2\gamma}$:

$$\|v_\gamma^{\pi_\varepsilon} - v_{n+1}\|_\infty \leq \frac{\gamma}{1-\gamma} \cdot \frac{\varepsilon(1-\gamma)}{2\gamma} = \frac{\varepsilon}{2} \tag{365}$$

$$\|v_{n+1} - v_\gamma^*\|_\infty \leq \frac{\gamma}{1-\gamma} \cdot \frac{\varepsilon(1-\gamma)}{2\gamma} = \frac{\varepsilon}{2} \tag{366}$$

Combining these:

$$\|v_\gamma^{\pi_\varepsilon} - v_\gamma^*\|_\infty \leq \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon \tag{367}$$

This completes the proof, showing that v_{n+1} is within $\varepsilon/2$ of v_γ^* (part 4) and π_ε is ε-optimal (part 3).

□

Newton-Kantorovich Applied to Bellman Optimality We now apply the Newton-Kantorovich framework to the Bellman optimality equation. Let

$$(\mathrm{L}v)(s) = \max_{a \in A(s)} \left\{ r(s,a) + \gamma \sum_{s'} p(s' \mid s, a) v(s') \right\}. \tag{368}$$

The problem is to find v such that $\mathrm{L}v = v$, or equivalently $\mathrm{B}(v) := \mathrm{L}v - v = 0$. The operator L is piecewise affine, hence not globally differentiable, but it is directionally differentiable everywhere in the Hadamard sense and Fréchet differentiable at points where the maximizer is unique.

We consider three complementary perspectives for understanding and computing its derivative.

Perspective 1: Max of Affine Maps In tabular form, for finite state and action spaces, the Bellman operator can be written as a pointwise maximum of affine maps:

$$(\mathrm{L}v)(s) = \max_{a \in A(s)} \{ r(s,a) + \gamma (P_a v)(s) \}, \tag{369}$$

where $P_a \in R^{|S| \times |S|}$ is the transition matrix associated with action a. Each $Q_a v := r^a + \gamma P_a v$ is affine in v. The operator L therefore computes the upper envelope of a finite set of affine functions at each state.

At any v, let the **active set** at state s be

$$\mathcal{A}^*(s; v) := \arg\max_{a \in A(s)} (Q_a v)(s). \tag{370}$$

Then the Hadamard directional derivative exists and is given by

$$(\mathrm{L}'(v; h))(s) = \max_{a \in \mathcal{A}^*(s;v)} \gamma (P_a h)(s). \tag{371}$$

If the active set is a singleton, this expression becomes linear in h, and L is Fréchet differentiable at v, with

$$\mathrm{L}'(v) = \gamma P_{\pi_v}, \tag{372}$$

where $\pi_v(s) := a^*(s)$ is the greedy policy at v.

Perspective 2: Envelope Theorem Consider now a value function approximated as a linear combination of basis functions:

$$v_c(s) = \sum_j c_j \phi_j(s). \tag{373}$$

At a node s_i, define the parametric maximization

$$v_i(c) := (\mathrm{L}v_c)(s_i) = \max_{a \in A(s_i)} \left\{ r(s_i, a) + \gamma \sum_j c_j E_{s'|s_i,a}[\phi_j(s')] \right\}. \tag{374}$$

Define

$$F_i(a, c) := r(s_i, a) + \gamma \sum_j c_j E_{s'|s_i,a}[\phi_j(s')], \tag{375}$$

so that $v_i(c) = \max_a F_i(a, c)$. Since F_i is linear in c, we can apply the **envelope theorem** (Danskin's theorem): if the optimizer $a_i^*(c)$ is unique or selected measurably, then

$$\frac{\partial v_i}{\partial c_j}(c) = \gamma E_{s'|s_i,a_i^*(c)}[\phi_j(s')]. \tag{376}$$

We do not need to differentiate the optimizer $a_i^*(c)$ itself. The result extends to the subdifferential case when ties occur, where the Jacobian becomes set-valued.

This result is useful when solving the collocation equation $\Phi c = v(c)$. Newton's method requires the Jacobian $v'(c)$, and this expression allows us to compute it without involving any derivatives of the optimal action.

Perspective 3: The Implicit Function Theorem The third perspective applies the implicit function theorem to understand when the Bellman operator is differentiable despite containing a max operator. The maximization problem defines an implicit relationship between the value function and the optimal action, and the implicit function theorem tells us when this relationship is smooth enough to differentiate through.

The Bellman operator is defined as

$$(\mathrm{L}v)(s) = \max_a \left\{ r(s, a) + \gamma \sum_j p(j \mid s, a) v(j) \right\}. \tag{377}$$

The difficulty is that the max operator encodes a discrete selection: which action achieves the maximum. To apply the implicit function theorem, we reformulate this as follows. For each action a, define the **action-value function**:

$$Q_a(v, s) := r(s, a) + \gamma \sum_j p(j \mid s, a) v(j). \tag{378}$$

The optimal action at v satisfies the **optimality condition**:

$$Q_{a^*(s)}(v, s) \geq Q_a(v, s) \quad \text{for all } a. \tag{379}$$

Now suppose that at a particular v, action $a^*(s)$ is a **strict local maximizer** in the sense that there exists $\delta > 0$ such that

$$Q_{a^*(s)}(v, s) > Q_a(v, s) + \delta \quad \text{for all } a \neq a^*(s). \tag{380}$$

This strict inequality is the regularity condition needed for the implicit function theorem. It ensures that the optimal action is unique at v and remains so in a neighborhood of v.

To see why, consider any perturbation $v + h$ with $\|h\|$ small. Since Q_a is linear in v, we have:

$$Q_a(v + h, s) = Q_a(v, s) + \gamma \sum_j p(j \mid s, a)h(j). \tag{381}$$

The perturbation term is bounded: $|\gamma \sum_j p(j \mid s, a)h(j)| \leq \gamma \|h\|$. Therefore, for $\|h\| < \delta/\gamma$, the strict gap ensures that

$$Q_{a^*(s)}(v + h, s) > Q_a(v + h, s) \quad \text{for all } a \neq a^*(s). \tag{382}$$

Thus $a^*(s)$ remains the unique maximizer throughout the neighborhood $\{v + h : \|h\| < \delta/\gamma\}$.

The implicit function theorem now applies: in this neighborhood, the mapping $v \mapsto a^*(s; v)$ is **constant** (and hence smooth), taking the value $a^*(s)$. This allows us to write

$$(\mathrm{L}v)(s) = Q_{a^*(s)}(v, s) = r(s, a^*(s)) + \gamma \sum_j p(j \mid s, a^*(s))v(j) \tag{383}$$

as an explicit formula that holds throughout the neighborhood. Since $Q_{a^*(s)}(\cdot, s)$ is an affine (hence smooth) function of v, we can differentiate it:

$$\frac{d}{dv}(\mathrm{L}v)(s) = \gamma P_{a^*(s)}. \tag{384}$$

More precisely, for any perturbation h:

$$(\mathrm{L}(v + h))(s) = (\mathrm{L}v)(s) + \gamma \sum_j p(j \mid s, a^*(s))h(j) + o(\|h\|). \tag{385}$$

This is the Fréchet derivative:

$$\mathrm{L}'(v) = \gamma P_{\pi_v}, \tag{386}$$

where $\pi_v(s) = a^*(s)$ is the greedy policy.

The role of the implicit function theorem: It guarantees that when the maximizer is unique with a strict gap (the regularity condition), the argmax function $v \mapsto a^*(s; v)$ is locally constant, which removes the non-differentiability of the max operator. Without this regularity condition (specifically, at points where multiple actions tie for optimality), the implicit function theorem does not apply, and the operator is not Fréchet differentiable. The active set perspective (Perspective 1) and the envelope theorem (Perspective 2) provide the tools to handle these non-smooth points.

Connection to Policy Iteration We return to the Newton-Kantorovich step:

$$(I - \mathrm{L}'(v_n))h_n = v_n - \mathrm{L}v_n, \quad v_{n+1} = v_n - h_n. \tag{387}$$

Suppose $\mathrm{L}'(v_n) = \gamma P_{\pi_{v_n}}$ for the greedy policy π_{v_n}. Then

$$(I - \gamma P_{\pi_{v_n}})v_{n+1} = r^{\pi_{v_n}}, \tag{388}$$

which is exactly policy evaluation for π_{v_n}. Recomputing the greedy policy from v_{n+1} yields the next iterate.

Thus, **policy iteration is Newton-Kantorovich** applied to the Bellman optimality equation. At points of nondifferentiability (when ties occur), the operator is still semismooth, and policy iteration corresponds to a semismooth Newton method. The envelope theorem is what justifies the simplification of the Jacobian to γP_{π_v}, bypassing the need to differentiate through the optimizer. This completes the equivalence.

The Semismooth Newton Perspective The three perspectives we developed above (the active set view, the envelope theorem, and the implicit function theorem) all point toward a deeper framework for understanding Newton-type methods on non-smooth operators. This framework, known as semismooth Newton methods, was developed precisely to handle operators like the Bellman operator that are piecewise smooth but not globally differentiable. The connection between policy iteration and semismooth Newton methods has been rigorously developed in recent work Gargiani et al. [2022].

The classical Newton-Kantorovich method assumes the operator is Fréchet differentiable everywhere. The derivative exists, is unique, and varies continuously with the base point. But the Bellman operator L violates this assumption at any value function where multiple actions tie for optimality at some state. At such points, the implicit function theorem fails, and there is no unique Fréchet derivative.

Semismooth Newton methods address this by replacing the notion of a single Jacobian with a generalized derivative that captures the behavior of the operator near non-smooth points. The most commonly used generalized derivative is the Clarke subdifferential, which we can think of as the convex hull of all possible "candidate Jacobians" that arise from limits approaching the non-smooth point from different directions.

For the Bellman residual $\mathrm{B}(v) = \mathrm{L}v - v$, the Clarke subdifferential at a point v can be characterized explicitly using our first perspective. Recall that at each state s, we defined the active set $\mathcal{A}^*(s; v) = \arg\max_a Q_a(v, s)$. When this set contains multiple actions, the operator is not Fréchet differentiable. However, it remains directionally differentiable in all directions, and the Clarke subdifferential consists of all matrices of the form

$$\partial \mathrm{B}(v) = \{I - \gamma P_\pi : \pi(s) \in \mathcal{A}^*(s; v) \text{ for all } s\}. \tag{389}$$

In words, the generalized Jacobian is the set of all matrices $I - \gamma P_\pi$ where π is any policy that selects an action from the active set at each state. When the maximizer is unique everywhere, this set reduces to a singleton, and we recover the classical Fréchet derivative. When ties occur, the set has multiple elements: precisely the convex combinations mentioned in Perspective 1.

The semismooth Newton method for solving $\mathrm{B}(v) = 0$ proceeds by selecting an element $J_k \in \partial \mathrm{B}(v_k)$ at each iteration and solving

$$J_k h_k = -\mathrm{B}(v_k), \quad v_{k+1} = v_k + h_k. \tag{390}$$

What this tells us is that any choice from the Clarke subdifferential yields a valid Newton-like update. In the context of the Bellman equation, choosing $J_k = I - \gamma P_{\pi_k}$ where π_k is any greedy policy corresponds exactly to the policy evaluation step in policy iteration. The freedom in selecting which action to choose when ties occur translates to the freedom in selecting which element of the subdifferential to use.

Under appropriate regularity conditions (specifically, when the residual function is BD-regular or CD-regular), the semismooth Newton method converges locally at a quadratic rate Gargiani et al. [2022]. This means that near the solution, the error decreases quadratically:

$$\|v_{k+1} - v^*\| \leq C\|v_k - v^*\|^2. \tag{391}$$

This theoretical result explains an empirical observation that has long been noted in practice: policy iteration typically converges in very few iterations, often just a handful, even when the state and action spaces are enormous and the space of possible policies is exponentially large.

The semismooth Newton framework also suggests a spectrum of methods interpolating between value iteration and policy iteration. Value iteration can be interpreted as a Newton-like method where we choose $J_k = I$ at every iteration, ignoring the dependence of L on v entirely. This choice guarantees global convergence through the contraction property but sacrifices the quadratic local convergence rate. Policy iteration, at the other extreme, uses the full generalized Jacobian $J_k = I - \gamma P_{\pi_k}$, achieving quadratic convergence but at the cost of solving a linear system at each iteration.

Between these extremes lie methods that use approximate Jacobians. One natural variant is to choose $J_k = \alpha I$ for some scalar $\alpha > 1$. This leads to the update

$$v_{k+1} = \frac{\alpha - 1}{\alpha} v_k + \frac{1}{\alpha} \mathrm{L} v_k. \tag{392}$$

This is known as α-value iteration or successive over-relaxation when $\alpha > 1$. For appropriate choices of α, this method retains global convergence while achieving better local rates than standard value iteration, and it requires only pointwise operations rather than solving a linear system. The Newton perspective thus unifies existing algorithms and generates new ones by systematically exploring different approximations to the generalized Jacobian.

The connection to semismooth Newton methods places policy iteration within a broader mathematical framework that extends far beyond dynamic programming. Semismooth Newton methods are used in optimization (for complementarity problems and variational inequalities), in PDE-constrained optimization (for problems with control constraints), and in economics (for equilibrium problems). The Bellman equation, viewed through this lens, is simply one instance of a piecewise smooth equation, and the tools developed for such equations apply directly.

Policy Iteration While we derived policy iteration-like steps from the Newton-Kantorovich method, it's worth examining policy iteration as a standalone algorithm, as it has been traditionally presented in the field of dynamic programming.

The policy iteration algorithm for discounted Markov decision problems is as follows:

As opposed to value iteration, this algorithm produces a sequence of both deterministic Markovian decision rules $\{\pi_n\}$ and value functions $\{\mathbf{v}^n\}$. We recognize in this algorithm the linearization step of the Newton-Kantorovich procedure, which takes place here in the policy evaluation step 3 where we solve the linear system $(\mathbf{I} - \gamma \mathbf{P}_{\pi_n})\mathbf{v} = \mathbf{r}_{\pi_n}$. In practice, this linear system could be solved either using direct methods (eg. Gaussian elimination), using simple iterative methods such as the successive approximation method for policy evaluation, or more sophisticated methods such as GMRES.

4 Approximate Dynamic Programming

Dynamic programming methods suffer from the curse of dimensionality and can quickly become difficult to apply in practice. We may also be dealing with large or continuous state or action spaces. We have seen so far that we could address this problem using discretization, or interpolation. These were already examples of approximate dynamic programming. In this chapter, we will see other forms of approximations meant to facilitate the optimization problem, either by approximating the optimality equations, the value function, or the policy itself. Approximation theory is at the heart of learning methods, and fundamentally, this chapter will be about the application of learning ideas to solve complex decision-making problems.

4.1 Smooth Bellman Optimality Equations

While the standard Bellman optimality equations use the max operator to determine the best action, an alternative formulation known as the smooth or soft Bellman optimality equations replaces this with a softmax operator. This approach originated from Rust [1987] and was later rediscovered in the context of maximum entropy inverse reinforcement learning Ziebart et al. [2008], which then led to soft Q-learning Haarnoja et al. [2017] and soft actor-critic Haarnoja et al. [2018], a state-of-the-art deep reinforcement learning algorithm.

In the infinite-horizon setting, the smooth Bellman optimality equations take the form:

$$v^\star_\gamma(s) = \frac{1}{\beta} \log \sum_{a \in A_s} \exp\left(\beta \left(r(s,a) + \gamma \sum_{j \in S} p(j|s,a) v^\star_\gamma(j)\right)\right) \tag{393}$$

Adopting an operator-theoretic perspective, we can define a nonlinear operator L_β such that the smooth value function of an MDP is then the solution to the following fixed-point equation:

$$(\mathrm{L}_\beta \mathbf{v})(s) = \frac{1}{\beta} \log \sum_{a \in \mathcal{A}_s} \exp\left(\beta \left(r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v(j)\right)\right) \tag{394}$$

As $\beta \to \infty$, L_β converges to the standard Bellman operator L. Furthermore, it can be shown that the smooth Bellman operator is a contraction mapping in the supremum norm, and therefore has a unique fixed point. However, as opposed to the usual "hard" setting, the fixed point of L_β is associated with the value function of an optimal stochastic policy defined by the softmax distribution:

$$\pi(a|s) = \frac{\exp\left(\beta \left(r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v^\star_\gamma(j)\right)\right)}{\sum_{a' \in \mathcal{A}_s} \exp\left(\beta \left(r(s,a') + \gamma \sum_{j \in \mathcal{S}} p(j|s,a') v^\star_\gamma(j)\right)\right)} \tag{395}$$

Despite the confusing terminology, the above "softmax" policy is simply the smooth counterpart to the argmax operator in the original optimality equation: it acts as a soft-argmax.

This formulation is interesting for several reasons. First, smoothness is a desirable property from an optimization standpoint. Unlike γ, we view β as a hyperparameter of our algorithm, which we can control to achieve the desired level of accuracy.

Second, while presented from an intuitive standpoint where we replace the max by the log-sum-exp (a smooth maximum) and the argmax by the softmax (a smooth argmax), this formulation can also be obtained from various other perspectives, offering theoretical tools and solution methods. For example, Rust [1987] derived this algorithm by considering a setting in which the rewards are stochastic and perturbed by a Gumbel noise variable. When considering the corresponding augmented state space and integrating the noise, we obtain smooth equations. This interpretation is leveraged by Rust for modeling purposes.

Smooth Value Iteration Algorithm The smooth value iteration algorithm replaces the max operator in standard value iteration with the logsumexp operator. Here's the algorithm structure:

Differences from standard value iteration:

- Line 8 uses $\frac{1}{\beta} \log \sum_a \exp(\beta \cdot q(s, a))$ instead of $\max_a q(s, a)$
- Line 15 extracts a stochastic policy using softmax instead of a deterministic argmax policy
- As $\beta \to \infty$, the algorithm converges to standard value iteration
- Lower β values produce more stochastic policies with higher entropy

There is also a way to obtain this equation by starting from the energy-based formulation often used in supervised learning, in which we convert an unnormalized probability distribution into a distribution using the softmax transformation. This is essentially what Ziebart et al. [2008] did in their paper. Furthermore, this perspective bridges with the literature on probabilistic graphical models, in which we can now cast the problem of finding an optimal smooth policy into one of maximum likelihood estimation (an inference problem). This is the idea of control as inference, which also admits the converse - that of inference as control - used nowadays for deriving fast samples and amortized inference techniques using reinforcement learning Levine et al. [2018].

Finally, it's worth noting that we can also derive this form by considering an entropy-regularized formulation in which we penalize for the entropy of our policy in the reward function term. This formulation admits a solution that coincides with the smooth Bellman equations Haarnoja et al. [2017].

4.1.1 Gumbel Noise on the Rewards

We can obtain the smooth Bellman equation by considering a setting in which we have Gumbel noise added to the reward function. This derivation provides both theoretical insight and connects to practical modeling scenarios where rewards have random perturbations.

Step 1: Define the Augmented MDP with Gumbel Noise At each time period and state s, we draw an **action-indexed shock vector**:

$$\boldsymbol{\epsilon}_t(s) = \big(\epsilon_t(s,a)\big)_{a\in\mathcal{A}_s}, \quad \text{where } \epsilon_t(s,a) \text{ i.i.d.} \sim \text{Gumbel}(\mu_\epsilon, 1/\beta) \tag{396}$$

These shocks are independent across time periods, states, and actions, and are independent of the MDP transition dynamics $p(\cdot|s,a)$.

The **Gumbel distribution** with location parameter μ and scale parameter $1/\beta$ has probability density function:

$$f(x;\mu,\beta) = \beta \exp\left(-\beta(x-\mu) - \exp(-\beta(x-\mu))\right) \tag{397}$$

To generate a Gumbel-distributed random variable, we can use inverse transform sampling: $X = \mu - \frac{1}{\beta}\ln(-\ln(U))$ where U is uniform on $(0,1)$.

> **Zero-Mean Shocks**
> To ensure the shocks have zero mean, we set $\mu_\epsilon = -\gamma_E/\beta$ where $\gamma_E \approx 0.5772$ is the Euler-Mascheroni constant. This choice eliminates an additive constant that would otherwise appear in the smooth Bellman equation. For simplicity, we will adopt this convention throughout.

We now define an **augmented MDP** with:

- **Augmented state**: $\tilde{s} = (s, \boldsymbol{\epsilon})$ where $s \in \mathcal{S}$ and $\boldsymbol{\epsilon} \in R^{|\mathcal{A}_s|}$
- **Augmented reward**: $\tilde{r}(\tilde{s}, a) = r(s,a) + \epsilon(a)$
- **Augmented transition**: $\tilde{p}(\tilde{s}'|\tilde{s}, a) = p(s'|s,a) \cdot p(\boldsymbol{\epsilon}')$

The transition factorizes because the next shock vector $\boldsymbol{\epsilon}'$ is drawn independently of the current state and action (conditional independence).

> **The Augmented State Space is Infinite-Dimensional**
> Even if the original state space $\mathcal{S}$ and action space $\mathcal{A}$ are finite, the augmented state space $\tilde{\mathcal{S}} = \mathcal{S} \times R^{|\mathcal{A}|}$ is **uncountably infinite** because each shock vector $\boldsymbol{\epsilon}$ is a continuous random variable. Therefore:
>
> - We cannot enumerate the augmented states

- Tabular dynamic programming methods do not apply directly
- The augmented value function $\tilde{v}(s, \boldsymbol{\epsilon})$ maps a continuous space to R

This motivates why we immediately marginalize over the shocks to obtain a finite-dimensional representation.

Step 2: The Hard Bellman Equation on the Augmented State Space The Bellman optimality equation for the augmented MDP is:

$$\tilde{v}(s, \boldsymbol{\epsilon}) = \max_{a \in \mathcal{A}_s} \{r(s, a) + \epsilon(a) + \gamma E_{s', \boldsymbol{\epsilon}'} [\tilde{v}(s', \boldsymbol{\epsilon}') \mid s, a]\} \tag{398}$$

Here the expectation is over the **next augmented state** $(s', \boldsymbol{\epsilon}')$, which includes both the next state $s' \sim p(\cdot|s, a)$ and the next shock vector $\boldsymbol{\epsilon}' \sim p(\cdot)$.

This is a perfectly well-defined Bellman equation, and an optimal stationary policy exists:

$$\pi(s, \boldsymbol{\epsilon}) \in \text{argmax}_{a \in \mathcal{A}_s} \{r(s, a) + \epsilon(a) + \gamma E_{s', \boldsymbol{\epsilon}'} [\tilde{v}(s', \boldsymbol{\epsilon}') \mid s, a]\} \tag{399}$$

However, this equation is **computationally intractable** because:

- The state space is continuous and infinite-dimensional
- The shocks are fresh each period
- We would need to solve for $\tilde{v}$ over an uncountable domain

We never solve this equation directly. Instead, we use it as a mathematical device to derive the smooth Bellman equation.

Step 3: Define the Ex-Ante (Inclusive) Value Function The idea here is to consider the **expected value before observing the current shocks**. We define what some authors in econometrics call the **inclusive value** or **ex-ante value**:

$$v(s) := E_{\boldsymbol{\epsilon}}\big[\tilde{v}(s, \boldsymbol{\epsilon})\big] \tag{400}$$

This is the value of being in state s **before** we observe the current-period shock vector $\boldsymbol{\epsilon}$.

Two Different Value Functions
It is crucial to distinguish:

- $\tilde{v}(s, \boldsymbol{\epsilon})$: the value **after** observing shocks (conditional on $\boldsymbol{\epsilon}$), defined on the augmented state space

- $v(s)$: the value **before** observing shocks (marginalizing over $\boldsymbol{\epsilon}$), defined on the original state space

The function $v(s)$ is what we actually compute and care about. The augmented value $\tilde{v}$ exists only as a proof device.

Step 4: Separate the Deterministic and Random Components Now we take the expectation of the augmented Bellman equation with respect to the **current shocks only** (everything that does not depend on the current $\boldsymbol{\epsilon}$ can be pulled out).

First, note that by the law of iterated expectations and independence of shocks across time:

$$E_{\boldsymbol{\epsilon}'}\left[\tilde{v}(s', \boldsymbol{\epsilon}')\right] = v(s') \tag{401}$$

This follows from our definition of v and the fact that the next shock is independent of everything else.

Now define the **deterministic part** of the right-hand side:

$$x_a(s) := r(s, a) + \gamma \sum_{j \in \mathcal{S}} p(j|s, a)v(j) \tag{402}$$

This is the expected return from taking action a in state s **without the shock**. Using this notation, the augmented Bellman equation becomes:

$$\tilde{v}(s, \boldsymbol{\epsilon}) = \max_{a \in \mathcal{A}_s} \{x_a(s) + \epsilon(a)\} \tag{403}$$

Taking the expectation over $\boldsymbol{\epsilon}$ on both sides:

$$v(s) = E_{\boldsymbol{\epsilon}}\left[\max_{a \in \mathcal{A}_s} \{x_a(s) + \epsilon(a)\}\right] \tag{404}$$

Expectation of a Max, Not Max of an Expectation
Notice carefully: we have $E[\max(\cdot)]$, **not** $\max E[\cdot]$. We are **not** swapping max and expectation.

The expression $E_{\boldsymbol{\epsilon}}[\max_a\{x_a + \epsilon(a)\}]$ is the expected value of the maximum of Gumbel-perturbed utilities. The Gumbel random utility identity evaluates this quantity in closed form.

Step 5: Apply the Gumbel Random Utility Identity We now invoke a result from extreme value theory:

Lemma 4.1 (Gumbel Random Utility Identity)**.** *Let $\epsilon_1, \ldots, \epsilon_m$ be i.i.d.* Gumbel$(\mu_\epsilon, 1/\beta)$ *random variables. For any deterministic values $x_1, \ldots, x_m \in R$:*

$$\max_{i=1,\ldots,m} \{x_i + \epsilon_i\} \stackrel{d}{=} \frac{1}{\beta} \log \sum_{i=1}^{m} \exp(\beta x_i) + \zeta \tag{405}$$

where $\zeta \sim$ Gumbel$(\mu_\epsilon, 1/\beta)$ (same distribution as the original shocks). Taking expectations:

$$E\left[\max_{i=1,\ldots,m} \{x_i + \epsilon_i\}\right] = \frac{1}{\beta} \log \sum_{i=1}^{m} \exp(\beta x_i) + \mu_\epsilon + \frac{\gamma_E}{\beta} \tag{406}$$

where $\gamma_E \approx 0.5772$ is the Euler-Mascheroni constant.
With mean-zero shocks *($\mu_\epsilon = -\gamma_E/\beta$), the constant term vanishes:*

$$E\left[\max_{i=1,\ldots,m} \{x_i + \epsilon_i\}\right] = \frac{1}{\beta} \log \sum_{i=1}^{m} \exp(\beta x_i) \tag{407}$$

Applying this identity to our problem (with mean-zero shocks):

$$v(s) = E_{\boldsymbol{\epsilon}}\left[\max_{a \in \mathcal{A}_s} \{x_a(s) + \epsilon(a)\}\right] = \frac{1}{\beta} \log \sum_{a \in \mathcal{A}_s} \exp(\beta x_a(s)) \tag{408}$$

Substituting the definition of $x_a(s)$:

$$v(s) = \frac{1}{\beta} \log \sum_{a \in \mathcal{A}_s} \exp\left(\beta \left(r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v(j)\right)\right) \tag{409}$$

We have arrived at the **smooth Bellman equation**.

Step 6: Summary of the Derivation To recap the logical flow:

1. We constructed an augmented MDP with state $(s, \boldsymbol{\epsilon})$ where shocks perturb rewards
2. We wrote the standard Bellman equation for this augmented MDP (hard max, but over an infinite-dimensional state space)
3. We defined the ex-ante value $v(s) = E_{\boldsymbol{\epsilon}}[\tilde{v}(s, \boldsymbol{\epsilon})]$ to eliminate the continuous shock component
4. We separated deterministic and random terms: $\tilde{v}(s, \boldsymbol{\epsilon}) = \max_a \{x_a(s) + \epsilon(a)\}$
5. We applied the Gumbel identity to evaluate $E_{\boldsymbol{\epsilon}}[\max_a \{\cdots\}]$ in closed form as a log-sum-exp

The augmented MDP with shocks exists **only as a mathematical device**. We never approximate $\tilde{v}$, never discretize $\boldsymbol{\epsilon}$, and never enumerate the augmented state space. The only computational object we work with is $v(s)$ on the original (finite) state space, which satisfies the smooth Bellman equation.

Deriving the Optimal Smooth Policy Now that we have derived the smooth value function, we can also obtain the corresponding optimal policy. The question is: **what policy should we follow in the original MDP (without explicitly conditioning on shocks)?**

In the augmented MDP, the optimal policy is deterministic but depends on the shock realization:

$$\pi(s, \boldsymbol{\epsilon}) \in \text{argmax}_{a \in \mathcal{A}_s} \left\{ r(s,a) + \epsilon(a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v(j) \right\} \tag{410}$$

However, we want a policy for the **original** state space s (not the augmented state). We obtain this by **marginalizing over the current shocks**, essentially asking: "what is the probability that action a is optimal when we average over all possible shock realizations?"

Define an indicator function:

$$I_a(\boldsymbol{\epsilon}) = \begin{cases} 1 & \text{if } a \in \text{argmax}_{a' \in \mathcal{A}_s} \{x_{a'}(s) + \epsilon(a')\} \\ 0 & \text{otherwise} \end{cases} \tag{411}$$

where $x_a(s) = r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v(j)$ as before.

The ex-ante probability that action a is optimal at state s is:

$$\pi(a|s) = E_{\boldsymbol{\epsilon}}[I_a(\boldsymbol{\epsilon})] = P_{\boldsymbol{\epsilon}} \left(a \in \text{argmax}_{a'} \{x_{a'}(s) + \epsilon(a')\} \right) \tag{412}$$

This is the probability that action a achieves the maximum when utilities are perturbed by Gumbel noise.

Lemma 4.2 (Gumbel-Max Probability (Softmax))**.** *Let $\epsilon_1, \ldots, \epsilon_m$ be i.i.d.* Gumbel$(\mu_\epsilon, 1/\beta)$ *random variables. For any deterministic values $x_1, \ldots, x_m \in R$, the probability that index i achieves the maximum is:*

$$P\left(i \in \text{argmax}_j \{x_j + \epsilon_j\}\right) = \frac{\exp(\beta x_i)}{\sum_{j=1}^m \exp(\beta x_j)} \tag{413}$$

This holds regardless of the location parameter μ_ϵ.

Applying this result to our problem:

$$\pi(a|s) = \frac{\exp\left(\beta x_a(s)\right)}{\sum_{a' \in \mathcal{A}_s} \exp\left(\beta x_{a'}(s)\right)} = \frac{\exp\left(\beta \left(r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v(j) \right)\right)}{\sum_{a' \in \mathcal{A}_s} \exp\left(\beta \left(r(s,a') + \gamma \sum_{j \in \mathcal{S}} p(j|s,a') v(j) \right)\right)} \tag{414}$$

This is the **softmax policy** or **Gibbs/Boltzmann policy** with inverse temperature β.

Properties:

- As $\beta \to \infty$: the policy becomes deterministic, concentrating on the action(s) with highest $x_a(s)$ (recovers standard greedy policy)
- As $\beta \to 0$: the policy becomes uniform over all actions (maximum entropy)
- For finite $\beta > 0$: the policy is stochastic, with probability mass proportional to exponentiated Q-values

This completes the derivation: the smooth Bellman equation yields a value function $v(s)$, and the corresponding optimal policy is the softmax over Q-values.

4.1.2 Regularized Markov Decision Processes

Regularized MDPs Geist et al. [2019] provide another perspective on how the smooth Bellman equations come to be. This framework offers a more general approach in which we seek to find optimal policies under the infinite horizon criterion while also accounting for a regularizer that influences the kind of policies we try to obtain.

Let's set up some necessary notation. First, recall that the policy evaluation operator for a stationary policy with decision rule π is defined as:

$$\mathrm{L}_\pi \mathbf{v} = \mathbf{r}_\pi + \gamma \mathbf{P}_\pi \mathbf{v} \tag{415}$$

where $\mathbf{r}_\pi$ is the expected reward vector under policy π, γ is the discount factor, and $\mathbf{P}_\pi$ is the state transition probability matrix under π. A complementary object to the value function is the q-function (or Q-factor) representation:

$$q_\gamma^\pi(s,a) = r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v_\gamma^\pi(j)$$
$$v_\gamma^\pi(s) = \sum_{a \in \mathcal{A}_s} \pi(a|s) q_\gamma^\pi(s,a)$$

The policy evaluation operator can then be written in terms of the q-function as:

$$[\mathrm{L}_\pi v](s) = \langle \pi(\cdot|s), q(s,\cdot) \rangle \tag{416}$$

Legendre-Fenchel Transform The workhorse behind the theory of regularized MDPs is the Legendre-Fenchel transform, also known as the convex conjugate. For a strongly convex function $\Omega : \Delta_{\mathcal{A}} \to R$, its Legendre-Fenchel transform $\Omega^* : R^{\mathcal{A}} \to R$ is defined as:

$$\Omega^*(q(s,\cdot)) = \max_{\pi(\cdot|s) \in \Delta_{\mathcal{A}}} \langle \pi(\cdot|s), q(s,\cdot) \rangle - \Omega(\pi(\cdot|s)) \tag{417}$$

An important property of this transform is that it has a unique maximizing argument, given by the gradient of Ω^*. This gradient is Lipschitz and satisfies:

$$\nabla\Omega^*(q(s,\cdot)) = \arg\max_{\pi} \langle \pi(\cdot|s), q(s,\cdot)\rangle - \Omega(\pi(\cdot|s)) \tag{418}$$

An important example of a regularizer is the negative entropy, which gives rise to the smooth Bellman equations as we are about to see.

4.1.3 Regularized Bellman Operators

With these concepts in place, we can now define the regularized Bellman operators:

1. **Regularized Policy Evaluation Operator** ($\mathrm{L}_{\pi,\Omega}$):

$$[\mathrm{L}_{\pi,\Omega} v](s) = \langle q(s,\cdot), \pi(\cdot|s)\rangle - \Omega(\pi(\cdot|s)) \tag{419}$$

2. **Regularized Bellman Optimality Operator** (L_{Ω}):

$$[\mathrm{L}_{\Omega} v](s) = [\max_{\pi} \mathrm{L}_{\pi,\Omega} v](s) = \Omega^*(q(s,\cdot)) \tag{420}$$

It can be shown that the addition of a regularizer in these regularized operators still preserves the contraction properties, and therefore the existence of a solution to the optimality equations and the convergence of successive approximation.

The regularized value function of a stationary policy with decision rule π, denoted by $v_{\pi,\Omega}$, is the unique fixed point of the operator equation:

$$\text{find } v \text{ such that } \quad v = \mathrm{L}_{\pi,\Omega} v \tag{421}$$

Under the usual assumptions on the discount factor and the boundedness of the reward, the value of a policy can also be found in closed form by solving for $\mathbf{v}$ in the linear system of equations:

$$(\mathbf{I} - \gamma \mathbf{P}_{\pi})\mathbf{v} = \mathbf{r}_{\pi} - \boldsymbol{\Omega}_{\pi} \tag{422}$$

where $[\boldsymbol{\Omega}_{\pi}](s) = \Omega(\pi(\cdot|s))$ is the vector of regularization terms at each state. The associated state-action value function $q_{\pi,\Omega}$ is given by:

$$q_{\pi,\Omega}(s,a) = r(s,a) + \sum_{j\in\mathcal{S}} \gamma p(j|s,a) v_{\pi,\Omega}(j)$$
$$v_{\pi,\Omega}(s) = \sum_{a\in\mathcal{A}_s} \pi(a|s) q_{\pi,\Omega}(s,a) - \Omega(\pi(\cdot|s))$$

The regularized optimal value function v^*_{Ω} is then the unique fixed point of L_{Ω} in the fixed point equation:

$$\text{find } v \text{ such that } v = \mathrm{L}_{\Omega} v \tag{423}$$

The associated state-action value function q^*_Ω is given by:

$$q^*_\Omega(s,a) = r(s,a) + \sum_{j \in \mathcal{S}} \gamma p(j|s,a) v^*_\Omega(j)$$
$$v^*_\Omega(s) = \Omega^*(q^*_\Omega(s,\cdot))$$

An important result in the theory of regularized MDPs is that there exists a unique optimal regularized policy. Specifically, if π^*_Ω is a conserving decision rule (i.e., $\pi^*_\Omega = \arg\max_\pi \mathrm{L}_{\pi,\Omega} v^*_\Omega$), then the randomized stationary policy $\boldsymbol{\pi} = \mathrm{const}(\pi^*_\Omega)$ is the unique optimal regularized policy.

In practice, once we have found v^*_Ω, we can derive the optimal decision rule by taking the gradient of the convex conjugate evaluated at the optimal action-value function:

$$\pi^*(\cdot|s) = \nabla \Omega^*(q^*_\Omega(s,\cdot)) \tag{424}$$

Recovering the Smooth Bellman Equations Under this framework, we can recover the smooth Bellman equations by choosing Ω to be the negative entropy, and obtain the softmax policy as the gradient of the convex conjugate. Let's show this explicitly:

1. Using the negative entropy regularizer:

$$\Omega(d(\cdot|s)) = \sum_{a \in \mathcal{A}_s} d(a|s) \ln d(a|s) \tag{425}$$

2. The convex conjugate:

$$\Omega^*(q(s,\cdot)) = \ln \sum_{a \in \mathcal{A}_s} \exp q(s,a) \tag{426}$$

3. Now, let's write out the regularized Bellman optimality equation:

$$v^*_\Omega(s) = \Omega^*(q^*_\Omega(s,\cdot)) \tag{427}$$

4. Substituting the expressions for Ω^* and q^*_Ω:

$$v^*_\Omega(s) = \ln \sum_{a \in \mathcal{A}_s} \exp \left(r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v^*_\Omega(j) \right) \tag{428}$$

This matches the form of the smooth Bellman equation we derived earlier, with the log-sum-exp operation replacing the max operation of the standard Bellman equation.

Furthermore, the optimal policy is given by the gradient of Ω^*:

$$d^*(a|s) = \nabla\Omega^*(q^*_\Omega(s,\cdot)) = \frac{\exp(q^*_\Omega(s,a))}{\sum_{a'\in\mathcal{A}_s}\exp(q^*_\Omega(s,a'))} \tag{429}$$

This is the familiar softmax policy we encountered in the smooth MDP setting.

Smooth Policy Iteration Algorithm Now that we've seen how the regularized MDP framework leads to smooth Bellman equations, we present smooth policy iteration. Unlike value iteration which directly iterates the Bellman operator, policy iteration alternates between policy evaluation and policy improvement steps.

Key properties of smooth policy iteration:

1. **Entropy-regularized evaluation**: The policy evaluation step (line 12 of Algorithm Algorithm **??**) accounts for the entropy bonus $\alpha H(\pi(\cdot|s))$ where $\alpha = 1/\beta$
2. **Stochastic policy improvement**: The policy improvement step (lines 12-14 of Algorithm Algorithm **??**) uses softmax instead of deterministic argmax, producing a stochastic policy
3. **Temperature parameter**:
   - Higher $\beta \to policiesclosertodeterministic(lowerentropy)Lower\beta \to morestochasticpolicies(higherentropy)$
   - As $\beta \to \infty \to recoversstandardpolicyiteration$
4. **Convergence**: Like standard policy iteration, this algorithm converges to the unique optimal regularized value function and policy

Equivalence Between Smooth Bellman Equations and Entropy-Regularized MDPs We have now seen two distinct ways to arrive at smooth Bellman equations. Earlier in this chapter, we introduced the logsumexp operator as a smooth approximation to the max operator, motivated by analytical tractability and the desire for differentiability. Just now, we derived the same equations through the lens of regularized MDPs, where we explicitly penalize the entropy of policies. These two perspectives are mathematically equivalent: solving the smooth Bellman equation with inverse temperature parameter β yields exactly the same optimal value function and optimal policy as solving the entropy-regularized MDP with regularization strength $\alpha = 1/\beta$. The two formulations are not merely similar. They describe identical optimization problems.

To see this equivalence clearly, consider the standard MDP problem with rewards $r(s,a)$ and transition probabilities $p(j|s,a)$. The regularized MDP framework tells us to solve:

$$\max_\pi E_\pi\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)\right] + \alpha E_\pi\left[\sum_{t=0}^{\infty}\gamma^t H(\pi(\cdot|s_t))\right], \tag{430}$$

where $H(\pi(\cdot|s)) = -\sum_a \pi(a|s) \ln \pi(a|s)$ is the entropy of the policy at state s, and $\alpha > 0$ is the entropy regularization strength.

We can rewrite this objective by absorbing the entropy term into a modified reward function. Define the entropy-augmented reward:

$$\tilde{r}(s, a, \pi) = r(s, a) + \alpha H(\pi(\cdot|s)). \tag{431}$$

However, this formulation makes the reward depend on the entire policy at each state, which is awkward. We can reformulate this more cleanly by expanding the entropy term. Recall that the entropy is:

$$H(\pi(\cdot|s)) = -\sum_a \pi(a|s) \ln \pi(a|s). \tag{432}$$

When we take the expectation over actions drawn from π, we have:

$$E_{a\sim\pi(\cdot|s)}[H(\pi(\cdot|s))] = \sum_a \pi(a|s) \left[-\sum_{a'} \pi(a'|s) \ln \pi(a'|s) \right] = -\sum_{a'} \pi(a'|s) \ln \pi(a'|s), \tag{433}$$

since the entropy doesn't depend on which action is actually sampled. But we can also write this as:

$$H(\pi(\cdot|s)) = -\sum_a \pi(a|s) \ln \pi(a|s) = E_{a\sim\pi(\cdot|s)}[-\ln \pi(a|s)]. \tag{434}$$

This shows that adding $\alpha H(\pi(\cdot|s))$ to the expected reward at state s is equivalent to adding $-\alpha \ln \pi(a|s)$ to the reward of taking action a at state s. More formally:

$$\begin{aligned}
&E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right] + \alpha E_\pi \left[\sum_{t=0}^{\infty} \gamma^t H(\pi(\cdot|s_t)) \right] \\
&= E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right] + \alpha E_\pi \left[\sum_{t=0}^{\infty} \gamma^t E_{a_t\sim\pi(\cdot|s_t)}[-\ln \pi(a_t|s_t)] \right] \\
&= E_\pi \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t, a_t) - \alpha \ln \pi(a_t|s_t) \right) \right].
\end{aligned}$$

The entropy bonus at each state, when averaged over the policy, becomes a per-action penalty proportional to the negative log probability of the action taken. This reformulation is more useful because the modified reward now depends only on the state, the action taken, and the probability assigned to that specific action by the policy, not on the entire distribution over actions.

This expression shows that entropy regularization is equivalent to adding a state-action dependent penalty term $-\alpha \ln \pi(a|s)$ to the reward. Intuititively, this terms amounts to paying a cost for low-entropy (deterministic) policies.

Now, when we write down the Bellman equation for this entropy-regularized problem, at each state s we need to find the decision rule $d(\cdot|s) \in \Delta(\mathcal{A}_s)$ (a probability distribution over actions) that maximizes:

$$v(s) = \max_{d(\cdot|s)\in\Delta(\mathcal{A}_s)} \sum_a d(a|s) \left[r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a)v(j) - \alpha \ln d(a|s) \right]. \tag{435}$$

Here $\Delta(\mathcal{A}_s) = \{d(\cdot|s) : d(a|s) \geq 0, \sum_a d(a|s) = 1\}$ denotes the probability simplex over actions available at state s. The optimization is over randomized decision rules at each state, constrained to be valid probability distributions.

This is a convex optimization problem with a linear constraint. We form the Lagrangian:

$$\mathcal{L}(d,\lambda) = \sum_a d(a|s) \left[r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a)v(j) - \alpha \ln d(a|s) \right] - \lambda \left(\sum_a d(a|s) - 1 \right), \tag{436}$$

where λ is the Lagrange multiplier enforcing the normalization constraint. Taking the derivative with respect to $d(a|s)$ and setting it to zero:

$$r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a)v(j) - \alpha(1 + \ln d^*(a|s)) - \lambda = 0. \tag{437}$$

Solving for $d^*(a|s)$:

$$d^*(a|s) = \exp\left(\frac{1}{\alpha} \left(r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a)v(j) - \lambda \right) \right). \tag{438}$$

Using the normalization constraint $\sum_a d^*(a|s) = 1$ to solve for λ:

$$\sum_a \exp\left(\frac{1}{\alpha} \left(r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a)v(j) \right) \right) = \exp\left(\frac{\lambda}{\alpha} \right). \tag{439}$$

Therefore:

$$\lambda = \alpha \ln \sum_a \exp\left(\frac{1}{\alpha} \left(r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a)v(j) \right) \right). \tag{440}$$

Substituting this back into the Bellman equation and simplifying:

$$v(s) = \alpha \ln \sum_a \exp\left(\frac{1}{\alpha} \left(r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a)v(j) \right) \right). \tag{441}$$

Setting $\beta = 1/\alpha$ (the inverse temperature), this becomes:

$$v(s) = \frac{1}{\beta} \ln \sum_a \exp\left(\beta\left(r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v(j)\right)\right). \tag{442}$$

We recover the smooth Bellman equation we derived earlier using the logsumexp operator. The inverse temperature parameter β controls how closely the logsumexp approximates the max: as $\beta \to \infty$, we recover the standard Bellman equation, while for finite β, we have a smooth approximation that corresponds to optimizing with entropy regularization strength $\alpha = 1/\beta$.

The optimal policy is:

$$\pi^*(a|s) = \frac{\exp\left(\beta q^*(s,a)\right)}{\sum_{a'} \exp\left(\beta q^*(s,a')\right)} = \text{softmax}_\beta(q^*(s,\cdot))(a), \tag{443}$$

which is exactly the softmax policy parametrized by inverse temperature.

The derivation establishes the complete equivalence: the value function v^* that solves the smooth Bellman equation is identical to the optimal value function v^*_Ω of the entropy-regularized MDP (with Ω being negative entropy and $\alpha = 1/\beta$), and the softmax policy that is greedy with respect to this value function achieves the maximum of the entropy-regularized objective. Both approaches yield the same numerical solution: the same values at every state and the same policy prescriptions. The only difference is how we conceptualize the problem: as smoothing the Bellman operator for computational tractability, or as explicitly trading off reward maximization against policy entropy.

This equivalence has important implications. When we use smooth Bellman equations with a logsumexp operator, we are implicitly solving an entropy-regularized MDP. Conversely, when we explicitly add entropy regularization to an MDP objective, we arrive at smooth Bellman equations as the natural description of optimality. This dual perspective will prove valuable in understanding various algorithms and theoretical results. For instance, in soft actor-critic methods and other maximum entropy reinforcement learning algorithms, the connection between smooth operators and entropy regularization provides both computational benefits (differentiability) and conceptual clarity (why we want stochastic policies).

Entropy-Regularized Dynamic Programming Algorithms While the smooth Bellman equations (using logsumexp) and entropy-regularized formulations are mathematically equivalent, it is instructive to present the algorithms explicitly in the entropy-regularized form, where the entropy bonus appears directly in the update equations.

Features:

- Line 11 updates the policy using the softmax of Q-values, with temperature α

- Line 14 explicitly computes the entropy-regularized value: expected Q-value plus entropy bonus
- The algorithm maintains and updates a stochastic policy throughout
- As $\alpha \to 0$ (or equivalently $\beta \to \infty$), this recovers standard value iteration

Features:

- **Policy Evaluation** (lines 3-15): Computes the value of the current policy including entropy bonus
  - Option 1: Iterative method (successive approximation)
  - Option 2: Direct solution via linear system
- **Policy Improvement** (lines 16-29): Updates policy to softmax over Q-values
- Line 14 shows the vector form: the linear system includes the entropy vector $\mathbf{H}_\pi$
- The algorithm alternates between evaluating the current stochastic policy and improving it
- Converges to the unique optimal entropy-regularized policy

4.2 Projection Methods for Functional Equations

The Bellman optimality equation $\mathrm{L}v = v$ is a functional equation: an equation where the unknown is an entire function rather than a finite-dimensional vector. When the state space is continuous or very large, we cannot represent the value function exactly on a computer. We must instead work with finite-dimensional approximations. This motivates projection methods, a general framework for transforming infinite-dimensional problems into tractable finite-dimensional ones.

4.2.1 What Does It Mean for a Residual to Be Zero?

Suppose we have found a candidate approximate solution $\hat{v}$ to the Bellman equation. To verify it satisfies $\mathrm{L}\hat{v} = \hat{v}$, we compute the **residual function** $R(s) = \mathrm{L}\hat{v}(s) - \hat{v}(s)$. For a true solution, this residual should be the **zero function**: $R(s) = 0$ for every state s. But what does it really mean for a function to equal zero?

In finite dimensions, a vector $\mathbf{r} \in R^n$ equals zero if and only if $\langle \mathbf{r}, \mathbf{y} \rangle = 0$ for every vector $\mathbf{y} \in R^n$. This follows because if $\mathbf{r} \neq \mathbf{0}$, we can always choose $\mathbf{y} = \mathbf{r}$, giving $\langle \mathbf{r}, \mathbf{r} \rangle = \|\mathbf{r}\|^2 > 0$. Conversely, if $\mathbf{r} = \mathbf{0}$, then $\langle \mathbf{r}, \mathbf{y} \rangle = 0$ trivially for all $\mathbf{y}$.

Inner products can distinguish the zero vector from any nonzero vector: for any $\mathbf{r} \neq \mathbf{0}$, there exists some test vector $\mathbf{y}$ that "witnesses" the fact that $\mathbf{r}$ is nonzero by producing $\langle \mathbf{r}, \mathbf{y} \rangle \neq 0$. This property, that we can tell apart (separate) different vectors by testing them with inner products, is what makes inner products so useful for verification.

The same principle extends to functions. A function R equals the zero function if and only if its "inner product" with every "test function" p vanishes:

$$R = 0 \quad \text{if and only if} \quad \langle R, p \rangle = \int_{\mathcal{S}} R(s)p(s)w(s)ds = 0 \quad \text{for all test functions } p, \tag{444}$$

where $w(s)$ is a weight function (often chosen to emphasize certain regions of the state space). Why does this work? For the same reason as in finite dimensions: if R is not the zero function, there must be some region where $R(s) \neq 0$. We can then choose a test function p that is nonzero in that same region (for instance, $p(s) = R(s)$ itself), which will produce $\langle R, p \rangle = \int R(s)p(s)w(s)ds > 0$, witnessing that R is nonzero. Conversely, if R is the zero function, then $\langle R, p \rangle = 0$ for any test function p.

This ability to **distinguish between different functions using inner products** is a fundamental principle from functional analysis. Just as we can test a vector by taking inner products with other vectors, we can test a function by taking inner products with other functions.

> **Connection to Functional Analysis**
> The principle that "a function equals zero if and only if it has zero inner

product with all test functions" is a consequence of the **Hahn-Banach theorem**, one of the cornerstones of functional analysis. The theorem guarantees that for any nonzero function R in a suitable function space, there exists a continuous linear functional (which can be represented as an inner product with some test function p) that produces a nonzero value when applied to R. This is often phrased as "the dual space separates points."

While you don't need to know the Hahn-Banach theorem to use projection methods, it provides the rigorous mathematical foundation ensuring that our inner product tests are theoretically sound. The constructive argument we gave above (choosing $p = R$) works in simple cases with well-behaved functions, but the Hahn-Banach theorem extends this guarantee to much more general settings.

Why is this useful? It transforms the pointwise condition "$R(s) = 0$ for all s" (infinitely many conditions, one per state) into an equivalent condition about inner products. We still cannot test against *all* possible test functions, since there are infinitely many of those too. But the inner product perspective suggests a natural computational strategy: choose a finite collection of test functions $\{p_1, \ldots, p_n\}$ and require

$$\langle R, p_i \rangle = 0, \quad i = 1, \ldots, n. \tag{445}$$

This gives us exactly n conditions that we can actually compute. This approach defines what are called **weighted residual methods**: we make the residual "small" by requiring it to satisfy certain weighted integral conditions.

Within weighted residual methods, there are two main families:

Projection methods (also called orthogonal projection methods) directly require the residual to have zero inner product with chosen test functions:

$$\langle R, p_i \rangle = 0, \quad i = 1, \ldots, n. \tag{446}$$

We are "projecting" the residual to be orthogonal to the span of the test functions. Different choices of test functions give different projection methods:

- **Galerkin**: Test against the basis functions used to represent $\hat{v}$, so $p_i = \varphi_i$
- **Collocation**: Test against delta functions $p_i = \delta(s - s_i)$, which reduces to pointwise evaluation $R(s_i) = 0$
- **Method of moments**: Test against polynomials $p_i = s^{i-1}$, ensuring low-order moments of the residual vanish
- **Subdomain method**: Test against indicator functions $p_i = I_{D_i}$ for subregions D_i, requiring zero average residual in each subdomain

The Special Role of Spectral Methods You may encounter the term **spectral methods** in the literature. This doesn't refer to a different choice of test functions, but rather to a choice of **basis functions** φ_i. Spectral methods use basis functions from families of orthogonal polynomials (like Chebyshev, Legendre, or Hermite polynomials) or trigonometric functions (Fourier series). The "spectral" name comes from the decomposition of the solution into these orthogonal components, analogous to decomposing a signal into frequency components.

What makes spectral bases special is their approximation properties: for smooth problems (functions with many continuous derivatives), spectral approximations achieve **exponential convergence**. As you add more basis functions, the approximation error decreases exponentially rather than polynomially. A function with k continuous derivatives approximated by piecewise polynomials of degree p has error $O(h^{p+1})$ where h is the grid spacing. But the same function approximated by a spectral method with n terms has error that decreases like $O(e^{-cn})$ for some constant $c > 0$. This dramatic difference makes spectral methods extremely efficient for smooth problems.

Now, spectral bases can be combined with any projection method. When we use a spectral basis with **Galerkin projection** (testing against the basis functions themselves), we get a **spectral Galerkin method**. The orthogonality of the basis functions often simplifies the resulting linear systems. When we use a spectral basis with **collocation**, we get what's often called a **pseudospectral method** or **spectral collocation method**.

Orthogonal Collocation **Orthogonal collocation** exploits a useful connection between collocation and quadrature. The idea is to:

1. Choose basis functions from an orthogonal polynomial family (say, Chebyshev polynomials $T_0, T_1, \ldots, T_{n-1}$)
2. Choose collocation points at the **zeros of the n-th polynomial** in that family

Why is this clever? Because these same points are also the optimal nodes for **Gauss quadrature** using the weight function associated with that polynomial family. For example, the zeros of the Chebyshev polynomial $T_n(s)$ are also the Chebyshev-Gauss quadrature nodes. This means:

- We get the computational simplicity of collocation: the projection conditions are just $R(s_i) = 0$ at the collocation points (no integrals to evaluate)
- When we do need to compute integrals (say, inside the operator N itself), we can use the collocation points as quadrature nodes and the resulting quadrature is **exact** for polynomials up to degree $2n - 1$
- For smooth problems, we inherit the exponential convergence of spectral approximations

This coordination between approximation and integration is why orthogonal collocation is so effective. You'll sometimes see it called a "pseudospectral method," though different authors use these terms with slight variations. The key point is that by carefully coordinating our choice of basis, test functions (collocation points), and quadrature nodes, we can achieve excellent accuracy with computational efficiency.

In summary, "spectral" describes the basis choice (orthogonal polynomials or Fourier), while "Galerkin," "collocation," etc. describe the projection choice (which test functions). Orthogonal collocation represents an optimal marriage of these choices for smooth problems.

Least squares methods take a different approach: instead of requiring orthogonality to specific test functions, we minimize the overall size of the residual measured in a weighted norm:

$$\min_\theta \|R(\cdot;\theta)\|^2 = \min_\theta \int_{\mathcal{S}} R(s;\theta)^2 w(s) ds. \tag{447}$$

This seeks the coefficients θ that make the residual as small as possible in the least squares sense. The first-order optimality conditions for this minimization problem turn out to be equivalent to a projection method with test functions $p_i = \partial R/\partial \theta_i$ (the derivatives of the residual with respect to the coefficients). So least squares can be viewed as a projection method with *data-dependent* test functions.

Both families aim to make the residual "close to zero," but projection methods do this by requiring orthogonality to chosen directions, while least squares does this by directly minimizing the norm of the residual. The term "projection methods" as used in the approximate dynamic programming literature often refers to both families, since they share the same computational framework of restricting the search to a finite-dimensional subspace and solving for coefficients that satisfy certain residual conditions.

In summary, we have transformed the impossible task of verifying "$R(s) = 0$ for all s" into a **finite-dimensional** problem: find coefficients $\theta = (\theta_1, \ldots, \theta_n)$ in our approximation $\hat{v}(s) = \sum_{i=1}^n \theta_i \varphi_i(s)$ such that either:

- The residual is orthogonal to n chosen test functions (projection methods), or
- The residual has minimum norm (least squares methods)

This is a major conceptual step forward: instead of infinitely many pointwise conditions, we have n conditions. However, these n conditions are not yet fully "feasible" computationally. Each projection condition $\langle R, p_i \rangle = \int_{\mathcal{S}} R(s) p_i(s) w(s) ds = 0$ still involves an integral that may need to be approximated numerically.

The computational cost hierarchy. Different methods have different computational burdens:

- **Collocation** is the cheapest: since $\langle R, \delta(\cdot - s_i) \rangle = R(s_i)$, we only evaluate the residual pointwise. No integration is needed in the projection conditions themselves.

- **Orthogonal collocation** shares this advantage (projection conditions are just pointwise evaluations), but adds a bonus: if integrals appear elsewhere, say inside the operator N, the collocation points double as optimal quadrature nodes. This synergy between approximation and integration is particularly valuable for smooth problems.
- **Galerkin methods** require evaluating integrals $\int R(s)\varphi_i(s)w(s)ds$ for each basis function. When using orthogonal polynomial bases (spectral Galerkin), these integrals can sometimes be simplified by orthogonality, but numerical quadrature is still typically needed.
- **Method of moments and subdomain methods** similarly require numerical quadrature to evaluate weighted integrals of the residual.
- **Least squares** requires computing $\int R(s)^2 w(s)ds$, which involves integrating the squared residual. This is potentially expensive, though the first-order conditions reduce this to a system similar to Galerkin.

The general pattern: collocation methods avoid integration in the projection step by testing at points rather than against functions, while methods that test against smooth functions (Galerkin, moments, subdomain) must pay the computational cost of numerical integration.

The rest of this chapter develops this framework systematically, showing how to choose bases, select test functions, evaluate or approximate the necessary integrals, and solve the resulting finite-dimensional problems.

4.2.2 The General Framework

Consider an operator equation of the form

$$N(f) = 0, \tag{448}$$

where $N : B_1 \to B_2$ is a continuous operator between complete normed vector spaces B_1 and B_2. For the Bellman equation, we have $N(v) = \mathrm{L}v - v$, so that solving $N(v) = 0$ is equivalent to finding the fixed point $v = \mathrm{L}v$.

The projection method approach consists of several conceptual steps that transform this infinite-dimensional problem into a finite-dimensional one.

Step 1: Choose a Finite-Dimensional Approximation Space We begin by selecting a basis $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ and approximating the unknown function as a linear combination:

$$\hat{f}(x) = \sum_{i=1}^{n} \theta_i \varphi_i(x). \tag{449}$$

The choice of basis functions φ_i is problem-dependent. Common choices include:

- **Polynomials**: For smooth problems, we might use Chebyshev polynomials or other orthogonal polynomial families
- **Splines**: For problems where we expect the solution to have regions of different smoothness
- **Radial basis functions**: For high-dimensional problems where tensor product methods become intractable

The number of basis functions n determines the flexibility of our approximation. In practice, we start with small n and increase it until the approximation quality is satisfactory. The only unknowns now are the coefficients $\theta = (\theta_1, \ldots, \theta_n)$.

While the classical presentation of projection methods focuses on polynomial bases, the framework applies equally well to other function classes. Neural networks, for instance, can be viewed through this lens: a neural network $\hat{f}(x; \theta)$ with parameters θ defines a flexible function class, and many training procedures can be interpreted as projection methods with specific choices of test functions or residual norms. The distinction is that classical methods typically use predetermined basis functions with linear coefficients, while neural networks use adaptive nonlinear features. Throughout this chapter, we focus on the classical setting to develop the core concepts, but the principles extend naturally to modern function approximators.

Step 2: Define the Residual Function Since we are approximating f with $\hat{f}$, the operator N will generally not vanish exactly. Instead, we obtain a **residual function**:

$$R(x; \theta) = N(\hat{f}(\cdot; \theta))(x). \tag{450}$$

This residual measures how far our candidate solution is from satisfying the equation at each point x. As we discussed in the introduction, we will assess whether this residual is "close to zero" by testing its inner products against chosen test functions.

Step 3: Choose Weighted Residual Conditions Having chosen our basis and defined the residual, we must decide how to make the residual "close to zero." As discussed in the introduction, we can either:

1. **Use projection conditions**: Select n test functions $\{p_1, \ldots, p_n\}$ and require:

$$\langle R(\cdot; \theta), p_i \rangle = \int_{\mathcal{S}} R(x; \theta) p_i(x) w(x) dx = 0, \quad i = 1, \ldots, n, \tag{451}$$

for some weight function $w(x)$. This yields n equations to determine the n coefficients in θ.

2. **Use a least squares condition**: Minimize the norm of the residual directly:

$$\min_{\theta} \int_{\mathcal{S}} R(x;\theta)^2 w(x) dx. \tag{452}$$

We begin by examining the main projection methods, distinguished entirely by their choice of test functions p_i, then discuss least squares as an alternative approach.

Let us examine the standard choices of test functions and what they tell us about the residual:

Galerkin Method: Test Against the Basis The Galerkin method chooses test functions $p_i = \varphi_i$, the same basis functions used to approximate $\hat{f}$:

$$\langle R(\cdot;\theta), \varphi_i \rangle = 0, \quad i = 1, \ldots, n. \tag{453}$$

To understand what this means, recall that in finite dimensions, two vectors are orthogonal when their inner product is zero. For functions, $\langle R, \varphi_i \rangle = \int R(x)\varphi_i(x)w(x)dx = 0$ expresses the same concept: R and φ_i are orthogonal as functions. But there's more to this than just testing against individual basis functions.

Consider our approximation space $\text{span}\{\varphi_1, \ldots, \varphi_n\}$ as an n-dimensional subspace within the infinite-dimensional space of all functions. Any function g in this space can be written as $g = \sum_{i=1}^{n} c_i \varphi_i$ for some coefficients c_i. If the residual R is orthogonal to all basis functions φ_i, then by linearity of the inner product, for any such function g:

$$\langle R, g \rangle = \left\langle R, \sum_{i=1}^{n} c_i \varphi_i \right\rangle = \sum_{i=1}^{n} c_i \langle R, \varphi_i \rangle = 0. \tag{454}$$

This shows that R is orthogonal to every function we can represent with our basis. The residual has "zero overlap" with our approximation space: we cannot express any part of it using our basis functions. In this sense, the residual is as "invisible" to our approximation as possible.

This condition is the defining property of optimality. By choosing our approximation $\hat{f}$ so that the residual $R = N(\hat{f})$ is orthogonal to the entire approximation space, we ensure that $\hat{f}$ is the orthogonal projection of the true solution onto $\text{span}\varphi_1, \ldots, \varphi_n$. Within this n-dimensional space, no better choice is possible: any other coefficients would yield a residual with a nonzero component inside the space, and therefore a larger norm.

The finite-dimensional analogy makes this concrete. Suppose you want to approximate a vector $\mathbf{v} \in R^3$ using only the xy-plane (a 2D subspace). The best approximation is to project $\mathbf{v}$ onto the plane, giving $\hat{\mathbf{v}} = (v_1, v_2, 0)$. The error is $\mathbf{r} = \mathbf{v} - \hat{\mathbf{v}} = (0, 0, v_3)$, which points purely in the z-direction, orthogonal to the entire xy-plane. We see the Galerkin condition in action: the error is orthogonal to the approximation space.

Method of Moments: Test Against Monomials The method of moments, for problems on $D \subset R$, chooses test functions $p_i(x) = x^{i-1}$ for $i = 1, \ldots, n$:

$$\langle R(\cdot;\theta), x^{i-1} \rangle = 0, \quad i = 1, \ldots, n. \tag{455}$$

This requires the first n moments of the residual function to vanish, ensuring the residual is "balanced" in the sense that it has no systematic trend captured by low-order polynomials. The moments $\int x^k R(x;\theta) w(x) dx$ measure weighted averages of the residual, with increasing powers of x giving more weight to larger values. Setting these to zero ensures the residual doesn't grow systematically with x. This approach is particularly useful when $w(x)$ is chosen as a probability measure, making the conditions natural moment restrictions familiar from statistics and econometrics.

Collocation Method: Test Against Delta Functions The collocation method chooses test functions $p_i(x) = \delta(x - x_i)$, the Dirac delta functions at points $\{x_1, \ldots, x_n\}$:

$$\langle R(\cdot;\theta), \delta(\cdot - x_i) \rangle = R(x_i;\theta) = 0, \quad i = 1, \ldots, n. \tag{456}$$

This is projection against the most localized test functions possible: delta functions that "sample" the residual at specific points, requiring the residual to vanish exactly where we test it. When using orthogonal polynomials with collocation points at the zeros of the n-th polynomial, the Chebyshev interpolation theorem guarantees that forcing $R(x_i;\theta) = 0$ at these specific points makes $R(x;\theta)$ small everywhere. Using the zeros of orthogonal polynomials as collocation points produces well-conditioned systems and near-optimal interpolation error. The computational advantage is significant. Collocation avoids numerical integration entirely, requiring only pointwise evaluation of R.

Subdomain Method: Test Against Indicator Functions The subdomain method partitions the domain into n subregions $\{D_1, \ldots, D_n\}$ and chooses test functions $p_i = I_{D_i}$, the indicator functions:

$$\langle R(\cdot;\theta), I_{D_i} \rangle = \int_{D_i} R(x;\theta) w(x) dx = 0, \quad i = 1, \ldots, n. \tag{457}$$

This requires the residual to have zero average over each subdomain, ensuring the approximation is good "on average" over each piece of the domain. This approach is particularly natural for finite element methods where the domain is divided into elements, ensuring local balance of the residual within each element.

Least Squares The least squares approach appears different at first glance, but it also fits the test function framework. We minimize:

$$\min_\theta \int_{\mathcal{S}} R(x;\theta)^2 w(x) dx = \min_\theta \langle R(\cdot;\theta), R(\cdot;\theta) \rangle. \tag{458}$$

The first-order conditions for this minimization problem are:

$$\left\langle R(\cdot;\theta), \frac{\partial R(\cdot;\theta)}{\partial \theta_i} \right\rangle = 0, \quad i = 1, \ldots, n. \tag{459}$$

Thus least squares implicitly uses test functions $p_i = \partial R/\partial \theta_i$, the gradients of the residual with respect to parameters. Unlike other methods where test functions are chosen a priori, here they depend on the current guess for θ and on the structure of our approximation.

We can now see the unifying structure of **weighted residual methods**: whether we use projection conditions or least squares minimization, all these methods follow the same template of restricting the search to an n-dimensional function space and imposing n conditions on the residual. For projection methods specifically, we pick n test functions and require $\langle R, p_i \rangle = 0$. They differ only in their philosophy about which test functions best detect whether the residual is "nearly zero." Galerkin tests against the approximation basis itself (natural for orthogonal bases), the method of moments tests against monomials (ensuring polynomial balance), collocation tests against delta functions (pointwise satisfaction), subdomain tests against indicators (local average satisfaction), and least squares tests against residual gradients (global norm minimization). Each choice reflects different priorities: computational efficiency, theoretical optimality, ease of implementation, or sensitivity to errors in different regions of the domain.

Step 4: Solve the Finite-Dimensional Problem The projection conditions give us a system to solve for the coefficients θ. For test function methods (Galerkin, collocation, moments, subdomain), we solve:

$$P_i(\theta) \equiv \langle R(\cdot;\theta), p_i \rangle = 0, \quad i = 1, \ldots, n. \tag{460}$$

This is a system of n (generally nonlinear) equations in n unknowns. For least squares, we solve the optimization problem $\min_\theta \langle R(\cdot;\theta), R(\cdot;\theta) \rangle$.

The **conditioning** of the system depends on the choice of test functions. The Jacobian matrix has entries:

$$J_{ij} = \frac{\partial P_i}{\partial \theta_j} = \left\langle \frac{\partial R(\cdot;\theta)}{\partial \theta_j}, p_i \right\rangle. \tag{461}$$

When test functions are orthogonal (or nearly so), the Jacobian tends to be well-conditioned. This is why orthogonal polynomial bases are preferred in Galerkin methods: they produce Jacobians with controlled condition numbers.

The **computational cost per iteration** varies significantly:

- **Collocation**: Cheapest to evaluate since $P_i(\theta) = R(x_i;\theta)$ requires only pointwise evaluation (no integration). The Jacobian is also cheap: $J_{ij} = \frac{\partial R(x_i;\theta)}{\partial \theta_j}$.

- **Galerkin and moments**: More expensive due to integration. Computing $P_i(\theta) = \int R(x;\theta)p_i(x)w(x)dx$ requires numerical quadrature. Each Jacobian entry requires integrating $\frac{\partial R}{\partial \theta_j}p_i$.

- **Least squares**: Most expensive when done via the objective function, which requires integrating R^2. However, the first-order conditions reduce it to a system like Galerkin, with test functions $p_i = \partial R/\partial \theta_i$.

For methods requiring integration, the choice of quadrature rule should match the basis. Gaussian quadrature with nodes at orthogonal polynomial zeros is efficient. When combined with collocation at those same points, the quadrature is exact for polynomials up to a certain degree. This coordination between quadrature and collocation makes **orthogonal collocation** effective.

The choice of solver depends on whether the finite-dimensional approximation preserves the structural properties of the original infinite-dimensional problem. This matters for the Bellman equation, where the original operator L is a contraction.

Successive approximation (fixed-point iteration) is the natural choice when the original operator is a contraction, as it preserves the global convergence guarantees. However, the finite-dimensional approximation $\hat{\mathrm{L}}$ may not inherit the contraction property of L. The approximation can introduce spurious fixed points or destroy the contraction constant, leading to divergence or slow convergence. This is especially problematic when using high-order polynomial approximations, which can create artificial oscillations that destabilize the iteration.

Newton's method is often the default choice for projection methods because it doesn't rely on the contraction property. Instead, it exploits the smoothness of the residual function. When the original problem is smooth and the approximation preserves this smoothness, Newton's method provides quadratic convergence near the solution. However, Newton's method requires good initial guesses and may converge to spurious solutions if the finite-dimensional problem has multiple fixed points that the original problem lacks.

The choice of basis and projection method affects which algorithm is most appropriate. For example:

- **Linear interpolation** often preserves contraction properties, making successive approximation reliable

- **High-order polynomials** may destroy contraction but provide smooth approximations suitable for Newton's method

- **Shape-preserving splines** can maintain both smoothness and structural properties

In practice, which algorithm should we use? When the operator equation can be written as a fixed-point problem $f = Tf$ and the operator T is known to be a contraction, successive approximation is often the best starting point: it

is computationally cheap and globally convergent. However, not all equations $N(f) = 0$ admit a natural fixed-point reformulation, and even when they do (e.g., $f = f - \alpha N(f)$ for some $\alpha > 0$), the resulting operator may not be a contraction in the finite-dimensional approximation space. In such cases, Newton's method becomes the primary option despite its requirement for good initial guesses and higher computational cost per iteration. A hybrid approach often works well: use successive approximation when applicable to generate an initial guess, then switch to Newton's method for refinement.

Another consideration is the conditioning of the resulting system. Poorly chosen basis functions or collocation points can lead to nearly singular Jacobians, causing numerical instability. Orthogonal bases and carefully chosen collocation points (like Chebyshev nodes) are preferred because they tend to produce well-conditioned systems.

Step 5: Verify the Solution Once we have computed a candidate solution $\hat{f}$, we must verify its quality. Projection methods optimize $\hat{f}$ with respect to specific criteria (specific test functions or collocation points), but we should check that the residual is small everywhere, including directions or points we did not optimize over.

Typical diagnostic checks include:

- Computing $\|R(\cdot;\theta)\|$ using a more accurate quadrature rule than was used in the optimization
- Evaluating $R(x;\theta)$ at many points not used in the fitting process
- If using Galerkin with the first n basis functions, checking orthogonality against higher-order basis functions

4.2.3 Application to the Bellman Equation

We now apply the projection method framework to the Bellman optimality equation. Recall that we seek a function v satisfying

$$v(s) = \mathrm{L}v(s) = \max_{a \in \mathcal{A}_s} \left\{ r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v(j) \right\}. \tag{462}$$

Writing this as an operator equation $N(v) = 0$ with $N(v) = \mathrm{L}v - v$, the residual function for a candidate approximation $\hat{v}(s) = \sum_{i=1}^{n} \theta_i \varphi_i(s)$ is:

$$R(s;\theta) = \mathrm{L}\hat{v}(s) - \hat{v}(s) = \max_{a \in \mathcal{A}_s} \left\{ r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) \hat{v}(j) \right\} - \sum_{i=1}^{n} \theta_i \varphi_i(s). \tag{463}$$

Any of the projection methods we discussed (Galerkin, method of moments, collocation, subdomain, or least squares) can be applied here. Each would

give us n conditions to determine the n coefficients in our approximation. For instance:

- **Galerkin** would require $\langle R(\cdot;\theta), \varphi_i \rangle = 0$ for $i = 1, \ldots, n$, involving integration of the residual weighted by basis functions
- **Method of moments** would require $\langle R(\cdot;\theta), s^{i-1} \rangle = 0$, setting the first n moments of the residual to zero
- **Collocation** would require $R(s_i;\theta) = 0$ at n chosen states, forcing the residual to vanish pointwise

In practice, **collocation is the most commonly used** projection method for the Bellman equation. The reason is computational: collocation avoids the numerical integration required by Galerkin and method of moments. Since the Bellman operator already involves integration (or summation) over next states, adding another layer of integration for the projection conditions would be computationally expensive. Collocation sidesteps this by requiring the equation to hold exactly at specific points.

We focus on collocation in detail, though the principles extend to other projection methods.

Collocation for the Bellman Equation The collocation approach chooses n states $\{s_1, \ldots, s_n\}$ (the collocation points) and requires:

$$R(s_i;\theta) = 0, \quad i = 1, \ldots, n. \tag{464}$$

This gives us a system of n nonlinear equations in n unknowns:

$$\sum_{j=1}^{n} \theta_j \varphi_j(s_i) = \max_{a \in \mathcal{A}_{s_i}} \left\{ r(s_i, a) + \gamma \sum_{j \in \mathcal{S}} p(j|s_i, a) \hat{v}(j) \right\}, \quad i = 1, \ldots, n. \tag{465}$$

The right-hand side requires evaluating the Bellman operator at the collocation points. For each collocation point s_i, we must:

1. For each action $a \in \mathcal{A}_{s_i}$, compute the expected continuation value $\sum_{j \in \mathcal{S}} p(j|s_i, a)\hat{v}(j)$
2. Take the maximum over actions

When the state space is continuous, the expectation involves integration, which typically requires numerical quadrature. When the state space is discrete but large, this is a straightforward (though potentially expensive) summation.

Solving the Collocation System To organize our thinking about solution methods, it helps to introduce the **collocation function** $v : R^n \to R^n$, which maps coefficient vectors to target values at the collocation points. Given a coefficient vector $\theta = (\theta_1, \ldots, \theta_n)$, the i-th component of $v(\theta)$ is defined as:

$$v_i(\theta) = \max_{a \in \mathcal{A}_{s_i}} \left\{ r(s_i, a) + \gamma \sum_{j \in \mathcal{S}} p(j|s_i, a) \sum_{\ell=1}^{n} \theta_\ell \varphi_\ell(j) \right\}. \tag{466}$$

In words: $v_i(\theta)$ is the value obtained by solving the Bellman maximization problem at collocation node s_i, using the approximation $\hat{v}(s; \theta) = \sum_{\ell=1}^{n} \theta_\ell \varphi_\ell(s)$ in place of the true value function. The collocation function evaluates the **right-hand side** of the Bellman equation at all collocation points, given a current guess for the coefficients.

Let $\mathbf{\Phi}$ denote the $n \times n$ **collocation matrix** with entries $\Phi_{ij} = \varphi_j(s_i)$. The **left-hand side** of the collocation equations is $\mathbf{\Phi}\theta$, which gives the values of $\hat{v}(s_i; \theta)$ at the collocation points. The collocation equation requires these to match:

$$\mathbf{\Phi}\theta = v(\theta). \tag{467}$$

This is the fixed-point problem we need to solve. We can approach it in two fundamentally different ways: as a **fixed-point iteration** (function iteration) or as a **rootfinding problem** (Newton's method).

Method 1: Function Iteration (Successive Approximation) We can rewrite the collocation equation as $\theta = \mathbf{\Phi}^{-1} v(\theta)$ (assuming $\mathbf{\Phi}$ is invertible, which holds when the basis functions are linearly independent at the collocation points). This suggests the **function iteration** scheme:

$$\theta^{(k+1)} = \mathbf{\Phi}^{-1} v(\theta^{(k)}). \tag{468}$$

This iteration has an intuitive interpretation when we break it into two steps:

Step 1 (Apply Bellman operator): For the current coefficient guess $\theta^{(k)}$, compute the Bellman operator values at all collocation points:

$$t_i^{(k)} = v_i(\theta^{(k)}) = \max_{a \in \mathcal{A}_{s_i}} \left\{ r(s_i, a) + \gamma \sum_{j \in \mathcal{S}} p(j|s_i, a) \sum_{\ell=1}^{n} \theta_\ell^{(k)} \varphi_\ell(j) \right\}, \quad i = 1, \ldots, n. \tag{469}$$

We now have a vector of **targets** $t^{(k)} = (t_1^{(k)}, \ldots, t_n^{(k)}) = v(\theta^{(k)})$.

Step 2 (Fit to targets): Find new coefficients $\theta^{(k+1)}$ such that the approximation matches the targets at the collocation points:

$$\sum_{\ell=1}^{n} \theta_\ell^{(k+1)} \varphi_\ell(s_i) = t_i^{(k)}, \quad i = 1, \ldots, n. \tag{470}$$

In matrix form: $\mathbf{\Phi}\theta^{(k+1)} = t^{(k)}$, which gives $\theta^{(k+1)} = \mathbf{\Phi}^{-1}t^{(k)} = \mathbf{\Phi}^{-1}v(\theta^{(k)})$.

This is **parametric value iteration** or **projection-based value iteration**: we iterate the Bellman operator in the finite-dimensional coefficient space, projecting back onto the span of the basis functions at each step. The method:

- Separates the nonlinear optimization (maximization in the Bellman operator) from the linear fitting problem
- Is globally convergent when the finite-dimensional approximation preserves the contraction property
- Requires only solving a linear system $\mathbf{\Phi}\theta^{(k+1)} = t^{(k)}$ at each iteration

Handling Stochastic Expectations
When the model includes a continuous random variable (e.g., $s' = g(s, a, \epsilon)$ where ϵ is a random disturbance, often called a "shock" in economics and econometrics), we must approximate the expectation using **numerical quadrature**. We replace the continuous ϵ with a discrete approximation taking values $\{\epsilon_1, \ldots, \epsilon_m\}$ with probabilities $\{w_1, \ldots, w_m\}$. The collocation function becomes:

$$v_i(\theta) = \max_{a \in \mathcal{A}_{s_i}} \left\{ r(s_i, a) + \gamma \sum_{k=1}^{m} w_k \sum_{\ell=1}^{n} \theta_\ell \varphi_\ell(g(s_i, a, \epsilon_k)) \right\}. \tag{471}$$

Common quadrature schemes include Gauss-Hermite (for normal random variables), Gauss-Legendre (for uniform random variables), or sparse grids for high-dimensional random variables.

Method 2: Newton's Method with the Envelope Theorem Alternatively, we can write the collocation equation as a **rootfinding problem**:

$$F(\theta) \equiv \mathbf{\Phi}\theta - v(\theta) = 0. \tag{472}$$

Newton's method for this system uses the update:

$$\theta^{(k+1)} = \theta^{(k)} - J_F(\theta^{(k)})^{-1} F(\theta^{(k)}), \tag{473}$$

where $J_F(\theta)$ is the Jacobian of F at θ. Since $J_F = \mathbf{\Phi} - J_v$ where J_v is the Jacobian of the collocation function v, we can rewrite this as:

$$\theta^{(k+1)} = \theta^{(k)} - [\mathbf{\Phi} - J_v(\theta^{(k)})]^{-1} [\mathbf{\Phi}\theta^{(k)} - v(\theta^{(k)})]. \tag{474}$$

The main challenge now is computing the Jacobian $J_v(\theta)$. The (i, j)-th entry is:

$$[J_v]_{ij} = \frac{\partial v_i}{\partial \theta_j}(\theta) = \frac{\partial}{\partial \theta_j}\left[\max_{a \in \mathcal{A}_{s_i}} \left\{ r(s_i, a) + \gamma \sum_{k \in \mathcal{S}} p(k|s_i, a) \sum_{\ell=1}^{n} \theta_\ell \varphi_\ell(k) \right\}\right]. \quad (475)$$

At first glance, this appears problematic because the max operator is not differentiable. However, we can apply the **Envelope Theorem** to compute this derivative without dealing with the non-differentiability of the max operator.

The Envelope Theorem

Setup: Consider a smooth objective function $f(\mathbf{x}, \boldsymbol{\theta})$ and define the optimal value:

$$v(\boldsymbol{\theta}) = \max_{\mathbf{x}} f(\mathbf{x}, \boldsymbol{\theta}). \quad (476)$$

Let $\mathbf{x}(\boldsymbol{\theta})$ denote the maximizer, satisfying the first-order condition:

$$\nabla_{\mathbf{x}} f(\mathbf{x}(\boldsymbol{\theta}), \boldsymbol{\theta}) = \mathbf{0}. \quad (477)$$

The Result: To find how the **optimal value** changes with $\boldsymbol{\theta}$, write $v(\boldsymbol{\theta}) = f(\mathbf{x}(\boldsymbol{\theta}), \boldsymbol{\theta})$ and apply the chain rule:

$$\nabla_{\boldsymbol{\theta}} v(\boldsymbol{\theta}) = \underbrace{\nabla_{\boldsymbol{\theta}} f(\mathbf{x}(\boldsymbol{\theta}), \boldsymbol{\theta})}_{\text{direct effect}} + \underbrace{\nabla_{\mathbf{x}} f(\mathbf{x}(\boldsymbol{\theta}), \boldsymbol{\theta})^\top}_{\mathbf{0} \text{ at optimum}} \frac{\partial \mathbf{x}}{\partial \boldsymbol{\theta}}. \quad (478)$$

Since $\nabla_{\mathbf{x}} f = \mathbf{0}$ at the optimum, the second term vanishes:

$$\boxed{\nabla_{\boldsymbol{\theta}} v(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} f(\mathbf{x}(\boldsymbol{\theta}), \boldsymbol{\theta})}. \quad (479)$$

This results tells us that you can compute the derivative of the optimal value by treating the maximizer as constant. You don't need to compute $\frac{\partial \mathbf{x}}{\partial \boldsymbol{\theta}}$.

In our Bellman collocation problem, $v_i(\theta) = \max_a\{r(s_i, a) + \gamma E[\sum_\ell \theta_\ell \varphi_\ell(s')]\}$. To compute $\frac{\partial v_i}{\partial \theta_j}$, we don't need to figure out how $a_i^*(\theta)$ changes with θ. We just evaluate the gradient at the optimal action:

$$\frac{\partial v_i}{\partial \theta_j}(\theta) = \gamma \sum_{s'} p(s'|s_i, a_i^*(\theta))\varphi_j(s'). \quad (480)$$

Important assumptions: f is smooth, the maximizer is unique and in the interior (or constraints are smooth with stable active sets), and the first-order condition holds.

Specifically, let $a_i^*(\theta)$ denote the optimal action at collocation point s_i given coefficients θ:

$$a_i^*(\theta) \in \text{argmax}_{a \in \mathcal{A}_{s_i}} \left\{ r(s_i, a) + \gamma \sum_{k \in \mathcal{S}} p(k|s_i, a) \sum_{\ell=1}^{n} \theta_\ell \varphi_\ell(k) \right\}. \tag{481}$$

By the Envelope Theorem, we can compute the derivative by treating a_i^* as constant:

$$\frac{\partial v_i}{\partial \theta_j}(\theta) = \gamma \sum_{k \in \mathcal{S}} p(k|s_i, a_i^*(\theta)) \varphi_j(k). \tag{482}$$

This is simply the **expected value of the j-th basis function at the next state**, evaluated at the optimal action for the current coefficient vector.

Why the Envelope Theorem Works Here

The Envelope Theorem applies to value functions of optimization problems. For a problem $V(\theta) = \max_x f(x, \theta)$, the derivative with respect to the parameter θ is:

$$\frac{dV}{d\theta} = \frac{\partial f}{\partial \theta}(x^*(\theta), \theta), \tag{483}$$

where $x^*(\theta)$ is the optimal choice. We don't need to account for $\frac{\partial x^*}{\partial \theta}$ because at the optimum, the first-order condition $\frac{\partial f}{\partial x} = 0$ makes that term vanish.

In our case, $v_i(\theta)$ is the value of maximizing over actions a, with θ playing the role of the parameter vector. The Envelope Theorem lets us compute $\frac{\partial v_i}{\partial \theta_j}$ by differentiating the objective (the Bellman right-hand side) with respect to θ_j while holding the optimal action a_i^* fixed. Since θ_j only appears in the continuation value $\sum_\ell \theta_\ell \varphi_\ell(k)$, the derivative picks out the coefficient of θ_j, which is the expected basis function value.

This approach is **equivalent** to the semi-smooth Newton method mentioned earlier: we're computing a generalized Jacobian by treating the optimal action as locally constant. As we converge to the solution, the optimal actions stabilize, and Newton's method achieves superlinear convergence.

Newton's method algorithm:

1. Start with initial guess $\theta^{(0)}$
2. For iteration $k = 0, 1, 2, \ldots$:
   - Compute $v(\theta^{(k)})$ and optimal actions $a_i^*(\theta^{(k)})$ for all collocation points
   - Compute Jacobian entries: $[J_v]_{ij} = \gamma \sum_{s' \in \mathcal{S}} p(s'|s_i, a_i^*(\theta^{(k)})) \varphi_j(s')$
   - Update: $\theta^{(k+1)} = \theta^{(k)} - [\mathbf{\Phi} - J_v(\theta^{(k)})]^{-1} [\mathbf{\Phi}\theta^{(k)} - v(\theta^{(k)})]$
   - Check convergence

This method offers **quadratic convergence** near the solution but requires a good initial guess. The Jacobian computation via the Envelope Theorem is typically cheaper than explicit semi-smooth calculus and has a clear economic interpretation: it tracks how the value propagates through the optimal decisions.

Comparison of Solution Methods We now have two approaches to solving the collocation fixed-point equation $\mathbf{\Phi}\theta = v(\theta)$:

Method	Formulation	Convergence	Per-iteration cost	Initial guess sensitivity
Function iteration	$\theta^{(k+1)} = \mathbf{\Phi}^{-1} v(\theta^{(k)})$	Linear (when contraction holds)	Low (one linear solve)	Robust
Newton's method	$\theta^{(k+1)} = \theta^{(k)} - [\mathbf{\Phi} - J_v]^{-1}[\mathbf{\Phi}\theta^{(k)} - v(\theta^{(k)})]$	Quadratic (near solution)	Moderate (Jacobian + linear solve)	Requires good initial guess

Function iteration exploits the fixed-point structure directly, treating the collocation problem as value iteration in coefficient space. When the finite-dimensional approximation preserves the contraction property of the Bellman operator, it converges globally from any initial guess. Each iteration is cheap: evaluate the Bellman operator at n points (the collocation nodes) and solve one linear system $\mathbf{\Phi}\theta^{(k+1)} = v(\theta^{(k)})$. However, convergence can be slow, especially when γ is close to 1.

Newton's method treats the problem as rootfinding, exploiting smoothness (via the Envelope Theorem) to achieve fast local convergence. Once close to the solution, Newton's method typically converges in just a few iterations. The per-iteration cost is higher: in addition to evaluating $v(\theta^{(k)})$, we must compute the Jacobian $J_v(\theta^{(k)})$, which requires evaluating expected basis function values at all collocation points. However, Newton's method is sensitive to initial conditions and may diverge or converge to spurious solutions when started far from the true fixed point.

Connection to Policy Iteration

The Newton update $\theta^{(k+1)} = [\mathbf{\Phi} - J_v(\theta^{(k)})]^{-1} v(\theta^{(k)})$ is equivalent to the **policy iteration** algorithm commonly used in discrete-state dynamic programming. To see this, note that at the optimal coefficients θ^*, we have $\mathbf{\Phi}\theta^* = v(\theta^*)$. The Newton step finds the coefficients that would be optimal if the policy (the optimal actions at each collocation point) were held fixed at the current iteration's policy. This connection explains why Newton's method often converges rapidly: like policy iteration, it implicitly performs

policy evaluation and policy improvement, which can converge in just a few iterations for well-behaved problems.

Practical recommendations:

1. **For problems with strong contraction** (small γ, well-conditioned $\mathbf{\Phi}$, shape-preserving bases): Start with function iteration. It's simple, robust, and often converges adequately in 20-50 iterations.

2. **For problems with weak contraction** (large γ, high-order polynomial bases): Use a **hybrid approach**:

   - Run function iteration for 5-10 iterations to get into the basin of attraction
   - Switch to Newton's method for fast convergence to high accuracy

3. **For problems where contraction fails** (non-monotone bases, approximation destroys contraction): Newton's method may be necessary from the start, but requires careful initialization (e.g., from a coarser approximation).

4. **Quasi-Newton methods** (like BFGS or Broyden) offer a middle ground: they approximate the Jacobian using function evaluations only, avoiding the Envelope Theorem calculation. This can be useful when computing J_v is expensive or when the Jacobian approximation is acceptable.

The choice often depends on the application domain. In economic models where the value function is guaranteed to be concave and monotone, simple bases (linear interpolation, shape-preserving splines) combined with function iteration are reliable. In control problems with complex dynamics, high-order approximations combined with Newton's method may be necessary for accuracy.

4.2.4 Monotone Projection and the Preservation of Contraction

The informal discussion of shape preservation hints at a deeper theoretical question: **when does the function iteration method converge?** Recall from our discussion of collocation that function iteration proceeds in two steps:

1. Apply the Bellman operator at collocation points: $t^{(k)} = v(\theta^{(k)})$ where $t_i^{(k)} = \mathrm{L}\hat{v}^{(k)}(s_i)$

2. Fit new coefficients to match these targets: $\mathbf{\Phi}\theta^{(k+1)} = t^{(k)}$, giving $\theta^{(k+1)} = \mathbf{\Phi}^{-1}v(\theta^{(k)})$

We can reinterpret this iteration in **function space** rather than coefficient space. Let Ψ be the **projection operator** that takes any function f and

returns its approximation in $\text{span}\{\varphi_1, \ldots, \varphi_n\}$. For collocation, Ψ is the interpolation operator: $(\Psi f)(s)$ is the unique linear combination of basis functions that matches f at the collocation points. Then Step 2 can be written as: fit $\hat{v}^{(k+1)}$ so that $\hat{v}^{(k+1)}(s_i) = \mathrm{L}\hat{v}^{(k)}(s_i)$ for all collocation points, which means $\hat{v}^{(k+1)} = \Psi(\mathrm{L}\hat{v}^{(k)})$.

In other words, function iteration is equivalent to **projected value iteration in function space**:

$$\hat{v}^{(k+1)} = \Psi \mathrm{L} \hat{v}^{(k)}. \tag{484}$$

We know that standard value iteration $v_{k+1} = \mathrm{L}v_k$ converges because L is a γ-contraction in the sup norm. But now we're iterating with the **composed operator** $\Psi\mathrm{L}$ instead of L alone.

This $\Psi\mathrm{L}$ structure is not specific to collocation. It is inherent in all projection methods. The general pattern is always the same: apply the Bellman operator to get a target function $\mathrm{L}\hat{v}^{(k)}$, then project it back onto our approximation space to get $\hat{v}^{(k+1)}$. The projection step defines an operator Ψ that depends on our choice of test functions:

- For **collocation**, Ψ interpolates values at collocation points
- For **Galerkin**, Ψ is orthogonal projection with respect to $\langle \cdot, \cdot \rangle_w$
- For **least squares**, Ψ minimizes the weighted residual norm

But regardless of which projection method we use, iteration takes the form $\hat{v}^{(k+1)} = \Psi\mathrm{L}\hat{v}^{(k)}$.

The critical question is: **does the composition $\Psi\mathrm{L}$ inherit the contraction property of L?** If not, the iteration may diverge, oscillate, or converge to a spurious fixed point even though the original problem is well-posed.

Monotone Approximators and Stability The answer turns out to depend on specific properties of the approximation operator Ψ. This theory was developed independently across multiple research communities—computational economics Judd [1992, 1996], Santos and Vigo-Aguiar [1998], economic dynamics Stachurski [2009], and reinforcement learning Gordon [1995, 1999]—arriving at essentially the same mathematical conditions.

Monotonicity Implies Nonexpansiveness It turns out that approximation operators satisfying simple structural properties automatically preserve contraction.

Proposition 4.1 (Monotone operators are nonexpansive (Stachurski))**.** *Let $\Psi : B(\mathcal{S}) \to B(\mathcal{S})$ be a linear operator on the space of bounded functions. If Ψ satisfies:*

1. ***Monotonicity**: $f \leq g$ pointwise implies $\Psi f \leq \Psi g$*

2. ***Constant preservation***: $\Psi \mathbf{1} = \mathbf{1}$ *where* $\mathbf{1}$ *is the constant function equal to 1*

Then Ψ *is nonexpansive in the sup norm:* $\|\Psi f - \Psi g\|_\infty \leq \|f - g\|_\infty$ *for all* $f, g \in B(\mathcal{S})$.

Proof. Let $M = \|f - g\|_\infty$. Then $-M \leq f(s) - g(s) \leq M$ for all s, which can be written as $g - M\mathbf{1} \leq f \leq g + M\mathbf{1}$. By monotonicity, $\Psi(g - M\mathbf{1}) \leq \Psi f \leq \Psi(g + M\mathbf{1})$. By linearity and constant preservation, $\Psi g - M\mathbf{1} \leq \Psi f \leq \Psi g + M\mathbf{1}$, which means $|\Psi f(s) - \Psi g(s)| \leq M$ for all s. Therefore $\|\Psi f - \Psi g\|_\infty \leq \|f - g\|_\infty$. □

This proposition shows that monotonicity and constant preservation automatically imply nonexpansiveness. There is no need to verify this separately. The intuition is that a monotone, constant-preserving operator acts like a weighted average that respects order structure and cannot amplify differences between functions.

Preservation of Contraction Combining nonexpansiveness with the contraction property of the Bellman operator yields the main stability result.

Theorem 4.3 (Stability of projected value iteration (Santos-Vigo-Aguiar))**.** *Let* $\mathrm{L} : B(\mathcal{S}) \to B(\mathcal{S})$ *be a* γ*-contraction on the space of bounded functions with respect to the sup norm. Let* $\Psi : B(\mathcal{S}) \to B(\mathcal{S})$ *be a linear approximation operator satisfying monotonicity and constant preservation.*

Then the composed operator $\Psi\mathrm{L}$ *is a* γ*-contraction, and projected value iteration* $v_{k+1} = \Psi\mathrm{L}v_k$ *converges globally to a unique fixed point* $v_\Psi \in$ *Range*(Ψ) *with approximation error:*

$$\|v_\Psi - v^*\|_\infty \leq \frac{1}{1-\gamma} \|\Psi v^* - v^*\|_\infty, \tag{485}$$

where v^* *is the true value function.*

Proof. Since L is a γ-contraction, we have $-\gamma\|f - g\|_\infty \leq \mathrm{L}f - \mathrm{L}g \leq \gamma\|f - g\|_\infty$ pointwise. By monotonicity of Ψ, $\Psi(-\gamma\|f - g\|_\infty) \leq \Psi(\mathrm{L}f - \mathrm{L}g) \leq \Psi(\gamma\|f - g\|_\infty)$. By constant preservation, $-\gamma\|f - g\|_\infty \leq \Psi(\mathrm{L}f - \mathrm{L}g) \leq \gamma\|f - g\|_\infty$, which implies $\|\Psi\mathrm{L}f - \Psi\mathrm{L}g\|_\infty \leq \gamma\|f - g\|_\infty$.

The error bound follows from fixed-point analysis: $v^* - v_\Psi = (I - \Psi\mathrm{L})^{-1}(v^* - \Psi v^*)$, and since $\Psi\mathrm{L}$ is a γ-contraction, $\|(I - \Psi\mathrm{L})^{-1}\| \leq (1 - \gamma)^{-1}$.

□

This error bound tells us that the fixed-point error is controlled by how well Ψ can represent v^*. If $v^* \in \mathrm{Range}(\Psi)$, then $\Psi v^* = v^*$ and the error vanishes. Otherwise, the error is proportional to the approximation error $\|\Psi v^* - v^*\|_\infty$, amplified by the factor $(1 - \gamma)^{-1}$.

Averagers in Discrete-State Problems For discrete-state problems, the monotonicity conditions have a natural interpretation as **averaging with nonnegative weights**. This characterization was developed by Gordon in the context of reinforcement learning.

Definition 4.1 (Averager (Gordon))**.** *An operator $\Psi : R^{|\mathcal{S}|} \to R^{|\mathcal{S}|}$ is an **averager** if $\Psi v = Wv$ where W is a $|\mathcal{S}| \times |\mathcal{S}|$ stochastic matrix: $w_{ij} \geq 0$ and $\sum_j w_{ij} = 1$ for all i.*

Averagers automatically satisfy the monotonicity conditions: linearity follows from matrix multiplication, monotonicity follows from nonnegativity of entries, and constant preservation follows from row sums equaling one.

Theorem 4.4 (Stability with averagers (Gordon))**.** *If Ψ is an averager and L is the Bellman operator (a γ-contraction), then $\Psi\mathrm{L}$ is a γ-contraction, and value iteration $v_{k+1} = \Psi\mathrm{L}v_k$ converges to a unique fixed point.*

This specializes the Santos-Vigo-Aguiar theorem to discrete states, expressed in the probabilistic language of stochastic matrices. The stochastic matrix characterization connects to Markov chain theory: Ψv represents expected values after one transition, and the monotonicity property reflects the fact that expectations preserve order.

Examples of averagers include state aggregation (averaging values within groups), K-nearest neighbors (averaging over nearest states), kernel smoothing with positive kernels, and multilinear interpolation on grids (barycentric weights are nonnegative and sum to one). **Counterexamples** include linear least squares regression (projection matrix may have negative entries) and high-order polynomial interpolation (Runge phenomenon produces negative weights).

Which Approximation Operators Are Monotone?

Method	Monotone?
Piecewise linear interpolation	Yes
Multilinear interpolation (grid)	Yes
Shape-preserving splines (Schumaker)	Yes
State aggregation	Yes
Kernel smoothing (positive kernels)	Yes
High-order polynomial interpolation	No
Least squares projection (arbitrary basis)	No
Fourier/spectral methods	No
Neural networks	No

distinction between "safe" (monotone) and "potentially unstable" (non-monotone) approximators provides rigorous foundation for the folk wisdom that linear interpolation is reliable while high-order polynomials can be dangerous for value iteration.

Practical Implications When using successive approximation (fixed-point iteration):

- Choose monotone approximators to guarantee convergence
- Piecewise linear interpolation, state aggregation, and kernel methods with positive kernels are safe choices
- High-order polynomials and least squares regression may fail to converge even when the Bellman operator is a strong contraction

When using rootfinding methods (Newton):

- Monotonicity is not required for convergence
- Can use smooth approximations (polynomials, splines, neural networks) for better approximation quality
- Requires good initial guesses and well-conditioned systems

- Stability depends on numerical properties of the Jacobian, not contraction preservation

Hybrid strategies:

1. Use smooth approximation for policy representation, but monotone averager for value iteration
2. Regularize smooth approximations with monotonicity constraints (monotone neural networks)
3. Run a few iterations with a monotone method to generate initial guess, then switch to Newton's method with smooth approximation
4. Solve projection equations directly (collocation with Newton) rather than iterating

This explains observed differences across research communities: reinforcement learning (traditionally using iterative TD methods) emphasized averagers, while computational economics (using collocation with Newton solvers) was more comfortable with polynomial bases.

Weighted Norms and Extensions The monotone approximation theory provides complete characterization for contraction in the sup norm. Several important extensions remain active research areas:

Weighted L^2 norms: For policy evaluation with Galerkin projection, the relevant norm is $\|\cdot\|_\xi$ where ξ is a state distribution. The contraction preservation condition becomes: ξ must be stationary under the policy's transition operator. On-policy TD methods converge while off-policy methods can diverge because the weighting distribution must match the policy dynamics.

Nonlinear approximation: Neural networks don't fit the linear operator framework. Recent work on monotone and convex neural networks attempts to recover stability through architectural constraints, but a complete theory is still emerging.

High-dimensional state spaces: Grid-based averagers become intractable due to curse of dimensionality. Understanding which non-averaging approximations provide acceptable stability-accuracy trade-offs is crucial for modern applications.

Off-policy learning: The averager framework assumes on-policy evaluation. Off-policy methods require additional machinery (importance sampling, gradient corrections) to maintain stability, even with averaging operators.

4.2.5 Galerkin Projection and Least Squares Temporal Difference

An important special case emerges when we apply Galerkin projection to the **policy evaluation** problem rather than the optimality problem. For a fixed policy π, the policy evaluation Bellman equation is:

$$v^\pi(s) = \mathrm{L}_\pi v^\pi(s) = r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, \pi(s)) v^\pi(s'). \tag{486}$$

This is a linear operator (no max), making the projection problem significantly simpler. Consider a linear function approximation $\hat{v}(s) = \boldsymbol{\varphi}(s)^\top \boldsymbol{\theta}$ where $\boldsymbol{\varphi}(s) = [\varphi_1(s), \ldots, \varphi_n(s)]^\top$ are basis functions and $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_n]^\top$ are coefficients to determine. The residual is:

$$R(s; \boldsymbol{\theta}) = \mathrm{L}_\pi \hat{v}(s) - \hat{v}(s) = r(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) \boldsymbol{\varphi}(s')^\top \boldsymbol{\theta} - \boldsymbol{\varphi}(s)^\top \boldsymbol{\theta}. \tag{487}$$

The Galerkin projection requires the residual to be orthogonal to all basis functions with respect to some weighting:

$$\sum_{s \in \mathcal{S}} \xi(s) R(s; \boldsymbol{\theta}) \varphi_j(s) = 0, \quad j = 1, \ldots, n, \tag{488}$$

where $\xi(s)$ is a distribution over states (often the stationary distribution under policy π, or uniform over visited states). Substituting the residual:

$$\sum_{s} \xi(s) \left[r(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) \boldsymbol{\varphi}(s')^\top \boldsymbol{\theta} - \boldsymbol{\varphi}(s)^\top \boldsymbol{\theta} \right] \varphi_j(s) = 0. \tag{489}$$

Rearranging and writing in matrix form, let $\boldsymbol{\Xi}$ be a diagonal matrix with $\Xi_{ss} = \xi(s)$, $\boldsymbol{\Phi}$ be the $|\mathcal{S}| \times n$ matrix with rows $\boldsymbol{\varphi}(s)^\top$, and $\mathbf{P}_\pi$ be the transition matrix under policy π. The Galerkin conditions become:

$$\boldsymbol{\Phi}^\top \boldsymbol{\Xi} (\mathbf{r}_\pi + \gamma \mathbf{P}_\pi \boldsymbol{\Phi} \boldsymbol{\theta} - \boldsymbol{\Phi} \boldsymbol{\theta}) = \mathbf{0}. \tag{490}$$

Solving for $\boldsymbol{\theta}$:

$$\boldsymbol{\Phi}^\top \boldsymbol{\Xi} (\boldsymbol{\Phi} - \gamma \mathbf{P}_\pi \boldsymbol{\Phi}) \boldsymbol{\theta} = \boldsymbol{\Phi}^\top \boldsymbol{\Xi} \mathbf{r}_\pi. \tag{491}$$

We have just derived the **Least Squares Temporal Difference (LSTD)** solution for policy evaluation. This shows that LSTD is Galerkin projection applied to the linear policy evaluation Bellman equation. The "least squares" name comes from the fact that this is the projection (in the weighted ℓ^2 sense) of the Bellman operator's output onto the span of the basis functions.

The projection perspective makes clear an important aspect of approximate dynamic programming. The solution $\boldsymbol{\theta}$ does not satisfy the true Bellman equation $v = \mathrm{L}_\pi v$ (which is typically impossible within our finite-dimensional approximation space). Instead, it satisfies $\hat{v} = \Pi \mathrm{L}_\pi \hat{v}$, where Π is the projection operator onto $\mathrm{span}\{\varphi_1, \ldots, \varphi_n\}$. We find the fixed point of the *projected* Bellman operator, not the Bellman operator itself. This is why approximation error persists even at convergence: the best we can do is find the value function whose Bellman operator output projects back onto itself.

The Projected Bellman Equations The LSTD solution gives a closed-form expression and connects to iterative algorithms developed in the next chapter. Understanding convergence of these methods requires analyzing when the projected Bellman operator $\Pi \mathrm{L}_\pi$ is a contraction.

Norms and projections. Fix a feature matrix $\mathbf{\Phi} \in R^{|\mathcal{S}| \times n}$ with full column rank and a probability distribution ξ over states. Define the ξ-weighted inner product and norm by

$$\langle u, v \rangle_\xi := \sum_s \xi(s) u(s) v(s) = u^\top \mathbf{\Xi} v, \qquad \|v\|_\xi := \sqrt{v^\top \mathbf{\Xi} v}, \tag{492}$$

where $\mathbf{\Xi} = \mathrm{diag}(\xi)$. The orthogonal projection onto $\mathrm{span}(\mathbf{\Phi})$ with respect to $\langle \cdot, \cdot \rangle_\xi$ is

$$\Pi = \mathbf{\Phi}(\mathbf{\Phi}^\top \mathbf{\Xi} \mathbf{\Phi})^{-1} \mathbf{\Phi}^\top \mathbf{\Xi}. \tag{493}$$

An operator T is a β**-contraction** in norm $\|\cdot\|$ if $\|\mathrm{T}v - \mathrm{T}w\| \leq \beta \|v - w\|$ for all v, w and some $\beta < 1$. It is a **non-expansion** if the same holds with $\beta = 1$.

Why Π is a non-expansion. This follows from the Pythagorean identity in weighted inner product spaces. For any $u \in R^{|\mathcal{S}|}$, the projection Πu and the residual $(I - \Pi)u$ are ξ-orthogonal: $\langle \Pi u, (I - \Pi)u \rangle_\xi = 0$. Therefore,

$$\|u\|_\xi^2 = \|\Pi u\|_\xi^2 + \|(I - \Pi)u\|_\xi^2. \tag{494}$$

Applying this to $u - v$ gives

$$\|\Pi u - \Pi v\|_\xi^2 = \|\Pi(u - v)\|_\xi^2 \leq \|\Pi(u - v)\|_\xi^2 + \|(I - \Pi)(u - v)\|_\xi^2 = \|u - v\|_\xi^2, \tag{495}$$

proving $\|\Pi u - \Pi v\|_\xi \leq \|u - v\|_\xi$.

When is L_π a contraction in $\|\cdot\|_\xi$? Write the policy evaluation operator as $\mathrm{L}_\pi v = r_\pi + \gamma \mathbf{P}_\pi v$, where $\mathbf{P}_\pi$ is the transition matrix under policy π. We know L_π is a γ-contraction in $\|\cdot\|_\infty$ from earlier chapters. However, whether it contracts in $\|\cdot\|_\xi$ depends on the relationship between ξ and $\mathbf{P}_\pi$.

We need to establish when the stochastic matrix $\mathbf{P}_\pi$ is non-expansive in $\|\cdot\|_\xi$. Following Bertsekas (Lemma 6.3.1), suppose ξ is a **steady-state probability vector** for $\mathbf{P}_\pi$ with positive components, meaning:

$$\xi^\top \mathbf{P}_\pi = \xi^\top, \qquad \text{or equivalently,} \qquad \xi(s') = \sum_s \xi(s) p(s' | s, \pi(s)) \text{ for all } s'. \tag{496}$$

Then for any $z \in R^{|\mathcal{S}|}$, using the convexity of the square function (Jensen's inequality):

$$\|\mathbf{P}_\pi z\|_\xi^2 = \sum_s \xi(s) \left(\sum_{s'} p(s'|s, \pi(s)) z(s') \right)^2$$
$$\leq \sum_s \xi(s) \sum_{s'} p(s'|s, \pi(s)) z(s')^2$$
$$= \sum_{s'} \left(\sum_s \xi(s) p(s'|s, \pi(s)) \right) z(s')^2$$

Using the defining property of steady-state probabilities $\sum_s \xi(s) p(s'|s, \pi(s)) = \xi(s')$:

$$= \sum_{s'} \xi(s') z(s')^2 = \|z\|_\xi^2. \tag{497}$$

Therefore $\|\mathbf{P}_\pi z\|_\xi \leq \|z\|_\xi$, showing that $\mathbf{P}_\pi$ is non-expansive in $\|\cdot\|_\xi$. Since $\|\mathrm{L}_\pi v - \mathrm{L}_\pi w\|_\xi = \gamma \|\mathbf{P}_\pi (v - w)\|_\xi$:

$$\|\mathrm{L}_\pi v - \mathrm{L}_\pi w\|_\xi \leq \gamma \|v - w\|_\xi. \tag{498}$$

Thus L_π is a γ-contraction in $\|\cdot\|_\xi$ when ξ is the steady-state distribution of π.

Contraction of the composition. Combining our two results: Π is a non-expansion and (under stationarity) L_π is a γ-contraction in $\|\cdot\|_\xi$. Therefore,

$$\|\Pi \mathrm{L}_\pi v - \Pi \mathrm{L}_\pi w\|_\xi \leq \|\mathrm{L}_\pi v - \mathrm{L}_\pi w\|_\xi \leq \gamma \|v - w\|_\xi. \tag{499}$$

By the Banach fixed-point theorem, $\Pi \mathrm{L}_\pi$ has a unique fixed point in $R^{|\mathcal{S}|}$, and iterates $v_{k+1} = \Pi \mathrm{L}_\pi v_k$ converge to it from any v_0. This fixed point satisfies the **projected Bellman equation**

$$v = \Pi(r_\pi + \gamma \mathbf{P}_\pi v), \qquad v \in \mathrm{span}(\mathbf{\Phi}). \tag{500}$$

Writing $v = \mathbf{\Phi}\boldsymbol{\theta}$ and left-multiplying by $\mathbf{\Phi}^\top \mathbf{\Xi}$ yields the **normal equations**

$$\mathbf{\Phi}^\top \mathbf{\Xi} (\mathbf{\Phi} - \gamma \mathbf{P}_\pi \mathbf{\Phi}) \boldsymbol{\theta} = \mathbf{\Phi}^\top \mathbf{\Xi} r_\pi, \tag{501}$$

which are precisely the LSTD equations we derived earlier. This result provides the theoretical foundation for temporal difference learning with linear function approximation: when learning on-policy (so ξ is stationary), convergence is guaranteed.

Off-policy instability. When ξ is not stationary for $\mathbf{P}_\pi$ (as occurs when data come from a different behavior policy), the Jensen argument breaks down. The transition operator $\mathbf{P}_\pi$ need not be non-expansive in $\|\cdot\|_\xi$, so $\Pi \mathrm{L}_\pi$ may fail to be a contraction. This is the root cause of off-policy divergence phenomena in linear TD learning (e.g., Baird's counterexample). Importance weighting and other corrections are designed to restore stability in this regime.

The linearity of the policy evaluation operator L_π is what gives us the closed-form solution. We could apply Galerkin projection to the Bellman optimality equation $v^* = \mathrm{L}v^*$, setting up orthogonality conditions $\sum_s \xi(s) R(s;\boldsymbol{\theta})\varphi_j(s) = 0$. The max operator makes these conditions nonlinear in $\boldsymbol{\theta}$, eliminating the closed form and requiring iterative solution. This brings us back to the successive approximation methods discussed earlier for collocation.

4.2.6 Fitted-Value Iteration: Beyond Linear Projection

Throughout this chapter, we have focused on polynomial approximations and linear projections, where the value function is represented as $v(s) = \sum_{j=1}^{d} \theta_j \varphi_j(s)$ and the projection operator Π solves a linear system to find the best coefficients $\boldsymbol{\theta}$. This framework, while analytically tractable, is just one instance of a much more general pattern that encompasses modern function approximation methods including neural networks, decision trees, kernel methods, and ensemble models.

The projection operator Π need not be a linear projection at all. Instead, it can be any computational procedure that fits an approximator to target data. This generalization leads us to **fitted-value iteration** (FVI), a universal template for approximate dynamic programming that subsumes classical projection methods as special cases while extending naturally to black-box function approximators.

The Fitting Operator Rather than thinking of Π as a mathematical projection onto a subspace, we can view it as a **fitting operator** `fit` that takes a dataset of state-value pairs $\{(s_i, y_i)\}_{i=1}^n$ and produces a function $\hat{v} \in \mathcal{F}$ from some hypothesis class $\mathcal{F}$:

$$\hat{v} = \texttt{fit}\big(\{(s_i, y_i)\}_{i=1}^n; \mathcal{F}\big). \tag{502}$$

The specifics of `fit` depend on the function approximator:

- **Linear basis functions**: `fit` solves the weighted least-squares problem $\min_{\boldsymbol{\theta}} \sum_{i=1}^n \xi(s_i)\big(y_i - \sum_j \theta_j \varphi_j(s_i)\big)^2$, yielding the Galerkin projection we studied earlier.
- **Neural networks**: `fit` runs stochastic gradient descent to minimize the loss $\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^n (y_i - v_{\boldsymbol{\theta}}(s_i))^2$, where $v_{\boldsymbol{\theta}}$ is a neural network with parameters $\boldsymbol{\theta}$.
- **Decision trees**: `fit` constructs a regression tree by recursively partitioning the state space to minimize the sum of squared residuals within each leaf.
- **Kernel methods**: `fit` computes kernel weights α_i such that $\hat{v}(s) = \sum_{i=1}^n \alpha_i k(s, s_i)$ for some kernel function k.

- **Ensemble methods**: `fit` trains multiple base learners (e.g., via boosting or bagging) and combines their predictions.

In each case, `fit` maps a training dataset to a function, but the internal mechanism varies widely: closed-form solution, iterative optimization, tree construction, or ensemble aggregation.

Fitted-Value Iteration as Successive Approximation Fitted-value iteration is the natural extension of the successive approximation methods we examined earlier. Starting from an initial approximation $v_0 \in \mathcal{F}$, we iterate:

1. **Apply the Bellman operator** at sample states $\{s_i\}_{i=1}^n$ to compute target values:

$$y_i = (\mathrm{L}v_k)(s_i) = \max_{a \in \mathcal{A}} \left\{ r(s_i, a) + \gamma \sum_{s'} p(s' \mid s_i, a) v_k(s') \right\}. \tag{503}$$

2. **Fit a new approximation** to the targets using the fitting operator:

$$v_{k+1} = \texttt{fit}\big(\{(s_i, y_i)\}_{i=1}^n; \mathcal{F}\big). \tag{504}$$

This two-step rhythm (operator application followed by function fitting) mirrors exactly the structure of projected value iteration, where we computed $\Pi \mathrm{L} v_k$. The difference is that Π has been replaced by the more general `fit`, which need not be a linear projection and need not have a closed form.

The abstraction `fit` encapsulates all the complexity of function approximation, whether that involves solving a linear system, running gradient descent for thousands of steps, growing a decision tree, or training an ensemble. From the algorithmic perspective, these are simply different implementations of the same conceptual operation: mapping a dataset to a function.

Connection to the Projection Framework How does fitted-value iteration relate to the projection methods we studied? The connection becomes clear when we recognize that the projection operator Π from earlier sections is simply one particular instantiation of `fit`:

- When $\mathcal{F}$ is a linear subspace spanned by basis functions $\{\varphi_j\}_{j=1}^d$, and `fit` minimizes the weighted squared error $\sum_i \xi(s_i)(y_i - \sum_j \theta_j \varphi_j(s_i))^2$, then `fit` coincides exactly with the Galerkin projection Π.
- The collocation method emerges when we choose evaluation points $\{s_i\}_{i=1}^d$ equal in number to the basis functions and `fit` enforces exact interpolation: $\sum_j \theta_j \varphi_j(s_i) = y_i$ for all i.
- Least-squares temporal difference (LSTD) methods, which we will encounter in the next chapter, can be viewed as performing `fit` on data collected from simulation rather than the full state space.

The residual function $R(s;\boldsymbol{\theta}) = v(s;\boldsymbol{\theta}) - (\mathrm{L}v(\cdot;\boldsymbol{\theta}))(s)$ still governs approximation quality. In the linear case, we minimized $\sum_s \xi(s)R(s;\boldsymbol{\theta})^2$ by setting up orthogonality conditions. For nonlinear approximators, the residual remains the fundamental object of interest, but we typically cannot enforce orthogonality analytically. Instead, `fit` implicitly seeks to make the residual small in an empirical sense over the training data.

Linear basis function approximation has many virtues: closed-form solutions, theoretical tractability, and well-understood convergence properties. But polynomial and radial basis functions require hand-crafted features, which can be difficult to design for high-dimensional state spaces or complex value function geometry. Neural networks, decision trees, kernel methods, and ensemble models can learn representations automatically or adapt their complexity to the data. The price is that we lose closed-form solutions and convergence guarantees, trading theoretical tractability for representational flexibility.

Toward Simulation-Based Methods A limitation of the fitted-value iteration algorithm as presented is that it assumes we can evaluate the Bellman operator exactly at every state. That is, computing $y_i = (\mathrm{L}v_k)(s_i)$ requires knowing the transition probabilities $p(j \mid s_i, a)$ and being able to sum or integrate over all possible next states j.

In many real-world problems, we have neither. We might have access only to a simulator that generates sample transitions (s, a, r, j), or to a dataset of observed trajectories. We might not even know the full state space in advance. This brings us to the threshold of **simulation-based approximate dynamic programming** and **reinforcement learning**, where the Bellman operator must be approximated from samples rather than computed exactly.

The projection and collocation methods developed in this chapter provide the conceptual foundation for these simulation-based methods. The residual $R(s;\boldsymbol{\theta}) = v(s;\boldsymbol{\theta}) - (\mathrm{L}v(\cdot;\boldsymbol{\theta}))(s)$ remains the central object. But instead of enforcing orthogonality conditions on the full state space, we will minimize empirical residuals on sampled data. The fitting operator `fit` will take in noisy, incomplete samples rather than exact values. And convergence will be probabilistic, characterized by sample complexity rather than deterministic fixed-point theorems.

The next chapter introduces Monte Carlo integration and temporal-difference learning, showing how to estimate the expectations in the Bellman operator from simulated experience. Together with the fitted-value iteration framework developed here, these tools form the backbone of modern approximate dynamic programming, connecting classical numerical methods to the data-driven paradigm of reinforcement learning.

4.3 Simulation-Based Approximate Dynamic Programming

In the previous chapter, we developed the projection method framework for solving functional equations. We saw how to transform the infinite-dimensional problem of finding v such that $\mathrm{L}v = v$ into a finite-dimensional one by choosing basis functions and projection conditions (least squares, Galerkin, collocation). The main idea was to make the residual $R(s) = \mathrm{L}v(s) - v(s)$ "small" according to some criterion—whether by minimizing its squared norm, requiring orthogonality to basis functions, or forcing it to vanish at specific collocation points.

However, we left one critical question unresolved: **how do we actually evaluate the Bellman operator** at a state s? Recall that applying L requires computing an expectation:

$$(\mathrm{L}v)(s) = \max_{a \in \mathcal{A}_s} \left\{ r(s,a) + \gamma \int v(s') p(ds'|s,a) \right\} \tag{505}$$

In our discussion of projection methods, we remained agnostic about numerical integration—we didn't commit to a specific technique for evaluating this integral. When the state space is discrete and small, this expectation is simply a finite sum we can compute exactly. When the state space is continuous or very large, we face a computational challenge.

This chapter addresses that challenge by introducing **Monte Carlo integration** as our method for approximating expectations. Rather than requiring explicit knowledge of the transition probability function $p(ds'|s,a)$ and using deterministic quadrature rules, we will work with **samples**—observed next states drawn from the transition dynamics. This is the defining feature of what the operations research community calls **simulation-based approximate dynamic programming**: the combination of projection methods (to handle infinite-dimensional function spaces) with Monte Carlo integration (to handle intractable expectations via sampling).

This same combination is precisely what the machine learning community recognizes as **reinforcement learning**. By relying on samples rather than exact probabilities, we move from planning with a known model to **learning from data**. The algorithms we develop will work in settings where we can simulate or interact with the system but may not have access to—or may not want to use—explicit transition probabilities. This is the bridge from the model-based dynamic programming of earlier chapters to the data-driven learning setting that defines modern RL.

Discretization and Numerical Quadrature Recall that the Bellman operator for continuous state spaces takes the form:

$$(\mathrm{L}v)(s) \equiv \max_{a \in \mathcal{A}_s} \left\{ r(s,a) + \gamma \int v(s') p(ds'|s,a) \right\}, \forall s \in \mathcal{S} \tag{506}$$

To make this computationally tractable for continuous state spaces, we can discretize the state space and approximate the integral over this discretized

space. While this allows us to evaluate the Bellman operator componentwise, we must first decide how to represent value functions in this discretized setting.

When working with discretized representations, we partition the state space into N_s cells with centers at grid points $\{s_i\}_{i=1}^{N_s}$. We then work with value functions that are piecewise constant on each cell: ie. for any $s \in S$: $v(s) = v(s_{k(s)})$ where $k(s)$ is the index of the cell containing s. We denote the discretized reward function by $r_h(s,a) \equiv r(s_{k(s)}, a)$.

For transition probabilities, we need to be more careful. While we similarly map any state-action pair to its corresponding cell, we must ensure that integrating over the discretized transition function yields a valid probability distribution. We achieve this by normalizing:

$$p_h(s'|s,a) \equiv \frac{p(s_{k(s')}|s_{k(s)},a)}{\int p(s_{k(s')}|s_{k(s)},a)ds'} \tag{507}$$

After defining our discretized reward and transition probability functions, we can write down our discretized Bellman operator. We start with the Bellman operator using our discretized functions r_h and p_h. While these functions map to grid points, they're still defined over continuous spaces - we haven't yet dealt with the computational challenge of the integral. With this discretization approach, the value function is piecewise constant over cells. This lets us express the integral as a sum over cells, where each cell's contribution is the probability of transitioning to that cell multiplied by the value at that cell's grid point:

$$\begin{aligned}
(\widehat{\mathrm{L}}_h v)(s) &= \max_{k=1,\ldots,N_a} \left\{ r_h(s,a_k) + \gamma \int v(s') p_h(s'|s,a_k) ds' \right\} \\
&= \max_{k=1,\ldots,N_a} \left\{ r_h(s,a_k) + \gamma \int v(s_{k(s')}) p_h(s'|s,a_k) ds' \right\} \\
&= \max_{k=1,\ldots,N_a} \left\{ r_h(s,a_k) + \gamma \sum_{i=1}^{N_s} v(s_i) \int_{cell_i} p_h(s'|s,a_k) ds' \right\} \\
&= \max_{k=1,\ldots,N_a} \left\{ r_h(s,a_k) + \gamma \sum_{i=1}^{N_s} v(s_i) p_h(s_i|s_{k(s)},a_k) \right\}
\end{aligned} \tag{508}$$

This form makes clear how discretization converts our continuous-space problem into a finite computation: we've replaced integration over continuous space with summation over grid points. The price we pay is that the number of terms in our sum grows exponentially with the dimension of our state space - the familiar curse of dimensionality.

Monte Carlo Integration Numerical quadrature methods scale poorly with increasing dimension. Specifically, for a fixed error tolerance ϵ, the number of required quadrature points grows exponentially with dimension d as $O\left(\left(\frac{1}{\epsilon}\right)^d\right)$. Furthermore, quadrature methods require explicit evaluation of the transition

probability function $p(s'|s,a)$ at specified points—a luxury we don't have in the "model-free" setting where we only have access to samples from the MDP.

Let $\mathcal{B} = \{s_1, \ldots, s_M\}$ be our set of base points where we will evaluate the operator. At each base point $s_k \in \mathcal{B}$, Monte Carlo integration approximates the expectation using N samples:

$$\int v_n(s')p(ds'|s_k,a) \approx \frac{1}{N}\sum_{i=1}^{N} v_n(s'_{k,i}), \quad s'_{k,i} \sim p(\cdot|s_k,a) \tag{509}$$

where $s'_{k,i}$ denotes the i-th sample drawn from $p(\cdot|s_k,a)$ for base point s_k. This approach has two properties making it particularly attractive for high-dimensional problems and model-free settings:

1. The convergence rate is $O\left(\frac{1}{\sqrt{N}}\right)$ regardless of the number of dimensions
2. It only requires samples from $p(\cdot|s_k,a)$, not explicit probability values

Using this approximation of the expected value over the next state, we can define a new "empirical" Bellman optimality operator:

$$(\widehat{\mathrm{L}}_N v)(s_k) \equiv \max_{a\in\mathcal{A}_{s_k}} \left\{ r(s_k,a) + \frac{\gamma}{N}\sum_{i=1}^{N} v(s'_{k,i}) \right\}, \quad s'_{k,i} \sim p(\cdot|s_k,a) \tag{510}$$

for each $s_k \in \mathcal{B}$. A direct adaptation of the successive approximation method for this empirical operator leads to:

Note that the original error bound derived as a termination criterion for value iteration need not hold in this approximate setting. Hence, we use a generic termination criterion based on computational budget and desired tolerance. While this aspect could be improved, we'll focus on a more pressing matter: the algorithm's tendency to produce upwardly biased values. In other words, this algorithm "thinks" the world is more rosy than it actually is - it overestimates values.

Overestimation Bias in Monte Carlo Value Iteration In statistics, bias refers to a systematic error where an estimator consistently deviates from the true parameter value. For an estimator $\hat{\theta}$ of a parameter θ, we define bias as: $\mathrm{Bias}(\hat{\theta}) = E[\hat{\theta}] - \theta$. While bias isn't always problematic — sometimes we deliberately introduce bias to reduce variance, as in ridge regression — uncontrolled bias can lead to significantly distorted results. In the context of value iteration, this distortion gets amplified even more due to the recursive nature of the algorithm.

Consider how the Bellman operator works in value iteration. At iteration n, we have a value function estimate $v_i(s)$ and aim to improve it by applying the Bellman operator L. The ideal update would be:

$$(\mathrm{L}v_i)(s) = \max_{a \in \mathcal{A}(s)} \left\{ r(s,a) + \gamma \int v_i(s') p(ds'|s,a) \right\} \tag{511}$$

However, we can't compute this integral exactly and use Monte Carlo integration instead, drawing N next-state samples for each state and action pair. The bias emerges when we take the maximum over actions:

$$(\widehat{\mathrm{L}}v_i)(s) = \max_{a \in \mathcal{A}(s)} \hat{q}_i(s,a), \text{ where } \hat{q}_i(s,a) \equiv r(s,a) + \frac{\gamma}{N} \sum_{j=1}^{N} v_i(s'_j), \quad s'_j \sim p(\cdot|s,a) \tag{512}$$

White the Monte Carlo estimate $\hat{q}_n(s,a)$ is unbiased for any individual action, the empirical Bellman operator is biased upward due to Jensen's inequality, which states that for any convex function f, we have $E[f(X)] \geq f(E[X])$. Since the maximum operator is convex, this implies:

$$E[(\widehat{\mathrm{L}}v_i)(s)] = E\left[\max_{a \in \mathcal{A}(s)} \hat{q}_i(s,a) \right] \geq \max_{a \in \mathcal{A}(s)} E[\hat{q}_i(s,a)] = (\mathrm{L}v_i)(s) \tag{513}$$

This means that our Monte Carlo approximation of the Bellman operator is biased upward:

$$b_i(s) = E[(\widehat{\mathrm{L}}v_i)(s)] - (\mathrm{L}v_i)(s) \geq 0 \tag{514}$$

Even worse, this bias compounds through iterations as each new value function estimate v_{n+1} is based on targets generated by the biased operator $\widehat{\mathrm{L}}$, creating a nested structure of bias accumulation. This bias remains nonnegative at every step, and each application of the Bellman operator potentially adds more upward bias. As a result, instead of converging to the true value function v^*, the algorithm typically stabilizes at a biased approximation that systematically overestimates true values.

The Keane-Wolpin Bias Correction Algorithm Keane and Wolpin proposed to de-bias such estimators by essentially "learning" the bias, then subtracting it when computing the empirical Bellman operator. If we knew this bias function, we could subtract it from our empirical estimate to get an unbiased estimate of the true Bellman operator:

$$(\widehat{\mathrm{L}}v_n)(s) - \text{bias}(s) = (\widehat{\mathrm{L}}v_n)(s) - (E[(\widehat{\mathrm{L}}v_n)(s)] - (\mathrm{L}v_n)(s)) \approx (\mathrm{L}v_n)(s) \tag{515}$$

This equality holds in expectation, though any individual estimate would still have variance around the true value.

So how can we estimate the bias function? The Keane-Wolpin manages this using an important fact from extreme value theory: for normal random

variables, the difference between the expected maximum and maximum of expectations scales with the standard deviation:

$$E\left[\max_{a\in\mathcal{A}} \hat{q}_i(s,a)\right] - \max_{a\in\mathcal{A}} E[\hat{q}_i(s,a)] \approx c \cdot \sqrt{\max_{a\in\mathcal{A}} \mathrm{Var}_i(s,a)} \tag{516}$$

The variance term $\max_{a\in\mathcal{A}} \mathrm{Var}_i(s,a)$ will typically be dominated by the action with the largest value – the greedy action $a_i^*(s)$. Rather than deriving the constant c theoretically, Keane-Wolpin proposed learning the relationship between variance and bias empirically through these steps:

1. Select a small set of "benchmark" states (typically 20-50) that span the state space
2. For these states, compute more accurate value estimates using many more Monte Carlo samples (10-100x more than usual)
3. Compute the empirical bias at each benchmark state s:

$$\hat{b}_i(s) = (\hat{\mathrm{L}}v_i)(s) - (\hat{\mathrm{L}}_{\text{accurate}}v_i)(s) \tag{517}$$

4. Fit a linear relationship between this bias and the variance at the greedy action:

$$\hat{b}_i(s) = \alpha_i \cdot \mathrm{Var}_i(s, a_i^*(s)) + \epsilon \tag{518}$$

This creates a dataset of pairs $(\mathrm{Var}_i(s, a_i^*(s)), \hat{b}_i(s))$ that can be used to estimate α_i through ordinary least squares regression. Once we have learned this bias function $\hat{b}$, we can define the bias-corrected Bellman operator:

$$(\widetilde{\mathrm{L}}v_i)(s)(\hat{\mathrm{L}}v_i)(s) - \hat{b}(s) \tag{519}$$

While this bias correction approach has been influential in econometrics, it hasn't gained much traction in the machine learning community. A major drawback is the need for accurate operator estimation at benchmark states, which requires allocating substantially more samples to these states. In the next section, we'll explore an alternative strategy that, while requiring the maintenance of two sets of value estimates, achieves bias correction without demanding additional samples.

Decoupling Selection and Evaluation A simpler approach to addressing the upward bias is to maintain two separate q-function estimates - one for action selection and another for evaluation. Let's first start by looking at the corresponding Monte Carlo value iteration algorithm and then convince ourselves that this is good idea using math. Assume a monte carlo integration setup over Q factors:

In this algorithm, we maintain two separate Q-functions (q^A and q^B) and use them asymmetrically: when updating q^A, we use network A to select the best

action ($a_i^* = \arg\max_{a'} q_i^A(s'_j, a')$) but then evaluate that action using network B's estimates ($q_i^B(s'_j, a_i^*)$). We do the opposite for updating q^B. You can see this separation in steps 3.2.2 and 3.2.3 of the algorithm, where for each network update, we first use one network to pick the action and then plug that chosen action into the other network for evaluation. We will see that this decomposition helps mitigate the positive bias that occurs due to Jensen's inequality.

An HVAC analogy Consider a building where each HVAC unit i has some true maximum power draw μ_i under worst-case conditions. Let's pretend that we don't have access to manufacturer datasheets, so we need to estimate these maxima from actual measurements. Now the challenge is that power draw fluctuates with environmental conditions. If we use a single day's measurements and look at the highest power draw, we systematically overestimate the true maximum draw across all units.

To see this, let X_A^i be unit i's power draw on day A and X_B^i be unit i's power draw on day. Wile both measurements are unbiased $E[X_A^i] = E[X_B^i] = \mu_i$, their maximum is not due to Jensen's inequality:

$$E[\max_i X_A^i] \geq \max_i E[X_A^i] = \max_i \mu_i \tag{520}$$

Intuitively, this problem occurs because reading tends to come from units that experienced particularly demanding conditions (e.g., direct sunlight, full occupancy, peak humidity) rather than just those with high true maximum draw. To estimate the true maximum power draw more accurately, we use the following measurement protocol:

1. Use day A measurements to **select** which unit hit the highest peak
2. Use day B measurements to **evaluate** that unit's power consumption

This yields the estimator:

$$Y = X_B^{\arg\max_i X_A^i} \tag{521}$$

We can show that by decoupling selection and evaluation in this fashion, our estimator Y will no longer systematically overestimate the true maximum draw. First, observe that $\arg\max_i X_A^i$ is a random variable (call it J) - it tells us which unit had highest power draw on day A. It has some probability distribution based on day A's conditions: $P(J = j) = P(\arg\max_i X_A^i = j)$. Using the law of total expectation:

$$\begin{aligned} E[Y] = E[X_B^J] &= E[E[X_B^J \mid J]] \text{ (by tower property)} \\ &= \sum_{j=1}^{n} E[X_B^j \mid J = j] P(J = j) \\ &= \sum_{j=1}^{n} E[X_B^j \mid \arg\max_i X_A^i = j] P(\arg\max_i X_A^i = j) \end{aligned}$$

Now we need to make an imporant observation: Unit j's power draw on day B (X_B^j) is independent of whether it had the highest reading on day A ($\{\arg\max_i X_A^i = j\}$). An extreme cold event on day A shouldn't affect day B's readings(especially in Quebec where the wheather tend to vary widely from day to day). Therefore:

$$E[X_B^j \mid \arg\max_i X_A^i = j] = E[X_B^j] = \mu_j \tag{522}$$

This tells us that the two-day estimator is now an average of the true underlying power consumptions:

$$E[Y] = \sum_{j=1}^{n} \mu_j P(\arg\max_i X_A^i = j) \tag{523}$$

To analyze $E[Y]$ more closely, let's use a general result: if we have a real-valued function f defined on a discrete set of units $\{1, \ldots, n\}$ and a probability distribution $q(\cdot)$ over these units, then the maximum value of f across all units is at least as large as the weighted sum of f values with weights q. Formally,

$$\max_{j \in \{1,\ldots,n\}} f(j) \geq \sum_{j=1}^{n} q(j) f(j). \tag{524}$$

Applying this to our setting, we set $f(j) = \mu_j$ (the true maximum power draw for unit j) and $q(j) = P(J = j)$ (the probability that unit j achieves the maximum reading on day A). This gives us:

$$\max_{j \in \{1,\ldots,n\}} \mu_j \geq \sum_{j=1}^{n} P(J = j)\mu_j = E[Y]. \tag{525}$$

Therefore, the expected value of Y (our estimator) will always be less than or equal to the true maximum value $\max_j \mu_j$. In other words, Y provides a **conservative estimate** of the true maximum: it tends not to overestimate $\max_j \mu_j$ but instead approximates it as closely as possible without systematic upward bias.

Consistency Even though Y is not an unbiased estimator of $\max_j \mu_j$ (since $E[Y] \leq \max_j \mu_j$), it is **consistent**. As more independent days (or measurements) are observed, the selection-evaluation procedure becomes more effective at isolating the intrinsic maximum, reducing the influence of day-specific environmental fluctuations. Over time, this approach yields a stable and increasingly accurate approximation of $\max_j \mu_j$.

To show that Y is a consistent estimator of $\max_i \mu_i$, we want to demonstrate that as the number of independent measurements (days, in this case) increases, Y converges in probability to $\max_i \mu_i$. Let's suppose we have m independent days of measurements for each unit. Denote:

- $X_A^{(k),i}$ as the power draw for unit i on day A_k, where $k \in \{1, \ldots, m\}$.

- $J_m = \arg\max_i \left(\frac{1}{m} \sum_{k=1}^{m} X_A^{(k),i} \right)$, which identifies the unit with the highest average power draw over m days.

The estimator we construct is: $Y_m = X_B^{(J_m)}$, where $X_B^{(J_m)}$ is the power draw of the selected unit J_m on an independent evaluation day B. We will now show that Y_m converges to $\max_i \mu_i$ as $m \to \infty$. This involves two main steps:

1. **Consistency of the Selection Step** J_m: As $m \to \infty$, the unit selected by J_m will tend to be the one with the true maximum power draw $\max_i \mu_i$.
2. **Convergence of** Y_m **to** μ_{J_m}: Since the evaluation day B measurement $X_B^{(J_m)}$ is unbiased with expectation μ_{J_m}, as $m \to \infty$, Y_m will converge to μ_{J_m}, which in turn converges to $\max_i \mu_i$.

The average power draw over m days for each unit i is:

$$\frac{1}{m} \sum_{k=1}^{m} X_A^{(k),i}. \tag{526}$$

By the law of large numbers, as $m \to \infty$, this sample average converges to the true expected power draw μ_i for each unit i:

$$\frac{1}{m} \sum_{k=1}^{m} X_A^{(k),i} \xrightarrow{m \to \infty} \mu_i. \tag{527}$$

Since J_m selects the unit with the highest sample average, in the limit, J_m will almost surely select the unit with the highest true mean, $\max_i \mu_i$. Thus, as $m \to \infty$,

$$\mu_{J_m} \to \max_i \mu_i. \tag{528}$$

Given that J_m identifies the unit with the maximum true mean power draw in the limit, we now look at $Y_m = X_B^{(J_m)}$, which is the power draw of unit J_m on the independent evaluation day B.

Since $X_B^{(J_m)}$ is an unbiased estimator of μ_{J_m}, we have:

$$E[Y_m \mid J_m] = \mu_{J_m}. \tag{529}$$

As $m \to \infty$, μ_{J_m} converges to $\max_i \mu_i$. Thus, Y_m will also converge in probability to $\max_i \mu_i$ because Y_m is centered around μ_{J_m} and J_m converges to the index of the unit with $\max_i \mu_i$.

Combining these two steps, we conclude that:

$$Y_m \xrightarrow{m \to \infty} \max_i \mu_i \text{ in probability.} \tag{530}$$

This establishes the **consistency** of Y as an estimator for $\max_i \mu_i$: as the number of independent measurements grows, Y_m converges to the true maximum power draw $\max_i \mu_i$.

4.3.1 Parametric Dynamic Programming

We have so far considered a specific kind of approximation: that of the Bellman operator itself. We explored a modified version of the operator with the desirable property of smoothness, which we deemed beneficial for optimization purposes and due to its rich multifaceted interpretations. We now turn our attention to another form of approximation, complementary to the previous kind, which seeks to address the challenge of applying the operator across the entire state space.

To be precise, suppose we can compute the Bellman operator $\mathrm{L}v$ at some state s, producing a new function U whose value at state s is $u(s) = (\mathrm{L}v)(s)$. Then, putting aside the problem of pointwise evaluation, we want to carry out this update across the entire domain of v. When working with small state spaces, this is not an issue, and we can afford to carry out the update across the entirety of the state space. However, for larger or infinite state spaces, this becomes a major challenge.

So what can we do? Our approach will be to compute the operator at chosen "grid points," then "fill in the blanks" for the states where we haven't carried out the update by "fitting" the resulting output function on a dataset of input-output pairs. The intuition is that for sufficiently well-behaved functions and sufficiently expressive function approximators, we hope to generalize well enough. Our community calls this "learning," while others would call it "function approximation" — a field of its own in mathematics. To truly have a "learning algorithm," we'll need to add one more piece of machinery: the use of samples — of simulation — to pick the grid points and perform numerical integration. But this is for the next section...

Partial Updates in the Tabular Case The ideas presented in this section apply more broadly to the successive approximation method applied to a fixed-point problem. Consider again the problem of finding the optimal value function $v^\star_\gamma$ as the solution to the Bellman optimality operator L:

$$\mathrm{L}\mathbf{v} \equiv \max_{\pi \in \Pi^{MD}} \{\mathbf{r}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}\} \tag{531}$$

Value iteration – the name for the method of successive approximation applied to L – computes a sequence of iterates $v_{n+1} = \mathrm{L}v_n$ from some arbitrary v_0. Let's pause to consider what the equality sign in this expression means: it represents an assignment (perhaps better denoted as :=) across the entire domain. This becomes clearer when writing the update in component form:

$$v_{n+1}(s) := (\mathrm{L}v_n)(s) \equiv \max_{a \in \mathcal{A}_s} \left\{ r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v_n(j) \right\}, \forall s \in \mathcal{S} \tag{532}$$

Pay particular attention to the $\forall s \in \mathcal{S}$ notation: what happens when we can't afford to update all components in each step of value iteration? A potential

solution is to use Gauss-Seidel Value Iteration, which updates states sequentially, immediately using fresh values for subsequent updates.

The Gauss-Seidel value iteration approach offers several advantages over standard value iteration: it can be more memory-efficient and often leads to faster convergence. This idea generalizes further (see for example Bertsekas [1983]) to accommodate fully asynchronous updates in any order. However, these methods, while more flexible in their update patterns, still fundamentally rely on a tabular representation—that is, they require storing and eventually updating a separate value for each state in memory. Even if we update states one at a time or in blocks, we must maintain this complete table of values, and our convergence guarantee assumes that every entry in this table will eventually be revised.

But what if maintaining such a table is impossible? This challenge arises naturally when dealing with continuous state spaces, where we cannot feasibly store values for every possible state, let alone update them. This is where function approximation comes into play.

Parametric Value Iteration In the parametric approach to dynamic programming, instead of maintaining an explicit table of values, we represent the value function using a parametric function approximator $v(s; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are parameters that get adjusted across iterations rather than the entries of a tabular representation. This idea traces back to the inception of dynamic programming and was described as early as 1963 by Bellman himself, who considered polynomial approximations. For a value function $v(s)$, we can write its polynomial approximation as:

$$v(s) \approx \sum_{i=0}^{n} \theta_i \phi_i(s) \tag{533}$$

where:

- $\{\phi_i(s)\}$ is the set of basis functions
- θ_i are the coefficients (our parameters)
- n is the degree of approximation

As we discussed earlier in the context of trajectory optimization, we can choose from different polynomial bases beyond the usual monomial basis $\phi_i(s) = s^i$, such as Legendre or Chebyshev polynomials. While polynomials offer attractive mathematical properties, they become challenging to work with in higher dimensions due to the curse of dimensionality. This limitation motivates our later turn to neural network parameterizations, which scale better with dimensionality.

Given a parameterization, our value iteration procedure must now update the parameters $\boldsymbol{\theta}$ rather than tabular values directly. At each iteration, we

aim to find parameters that best approximate the Bellman operator's output at chosen base points. More precisely, we collect a dataset:

$$\mathcal{D}_n = \{(s_i, (\mathrm{L}v)(s_i; \boldsymbol{\theta}_n)) \mid s_i \in B\} \tag{534}$$

and fit a regressor $v(\cdot; \boldsymbol{\theta}_{n+1})$ to this data.

This process differs from standard supervised learning in a specific way: rather than working with a fixed dataset, we iteratively generate our training targets using the previous value function approximation. During this process, the parameters $\boldsymbol{\theta}_n$ remain "frozen", entering only through dataset creation. This naturally leads to maintaining two sets of parameters:

- $\boldsymbol{\theta}_n$: parameters of the target model used for generating training targets
- $\boldsymbol{\theta}_{n+1}$: parameters being optimized in the current iteration

This target model framework emerges naturally from the structure of parametric value iteration — an insight that provides theoretical grounding for modern deep reinforcement learning algorithms where we commonly hear about the importance of the "target network trick" .

Parametric successive approximation, known in reinforcement learning literature as Fitted Value Iteration, offers a flexible template for deriving new algorithms by varying the choice of function approximator. Various instantiations of this approach have emerged across different fields:

- Using polynomial basis functions with linear regression yields Kortum's method Kortum [1992], known to econometricians. In reinforcement learning terms, this corresponds to value iteration with projected Bellman equations Rust [1996].
- Employing extremely randomized trees (via `ExtraTreesRegressor`) leads to the tree-based fitted value iteration of Ernst et al. Ernst et al. [2005b].
- Neural network approximation (via `MLPRegressor`) gives rise to Neural Fitted Q-Iteration as developed by Riedmiller Riedmiller [2005a].

The \texttt{fit} function in our algorithm represents this supervised learning step and can be implemented using any standard regression tool that follows the scikit-learn interface. This flexibility in choice of function approximator allows practitioners to leverage the extensive ecosystem of modern machine learning tools while maintaining the core dynamic programming structure.

The structure of the above algorithm mirrors value iteration in its core idea of iteratively applying the Bellman operator. However, several key modifications distinguish this fitted variant:

First, rather than applying updates across the entire state space, we compute the operator only at selected base points B. The resulting values are then stored implicitly through the parameter vector $\boldsymbol{\theta}$ via the fitting step, rather than explicitly as in the tabular case.

The fitting procedure itself may introduce an "inner optimization loop." For instance, when using neural networks, this involves an iterative gradient descent procedure to optimize the parameters. This creates an interesting parallel with modified policy iteration: just as we might truncate policy evaluation steps there, we can consider variants where this inner loop runs for a fixed number of iterations rather than to convergence.

Finally, the termination criterion from standard value iteration may no longer hold. The classical criterion relied on the sup-norm contractivity property of the Bellman operator — a property that isn't generally preserved under function approximation. While certain function approximation schemes can maintain this sup-norm contraction property (as we'll see later), this is the exception rather than the rule.

Parametric Policy Iteration We can extend this idea of fitting partial operator updates to the policy iteration setting. Remember, policy iteration involves iterating in the space of policies rather than in the space of value functions. Given an initial guess on a deterministic decision rule d_0, we iteratively:

1. Compute the value function for the current policy (policy evaluation)
2. Derive a new improved policy (policy improvement)

When computationally feasible and under the model-based setting, we can solve the policy evaluation step directly as a linear system equation. Alternatively, we could carry out policy evaluation by applying successive approximation to the operator L_{d_n} until convergence, or as in modified policy iteration, for just a few steps.

To apply the idea of fitting partial updates, we start at the level of the policy evaluation operator L_{d_n}. For a given decision rule d_n, this operator in component form is:

$$(L_{d_n} v)(s) = r(s, d_n(s)) + \gamma \int v(s') p(ds'|s, d_n(s)) \tag{535}$$

For a set of base points $B = \{s_1, ..., s_M\}$, we form our dataset:

$$\mathcal{D}_n = \{(s_k, y_k) : s_k \in B\} \tag{536}$$

where:

$$y_k = r(s_k, d_n(s_k)) + \gamma \int v_n(s') p(ds'|s_k, d_n(s_k)) \tag{537}$$

This gives us a way to perform approximate policy evaluation through function fitting. However, we now face the question of how to perform policy improvement in this parametric setting. The solution comes from the fact that in the exact form of policy iteration, we don't need to improve the policy everywhere to guarantee progress. In fact, improving the policy at even a single state is sufficient for convergence.

This suggests a natural approach: rather than trying to approximate an improved policy over the entire state space, we can simply:

1. Compute improved actions at our base points:

$$d_{n+1}(s_k) = \arg\max_{a \in \mathcal{A}} \left\{ r(s_k, a) + \gamma \int v_n(s') p(ds'|s_k, a) \right\}, \quad \forall s_k \in B \tag{538}$$

   [resume]Let the function approximation of the value function implicitly generalize these improvements to other states during the next policy evaluation phase.

This leads to the following algorithm:

As opposed to exact policy iteration, the iterates of parametric policy iteration need not converge monotonically to the optimal value function. Intuitively, this is because we use function approximation to generalize from base points to the entire state space which can lead to Value estimates improving at base points but degrading at other states or can cause interference between updates at different states due to the shared parametric representation

Q-Factor Representation As we discussed above, Monte Carlo integration is the method of choice when it comes to approximating the effect of the Bellman operator. This is due to both its computational advantages in higher dimensions and its compatibility with the model-free assumption. However, there is an additional important detail that we have neglected to properly cover: extracting actions from values in a model-free fashion. While we can obtain a value function using the Monte Carlo approach described above, we still face the challenge of extracting an optimal policy from this value function.

More precisely, recall that an optimal decision rule takes the form:

$$d(s) = \arg\max_{a \in \mathcal{A}} \left\{ r(s, a) + \gamma \int v(s') p(ds'|s, a) \right\} \tag{539}$$

Therefore, even given an optimal value function v, deriving an optimal policy would still require Monte Carlo integration every time we query the decision rule/policy at a state.

An important idea in dynamic programming is that rather than approximating a state-value function, we can instead approximate a state-action value function. These two functions are related: the value function is the expectation of the Q-function (called Q-factors by some authors in the operations research literature) over the conditional distribution of actions given the current state:

$$v(s) = E[q(s, a)|s] \tag{540}$$

If q^* is an optimal state-action value function, then $v^*(s) = \max_a q^*(s, a)$. Just as we had a Bellman operator for value functions, we can also define an optimality operator for Q-functions. In component form:

$$(\mathrm{L}q)(s,a) = r(s,a) + \gamma \int p(ds'|s,a) \max_{a' \in \mathcal{A}(s')} q(s',a') \tag{541}$$

Furthermore, this operator for Q-functions is also a contraction in the sup-norm and therefore has a unique fixed point q^*.

The advantage of iterating over Q-functions rather than value functions is that we can immediately extract optimal actions without having to represent the reward function or transition dynamics directly, nor perform numerical integration. Indeed, an optimal decision rule at state s is obtained as:

$$d(s) = \arg\max_{a \in \mathcal{A}(s)} q(s,a) \tag{542}$$

With this insight, we can adapt our parametric value iteration algorithm to work with Q-functions:

Initialization and Warmstarting Parametric dynamic programming involves solving a sequence of related optimization problems, one for each fitting procedure at each iteration. While we've presented these as independent fitting problems, in practice we can leverage the relationship between successive iterations through careful initialization. This "warmstarting" strategy can significantly impact both computational efficiency and solution quality.

The basic idea is simple: rather than starting each fitting procedure from scratch, we initialize the function approximator with parameters from the previous iteration. This can speed up convergence since successive Q-functions tend to be similar. However, recent work suggests that persistent warmstarting might sometimes be detrimental, potentially leading to a form of overfitting. Alternative "reset" strategies that occasionally reinitialize parameters have shown promise in mitigating this issue.

Here's how warmstarting can be incorporated into parametric Q-learning with one-step Monte Carlo integration:

The main addition here is the periodic reset of parameters (controlled by frequency k) which helps balance the benefits of warmstarting with the need to avoid potential overfitting. When $k = \infty$, we get traditional persistent warmstarting, while $k = 1$ corresponds to training from scratch each iteration.

Inner Loop Convergence Beyond the choice of initialization and whether to chain optimization problems through warmstarting, we can also control how we terminate the inner optimization procedure. In the templates presented above, we implicitly assumed that `fit` is run to convergence. However, this need not be the case, and different implementations handle this differently.

For example, scikit-learn's MLPRegressor terminates based on several criteria: when the improvement in loss falls below a tolerance (default `tol=1e-4`), when it reaches the maximum number of iterations (default `max_iter=200`), or when the loss fails to improve for `n_iter_no_change` consecutive epochs. In contrast, ExtraTreesRegressor builds trees deterministically to completion

based on its splitting criteria, with termination controlled by parameters like `min_samples_split` and `max_depth`.

The intuition for using early stopping in the inner optimization mirrors that of modified policy iteration in exact dynamic programming. Just as modified policy iteration truncates the Neumann series during policy evaluation rather than solving to convergence, we might only partially optimize our function approximator at each iteration. While this complicates the theoretical analysis, it often works well in practice and can be computationally more efficient.

This perspective helps us understand modern deep reinforcement learning algorithms. For instance, DQN can be viewed as an instance of fitted Q-iteration where the inner optimization is intentionally limited. Here's how we can formalize this approach:

This formulation makes explicit the two-level optimization structure and allows us to control the trade-off between inner loop optimization accuracy and overall computational efficiency. When $N_{inner} = 1$, we recover something closer to DQN's update rule, while larger values of N_{inner} bring us closer to the full fitted Q-iteration approach.

4.3.2 Example Methods

There are several moving parts we can swap in and out when working with parametric dynamic programming - from the function approximator we choose, to how we warm start things, to the specific methods we use for numerical integration and inner optimization. In this section, we'll look at some concrete examples and see how they fit into this general framework.

Kernel-Based Reinforcement Learning (2002) Ormoneit and Sen's Kernel-Based Reinforcement Learning (KBRL) Ormoneit and Sen [2002] helped establish the general paradigm of batch reinforcement learning later advocated by Ernst et al. [2005a]. KBRL is a purely offline method that first collects a fixed set of transitions and then uses kernel regression to solve the optimal control problem through value iteration on this dataset. While the dominant approaches at the time were online methods like temporal difference, KBRL showed that another path to developping reinforcement learning algorithm was possible: one that capable of leveraging advances in supervised learning to provide both theoretical and practical benefits.

As the name suggests, KBRL uses kernel based regression within the general framework of outlined above.

Step 3 is where KBRL uses kernel regression with a normalized weighting kernel:

$$k_b(x_t^l, x) = \frac{\phi(\|x_t^l - x\|/b)}{\sum_{l'} \phi(\|x_{t'}^l - x\|/b)} \tag{543}$$

where ϕ is a kernel function (often Gaussian) and b is the bandwidth parameter. Each iteration reuses the entire fixed dataset to re-estimate Q-values through this kernel regression.

An important theoretical contribution of KBRL is showing that this kernel-based approach ensures convergence of the Q-function sequence. The authors prove that, with appropriate choice of kernel bandwidth decreasing with sample size, the method is consistent - the estimated Q-function converges to the true Q-function as the number of samples grows.

The main practical limitation of KBRL is computational - being a batch method, it requires storing and using all transitions at each iteration, leading to quadratic complexity in the number of samples. The authors acknowledge this limitation for online settings, suggesting that modifications like discarding old samples or summarizing data clusters would be needed for online applications. Ernst's later work with tree-based methods would help address this limitation while maintaining many of the theoretical advantages of the batch approach.

Ernst's Fitted Q Iteration (2005) Ernst's Ernst et al. [2005a] specific instantiation of parametric q-value iteration uses extremely randomized trees, an extension to random forests proposed by Geurts et al. [2006]. This algorithm became particularly well-known, partly because it was one of the first to demonstrate the advantages of offline reinforcement learning in practice on several challenging benchmarks at the time.

Random Forests and Extra-Trees differ primarily in how they construct individual trees. Random Forests creates diversity in two ways: it resamples the training data (bootstrap) for each tree, and at each node it randomly selects a subset of features but then searches exhaustively for the best cut-point within each selected feature. In contrast, Extra-Trees uses the full training set for each tree and injects randomization differently: at each node, it randomly selects both features and cut-points without searching for the optimal one. It then picks the best among these completely random splits according to a variance reduction criterion. This double randomization - in both feature and cut-point selection - combined with using the full dataset makes Extra-Trees faster than Random Forests while maintaining similar predictive accuracy.

An important implementation detail concerns how tree structures can be reused across iterations of fitted Q iteration. With parametric methods like neural networks, warmstarting is straightforward - you simply initialize the weights with values from the previous iteration. For decision trees, the situation is more subtle because the model structure is determined by how splits are chosen at each node. When the number of candidate splits per node is $K = 1$ (totally randomized trees), the algorithm selects both the splitting variable and threshold purely at random, without looking at the target values (the Q-values we're trying to predict) to evaluate the quality of the split. This means the tree structure only depends on the input variables and random choices, not on what we're predicting. As a result, we can build the trees once in the first iteration and reuse their structure throughout all iterations, only updating the prediction values at the leaves.

Standard Extra-Trees ($K > 1$), however, uses target values to choose the best among K random splits by calculating which split best reduces the variance of

the predictions. Since these target values change in each iteration of fitted Q iteration (as our estimate of Q evolves), we must rebuild the trees completely. While this is computationally more expensive, it allows the trees to better adapt their structure to capture the evolving Q-function.

The complete algorithm can be formalized as follows:

Neural Fitted Q Iteration (2005) Riedmiller's Neural Fitted Q Iteration (NFQI) Riedmiller [2005b] is a natural instantiation of parametric Q-value iteration where:

1. The function approximator $q(s, a; \boldsymbol{\theta})$ is a multi-layer perceptron
2. The `fit` function uses Rprop optimization trained to convergence on each iteration's pattern set
3. The expected next-state values are estimated through Monte Carlo integration with $N = 1$, using the observed next states from transitions

Specifically, rather than using numerical quadrature which would require known transition probabilities, NFQ approximates the expected future value using observed transitions:

$$\int q_n(s', a')p(ds'|s, a) \approx q_n(s'_{observed}, a') \tag{544}$$

where $s'_{observed}$ is the actual next state that was observed after taking action a in state s. This is equivalent to Monte Carlo integration with a single sample, making the algorithm fully model-free.

The algorithm follows from the parametric Q-value iteration template:

While NFQI was originally introduced as an offline method with base points collected a priori, the authors also present a variant where base points are collected incrementally. In this online variant, new transitions are gathered using the current policy (greedy with respect to Q_k) and added to the experience set. This approach proves particularly useful when random exploration cannot efficiently collect representative experiences.

Deep Q Networks (2013) DQN Mnih et al. [2013] is a close relative of NFQI - in fact, Riedmiller, the author of NFQI, was also an author on the DQN paper. What at first glance might look like a different algorithm can actually be understood as a special case of parametric dynamic programming with practical adaptations. Let's build this connection step by step.

First, let's start with basic parametric Q-value iteration using a neural network:

Next, let's open up the `fit` procedure to show the inner optimization loop using gradient descent:

Warmstarting and Partial Fitting A natural modification is to initialize the inner optimization loop with the previous iteration's parameters - a strategy known as warmstarting - rather than starting from $\boldsymbol{\theta}_0$ each time. Additionally, similar to how modified policy iteration performs partial policy evaluation rather than solving to convergence, we can limit ourselves to a fixed number of optimization steps. These pragmatic changes, when combined, yield:

Flattening the Updates with Target Swapping Now rather than maintaining two sets of indices for the outer and inner levels, we could also "flatten" this algorithm under a single loop structure using modulo arithmetics. Here's how we could rewrite it:

The flattened version with target parameters achieves exactly the same effect as our previous nested-loop structure with warmstarting and K gradient steps. In the nested version, we would create a dataset using parameters $\boldsymbol{\theta}_n$, then perform K gradient steps to obtain $\boldsymbol{\theta}_{n+1}$. In our flattened version, we maintain a separate $\boldsymbol{\theta}_{target}$ that gets updated every K steps, ensuring that the dataset $\mathcal{D}_n$ is created using the same parameters for K consecutive iterations - just as it would be in the nested version. The only difference is that we've restructured the algorithm to avoid explicitly nesting the loops, making it more suitable for continuous online training which we are about to introduce. The periodic synchronization of $\boldsymbol{\theta}_{target}$ with the current parameters $\boldsymbol{\theta}_n$ effectively marks the boundary of what would have been the outer loop in our previous version.

Exponential Moving Average Targets An alternative to this periodic swap of parameters is to use an exponential moving average (EMA) of the parameters:

Note that the original DQN used the periodic swap of parameters rather than EMA targets. EMA targets (also called "Polyak averaging") started becoming popular in deep RL with DDPG Lillicrap et al. [2015] where they used a "soft" target update: $\boldsymbol{\theta}_{target} \leftarrow \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta}_{target}$ with a small τ (like 0.001). This has since become a common choice in many algorithms like TD3 Fujimoto et al. [2018] and SAC Haarnoja et al. [2018].

Online Data Collection and Experience Replay Rather than using offline data, we now consider a modification where we incrementally gather samples under our current policy. A common exploration strategy is ε-greedy: with probability ε we select a random action, and with probability $1-\varepsilon$ we select the greedy action $\arg\max_a q(s,a;\boldsymbol{\theta}_n)$. This ensures we maintain some exploration even as our Q-function estimates improve. Typically ε is annealed over time, starting with a high value (e.g., 1.0) to encourage early exploration and gradually decreasing to a small final value (e.g., 0.01) to maintain a minimal level of exploration while mostly exploiting our learned policy.

This version faces two practical challenges. First, the transition dataset $\mathcal{T}_n$ grows unbounded over time, creating memory issues. Second, computing gradients over the entire dataset becomes increasingly expensive. These are

common challenges in online learning settings, and the standard solutions from supervised learning apply here:

1. Use a fixed-size circular buffer (often called replay buffer, in reference to "experience replay" by Lin [1992]) to limit memory usage

2. Compute gradients on mini-batches rather than the full dataset

Here's how we can modify our algorithm to incorporate these ideas:

This formulation naturally leads to an important concept in deep reinforcement learning: the replay ratio (or data reuse ratio). In our algorithm, for each new transition we collect, we sample a mini-batch of size b from our replay buffer and perform one update. This means we're reusing past experiences at a ratio of b:1 - for every new piece of data, we're learning from b experiences. This ratio can be tuned as a hyperparameter. Higher ratios mean more computation per environment step but better data efficiency, as we're extracting more learning from each collected transition. This highlights one of the key benefits of experience replay: it allows us to decouple the rate of data collection from the rate of learning updates. Some modern algorithms like SAC or TD3 explicitly tune this ratio, sometimes using multiple gradient steps per environment step to achieve higher data efficiency.

I'll write a subsection that naturally follows from the previous material and introduces double Q-learning in the context of DQN.

Double-Q Network Variant As we saw earlier, the max operator in the target computation can lead to overestimation of Q-values. This happens because we use the same network to both select and evaluate actions in the target computation: $y_i \leftarrow r_i + \gamma \max_{a' \in A} q(s'_i, a'; \boldsymbol{\theta}_{target})$. The max operator means we're both choosing the action that looks best under our current estimates and then using that same set of estimates to evaluate how good that action is, potentially compounding any optimization bias.

Double DQN Van Hasselt et al. [2016] addresses this by using the current network parameters to select actions but the target network parameters to evaluate them. This leads to a simple modification of the DQN algorithm:

The main difference from the original DQN is in step 7, where we now separate action selection from action evaluation. Rather than directly taking the max over the target network's Q-values, we first select the action using our current network ($\boldsymbol{\theta}_n$) and then evaluate that specific action using the target network ($\boldsymbol{\theta}_{target}$). This simple change has been shown to lead to more stable learning and better final performance across a range of tasks.

Deep Q Networks with Resets (2022) In flattening neural fitted Q-iteration, our field had perhaps lost sight of an important structural element: the choice of inner-loop initializer inherent in the original FQI algorithm. The traditional structure explicitly separated outer iterations (computing targets) from inner optimization (fitting to those targets), with each inner optimization starting fresh from parameters $\boldsymbol{\theta}_0$.

The flattened version with persistent warmstarting seemed like a natural optimization - why throw away learned parameters? However, recent work D'Oro et al. [2023] has shown that persistent warmstarting can actually be detrimental to learning. Neural networks tend to lose their ability to learn and generalize over the course of training, suggesting that occasionally starting fresh from $\boldsymbol{\theta}_0$ might be beneficial. Here's how this looks algorithmically in the context of DQN:

This algorithm change allows us to push the limits of our update ratio - the number of gradient steps we perform per environment interaction. Without resets, increasing this ratio leads to diminishing returns as the network's ability to learn degrades. However, by periodically resetting the parameters while maintaining our dataset of transitions, we can perform many more updates per interaction, effectively making our algorithm more "offline" and thus more sample efficient.

The hard reset strategy, while effective, might be too aggressive in some settings as it completely discards learned parameters. An alternative approach is to use a softer form of reset, adapting the "Shrink and Perturb" technique originally introduced by Ash and Adams [2020] in the context of continual learning. In their work, they found that neural networks that had been trained on one task could better adapt to new tasks if their parameters were partially reset - interpolated with a fresh initialization - rather than either kept intact or completely reset.

We can adapt this idea to our setting. Instead of completely resetting to $\boldsymbol{\theta}_0$, we can perform a soft reset by interpolating between our current parameters and a fresh random initialization:

The interpolation coefficient β controls how much of the learned parameters we retain, with $\beta = 0$ recovering the hard reset case and $\beta = 1$ corresponding to no reset at all. This provides a more flexible approach to restoring learning capability while potentially preserving useful features that have been learned. Like hard resets, this softer variant still enables high update ratios by preventing the degradation of learning capability, but does so in a more gradual way.

4.3.3 Does Parametric Dynamic Programming Converge?

So far we have avoided the discussion of convergence and focused on intuitive algorithm development, showing how we can extend successive approximation by computing only a few operator evaluations which then get generalized over the entire domain at each step of the value iteration procedure. Now we turn our attention to understanding the conditions under which this general idea can be shown to converge.

A crucial question to ask is whether our algorithm maintains the contraction property that made value iteration so appealing in the first place - the property that allowed us to show convergence to a unique fixed point. We must be careful here because the contraction mapping theorem is specific to a given norm. In the case of value iteration, we showed the Bellman optimality operator is a contraction in the sup-norm, which aligns naturally with how we compare

policies based on their value functions.

The situation becomes more complicated with fitted methods because we are not dealing with just a single operator. At each iteration, we perform exact, unbiased pointwise evaluations of the Bellman operator, but instead of obtaining the next function exactly, we get the closest representable one under our chosen function approximation scheme. Gordon Gordon [1995] showed that the fitting step can be conceptualized as an additional operator that gets applied on top of the exact Bellman operator to produce the next function parameters. This leads to viewing fitted value methods - which for simplicity we describe only for the value case, though the Q-value setting follows similarly - as the composition of two operators:

$$v_{n+1} = \Gamma(\mathrm{L}(v_n)) \tag{545}$$

where L is the Bellman operator and Γ represents the function approximation mapping.

Now we arrive at the central question: if L was a sup-norm contraction, is Γ composed with L still a sup-norm contraction? What conditions must hold for this to be true? This question is fundamental because if we can establish that the composition of these two operators maintains the contraction property in the sup-norm, we get directly that our resulting successive approximation method will converge.

The Search for Nonexpansive Operators Consider what happens in the fitting step: we have two value functions v and w, and after applying the Bellman operator L to each, we get new target values that differ by at most γ times their original difference in sup-norm (due to L being a γ-contraction in the sup norm). But what happens when we fit to these target values? If the function approximator can exaggerate differences between its target values, even a small difference in the targets could lead to a larger difference in the fitted functions. This would be disastrous - even though the Bellman operator shrinks differences between value functions by a factor of γ, the fitting step could amplify them back up, potentially breaking the contraction property of the composite operator.

In order to ensure that the composite operator is contractive, we need conditions on Γ such that if L is a sup-norm contraction then the composition also is. A natural property to consider is when Γ is a non-expansion. By definition, this means that for any functions v and w:

$$\|\Gamma(v) - \Gamma(w)\|_\infty \leq \|v - w\|_\infty \tag{546}$$

This turns out to be exactly what we need, since if Γ is a non-expansion, then for any functions v and w:

$$\|\Gamma(\mathrm{L}(v)) - \Gamma(\mathrm{L}(w))\|_\infty \leq \|L(v) - L(w)\|_\infty \leq \gamma\|v - w\|_\infty \tag{547}$$

The first inequality uses the non-expansion property of Γ, while the second uses the fact that L is a γ-contraction. Together they show that the composite operator $\Gamma \circ L$ remains a γ-contraction.

Gordon's Averagers But which function approximators satisfy this non-expansion property? Gordon shows that "averagers", approximators that compute their outputs as weighted averages of their training values, are always non-expansions in sup-norm. This includes many common approximation schemes like k-nearest neighbors, linear interpolation, and kernel smoothing with normalized weights. The intuition is that if you're taking weighted averages with weights that sum to one, you can never extrapolate beyond the range of your training values. These methods "interpolate". This theoretical framework explains why simple interpolation methods like k-nearest neighbors have proven very stable in practice, while more complex approximators can fail catastrophically. To guarantee convergence, we should either use averagers directly or modify other approximators to ensure they never extrapolate beyond their training targets.

More precisely, a function approximator Γ is an averager if for any state s and any target function v, the fitted value can be written as:

$$\Gamma(v)(s) = \sum_{i=1}^{n} w_i(s) v(s_i) \tag{548}$$

where the weights $w_i(s)$ satisfy:

1. $w_i(s) \geq 0$ for all i and s
2. $\sum_{i=1}^{n} w_i(s) = 1$ for all s
3. The weights $w_i(s)$ depend only on s and the training points $\{s_i\}$, not on the values $v(s_i)$

Let $m = \min_i v(s_i)$ and $M = \max_i v(s_i)$. Then:

$$m = m \sum_i w_i(s) \leq \sum_i w_i(s) v(s_i) \leq M \sum_i w_i(s) = M \tag{549}$$

So $\Gamma(v)(s) \in [m, M]$ for all s, meaning the fitted function cannot take values outside the range of its training values. This property is what makes averagers "interpolate" rather than "extrapolate" and is directly related to why they preserve the contraction property when composed with the Bellman operator. To see why averagers are non-expansions, consider two functions v and w. At any state s:

$$
\begin{aligned}
|\Gamma(v)(s) - \Gamma(w)(s)| &= \left| \sum_{i=1}^{n} w_i(s) v(s_i) - \sum_{i=1}^{n} w_i(s) w(s_i) \right| \\
&= \left| \sum_{i=1}^{n} w_i(s) (v(s_i) - w(s_i)) \right| \\
&\leq \sum_{i=1}^{n} w_i(s) |v(s_i) - w(s_i)| \\
&\leq \|v - w\|_\infty \sum_{i=1}^{n} w_i(s) \\
&= \|v - w\|_\infty
\end{aligned}
$$

Since this holds for all s, we have $\|\Gamma(v) - \Gamma(w)\|_\infty \leq \|v - w\|_\infty$, proving that Γ is a non-expansion.

Which Function Approximators Interpolate vs Extrapolate?

K-nearest neighbors (KNN) Let's look at specific examples, starting with k-nearest neighbors. For any state s, let $s_{(1)}, ..., s_{(k)}$ denote the k nearest training points to s. Then:

$$
\Gamma(v)(s) = \frac{1}{k} \sum_{i=1}^{k} v(s_{(i)}) \tag{550}
$$

This is an averager with weights $w_i(s) = \frac{1}{k}$ for the k nearest neighbors and 0 for all other points.

For kernel smoothing with a kernel function K, the fitted value is:

$$
\Gamma(v)(s) = \frac{\sum_{i=1}^{n} K(s - s_i) v(s_i)}{\sum_{i=1}^{n} K(s - s_i)} \tag{551}
$$

The denominator normalizes the weights to sum to 1, making this an averager with weights $w_i(s) = \frac{K(s - s_i)}{\sum_{j=1}^{n} K(s - s_j)}$.

Linear Regression In contrast, methods like linear regression and neural networks can and often do extrapolate beyond their training targets. More precisely, given a dataset of state-value pairs $\{(s_i, v(s_i))\}_{i=1}^{n}$, these methods fit parameters to minimize some error criterion, and the resulting function $\Gamma(v)(s)$ may take values outside the interval $[\min_i v(s_i), \max_i v(s_i)]$ even when evaluated at a new state s. For instance, linear regression finds parameters by minimizing squared error:

$$
\min_{\theta} \sum_{i=1}^{n} (v(s_i) - \theta^T \phi(s_i))^2 \tag{552}
$$

The resulting fitted function is:

$$\Gamma(v)(s) = \phi(s)^T(\Phi^T\Phi)^{-1}\Phi^T v \tag{553}$$

where Φ is the feature matrix with rows $\phi(s_i)^T$. This cannot be written as a weighted average with weights independent of v. Indeed, we can construct examples where the fitted value at a point lies outside the range of training values. For example, consider two sets of target values defined on just three points $s_1 = 0$, $s_2 = 1$, and $s_3 = 2$:

$$v = \begin{bmatrix}0\\0\\1\end{bmatrix}, \quad w = \begin{bmatrix}0\\1\\1\end{bmatrix} \tag{554}$$

Using a single feature $\phi(s) = s$, our feature matrix is:

$$\Phi = \begin{bmatrix}0\\1\\2\end{bmatrix} \tag{555}$$

For function v, the fitted parameters are:

$$\theta_v = (\Phi^T\Phi)^{-1}\Phi^T v = \frac{1}{14}(2) \tag{556}$$

And for function w:

$$\theta_w = (\Phi^T\Phi)^{-1}\Phi^T w = \frac{1}{14}(8) \tag{557}$$

Now if we evaluate these fitted functions at $s = 3$ (outside our training points):

$$\Gamma(v)(3) = 3\theta_v = \frac{6}{14} \approx 0.43 \tag{558}$$

$$\Gamma(w)(3) = 3\theta_w = \frac{24}{14} \approx 1.71 \tag{559}$$

Therefore:

$$|\Gamma(v)(3) - \Gamma(w)(3)| = \frac{18}{14} > 1 = \|v - w\|_\infty \tag{560}$$

Spline Interpolation Linear interpolation between points – the technique used earlier in this chapter – is an averager since for any point s between knots s_i and s_{i+1}:

$$\Gamma(v)(s) = \left(\frac{s_{i+1} - s}{s_{i+1} - s_i}\right) v(s_i) + \left(\frac{s - s_i}{s_{i+1} - s_i}\right) v(s_{i+1}) \tag{561}$$

The weights sum to 1 and are non-negative. However, cubic splines, despite their smoothness advantages, can violate the non-expansion property. To see this, consider fitting a natural cubic spline to three points:

$$s_1 = 0,\ s_2 = 1,\ s_3 = 2 \tag{562}$$

with two different sets of values:

$$v = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad w = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{563}$$

The natural cubic spline for v will overshoot at $s \approx 0.5$ and undershoot at $s \approx 1.5$ due to its attempt to minimize curvature, giving values outside the range $[0, 1]$. Meanwhile, w fits a flat line at 0. Therefore:

$$\|v - w\|_\infty = 1 \tag{564}$$

but

$$\|\Gamma(v) - \Gamma(w)\|_\infty > 1 \tag{565}$$

This illustrates a general principle: methods that try to create smooth functions by minimizing some global criterion (like curvature in splines) often sacrifice the non-expansion property to achieve their smoothness goals.

4.4 Policy Parametrization Methods

In the previous chapter, we explored various approaches to approximate dynamic programming, focusing on ways to handle large state spaces through function approximation. However, these methods still face significant challenges when dealing with large or continuous action spaces. The need to maximize over actions during the Bellman operator evaluation becomes computationally prohibitive as the action space grows.

This chapter explores a natural evolution of these ideas: rather than exhaustively searching over actions, we can parameterize and directly optimize the policy itself. We begin by examining how fitted Q methods, while effective for handling large state spaces, still struggle with action space complexity.

4.4.1 Embedded Optimization

Recall that in fitted Q methods, the main idea is to compute the Bellman operator only at a subset of all states, relying on function approximation to generalize to the remaining states. At each step of the successive approximation loop, we build a dataset of input state-action pairs mapped to their corresponding optimality operator evaluations:

$$\mathcal{D}_n = \{((s,a), (Lq)(s,a;\boldsymbol{\theta}_n)) \mid (s,a) \in \mathcal{B}\} \tag{566}$$

This dataset is then fed to our function approximator (neural network, random forest, linear model) to obtain the next set of parameters:

$$\boldsymbol{\theta}_{n+1} \leftarrow \texttt{fit}(\mathcal{D}_n) \tag{567}$$

While this strategy allows us to handle very large or even infinite (continuous) state spaces, it still requires maximizing over actions ($\max_{a \in A}$) during the dataset creation when computing the operator L for each basepoint. This maximization becomes computationally expensive for large action spaces. A natural improvement is to add another level of optimization: for each sample added to our regression dataset, we can employ numerical optimization methods to find actions that maximize the Bellman operator for the given state.

The above pseudocode introduces a generic `maximize` routine which represents any numerical optimization method that searches for an action maximizing the given function. This approach is versatile and can be adapted to different types of action spaces. For continuous action spaces, we can employ standard nonlinear optimization methods like gradient descent or L-BFGS (e.g., using scipy.optimize.minimize). For large discrete action spaces, we can use integer programming solvers - linear integer programming if the Q-function approximator is linear in actions, or mixed-integer nonlinear programming (MINLP) solvers for nonlinear Q-functions. The choice of solver depends on the structure of our Q-function approximator and the constraints on our action space.

Amortized Optimization Approach This process is computationally intensive. A natural question is whether we can "amortize" some of this computation by replacing the explicit optimization for each sample with a direct mapping that gives us an approximate maximizer directly. For Q-functions, recall that the operator is given by:

$$(\mathrm{L}q)(s,a) = r(s,a) + \gamma \int p(ds'|s,a) \max_{a' \in \mathcal{A}(s')} q(s',a') \tag{568}$$

If q^* is the optimal state-action value function, then $v^*(s) = \max_a q^*(s,a)$, and we can derive the optimal policy directly by computing the decision rule:

$$\pi^\star(s) = \arg\max_{a \in \mathcal{A}(s)} q^\star(s,a) \tag{569}$$

Since q^* is a fixed point of L, we can write:

$$\begin{aligned} q^\star(s,a) &= (Lq^*)(s,a) \\ &= r(s,a) + \gamma \int p(ds'|s,a) \max_{a' \in \mathcal{A}(s')} q^\star(s',a') \\ &= r(s,a) + \gamma \int p(ds'|s,a) q^\star(s', \pi^\star(s')) \end{aligned}$$

Note that $\pi^\star$ is implemented by our `maximize` numerical solver in the procedure above. A practical strategy would be to collect these maximizer values at each step and use them to train a function approximator that directly predicts these solutions. Due to computational constraints, we might want to compute these exact maximizer values only for a subset of states, based on some computational budget, and use the fitted decision rule to generalize to the remaining states. This leads to the following amortized version:

An important observation about this procedure is that the policy $d(s; \boldsymbol{w})$ is being trained on a dataset $\mathcal{D}_d$ containing optimal actions computed with respect to an evolving Q-function. Specifically, at iteration n, we collect pairs $(s', a^*_{s'})$ where $a^*_{s'} = \arg\max_a q(s', a; \boldsymbol{\theta}_n)$. However, after updating to $\boldsymbol{\theta}_{n+1}$, these actions may no longer be optimal with respect to the new Q-function.

A natural approach to handle this staleness would be to maintain only the most recent optimization data. We could modify our procedure to keep a sliding window of K iterations, where at iteration n, we only use data from iterations max(0, n-K) to n. This would be implemented by augmenting each entry in $\mathcal{D}_d$ with a timestamp:

$$\mathcal{D}^t_\pi = \{(s', a^*_{s'}, t) \mid t \in \{n-K, \ldots, n\}\} \tag{570}$$

where t indicates the iteration at which the optimal action was computed. When fitting the policy network, we would then only use data points that are at most K iterations old:

$$\boldsymbol{w}_{n+1} \leftarrow \texttt{fit}(\{(s', a^*_{s'}) \mid (s', a^*_{s'}, t) \in \mathcal{D}^t_\pi, n - K \leq t \leq n\}) \tag{571}$$

This introduces a trade-off between using more data (larger K) versus using more recent, accurate data (smaller K). The choice of K would depend on how quickly the Q-function evolves and the computational budget available for computing exact optimal actions.

Now the main issue with this approach, apart from the intrinsic out-of-distribution drift that we are trying to track, is that it requires "ground truth" - samples of optimal actions computed by the actual solver. This raises a natural question: how few samples do we actually need? Could we even envision eliminating the solver entirely? What seems impossible at first glance turns out to be achievable. The intuition is that as our policy improves at selecting actions, we can bootstrap from these increasingly better choices. As we continuously amortize these improving actions over time, it creates a virtuous cycle of self-improvement towards the optimal policy. But for this bootstrapping process to work, we need careful management. Move too quickly and the process may become unstable. Let's examine how this balance can be achieved.

4.4.2 Deterministic Parametrized Policies

In this section, we consider deterministic parametrized policies of the form $d(s; \boldsymbol{w})$ which directly output an action given a state. This approach differs from stochastic policies that output probability distributions over actions, making it particularly suitable for continuous control problems where the optimal policy is often deterministic. We'll see how fitted Q-value methods can be naturally extended to simultaneously learn both the Q-function and such a deterministic policy.

Neural Fitted Q-iteration for Continuous Actions (NFQCA) To develop this approach, let's first consider an idealized setting where we have access to $q^\star$, the optimal Q-function. Then we can state our goal as finding policy parameters $\boldsymbol{w}$ that maximize $q^\star$ with respect to the actions chosen by our policy across the state space:

$$\max_{\boldsymbol{w}} q^*(s, d(s; \boldsymbol{w})) \quad \text{for all } s \tag{572}$$

However, it's computationally infeasible to satisfy this condition for every possible state s, especially in large or continuous state spaces. To address this, we assume a distribution of states, denoted $\mu(s)$, and take the expectation, leading to the problem:

$$\max_{\boldsymbol{w}} E_{s\sim\mu(s)}[q^*(s, d(s; \boldsymbol{w}))] \tag{573}$$

However in practice, we do not have access to q^*. Instead, we need to approximate q^* with a Q-function $q(s, a; \boldsymbol{\theta})$, parameterized by $\boldsymbol{\theta}$, which we will

learn simultaneously with the policy function $d(s; \boldsymbol{w})$. Given a samples of initial states drawn from μ, we then maximize this objective via a Monte Carlo surrogate problem:

$$\max_{\boldsymbol{w}} E_{s\sim\mu(s)}[q(s, d(s; \boldsymbol{w}); \boldsymbol{\theta})] \approx \max_{\boldsymbol{w}} \frac{1}{|\mathcal{B}|} \sum_{s\in\mathcal{B}} q(s, d(s; \boldsymbol{w}); \boldsymbol{\theta}) \tag{574}$$

When using neural networks to parametrize q and d, we obtain the Neural Fitted Q-Iteration with Continuous Actions (NFQCA) algorithm proposed by Hafner and Riedmiller [2011].

In practice, both the `fit` and `minimize` operations above are implemented using gradient descent. For the Q-function, the `fit` operation minimizes the mean squared error between the network's predictions and the target values:

$$\texttt{fit}(\mathcal{D}_q) = \arg\min_{\boldsymbol{\theta}} \frac{1}{|\mathcal{D}_q|} \sum_{((s,a),y)\in\mathcal{D}_q} (q(s, a; \boldsymbol{\theta}) - y)^2 \tag{575}$$

For the policy update, the `minimize` operation uses gradient descent on the composition of the "critic" network q and the "actor" network d. This results in the following update rule:

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n + \alpha \nabla_{\boldsymbol{w}} \left(\frac{1}{|\mathcal{B}|} \sum_{(s,a,r,s')\in\mathcal{B}} q(s, d(s; \boldsymbol{w}); \boldsymbol{\theta}_{n+1}) \right) \tag{576}$$

where α is the learning rate. Both operations can be efficiently implemented using modern automatic differentiation libraries and stochastic gradient descent variants like Adam or RMSProp.

Deep Deterministic Policy Gradient (DDPG) Just as DQN adapted Neural Fitted Q-Iteration to the online setting, DDPG Lillicrap et al. [2015] extends NFQCA to learn from data collected online. Like NFQCA, DDPG simultaneously learns a Q-function and a deterministic policy that maximizes it, but differs in how it collects and processes data.

Instead of maintaining a fixed set of basepoints, DDPG uses a replay buffer that continuously stores new transitions as the agent interacts with the environment. Since the policy is deterministic, exploration becomes challenging. DDPG addresses this by adding noise to the policy's actions during data collection:

$$a = d(s; \boldsymbol{w}) + \mathcal{N} \tag{577}$$

where $\mathcal{N}$ represents exploration noise drawn from an Ornstein-Uhlenbeck (OU) process. The OU process is particularly well-suited for control tasks as it generates temporally correlated noise, leading to smoother exploration trajectories compared to independent random noise. It is defined by the stochastic differential equation:

$$d\mathcal{N}_t = \theta(\mu - \mathcal{N}_t)dt + \sigma dW_t \tag{578}$$

where μ is the long-term mean value (typically set to 0), θ determines how strongly the noise is pulled toward this mean, σ scales the random fluctuations, and dW_t is a Wiener process (continuous-time random walk). For implementation, we discretize this continuous-time process using the Euler-Maruyama method:

$$\mathcal{N}_{t+1} = \mathcal{N}_t + \theta(\mu - \mathcal{N}_t)\Delta t + \sigma\sqrt{\Delta t}\epsilon_t \tag{579}$$

where Δt is the time step and $\epsilon_t \sim \mathcal{N}(0, 1)$ is standard Gaussian noise. Think of this process like a spring mechanism: when the noise value $\mathcal{N}_t$ deviates from μ, the term $\theta(\mu - \mathcal{N}_t)\Delta t$ acts like a spring force, continuously pulling it back. Unlike a spring, however, this return to μ is not oscillatory - it's more like motion through a viscous fluid, where the force simply decreases as the noise gets closer to μ. The random term $\sigma\sqrt{\Delta t}\epsilon_t$ then adds perturbations to this smooth return trajectory. This creates noise that wanders away from μ (enabling exploration) but is always gently pulled back (preventing the actions from wandering too far), with θ controlling the strength of this pulling force.

The policy gradient update follows the same principle as NFQCA:

$$\nabla_{\boldsymbol{w}} E_{s\sim\mu(s)}[q(s, d(s; \boldsymbol{w}); \boldsymbol{\theta})] \tag{580}$$

We then embed this exploration mechanism into the data collection procedure and use the same flattened FQI structure that we adopted in DQN. Similar to DQN, flattening the outer-inner optimization structure leads to the need for target networks - both for the Q-function and the policy.

Twin Delayed Deep Deterministic Policy Gradient (TD3) While DDPG provided a foundation for continuous control with deep RL, it suffers from similar overestimation issues as DQN. TD3 Fujimoto et al. [2018] addresses these challenges through three key modifications: double Q-learning to reduce overestimation bias, delayed policy updates to reduce per-update error, and target policy smoothing to prevent exploitation of Q-function errors.

Similar to Double Q-learning, TD3 decouples selection from evaluation when forming the targets. However, instead of intertwining the two existing online and target networks, TD3 suggests learning two Q-functions simultaneously and uses their minimum when computing target values to help combat the overestimation bias further.

Furthermore, when computing target Q-values, TD3 adds small random noise to the target policy's actions and clips it to keep the perturbations bounded. This regularization technique essentially implements a form of "policy smoothing" that prevents the policy from exploiting areas where the Q-function may have erroneously high values:

```
$$\tilde{a} = d(s'; \boldsymbol{w}_{target}) + \text{clip}(\mathcal{N}(0, \sigma), -c, c)$$
```

While DDPG used the OU process which generates temporally correlated noise, TD3's authors found that simple uncorrelated Gaussian noise works just as well for exploration. It is also easier to implement and tune since you only need to set a single parameter ($\sigma_{explore}$) for exploration rather than the multiple parameters required by the OU process (θ, μ, σ).

Finally, TD3 updates the policy network (and target networks) less frequently than the Q-networks, typically once every d Q-function updates. This helps reduce the per-update error and gives the Q-functions time to become more accurate before they are used to update the policy.

4.4.3 Soft Actor-Critic

Adapting the intuition of NFQCA to the smooth Bellman optimality equations leads us to the soft actor-critic algorithm Haarnoja et al. [2018]. To understand this connection, let's first examine how the smooth Bellman equations emerge naturally from entropy regularization.

Consider the standard Bellman operator augmented with an entropy term. The smooth Bellman operator L_β takes the form:

$$(\mathrm{L}_\beta v)(s) = \max_{\pi \in \Pi^{MR}} \{E_{a\sim\pi}[r(s,a) + \gamma v(s')] + \beta \mathcal{H}(\pi)\} \tag{581}$$

where $\mathcal{H}(\pi) = -E_{a\sim\pi}[\log \pi(a|s)]$ represents the entropy of the policy. To find the solution to the optimization problem embedded in the operator L_β, we set the functional derivative of the objective with respect to the decision rule to zero:

$$\frac{\delta}{\delta \pi(a|s)} \left[\int_A \pi(a|s)(r(s,a) + \gamma v(s'))da - \beta \int_A \pi(a|s) \log \pi(a|s) da \right] = 0 \tag{582}$$

Enforcing that $\int_A \pi(a|s)da = 1$ leads to the following Lagrangian:

$$r(s,a) + \gamma v(s') - \beta(1 + \log \pi(a|s)) - \lambda(s) = 0 \tag{583}$$

Solving for π shows that the optimal policy is a Boltzmann distribution

$$\pi^*(a|s) = \frac{\exp(\frac{1}{\beta}(r(s,a) + \gamma E_{s'}[v(s')]))}{Z(s)} \tag{584}$$

When we substitute this optimal policy back into the entropy-regularized objective, we obtain:

$$\begin{aligned} v(s) &= E_{a\sim d^*}[r(s,a) + \gamma v(s')] + \beta \mathcal{H}(d^*) \\ &= \beta \log \int_A \exp(\frac{1}{\beta}(r(s,a) + \gamma E_{s'}[v(s')]))da \end{aligned}$$

As we saw at the beginning of this chapter, the smooth Bellman optimality operator for Q-factors is defined as:

$$(\mathrm{L}_\beta q)(s,a) = r(s,a) + \gamma E_{s'}\left[\beta \log \int_A \exp(\frac{1}{\beta} q(s',a'))da'\right] \tag{585}$$

This operator maintains the contraction property of its standard counterpart, guaranteeing a unique fixed point q^*. The optimal policy takes the form:

$$d^*(a|s) = \frac{\exp(\frac{1}{\beta} q^*(s,a))}{Z(s)} \tag{586}$$

where $Z(s) = \int_A \exp(\frac{1}{\beta} q^*(s,a))da$. The optimal value function can be recovered as:

$$v^*(s) = \beta \log \int_A \exp(\frac{1}{\beta} q^*(s,a))da \tag{587}$$

Fitted Q-Iteration for the Smooth Bellman Equations Following the principles of fitted value iteration, we can approximate approximate the effect of the smooth Bellman operator by computing it exactly at a number of basepoints and generalizing elsewhere using function approximation. Concretely, given a collection of states s_i and actions a_i, we would compute regression target values:

$$y_i = r(s_i,a_i) + \gamma E_{s'}\left[\beta \log \int_A \exp(\frac{1}{\beta} q_\theta(s',a'))da'\right] \tag{588}$$

and fit our Q-function approximator by minimizing:

$$\min_\theta \sum_i (q_\theta(s_i,a_i) - y_i)^2 \tag{589}$$

The expectation over next states can be handled through Monte Carlo estimation using samples from the environment: given a transition (s_i, a_i, s_i'), we can approximate:

$$E_{s'}\left[\beta \log \int_A \exp(\frac{1}{\beta} q_\theta(s',a'))da'\right] \approx \beta \log \int_A \exp(\frac{1}{\beta} q_\theta(s_i',a'))da' \tag{590}$$

However, we still face the challenge of computing the integral over actions. This motivates maintaining separate function approximators for both Q and V, using samples from the current policy to estimate the value function:

$$v_\psi(s) \approx E_{a\sim d(\cdot|s;\phi)}\left[q_\theta(s,a) - \beta \log d(a|s;\phi)\right] \tag{591}$$

By maintaining both approximators, we can estimate targets using sampled actions from our policy. Specifically, if we have a transition (s_i, a_i, s_i') and sample $a_i' \sim d(\cdot|s_i';\phi)$, our target becomes:

$$y_i = r(s_i,a_i) + \gamma\left(q_\theta(s_i',a_i') - \beta \log d(a_i'|s_i';\phi)\right) \tag{592}$$

This approach exists only due to the dual representation of the smooth Bellman equations as an entropy-regularized problem, which transforms the intractable log-sum-exp into a form we can estimate efficiently through sampling.

Approximating Boltzmann Policies by Gaussians The entropy-regularized objective and the smooth Bellman equation are mathematically equivalent. However, both formulations face a practical challenge: they require evaluating an intractable integral due to the Boltzmann distribution. Soft Actor-Critic (SAC) addresses this problem by approximating the optimal policy with a simpler, more tractable Gaussian distribution. Given the optimal soft policy:

$$d^*(a|s) = \frac{\exp(\frac{1}{\beta}q^*(s,a))}{Z(s)} \tag{593}$$

we seek to approximate it with a Gaussian policy:

$$d(a|s;\phi) = \mathcal{N}(\mu_\phi(s), \sigma_\phi(s)) \tag{594}$$

This approximation task naturally raises the question of how to measure the "closeness" between the target Boltzmann distribution and a candidate Gaussian approximation. Following common practice in deep learning, we employ the Kullback-Leibler (KL) divergence as our measure of distributional distance. To find the best approximation, we minimize the KL divergence between our policy and the optimal policy, using our current estimate q_θ of q^*:

$$\text{minimize}_\phi \, E_{s\sim\mu(s)}\left[D_{KL}\left(d(\cdot|s;\phi)\|\frac{\exp(\frac{1}{\beta}q_\theta(s,\cdot))}{Z(s)}\right)\right] \tag{595}$$

However, an important question remains: how can we solve this optimization problem when it involves the intractable partition function $Z(s)$? To see this, recall that for two distributions p and q, the KL divergence takes the form $D_{KL}(p\|q) = E_{x\sim p}[\log p(x) - \log q(x)]$. Let's denote the target Boltzmann distribution based on our current Q-estimate as:

$$d_\theta(a|s) = \frac{\exp(\frac{1}{\beta}q_\theta(s,a))}{Z_\theta(s)} \tag{596}$$

Then the KL minimization becomes:

$$\begin{aligned} D_{KL}(d(\cdot|s;\phi)\|d_\theta) &= E_{a\sim d(\cdot|s;\phi)}[\log d(a|s;\phi) - \log d_\theta(a|s)] \\ &= E_{a\sim d(\cdot|s;\phi)}\left[\log d(a|s;\phi) - \log\left(\frac{\exp(\frac{1}{\beta}q_\theta(s,a))}{Z_\theta(s)}\right)\right] \\ &= E_{a\sim d(\cdot|s;\phi)}\left[\log d(a|s;\phi) - \frac{1}{\beta}q_\theta(s,a) + \log Z_\theta(s)\right] \end{aligned}$$

Since $\log Z(s)$ is constant with respect to ϕ, minimizing this KL divergence is equivalent to:

$$\text{minimize}_\phi \, E_{s\sim\mu(s)} E_{a\sim d(\cdot|s;\phi)}[\log d(a|s;\phi) - \frac{1}{\beta} q_\theta(s,a)] \tag{597}$$

Reparameterizating the Objective One last challenge remains: ϕ appears in the distribution underlying the inner expectation, as well as in the integrand. This setting departs from standard empirical risk minimization (ERM) in supervised learning where the distribution of the data (e.g., cats and dogs in image classification) remains fixed regardless of model parameters. Here, however, the "data" - our sampled actions - depends directly on the parameters ϕ we're trying to optimize.

This dependence prevents us from simply using sample average estimators and differentiating through them, as we typically do in supervised learning. The challenge of correctly and efficiently estimating such derivatives has been extensively studied in the simulation literature under the umbrella of "derivative estimation." SAC adopts a particular solution known as the reparameterization trick in deep learning (or the IPA estimator in simulation literature). This approach transforms the problem by pushing ϕ inside the expectation through a change of variables.

To address this, we can express our Gaussian policy through a deterministic function f_ϕ that transforms noise samples to actions:

$$f_\phi(s,\epsilon) = \mu_\phi(s) + \sigma_\phi(s)\epsilon, \quad \epsilon \sim \mathcal{N}(0,1) \tag{598}$$

This transformation allows us to rewrite our objective using an expectation over the fixed noise distribution:

$$E_{s\sim\mu(s)} E_{\epsilon\sim\mathcal{N}(0,1)}[\log d(f_\phi(s,\epsilon)|s;\phi) - \frac{1}{\beta} q_\theta(s, f_\phi(s,\epsilon))]$$

Now ϕ appears only in the integrand through the function f_ϕ, not in the sampling distribution. The objective involves two terms. First, the log-probability of our Gaussian policy has a simple closed form:

$$\log d(f_\phi(s,\epsilon)|s;\phi) = -\frac{1}{2}\log(2\pi\sigma_\phi(s)^2) - \frac{(f_\phi(s,\epsilon) - \mu_\phi(s))^2}{2\sigma_\phi(s)^2} \tag{599}$$

Second, ϕ enters through the composition of $q^\star$ with f_ϕ: $q^\star(s, f_\phi(s,\epsilon))$. The chain rule for this composition would involve derivatives of both functions. While this might be problematic if the Q-factors were to come from outside of our control (ie. not in the computational graph), but since SAC learns it simultaneously with the policy, then we can simply compute all required derivatives through automatic differentiation.

This composition of policy and value functions - where f_ϕ enters as input to q_θ - directly parallels the structure we encountered in deterministic policy methods like NFQCA and DDPG. In those methods, we optimized:

$$\max_\phi E_{s\sim\mu(s)}[q_\theta(s, f_\phi(s))] \tag{600}$$

where $f_\phi(s)$ was a deterministic policy. SAC extends this idea to stochastic policies by having f_ϕ transform both state and noise:

$$\max_\phi E_{s\sim\mu(s)} E_{\epsilon\sim\mathcal{N}(0,1)}[q_\theta(s, f_\phi(s, \epsilon))] \tag{601}$$

Thus, rather than learning a single action for each state as in DDPG, we learn a function that transforms random noise into actions, explicitly parameterizing a distribution over actions while maintaining the same underlying principle of differentiating through composed policy and value functions.

4.4.4 Derivative Estimation for Stochastic Optimization

Consider optimizing an objective that involves an expectation:

$$J(\theta) = E_{x\sim p(x;\theta)}[f(x, \theta)] \tag{602}$$

For concreteness, let's examine a simple example where $x \sim \mathcal{N}(\theta, 1)$ and $f(x, \theta) = x^2\theta$. The derivative we seek is:

$$\frac{d}{d\theta}J(\theta) = \frac{d}{d\theta}\int x^2\theta p(x;\theta)dx \tag{603}$$

While we can compute this exactly for the Gaussian example, this is often impossible for more general problems. We might then be tempted to approximate our objective using samples:

$$J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i, \theta), \quad x_i \sim p(x;\theta) \tag{604}$$

Then differentiate this approximation:

$$\frac{d}{d\theta}J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial\theta} f(x_i, \theta) \tag{605}$$

However, this naive approach ignores that the samples themselves depend on θ. The correct derivative requires the product rule:

$$\frac{d}{d\theta}J(\theta) = \int \frac{\partial}{\partial\theta}[f(x, \theta)p(x;\theta)]dx = \int \left[\frac{\partial f}{\partial\theta}p(x;\theta) + f(x, \theta)\frac{\partial p(x;\theta)}{\partial\theta}\right] dx \tag{606}$$

The issue here is that while the first term could be numerically integrated using the Monte Carlo, the second one can't as it's not an expectation.

Would there be a way to transform our objective in such a way that the Monte Carlo estimator for the objective could be differentiated directly while ensuring that the resulting derivative is unbiased? We will see that there are two main solutions to that problem: by doing a change of measure, or a change of variables.

The Likelihood Ratio Method One solution comes from rewriting our objective using any distribution $q(x)$:

$$J(\theta) = \int f(x,\theta)\frac{p(x;\theta)}{q(x)}q(x)dx = E_{x\sim q(x)}\left[f(x,\theta)\frac{p(x;\theta)}{q(x)}\right] \tag{607}$$

Let's write this more functionally by defining:

$$J(\theta) = E_{x\sim q(x)}[h(x,\theta)], \;\; h(x,\theta) \equiv f(x,\theta)\frac{p(x;\theta)}{q(x)} \tag{608}$$

Now when we differentiate J, it's clear that we must take the partial derivative of h with respect to its second argument:

$$\frac{d}{d\theta}J(\theta) = E_{x\sim q(x)}\left[\frac{\partial h}{\partial \theta}(x,\theta)\right] = E_{x\sim q(x)}\left[f(x,\theta)\frac{\partial}{\partial\theta}\frac{p(x;\theta)}{q(x)} + \frac{p(x;\theta)}{q(x)}\frac{\partial f}{\partial\theta}(x,\theta)\right] \tag{609}$$

The so-called "score function" derivative estimator is obtained for the choice of $q(x) = p(x;\theta)$, where the ratio simplifies to 1 and its derivative becomes the score function:

$$\frac{d}{d\theta}J(\theta) = E_{x\sim p(x;\theta)}\left[f(x,\theta)\frac{\partial \log p(x,\theta)}{\partial\theta} + \frac{\partial f(x,\theta)}{\partial\theta}\right] \tag{610}$$

The Reparameterization Trick An alternative approach eliminates the θ-dependence in the sampling distribution by expressing x through a deterministic transformation of the noise:

$$x = g(\epsilon,\theta), \quad \epsilon \sim q(\epsilon) \tag{611}$$

Therefore if we want to sample from some target distribution $p(x;\theta)$, we can do so by first sampling from a simple base distribution $q(\epsilon)$ (like a standard normal) and then transforming those samples through a carefully chosen function g. If $g(\cdot,\theta)$ is invertible, the change of variables formula tells us how these distributions relate:

$$p(x;\theta) = q(g^{-1}(x,\theta))\left|\det\frac{\partial g^{-1}(x,\theta)}{\partial x}\right| = q(\epsilon)\left|\det\frac{\partial g(\epsilon,\theta)}{\partial\epsilon}\right|^{-1} \tag{612}$$

For example, if we want to sample from any multivariate Gaussian distributions with covariance matrix Σ and mean μ, it suffices to be able to sample from a standard normal noise and compute the linear transformation:

$$x = \mu + \Sigma^{1/2}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{613}$$

where $\Sigma^{1/2}$ is the matrix square root obtained via Cholesky decomposition. In the univariate case, this transformation is simply:

$$x = \mu + \sigma\epsilon, \quad \epsilon \sim \mathcal{N}(0, 1) \tag{614}$$

where $\sigma = \sqrt{\sigma^2}$ is the standard deviation (square root of the variance).

Common Examples of Reparameterization

The Truncated Normal Distribution When we need samples constrained to an interval $[a, b]$, we can use the truncated normal distribution. To sample from it, we transform uniform noise through the inverse cumulative distribution function (CDF) of the standard normal:

$$x = \Phi^{-1}(u\Phi(b) + (1 - u)\Phi(a)), \quad u \sim \text{Uniform}(0, 1) \tag{615}$$

Here:

- $\Phi(z) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right]$ is the CDF of the standard normal distribution
- Φ^{-1} is its inverse (the quantile function)
- $\text{erf}(z) = \frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$ is the error function

The resulting samples follow a normal distribution restricted to $[a, b]$, with the density properly normalized over this interval.

The Kumaraswamy Distribution When we need samples in the unit interval [0,1], a natural choice might be the Beta distribution. However, its inverse CDF doesn't have a closed form. Instead, we can use the Kumaraswamy distribution as a convenient approximation, which allows for a simple reparameterization:

$$x = (1 - (1 - u^{\alpha})^{1/\beta}), \quad u \sim \text{Uniform}(0, 1) \tag{616}$$

where:

- $\alpha, \beta > 0$ are shape parameters that control the distribution
- α determines the concentration around 0
- β determines the concentration around 1
- The distribution is similar to Beta(α,β) but with analytically tractable CDF and inverse CDF

The Kumaraswamy distribution has density:

$$f(x; \alpha, \beta) = \alpha\beta x^{\alpha-1}(1 - x^{\alpha})^{\beta-1}, \quad x \in [0, 1] \tag{617}$$

The Gumbel-Softmax Distribution When sampling from a categorical distribution with probabilities $\{\pi_i\}$, one approach uses Gumbel$(0, 1)$ noise combined with the argmax of log-perturbed probabilities:

$$\text{argmax}_i(\log \pi_i + g_i), \quad g_i \sim \text{Gumbel}(0, 1) \tag{618}$$

This approach, known in machine learning as the Gumbel-Max trick, relies on sampling Gumbel noise from uniform random variables through the transformation $g_i = -\log(-\log(u_i))$ where $u_i \sim \text{Uniform}(0, 1)$. To see why this gives us samples from the categorical distribution, consider the probability of selecting category i:

$$\begin{aligned} P(\text{argmax}_j(\log \pi_j + g_j) = i) &= P(\log \pi_i + g_i > \log \pi_j + g_j \text{ for all } j \neq i) \\ &= P(g_i - g_j > \log \pi_j - \log \pi_i \text{ for all } j \neq i) \end{aligned}$$

Since the difference of two Gumbel random variables follows a logistic distribution, $g_i - g_j \sim \text{Logistic}(0, 1)$, and these differences are independent for different j (due to the independence of the original Gumbel variables), we can write:

$$\begin{aligned} P(\text{argmax}_j(\log \pi_j + g_j) = i) &= \prod_{j \neq i} P(g_i - g_j > \log \pi_j - \log \pi_i) \\ &= \prod_{j \neq i} \frac{\pi_i}{\pi_i + \pi_j} = \pi_i \end{aligned}$$

The last equality requires some additional algebra to show, but follows from the fact that these probabilities must sum to 1 over all i.

While we have shown that the Gumbel-Max trick gives us exact samples from a categorical distribution, the argmax operation isn't differentiable. For stochastic optimization problems of the form:

$$E_{x \sim p(x;\theta)}[f(x)] = E_{\epsilon \sim \text{Gumbel}(0,1)}[f(g(\epsilon, \theta))] \tag{619}$$

we need g to be differentiable with respect to θ. This leads us to consider a continuous relaxation where we replace the hard argmax with a temperature-controlled softmax:

$$z_i = \frac{\exp((\log \pi_i + g_i)/\tau)}{\sum_j \exp((\log \pi_j + g_j)/\tau)} \tag{620}$$

As $\tau \to 0$, this approximation approaches the argmax:

$$\lim_{\tau \to 0} \frac{\exp(x_i/\tau)}{\sum_j \exp(x_j/\tau)} = \begin{cases} 1 & \text{if } x_i = \max_j x_j \\ 0 & \text{otherwise} \end{cases} \tag{621}$$

The resulting distribution over the probability simplex is called the Gumbel-Softmax (or Concrete) distribution. The temperature parameter τ controls the discreteness of our samples: smaller values give samples closer to one-hot vectors but with less stable gradients, while larger values give smoother gradients but more diffuse samples.

Numerical Analysis of Gradient Estimators Let us examine the behavior of our three gradient estimators for the stochastic optimization objective:

$$J(\theta) = E_{x\sim\mathcal{N}(\theta,1)}[x^2\theta] \tag{622}$$

To get an analytical expression for the derivative, first note that we can factor out θ to obtain $J(\theta) = \theta E[x^2]$ where $x \sim \mathcal{N}(\theta, 1)$. By definition of the variance, we know that $\text{Var}(x) = E[x^2] - (E[x])^2$, which we can rearrange to $E[x^2] = \text{Var}(x) + (E[x])^2$. Since $x \sim \mathcal{N}(\theta, 1)$, we have $\text{Var}(x) = 1$ and $E[x] = \theta$, therefore $E[x^2] = 1 + \theta^2$. This gives us:

$$J(\theta) = \theta(1 + \theta^2) \tag{623}$$

Now differentiating with respect to θ using the product rule yields:

$$\frac{d}{d\theta}J(\theta) = 1 + 3\theta^2 \tag{624}$$

For concreteness, we fix $\theta = 1.0$ and analyze samples drawn using Monte Carlo estimation with batch size 1000 and 1000 independent trials. Evaluating at $\theta = 1$ gives us $\frac{d}{d\theta}J(\theta)\big|_{\theta=1} = 1 + 3(1)^2 = 4$, which serves as our ground truth against which we compare our estimators:

1. First, we consider the naive estimator that incorrectly differentiates the Monte Carlo approximation:

$$\hat{g}_{\text{naive}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} x_i^2 \tag{625}$$

   For $x \sim \mathcal{N}(1, 1)$, we have $E[x^2] = \theta^2 + 1 = 2.0$ and $E[\hat{g}_{\text{naive}}] = 2.0$. We should therefore expect a bias of about -2 in our experiment.

2. Then we compute the score function estimator:

$$\hat{g}_{\text{SF}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[x_i^2\theta(x_i - \theta) + x_i^2\right] \tag{626}$$

   This estimator is unbiased with $E[\hat{g}_{\text{SF}}] = 4$

3. Finally, through the reparameterization $x = \theta + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$, we obtain:

$$\hat{g}_{\text{RT}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[2\theta(\theta + \epsilon_i) + (\theta + \epsilon_i)^2 \right] \tag{627}$$

This estimator is also unbiased with $E[\hat{g}_{\text{RT}}] = 4$.

The numerical experiments coroborate our theory. The naive estimator consistently underestimates the true gradient by 2.0, though it maintains a relatively small variance. This systematic bias would make it unsuitable for optimization despite its low variance. The score function estimator corrects this bias but introduces substantial variance. While unbiased, this estimator would require many samples to achieve reliable gradient estimates. Finally, the reparameterization trick achieves a much lower variance while remaining unbiased. While this experiment is for didactic purposes only, it reproduces what is commonly found in practice: that when applicable, the reparameterization estimator tends to perform better than the score function counterpart.

4.4.5 Score Function Gradient Estimation in Reinforcement Learning

Let $G(\tau) \equiv \sum_{t=0}^{T} r(s_t, a_t)$ be the sum of undiscounted rewards in a trajectory τ. The stochastic optimization problem we face is to maximize:

$$J(\boldsymbol{w}) = E_{\tau \sim p(\tau; \boldsymbol{w})}[G(\tau)] \tag{628}$$

where $\tau = (s_0, a_0, s_1, a_1, ...)$ is a trajectory and $G(\tau)$ is the total return. Applying the score function estimator, we get:

$$\begin{aligned}
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &= \nabla_{\boldsymbol{w}} E_{\tau}[G(\tau)] \\
&= E_{\tau}\left[G(\tau) \nabla_{\boldsymbol{w}} \log p(\tau; \boldsymbol{w}) \right] \\
&= E_{\tau}\left[G(\tau) \nabla_{\boldsymbol{w}} \sum_{t=0}^{T} \log d(a_t | s_t; \boldsymbol{w}) \right] \\
&= E_{\tau}\left[G(\tau) \sum_{t=0}^{T} \nabla_{\boldsymbol{w}} \log d(a_t | s_t; \boldsymbol{w}) \right]
\end{aligned}$$

We have eliminated the need to know the transition probabilities in this estimator since the probability of a trajectory factorizes as:

$$p(\tau; \boldsymbol{w}) = p(s_0) \prod_{t=0}^{T} d(a_t | s_t; \boldsymbol{w}) p(s_{t+1} | s_t, a_t) \tag{629}$$

Therefore, only the policy depends on $\boldsymbol{w}$. When taking the logarithm of this product, we get a sum where all the $\boldsymbol{w}$-independent terms vanish. The final estimator samples trajectories under the distribution $p(\tau; \boldsymbol{w})$ and computes:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx \frac{1}{N} \sum_{i=1}^{N} \left[G(\tau^{(i)}) \sum_{t=0}^{T} \nabla_{\boldsymbol{w}} \log d(a_t^{(i)} | s_t^{(i)}; \boldsymbol{w}) \right] \tag{630}$$

This is a direct application of the score function estimator. However, we rarely use this form in practice and instead make several improvements to further reduce the variance.

Leveraging Conditional Independence Given the Markov property of the MDP, rewards r_k for $k < t$ are conditionally independent of action a_t given the history $h_t = (s_0, a_0, ..., s_{t-1}, a_{t-1}, s_t)$. This allows us to only need to consider future rewards when computing policy gradients.

$$\begin{aligned}
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &= E_\tau \left[\sum_{t=0}^{T} \nabla_{\boldsymbol{w}} \log d(a_t | s_t; \boldsymbol{w}) \sum_{k=0}^{T} r_k \right] \\
&= E_\tau \left[\sum_{t=0}^{T} \nabla_{\boldsymbol{w}} \log d(a_t | s_t; \boldsymbol{w}) \left(\sum_{k=0}^{t-1} r_k + \sum_{k=t}^{T} r_k \right) \right] \\
&= E_\tau \left[\sum_{t=0}^{T} \nabla_{\boldsymbol{w}} \log d(a_t | s_t; \boldsymbol{w}) \sum_{k=t}^{T} r_k \right]
\end{aligned}$$

The condition independence assumption means that the term $E_\tau \left[\sum_{t=0}^{T} \nabla_{\boldsymbol{w}} \log d(a_t | s_t; \boldsymbol{w}) \sum_{k=0}^{t-1} r_k \right]$ vanishes. To see this, let's factor the trajectory distribution as:

$$p(\tau) = p(s_0, ..., s_t, a_0, ..., a_{t-1}) \cdot d(a_t | s_t; \boldsymbol{w}) \cdot p(s_{t+1}, ..., s_T, a_{t+1}, ..., a_T | s_t, a_t) \tag{631}$$

We can now re-write a single term of this summation as:

$$E_\tau \left[\nabla_{\boldsymbol{w}} \log d(a_t | s_t; \boldsymbol{w}) \sum_{k=0}^{t-1} r_k \right] = E_{s_{0:t}, a_{0:t-1}} \left[\sum_{k=0}^{t-1} r_k \cdot E_{a_t} \left[\nabla_{\boldsymbol{w}} \log d(a_t | s_t; \boldsymbol{w}) \right] \right] \tag{632}$$

The inner expectation is zero because

$$
\begin{aligned}
E_{a_t}\left[\nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})\right] &= \int \nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w}) d(a_t|s_t;\boldsymbol{w}) da_t \\
&= \int \frac{\nabla_{\boldsymbol{w}} d(a_t|s_t;\boldsymbol{w})}{d(a_t|s_t;\boldsymbol{w})} d(a_t|s_t;\boldsymbol{w}) da_t \\
&= \int \nabla_{\boldsymbol{w}} d(a_t|s_t;\boldsymbol{w}) da_t \\
&= \nabla_{\boldsymbol{w}} \int d(a_t|s_t;\boldsymbol{w}) da_t \\
&= \nabla_{\boldsymbol{w}} 1 = 0
\end{aligned}
$$

The Monte Carlo estimator becomes:

$$
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T} \nabla_{\boldsymbol{w}} \log d(a_t^{(i)}|s_t^{(i)};\boldsymbol{w}) \sum_{k=t}^{T} r_k^{(i)} \right] \tag{633}
$$

The benefit of this estimator compared to the naive one is that it generally has less variance. More formally, we can show that this estimator arises from the application of a variance reduction technique known as the Extended Conditional Monte Carlo Method.

Variance Reduction via Control Variates A control variate is a zero-mean random variable that we subtract from our estimator to reduce variance. Given an estimator Z and a control variate C with $E[C] = 0$, we can construct a new unbiased estimator:

$$
Z_{\text{cv}} = Z - \alpha C \tag{634}
$$

where α is a coefficient we can choose. The variance of this new estimator is:

$$
\text{Var}(Z_{\text{cv}}) = \text{Var}(Z) + \alpha^2 \text{Var}(C) - 2\alpha \text{Cov}(Z, C) \tag{635}
$$

The optimal α that minimizes this variance is:

$$
\alpha^* = \frac{\text{Cov}(Z, C)}{\text{Var}(C)} \tag{636}
$$

In the reinforcement learning setting, we usually choose $C_t = \nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})$ as our control variate at each timestep. For a given state s_t, our estimator at time t is:

$$
Z_t = \nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w}) \sum_{k=t}^{T} r_k \tag{637}
$$

Our control variate estimator becomes:

$$Z_{t,\text{cv}} = Z_t - \alpha_t^* C_t = \nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})(\sum_{k=t}^{T} r_k - \alpha_t^*) \tag{638}$$

Following the general theory, and using the fact that $E[C_t|s_t] = 0$ the optimal coefficient is:

$$\begin{aligned}
\alpha_t^* = \frac{\text{Cov}(Z_t, C_t|s_t)}{\text{Var}(C_t|s_t)} &= \frac{E[Z_t C_t^T|s_t] - E[Z_t|s_t]E[C_t^T|s_t]}{E[C_t C_t^T|s_t] - E[C_t|s_t]E[C_t^T|s_t]} \\
&= \frac{E[\nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w}) \nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})^T \sum_{k=t}^{T} r_k|s_t] - 0}{E[\nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w}) \nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})^T|s_t] - 0} \\
&= \frac{E[\|\nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})\|^2 \sum_{k=t}^{T} r_k|s_t]}{E[\|\nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})\|^2|s_t]} \\
&= \frac{E_{a_t|s_t}[\|\nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})\|^2] E[\sum_{k=t}^{T} r_k|s_t]}{E_{a_t|s_t}[\|\nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})\|^2]} \\
&= E[\sum_{k=t}^{T} r_k|s_t] = v^{d_{\boldsymbol{w}}}(s_t)
\end{aligned}$$

Therefore, our variance-reduced estimator becomes:

$$Z_{\text{cv},t} = \nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w}) \left(\sum_{k=t}^{T} r_k - v^{d_{\boldsymbol{w}}}(s_t) \right) \tag{639}$$

In practice when implementing this estimator, we won't have access to the true value function. So as we did earlier for NFQCA or SAC, we commonly learn that value function simultaneously with the policy. Do do so, we could either using a fitted value approach, or even more simply just regress from states to sum of rewards to learn what Williams 1992 called a "baseline":

When implementing this algorithm nowadays, we always using mini-batching to make full use of our GPUs. Therefore, a more representative variant for this algorithm would be:

Generalized Advantage Estimator Given our control variate estimator with baseline $v(s)$, we have:

$$\nabla_{\boldsymbol{w}} \log d(a_t|s_t;\boldsymbol{w})(G_t - v(s_t)) \tag{640}$$

where G_t is the return $\sum_{k=t}^{T} r_k$. We can improve this estimator by considering how it relates to the advantage function, defined as:

$$A(s_t, a_t) = q(s_t, a_t) - v(s_t) \tag{641}$$

where $q(s_t, a_t)$ is the action-value function. Due to the Bellman equation:

$$q(s_t, a_t) = E_{s_{t+1}, r_t}[r_t + \gamma v(s_{t+1}) | s_t, a_t] \tag{642}$$

This leads to the one-step TD error:

$$\delta_t = r_t + \gamma v(s_{t+1}) - v(s_t) \tag{643}$$

Now, let's decompose our original term:

$$\begin{aligned} G_t - v(s_t) &= r_t + \gamma G_{t+1} - v(s_t) \\ &= r_t + \gamma v(s_{t+1}) - v(s_t) + \gamma(G_{t+1} - v(s_{t+1})) \\ &= \delta_t + \gamma(G_{t+1} - v(s_{t+1})) \end{aligned}$$

Expanding recursively:

$$\begin{aligned} G_t - v(s_t) &= \delta_t + \gamma(G_{t+1} - v(s_{t+1})) \\ &= \delta_t + \gamma[\delta_{t+1} + \gamma(G_{t+2} - v(s_{t+2}))] \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + ... + \gamma^{T-t}\delta_T \end{aligned}$$

GAE generalizes this by introducing a weighted version with parameter λ:

$$A^{\text{GAE}(\gamma,\lambda)}(s_t, a_t) = (1-\lambda)\sum_{k=0}^{\infty}\lambda^k \sum_{l=0}^{k}\gamma^l \delta_{t+l} \tag{644}$$

Which simplifies to:

$$A^{\text{GAE}(\gamma,\lambda)}(s_t, a_t) = \sum_{l=0}^{\infty}(\gamma\lambda)^l \delta_{t+l} \tag{645}$$

This formulation allows us to trade off bias and variance through λ:

- $\lambda = 0$: one-step TD error (low variance, high bias)
- $\lambda = 1$: Monte Carlo estimate (high variance, low bias)

The general GAE algorithm with mini-batches is the following:

With $\lambda = 0$, the GAE advantage estimator becomes just the one-step TD error:

$$A^{\text{GAE}(\gamma,0)}(s_t, a_t) = \delta_t = r_t + \gamma v(s_{t+1}) - v(s_t) \tag{646}$$

The non-batched, purely online, GAE(0) algorithm then becomes:

This version was first derived by Richard Sutton in his 1984 PhD thesis. He called it the Adaptive Heuristic Actor-Critic algorithm. As far as I know, it was not derived using the score function method outlined here, but rather through intuitive reasoning (great intuition!).

The Policy Gradient Theorem Sutton 1999 provided an expression for the gradient of the infinite discounted return with respect to the parameters of a parameterized policy. Here is an alternative derivation by considering a bilevel optimization problem:

$$\max_{\mathbf{w}} \alpha^\top \mathbf{v}_\gamma^{d^\infty} \tag{647}$$

subject to:

$$(\mathbf{I} - \gamma \mathbf{P}_d)\mathbf{v}_\gamma^{d^\infty} = \mathbf{r}_d \tag{648}$$

The Implicit Function Theorem states that if there is a solution to the problem $F(\mathbf{v}, \mathbf{w}) = 0$, then we can "reparameterize" our problem as $F(\mathbf{v}(\mathbf{w}), \mathbf{w})$ where $\mathbf{v}(\mathbf{w})$ is an implicit function of $\mathbf{w}$. If the Jacobian $\frac{\partial F}{\partial \mathbf{v}}$ is invertible, then:

$$\frac{d\mathbf{v}(\mathbf{w})}{d\mathbf{w}} = -\left(\frac{\partial F(\mathbf{v}(\mathbf{w}), \mathbf{w})}{\partial \mathbf{x}}\right)^{-1} \frac{\partial F(\mathbf{v}(\mathbf{w}), \mathbf{w})}{\partial \mathbf{w}} \tag{649}$$

Here we made it clear in our notation that the derivative must be evaluated at root $(\mathbf{v}(\mathbf{w}), \mathbf{w})$ of F. For the remaining of this derivation, we will drop this dependence to make notation more compact.

Applying this to our case with $F(\mathbf{v}, \mathbf{w}) = (\mathbf{I} - \gamma \mathbf{P}_d)\mathbf{v} - \mathbf{r}_d$:

$$\frac{\partial \mathbf{v}_\gamma^{d^\infty}}{\partial \mathbf{w}} = (\mathbf{I} - \gamma \mathbf{P}_d)^{-1} \left(\frac{\partial \mathbf{r}_d}{\partial \mathbf{w}} + \gamma \frac{\partial \mathbf{P}_d}{\partial \mathbf{w}} \mathbf{v}_\gamma^{d^\infty}\right) \tag{650}$$

Then:

$$\begin{aligned} \nabla_{\mathbf{w}} J(\mathbf{w}) &= \alpha^\top \frac{\partial \mathbf{v}_\gamma^{d^\infty}}{\partial \mathbf{w}} \\ &= \mathbf{x}_\alpha^\top \left(\frac{\partial \mathbf{r}_d}{\partial \mathbf{w}} + \gamma \frac{\partial \mathbf{P}_d}{\partial \mathbf{w}} \mathbf{v}_\gamma^{d^\infty}\right) \end{aligned}$$

where we have defined the discounted state visitation distribution:

$$\mathbf{x}_\alpha^\top \equiv \alpha^\top (\mathbf{I} - \gamma \mathbf{P}_d)^{-1}. \tag{651}$$

Remember the vector notation for MDPs:

$$\begin{aligned} [\mathbf{r}_d]_s &= \sum_a d(a|s) r(s, a) \\ [\mathbf{P}_d]_{ss'} &= \sum_a d(a|s) P(s'|s, a) \end{aligned}$$

Then taking the derivatives gives us:

$$\left[\frac{\partial \mathbf{r}_d}{\partial \mathbf{w}}\right]_s = \sum_a \nabla_{\mathbf{w}} d(a|s) r(s,a)$$
$$\left[\frac{\partial \mathbf{P}_d}{\partial \mathbf{w}} \mathbf{v}_\gamma^{d^\infty}\right]_s = \sum_a \nabla_{\mathbf{w}} d(a|s) \sum_{s'} P(s'|s,a) v_\gamma^{d^\infty}(s')$$

Substituting back:

$$\begin{aligned}\nabla_{\mathbf{w}} J(\mathbf{w}) &= \sum_s x_\alpha(s) \left(\sum_a \nabla_{\mathbf{w}} d(a|s) r(s,a) + \gamma \sum_a \nabla_{\mathbf{w}} d(a|s) \sum_{s'} P(s'|s,a) v_\gamma^{d^\infty}(s') \right) \\ &= \sum_s x_\alpha(s) \sum_a \nabla_{\mathbf{w}} d(a|s) \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_\gamma^{d^\infty}(s') \right)\end{aligned}$$

This is the policy gradient theorem, where $x_\alpha(s)$ is the discounted state visitation distribution and the term in parentheses is the state-action value function $q(s,a)$.

Normalized Discounted State Visitation Distribution The discounted state visitation $x_\alpha(s)$ is not normalized. Therefore the expression we obtained above is not an expectation. However, we can tranform it into one by normalizing by $1-\gamma$. Note that for any initial distribution α:

$$\sum_s x_\alpha(s) = \alpha^\top (\mathbf{I} - \gamma \mathbf{P}_d)^{-1} \mathbf{1} = \frac{\alpha^\top \mathbf{1}}{1-\gamma} = \frac{1}{1-\gamma} \tag{652}$$

Therefore, defining the normalized state distribution $\mu_\alpha(s) = (1-\gamma) x_\alpha(s)$, we can write:

$$\begin{aligned}\nabla_{\mathbf{w}} J(\mathbf{w}) &= \frac{1}{1-\gamma} \sum_s \mu_\alpha(s) \sum_a \nabla_{\mathbf{w}} d(a|s) \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_\gamma^{d^\infty}(s') \right) \\ &= \frac{1}{1-\gamma} E_{s \sim \mu_\alpha} \left[\sum_a \nabla_{\mathbf{w}} d(a|s) Q(s,a) \right]\end{aligned}$$

Now we have expressed the policy gradient theorem in terms of expectations under the normalized discounted state visitation distribution. But what does sampling from μ_α means? Recall that $\mathbf{x}_\alpha^\top = \alpha^\top (\mathbf{I} - \gamma \mathbf{P}_d)^{-1}$. Using the Neumann series expansion (valid when $\|\gamma \mathbf{P}_d\| < 1$, which holds for $\gamma < 1$ since $\mathbf{P}_d$ is a stochastic matrix) we have:

$$\boldsymbol{\mu}_\alpha^\top = (1-\gamma) \alpha^\top \sum_{k=0}^{\infty} (\gamma \mathbf{P}_d)^k \tag{653}$$

We can then factor out the first term from this summation to obtain: s

$$\begin{aligned}\boldsymbol{\mu}_\alpha^\top &= (1-\gamma)\alpha^\top \sum_{k=0}^{\infty}(\gamma \mathbf{P}_d)^k \\ &= (1-\gamma)\alpha^\top + (1-\gamma)\alpha^\top \sum_{k=1}^{\infty}(\gamma \mathbf{P}_d)^k \\ &= (1-\gamma)\alpha^\top + (1-\gamma)\alpha^\top \gamma \mathbf{P}_d \sum_{k=0}^{\infty}(\gamma \mathbf{P}_d)^k \\ &= (1-\gamma)\alpha^\top + \gamma \boldsymbol{\mu}_\alpha^\top \mathbf{P}_d\end{aligned}$$

The balance equation :

$$\boldsymbol{\mu}_\alpha^\top = (1-\gamma)\alpha^\top + \gamma \boldsymbol{\mu}_\alpha^\top \mathbf{P}_d \tag{654}$$

shows that $\boldsymbol{\mu}_\alpha$ is a mixture distribution: with probability $1-\gamma$ you draw a state from the initial distribution α (reset), and with probability γ you follow the policy dynamics $\mathbf{P}_d$ from the current state (continue). This interpretation directly connects to the geometric process: at each step you either terminate and resample from α (with probability $1-\gamma$) or continue following the policy (with probability γ).

```
import numpy as np

def sample_from_discounted_visitation(
    alpha,
    policy,
    transition_model,
    gamma,
    n_samples=1000
):
    """Sample states from the discounted visitation distribution.

    Args:
        alpha: Initial state distribution (vector of probabilities)
        policy: Function (state -> action probabilities)
        transition_model: Function (state, action -> next state probabilities)
        gamma: Discount factor
        n_samples: Number of states to sample

    Returns:
        Array of sampled states
    """
    samples = []
    n_states = len(alpha)
```

```
    # Initialize state from alpha
    current_state = np.random.choice(n_states, p=alpha)

    for _ in range(n_samples):
        samples.append(current_state)

        # With probability (1 -gamma): reset
        if np.random.random() > gamma:
            current_state = np.random.choice(n_states, p=alpha)
        # With probability gamma: continue
        else:
            # Sample action from policy
            action_probs = policy(current_state)
            action = np.random.choice(len(action_probs), p=action_probs)

            # Sample next state from transition model
            next_state_probs = transition_model(current_state, action)
            current_state = np.random.choice(n_states, p=next_state_probs)

    return np.array(samples)

# Example usage for a simple 2 -state MDP
alpha = np.array([0.7, 0.3])  # Initial distribution
policy = lambda s: np.array([0.8, 0.2])  # Dummy policy
transition_model = lambda s, a: np.array([0.9, 0.1])  # Dummy transitions
gamma = 0.9

samples = sample_from_discounted_visitation(alpha, policy, transition_model, gamma)

# Check empirical distribution
print("Empirical state distribution:")
print(np.bincount(samples) / len(samples))
```

While the math shows that sampling from the discounted visitation distribution $\boldsymbol{\mu}_\alpha$ would give us unbiased policy gradient estimates, Thomas (2014) demonstrated that this implementation can be detrimental to performance in practice. The issue arises because terminating trajectories early (with probability $1 - \gamma$) reduces the effective amount of data we collect from each trajectory. This early termination weakens the learning signal, as many trajectories don't reach meaningful terminal states or rewards.

Therefore, in practice, we typically sample complete trajectories from the undiscounted process (i.e., run the policy until natural termination or a fixed horizon) while still using γ in the advantage estimation. This approach preserves the full learning signal from each trajectory and has been empirically shown to lead to better performance.

This is one of several cases in RL where the theoretically optimal procedure

differs from the best practical implementation.

4.4.6 Policy Optimization with a Model

In this section, we'll explore how incorporating a model of the dynamics can help us design better policy gradient estimators. Let's begin with a pure model-free approach that uses a critic to maximize a deterministic policy:

$$J(\boldsymbol{w}) = E_{s\sim\rho}[Q(s, d(s;\boldsymbol{w}))], \;\; \nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = E_{s\sim\rho}[\nabla_a Q(s,a)|_{a=d(s;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s;\boldsymbol{w})] \tag{655}$$

Using the recursive structure of the Bellman equations, we can unroll our objective one step ahead:

$$J(\boldsymbol{w}) = E_{s\sim\rho}[r(s, d(s;\boldsymbol{w})) + \gamma E_{s'\sim p(\cdot|s,d(s;\boldsymbol{w}))}[Q(s', d(s';\boldsymbol{w}))]], \tag{656}$$

To differentiate this objective, we need access to both a model of the dynamics and the reward function, as shown in the following expression:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = E_{s\sim\rho}[\nabla_a r(s,a)|_{a=d(s;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s;\boldsymbol{w}) + \gamma(E_{s'\sim p(\cdot|s,d(s;\boldsymbol{w}))}[\nabla_a Q(s',a)|_{a=d(s';\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s';\boldsymbol{w})] + \nabla_{\boldsymbol{w}} d(s;\boldsymbol{w}) \tag{657}$$

While $\boldsymbol{w}$ doesn't appear in the outer expectation over initial states, it affects the inner expectation over next states—a distributional dependency. As a result, the product rule of calculus yields two terms: the first being an expectation, and the second being problematic for sample average estimation. However, we have tools to address this challenge: we can either apply the reparameterization trick to the dynamics or use score function estimators.

For the reparameterization approach, assuming $s' = f(s, a, \xi; \boldsymbol{w})$ where ξ is the noise variable:

$$\begin{aligned} J^{\text{DPMB-R}}(\boldsymbol{w}) &= E_{s\sim\rho,\xi}[r(s, d(s;\boldsymbol{w})) + \gamma Q(f(s, d(s;\boldsymbol{w}), \xi), d(f(s, d(s;\boldsymbol{w}), \xi); \boldsymbol{w}))] \\ \nabla_{\boldsymbol{w}} J^{\text{DPMB-R}}(\boldsymbol{w}) &= E_{s\sim\rho,\xi}[\nabla_a r(s,a)|_{a=d(s;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s;\boldsymbol{w}) + \\ &\qquad \gamma(\nabla_a Q(s',a)|_{a=d(s';\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s';\boldsymbol{w}) + \nabla_{s'} Q(s', d(s';\boldsymbol{w})) \nabla_{\boldsymbol{w}} f(s, d(s;\boldsymbol{w}), \xi) \nabla_{\boldsymbol{w}} d(s;\boldsymbol{w}))] \end{aligned}$$

As for the score function approach:

$$\begin{aligned} \nabla_{\boldsymbol{w}} J^{\text{DPMB-SF}}(\boldsymbol{w}) = E_{s\sim\rho}[&\nabla_a r(s,a)|_{a=d(s;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s;\boldsymbol{w}) + \\ &\gamma E_{s'\sim p(\cdot|s,d(s;\boldsymbol{w}))}[\nabla_{\boldsymbol{w}} \log p(s'|s, d(s;\boldsymbol{w})) Q(s', d(s';\boldsymbol{w})) + \nabla_a Q(s',a)|_{a=d(s';\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s';\boldsymbol{w}) \end{aligned}$$

We've now developed a hybrid algorithm that combines model-based and model-free approaches while integrating derivative estimators for stochastic dynamics with a deterministic policy parameterization. While this hybrid estimator remains relatively unexplored in practice, it could prove valuable for systems with specific structural properties.

Consider a robotics scenario with hybrid continuous-discrete dynamics: a robotic arm operates in continuous space but interacts with discrete object states. While the arm's policy remains differentiable ($\nabla_{\boldsymbol{w}} d$), the object state transitions follow categorical distributions. In this case, reparameterization becomes impractical, but the score function approach is viable if we can compute $\nabla_{\boldsymbol{w}} \log p(s'|s, d(s; \boldsymbol{w}))$ from the known transition model. Similar structures arise in manufacturing processes, where machine actions might be continuous and differentiable, but material state transitions often follow discrete steps with known probabilities. Note that both approaches require knowledge of transition probabilities and won't work with pure black-box simulators or systems where we can only sample transitions without probability estimates.

Another dimension to explore in our algorithm design is the number of steps we wish to unroll our model. This allows us to better understand and control the bias-variance tradeoffs in our methods.

Backpropagation Policy Optimization The questions of derivative estimators only arise with stochastic dynamics. For systems with deterministic dynamics and a deterministic policy, the one-step gradient unroll simplifies to:

$$\begin{aligned}\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) =& E_{s\sim\rho}[\nabla_a r(s,a)|_{a=d(s;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s;\boldsymbol{w}) + \\ & \gamma(\nabla_a Q(s',a)|_{a=d(s';\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s';\boldsymbol{w}) + \nabla_{\boldsymbol{w}} d(s;\boldsymbol{w}) \nabla_a f(s,a)|_{a=d(s;\boldsymbol{w})} \nabla_{s'} Q(s', d(s';\boldsymbol{w})))]\end{aligned}$$

where $s' = f(s, d(s; \boldsymbol{w}))$ is the deterministic next state. Notice the resemblance between this expression and that obtained for $\nabla_{\boldsymbol{w}} J^{\text{DPMB -R}}$ above. They are essentially the same except that in the reparameterized case, the dynamics have made the dependence on the noise variable explicit and the outer expectation has been updated accordingly. This similarity arises because differentiation through reparameterized dynamics models is, in essence, backpropagation: it tracks the propagation of perturbations through a computation graph—which we refer to as a stochastic computation graph in this setting.

Still under this simplified setting with deterministic policies and dynamics, we can extend the expression for the gradient through n-steps of model unroll, leading to:

$$\begin{aligned}\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) =& E_{s\sim\rho}[\sum_{t=0}^{n-1} \gamma^t (\nabla_a r(s_t,a_t)|_{a_t=d(s_t;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s_t;\boldsymbol{w}) + \nabla_{s_t} r(s_t, d(s_t;\boldsymbol{w})) \nabla_{\boldsymbol{w}} s_t) + \\ & \gamma^n (\nabla_a Q(s_n,a)|_{a=d(s_n;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s_n;\boldsymbol{w}) + \nabla_{s_n} Q(s_n, d(s_n;\boldsymbol{w})) \nabla_{\boldsymbol{w}} s_n)]\end{aligned}$$

where $s_{t+1} = f(s_t, d(s_t; \boldsymbol{w}))$ for $t = 0, \ldots, n-1$, $s_0 = s$, and $\nabla_{\boldsymbol{w}} s_t$ follows the recursive relationship:

$$\nabla_{\boldsymbol{w}} s_{t+1} = \nabla_a f(s_t,a)|_{a=d(s_t;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s_t;\boldsymbol{w}) + \nabla_s f(s_t, d(s_t;\boldsymbol{w})) \nabla_{\boldsymbol{w}} s_t \tag{658}$$

with base case $\nabla_{\boldsymbol{w}} s_0 = 0$ since the initial state does not depend on the policy parameters.

At both ends of the spectrum, we have that for n=0, we fall back to the pure critic NFQCA-like approach, and for $n = \infty$, we don't bootstrap at all and are fully model-based without a critic. The pure model-based critic-free approach to optimization is what we may refer to as a backpropagation-based policy optimization (BPO).

But just as backpropagation through RNNs or very deep networks can be challenging due to exploding and vanishing gradients, "vanilla" Backpropagation Policy Optimization (BPO) without a critic can severely suffer from the curse of horizon. This is because it essentially implements single shooting optimization. Using a critic can be an effective remedy to this problem, allowing us to better control the bias-variance tradeoff while preserving gradient information that would be lost with a more drastic truncated backpropagation approach.

Stochastic Value Gradient (SVG) The stochastic value gradient framework of Heess (2015) applies the recipe for policy optimization with a model using reparameterized dynamics and action selection via randomized policies. In this setting, the stochastic policy model based reparameterized estimator over n steps is

$$J^{\text{SPMB-R-N}}(\boldsymbol{w}) = E_{s\sim\rho,\{\epsilon_i\}_{i=0}^{n-1},\{\xi_i\}_{i=0}^{n-1}}[\sum_{i=0}^{n-1} \gamma^i r(s_i, d(s_i, \epsilon_i; \boldsymbol{w})) + \gamma^n Q(s_n, d(s_n, \epsilon_n; \boldsymbol{w}))]$$

where $s_0 = s$ and for $i \geq 0$, $s_{i+1} = f(s_i, d(s_i, \epsilon_i; \boldsymbol{w}), \xi_i)$. The gradient becomes:

$$\begin{aligned}\nabla_{\boldsymbol{w}} J^{\text{SPMB-R-N}}(\boldsymbol{w}) = E_{s\sim\rho,\{\epsilon_i\}_{i=0}^{n-1},\{\xi_i\}_{i=0}^{n-1}}[\sum_{i=0}^{n-1} \gamma^i \left(\nabla_a r(s_i, a)|_{a=d(s_i,\epsilon_i;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s_i, \epsilon_i; \boldsymbol{w}) + \nabla_{s_i} r(s_i, d(s_i, \epsilon_i; \boldsymbol{w})) \nabla_{\boldsymbol{w}}\right. \\ \left. \gamma^n (\nabla_a Q(s_n, a)|_{a=d(s_n,\epsilon_n;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s_n, \epsilon_n; \boldsymbol{w}) + \nabla_{s_n} Q(s_n, d(s_n, \epsilon_n; \boldsymbol{w})) \nabla_{\boldsymbol{w}} s_n)\right]\end{aligned}$$

where $\nabla_{\boldsymbol{w}} s_0 = 0$ and for $i \geq 0$:

$$\nabla_{\boldsymbol{w}} s_{i+1} = \nabla_a f(s_i, a, \xi_i)|_{a=d(s_i,\epsilon_i;\boldsymbol{w})} \nabla_{\boldsymbol{w}} d(s_i, \epsilon_i; \boldsymbol{w}) + \nabla_{s_i} f(s_i, d(s_i, \epsilon_i; \boldsymbol{w}), \xi_i) \nabla_{\boldsymbol{w}} s_i \tag{659}$$

While we could implement this expression for the gradient ourselves, this approach is much easier, less error-prone, and most likely better optimized for performance when using automatic differentiation. Given a set of rollouts (for which we know the primitive noise variables), then we can compute the monte carlo objective:

$$\hat{J}^{\text{SPMB-R-N}}(\boldsymbol{w}) = \frac{1}{M}\sum_{m=1}^{M}[\sum_{i=0}^{n-1}\gamma^i r(s_i^m, d(s_i^m, \epsilon_i^m; \boldsymbol{w})) + \gamma^n Q(s_n^m, d(s_n^m, \epsilon_n^m; \boldsymbol{w}))] \tag{660}$$

where the states are generated recursively using the known noise variables: starting with initial state s_0^m, each subsequent state is computed as $s_{i+1}^m = f(s_i^m, d(s_i^m, \epsilon_i^m; \boldsymbol{w}), \xi_i^m)$. Thus, a trajectory is completely determined by just the sequence of noise variables:$(\epsilon_0^m, \xi_0^m, \epsilon_1^m, \xi_1^m, ..., \epsilon_{n-1}^m, \xi_{n-1}^m, \epsilon_n^m)$ where ϵ_i^m are the action noise variables and ξ_i^m are the dynamics noise variables.

The choice of unroll steps n gives us precise control over the balance between model-based and critic-based components in our gradient estimator. At one extreme, setting $n = 0$ yields a purely model-free algorithm known as SVG(0) (Heess et al., 2015), which relies entirely on the critic for value estimation:

$$\hat{J}^{\text{SPMB-R-0}}(\boldsymbol{w}) = \frac{1}{M}\sum_{m=1}^{M} Q(s_0^m, d(s_0^m, \epsilon_0^m; \boldsymbol{w})) \tag{661}$$

At the other extreme, as $n \to \infty$, we can drop the critic term (since $\gamma^n Q \to 0$) to obtain a purely model-based algorithm, SVG(∞):

$$\hat{J}^{\text{SPMB-R-}\infty}(\boldsymbol{w}) = \frac{1}{M}\sum_{m=1}^{M}\sum_{i=0}^{\infty}\gamma^i r(s_i^m, d(s_i^m, \epsilon_i^m; \boldsymbol{w})) \tag{662}$$

In the 2015 paper, the authors make a specific choice to reparameterize both the dynamics and action selections using normal distributions. For the policy, they use:

$$a_t = d(s_t; \boldsymbol{w}) + \sigma(s_t; \boldsymbol{w})\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, I) \tag{663}$$

where $d(s_t; \boldsymbol{w})$ predicts the mean action and $\sigma(s_t; \boldsymbol{w})$ predicts the standard deviation. For the dynamics:

$$s_{t+1} = \mu(s_t, a_t; \boldsymbol{\phi}) + \Sigma(s_t, a_t; \boldsymbol{\phi})\xi_t, \quad \xi_t \sim \mathcal{N}(0, I) \tag{664}$$

where $\mu(s_t, a_t; \boldsymbol{\phi})$ predicts the mean next state and $\Sigma(s_t, a_t; \boldsymbol{\phi})$ predicts the covariance matrix.

Under this reparameterization, the n-step surrogate loss becomes:

$$\hat{J}^{\text{SPMB-R-n}}(\boldsymbol{w}) = \frac{1}{M}\sum_{m=1}^{M}[\sum_{t=0}^{n-1}\gamma^t r(s_t^m(\boldsymbol{w}), d(s_t^m(\boldsymbol{w}); \boldsymbol{w}) + \sigma(s_t^m(\boldsymbol{w}); \boldsymbol{w})\epsilon_t^m) + \gamma^n Q(s_n^m(\boldsymbol{w}), d(s_n^m(\boldsymbol{w}); \boldsymbol{w}) + \sigma(s_n^m(\boldsymbol{w}); \boldsymbol{w})\epsilon_n^m)]$$

where:

$$s_{t+1}^m(\boldsymbol{w}) = \mu(s_t^m(\boldsymbol{w}), d(s_t^m(\boldsymbol{w}); \boldsymbol{w}) + \sigma(s_t^m(\boldsymbol{w}); \boldsymbol{w})\epsilon_t^m; \boldsymbol{\phi}) + \Sigma(s_t^m(\boldsymbol{w}), a_t^m(\boldsymbol{w}); \boldsymbol{\phi})\xi_t^m \tag{665}$$

Noise Inference in SVG The method we've presented so far assumes we have direct access to the noise variables ϵ and ξ used to generate trajectories. This works well in the on-policy setting where we generate our own data. However, in off-policy scenarios where we receive externally generated trajectories, we need to infer these noise variables—a process the authors call noise inference.

For the Gaussian case discussed above, this inference is straightforward. Given an observed scalar action a_t and the current policy parameters $\boldsymbol{w}$, we can recover the action noise ϵ_t through inverse reparameterization:

$$\epsilon_t = \frac{a_t - d(s_t; \boldsymbol{w})}{\sigma(s_t; \boldsymbol{w})} \tag{666}$$

Similarly, for the dynamics noise where states are typically vector-valued:

$$\xi_t = \Sigma(s_t, a_t; \boldsymbol{\phi})^{-1}(s_{t+1} - \mu(s_t, a_t; \boldsymbol{\phi})) \tag{667}$$

This simple inversion is possible because the Gaussian reparameterization is an affine transformation, which is invertible as long as $\sigma(s_t; \boldsymbol{w})$ is non-zero for scalar actions and $\Sigma(s_t, a_t; \boldsymbol{\phi})$ is full rank for vector-valued states. The same principle extends naturally to vector-valued actions, where σ would be replaced by a full covariance matrix.

More generally, this idea of invertible transformations can be extended far beyond simple Gaussian reparameterization. We could consider a sequence of invertible transformations:

$$z_0 \sim \mathcal{N}(0, I) \xrightarrow{f_1} z_1 \xrightarrow{f_2} z_2 \xrightarrow{f_3} \cdots \xrightarrow{f_K} z_K = a_t \tag{668}$$

where each f_k is an invertible neural network layer. The forward process can be written compactly as:

$$a_t = (f_K \circ f_{K-1} \circ \cdots \circ f_1)(z_0; \boldsymbol{w}) \tag{669}$$

This is the idea behind normalizing flows: a series of invertible transformations that can transform a simple base distribution (like a standard normal) into a complex target distribution while maintaining exact invertibility.

The noise inference in this case would involve applying the inverse transformations:

$$z_0 = (f_1^{-1} \circ \cdots \circ f_K^{-1})(a_t; \boldsymbol{w}) \tag{670}$$

This approach offers several advantages:

1. More expressive policies and dynamics models capable of capturing multimodal distributions
2. Exact likelihood computation through the change of variables formula (can be useful for computing the log prob terms in entropy regularization for example)
3. Precise noise inference through the guaranteed invertibility of the flow

As far as I know, this approach has not been explored in the literature.

DPG as a Special Case of SAC At first glance, SAC and DPG might appear to be fundamentally different approaches to policy optimization. SAC begins with the principle of entropy maximization and policy distribution matching through KL divergence minimization, while DPG directly optimizes a deterministic policy to maximize expected Q-values. However, we can show that DPG emerges as a special case of SAC as we take the temperature parameter to zero.

Proposition 4.2 (Convergence of SAC to DPG)**.** *Let $d(\cdot|s; \boldsymbol{w}_\alpha)$ be the optimal stochastic policy for SAC with temperature α, and $d(s; \boldsymbol{w}_{DPG})$ be the optimal deterministic policy gradient solution. Under appropriate assumptions, as $\alpha \to 0$:*

$$d(a|s; \boldsymbol{w}_\alpha) \to \delta(a - d(s; \boldsymbol{w}_{DPG})) \tag{671}$$

Assumptions:

1. *The stochastic policy class is Gaussian with learnable mean and standard deviation:*

$$d(a|s; \boldsymbol{w}) = \mathcal{N}(\mu_w(s), \sigma_w(s)^2) \tag{672}$$

2. *The SAC objective for policy improvement uses the soft Q-function:*

$$\boldsymbol{w}_\alpha^* = \arg\min_w E_{s\sim\rho}\left[D_{KL}\left(d(\cdot|s; \boldsymbol{w}) \| \frac{\exp(Q_{soft}(s, \cdot)/\alpha)}{\int \exp(Q_{soft}(s, b)/\alpha) db}\right)\right] \tag{673}$$

where Q_{soft} follows the soft Bellman equation:

$$Q_{soft}(s, a) = r(s, a) + \gamma E_{s'\sim P}\left[E_{a'\sim d(\cdot|s')}\left[Q_{soft}(s', a') - \alpha \log d(a'|s')\right]\right] \tag{674}$$

3. *The DPG objective with a deterministic policy uses the standard Q-function:*

$$\boldsymbol{w}_{DPG}^* = \arg\max_w E_{s\sim\rho}\left[Q(s, d(s; \boldsymbol{w}))\right] \tag{675}$$

where Q follows the standard Bellman equation:

$$Q(s, a) = r(s, a) + \gamma E_{s'\sim P}[Q(s', d(s'; \boldsymbol{w}))] \tag{676}$$

4. *$Q_{soft}(s, a)$ and $Q(s, a)$ are continuous and achieve their maxima for each state s.*

Proof. At the fixed point of the soft Bellman equation, as $\alpha \to 0$, the entropy term $-\alpha \log d(a|s)$ vanishes, and $Q_{soft} \to Q$. This implies that the SAC target distribution, which is proportional to $\exp(Q_{soft}(s,a)/\alpha)$, becomes:

$$\lim_{\alpha \to 0} \frac{\exp(Q(s,a)/\alpha)}{\int \exp(Q(s,b)/\alpha) db} = \delta(a - \arg\max_a Q(s,a)), \tag{677}$$

by Laplace's method. The target distribution thus collapses to a delta function centered at the deterministic optimal action $\arg\max_a Q(s,a)$.

The KL divergence term in the SAC objective measures the divergence between the stochastic policy $d(a|s;\boldsymbol{w})$ (Gaussian) and this target distribution. For a Gaussian $\mathcal{N}(\mu, \sigma^2)$ and a delta function $\delta(a - a^*)$, we derive:

$$D_{KL}(\mathcal{N}(\mu,\sigma^2) \| \delta(a-a^*)) = \lim_{\epsilon \to 0} D_{KL}(\mathcal{N}(\mu,\sigma^2) \| \mathcal{N}(a^*, \epsilon^2)), \tag{678}$$

where $\mathcal{N}(a^*, \epsilon^2)$ is a Gaussian approximation of the delta. Using the KL formula:

$$D_{KL} = \frac{1}{2}\left[\log\left(\frac{\epsilon^2}{\sigma^2}\right) + \frac{\sigma^2}{\epsilon^2} + \frac{(\mu - a^*)^2}{\epsilon^2} - 1\right]. \tag{679}$$

Taking the limit $\epsilon \to 0$, the divergence diverges unless $\mu = a^*$ and $\sigma = 0$, where it becomes zero. Thus, minimizing the SAC objective drives $\mu_w(s) \to \arg\max_a Q(s,a)$ and $\sigma_w(s) \to 0$.

Consequently, the stochastic policy converges to a delta function:

$$\lim_{\alpha \to 0} d(a|s;\boldsymbol{w}^*_\alpha) = \delta(a - \arg\max_a Q(s,a)) = \delta(a - d(s;\boldsymbol{w}^*_{DPG})). \tag{680}$$

□

4.4.7 Policy Optimization with a Trust Region

Trust region methods in optimization approximate the objective function with a simpler local model within a region where we "trust" this approximation to be good. This brings about the need to define what we mean by a local region, and therefore to pick a geometry which suits our problem.

In standard optimization in the Euclidean space on R^n, at each iteration k, we create a quadratic approximation around the current point x_k:

$$m_k(p) = f(x_k) + g_k^T p + \frac{1}{2} p^T B_k p \tag{681}$$

where $g_k = \nabla f(x_k)$ is the gradient and B_k approximates the Hessian. The trust region constrains updates using Euclidean distance:

$$\min_p m_k(p) \text{ subject to } \|p\| \leq \Delta_k \tag{682}$$

However, when optimizing over probability distributions $p(x;\theta)$, the Euclidean geometry becomes unnatural. Instead, the Kullback-Leibler divergence provides a more natural mean of measuring proximity:

$$D_{KL}(p(x;\theta)||p(x;\theta_k)) = \int p(x;\theta) \log\left(\frac{p(x;\theta)}{p(x;\theta_k)}\right) dx \tag{683}$$

This leads to the following trust region subproblem:

$$\min_{\theta} m_k(\theta) \text{ subject to } D_{KL}(p(x;\theta)||p(x;\theta_k)) \leq \Delta_k \tag{684}$$

For exponential families, the KL divergence locally reduces to a quadratic form involving the Fisher Information Matrix $I(\theta_k)$:

$$D_{KL}(p(x;\theta)||p(x;\theta_k)) \approx \frac{1}{2}(\theta - \theta_k)^T I(\theta_k)(\theta - \theta_k) \tag{685}$$

In both cases, after solving for the step, we evaluate the actual versus predicted reduction ratio:

$$\rho_k = \frac{f(x_k) - f(x_k + p)}{m_k(0) - m_k(p)} \tag{686}$$

This ratio determines both step acceptance and trust region adjustment:

$$\Delta_{k+1} = \begin{cases} \alpha_1 \Delta_k & \text{if } \rho_k < \eta_1 \text{ (poor prediction)} \\ \Delta_k & \text{if } \eta_1 \leq \rho_k < \eta_2 \text{ (acceptable)} \\ \alpha_2 \Delta_k & \text{if } \rho_k \geq \eta_2 \text{ (very good)} \end{cases} \tag{687}$$

The method accepts steps when the model prediction is sufficiently accurate:

$$x_{k+1} = \begin{cases} x_k + p & \text{if } \rho_k > \eta_1 \\ x_k & \text{otherwise} \end{cases} \tag{688}$$

5 Appendix

5.1 Example COCPs

5.1.1 Inverted Pendulum

The inverted pendulum is a classic problem in control theory and robotics that demonstrates the challenge of stabilizing a dynamic system that is inherently unstable. The objective is to keep a pendulum balanced in the upright position by applying a control force, typically at its base. This setup is analogous to balancing a broomstick on your finger: any deviation from the vertical position will cause the system to tip over unless you actively counteract it with appropriate control actions.

We typically assume that the pendulum is mounted on a cart or movable base, which can move horizontally. The system's state is then characterized by four variables:

1. **Cart position**: $x(t)$ — the horizontal position of the base.
2. **Cart velocity**: $\dot{x}(t)$ — the speed of the cart.
3. **Pendulum angle**: $\theta(t)$ — the angle between the pendulum and the vertical upright position.
4. **Angular velocity**: $\dot{\theta}(t)$ — the rate at which the pendulum's angle is changing.

This setup is more complex because the controller must deal with interactions between two different types of motion: linear (the cart) and rotational (the pendulum). This system is said to be "underactuated" because the number of control inputs (one) is less than the number of state variables (four). This makes the problem more challenging and interesting from a control perspective.

We can simplify the problem by assuming that the base of the pendulum is fixed. This is akin to having the bottom of the stick attached to a fixed pivot on a table. You can't move the base anymore; you can only apply small nudges at the pivot point to keep the stick balanced upright. In this case, you're only focusing on adjusting the stick's tilt without worrying about moving the base. This reduces the problem to stabilizing the pendulum's upright orientation using only the rotational dynamics. The system's state can now be described by just two variables:

1. **Pendulum angle**: $\theta(t)$ — the angle of the pendulum from the upright vertical position.
2. **Angular velocity**: $\dot{\theta}(t)$ — the rate at which the pendulum's angle is changing.

The evolution of these two varibles is governed by the following ordinary differential equation:

$$\begin{bmatrix}\dot{\theta}(t)\\ \ddot{\theta}(t)\end{bmatrix} = \begin{bmatrix}\dot{\theta}(t)\\ \frac{mgl}{J_t}\sin\theta(t) - \frac{\gamma}{J_t}\dot{\theta}(t) + \frac{l}{J_t}u(t)\cos\theta(t)\end{bmatrix}, \quad y(t) = \theta(t) \tag{689}$$

where:

- m is the mass of the pendulum
- g is the acceleration due to gravity
- l is the length of the pendulum
- γ is the coefficient of rotational friction
- $J_t = J + ml^2$ is the total moment of inertia, with J being the pendulum's moment of inertia about its center of mass
- $u(t)$ is the control force applied at the base
- $y(t) = \theta(t)$ is the measured output (the pendulum's angle)

We expect that when no control is applied to the system, the rod should be falling down when started from the upright position.

5.1.2 Pendulum in the Gym Environment

Let's examine the code and reverse-engineer the original continuous-time problem hidden behind the abstraction layer. Although the pendulum problem may have limited practical relevance as a real-world application, it serves as an excellent example for our analysis. In the current version of Pendulum, we find that the Gym implementation uses a simplified model. Like our implementation, it assumes a fixed base and doesn't model cart movement. The state is also represented by the pendulum angle and angular velocity. However, the equations of motion implemented in the Gym environment are different and correspond to the following ODE:

$$\begin{aligned}\dot{\theta} &= \theta_{dot}\\ \dot{\theta}_{dot} &= \frac{3g}{2l}\sin(\theta) + \frac{3}{ml^2}u\end{aligned}$$

Compared to our simplified model, the Gym implementation makes the following additional assumptions:

1. It omits the term $\frac{\gamma}{J_t}\dot{\theta}(t)$, which represents damping or air resistance. This means that it assumes an idealized pendulum that doesn't naturally slow down over time.

2. It uses ml^2 instead of $J_t = J + ml^2$, which assumes that all mass is concentrated at the pendulum's end (like a point mass on a massless rod), rather than accounting for mass distribution along the pendulum.

3. The control input u is applied directly, without a $\cos\theta(t)$ term, which means that the applied torque has the same effect regardless of the pendulum's position, rather than varying with angle. For example, imagine trying to push a door open. When the door is almost closed (pendulum near vertical), a small push perpendicular to the door (analogous to our control input) can easily start it moving. However, when the door is already wide open (pendulum horizontal), the same push has little effect on the door's angle. In a more detailed model, this would be captured by the $\cos\theta(t)$ term, which is maximum when the pendulum is vertical ($\cos 0 = 1$) and zero when horizontal ($\cos 90 = 0$).

The goal remains to stabilize the rod upright, but the way in which this encoded is through the following instantenous cost function:

$$\begin{aligned} c(\theta, \dot{\theta}, u) &= (\text{normalize}(\theta))^2 + 0.1\dot{\theta}^2 + 0.001u^2 \\ \text{normalize}(\theta) &= ((\theta + \pi) \bmod 2\pi) - \pi \end{aligned}$$

This cost function penalizes deviations from the upright position (first term), discouraging rapid motion (second term), and limiting control effort (third term). The relative weights has been manually chosen to balance the primary goal of upright stabilization with the secondary aims of smooth motion and energy efficiency. The normalization ensures that the angle is always in the range $[-\pi, \pi]$ so that the pendulum positions (e.g., 0 and 2π) are treated identically, which could otherwise confuse learning algorithms.

Studying the code further, we find that it imposes bound constraints on both the control input and the angular velocity through clipping operations:

$$\begin{aligned} u &= \max(\min(u, u_{max}), -u_{max}) \\ \dot{\theta} &= \max(\min(\dot{\theta}, \dot{\theta}_{max}), -\dot{\theta}_{max}) \end{aligned}$$

Where $u_{max} = 2.0$ and $\dot{\theta}_{max} = 8.0$. Finally, when inspecting the `step` function, we find that the dynamics are discretized using forward Euler under a fixed step size of $h = 0.0.5$. Overall, the discrete-time trajectory optimization

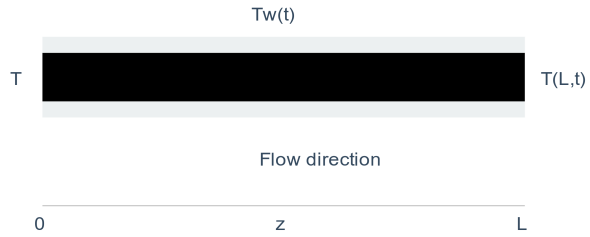
problem implemented in Gym is the following:

$$
\begin{aligned}
\min_{u_k} \quad & J = \sum_{k=0}^{N-1} c(\theta_k, \dot{\theta}_k, u_k) \\
\text{subject to:} \quad & \theta_{k+1} = \theta_k + \dot{\theta}_k \cdot h \\
& \dot{\theta}_{k+1} = \dot{\theta}_k + \left(\frac{3g}{2l} \sin(\theta_k) + \frac{3}{ml^2} u_k \right) \cdot h \\
& -u_{\max} \leq u_k \leq u_{\max} \\
& -\dot{\theta}_{\max} \leq \dot{\theta}_k \leq \dot{\theta}_{\max}, \quad k = 0, 1, ..., N-1 \\
\text{given:} \quad & \theta_0 = \theta_{\text{initial}}, \quad \dot{\theta}_0 = \dot{\theta}_{\text{initial}}, \quad N = 200
\end{aligned}
$$

with $g = 10.0$, $l = 1.0$, $m = 1.0$, $u_{max} = 2.0$, and $\dot{\theta}_{max} = 8.0$. This discrete-time problem corresponds to the following continuous-time optimal control problem:

$$
\begin{aligned}
\min_{u(t)} \quad & J = \int_0^T c(\theta(t), \dot{\theta}(t), u(t)) dt \\
\text{subject to:} \quad & \dot{\theta}(t) = \dot{\theta}(t) \\
& \ddot{\theta}(t) = \frac{3g}{2l} \sin(\theta(t)) + \frac{3}{ml^2} u(t) \\
& -u_{\max} \leq u(t) \leq u_{\max} \\
& -\dot{\theta}_{\max} \leq \dot{\theta}(t) \leq \dot{\theta}_{\max} \\
\text{given:} \quad & \theta(0) = \theta_0, \quad \dot{\theta}(0) = \dot{\theta}_0, \quad T = 10 \text{ seconds}
\end{aligned}
$$

5.1.3 Heat Exchanger



We are considering a system where fluid flows through a tube, and the goal is to control the temperature of the fluid by adjusting the temperature of the tube's wall over time. The wall temperature, denoted as $T_w(t)$, can be changed as a function of time, but it remains the same along the length of the tube. On the other hand, the temperature of the fluid inside the tube, $T(z,t)$, depends

both on its position along the tube z and on time t. It evolves according to the following partial differential equation:

$$\frac{\partial T}{\partial t} = -v\frac{\partial T}{\partial z} + \frac{h}{\rho C_p}(T_w(t) - T) \tag{690}$$

where we have:

- v: the average speed of the fluid moving through the tube,
- h: how easily heat transfers from the wall to the fluid,
- ρ and C_p: the fluid's density and heat capacity.

This equation describes how the fluid's temperature changes as it moves along the tube and interacts with the tube's wall temperature. The fluid enters the tube with an initial temperature T_0 at the inlet (where $z = 0$). Our objective is to adjust the wall temperature $T_w(t)$ so that by a specific final time t_f, the fluid's temperature reaches a desired distribution $T_s(z)$ along the length of the tube. The relationship for $T_s(z)$ under steady-state conditions (ie. when changes over time are no longer considered), is given by:

$$\frac{dT_s}{dz} = \frac{h}{v\rho C_p}[\theta - T_s] \tag{691}$$

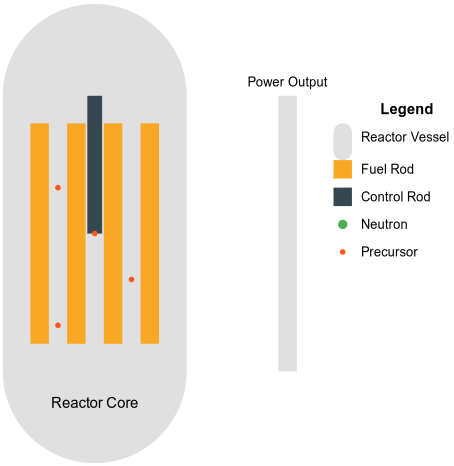
where θ is a constant temperature we want to maintain at the wall. The objective is to control the wall temperature $T_w(t)$ so that by the end of the time interval t_f, the fluid temperature $T(z, t_f)$ is as close as possible to the desired distribution $T_s(z)$. This can be formalized by minimizing the following quantity:

$$I = \int_0^L \left[T(z, t_f) - T_s(z)\right]^2 dz \tag{692}$$

where L is the length of the tube. Additionally, we require that the wall temperature cannot exceed a maximum allowable value $T_{\max}$:

$$T_w(t) \leq T_{\max} \tag{693}$$

5.1.4 Nuclear Reactor



In a nuclear reactor, neutrons interact with fissile nuclei, causing nuclear fission. This process produces more neutrons and smaller fissile nuclei called precursors. The precursors subsequently absorb more neutrons, generating "delayed" neutrons. The kinetic energy of these products is converted into thermal energy through collisions with neighboring atoms. The reactor's power output is determined by the concentration of neutrons available for nuclear fission.

The reaction kinetics can be modeled using a system of ordinary differential equations:

$$\begin{aligned} \dot{x}(t) &= \frac{r(t)x(t) - \alpha x^2(t) - \beta x(t)}{\tau} + \mu y(t), & x(0) &= x_0 \\ \dot{y}(t) &= \frac{\beta x(t)}{\tau} - \mu y(t), & y(0) &= y_0 \end{aligned}$$

where:

- $x(t)$: concentration of neutrons at time t
- $y(t)$: concentration of precursors at time t
- t: time
- $r(t) = r[u(t)]$: degree of change in neutron multiplication at time t as a function of control rod displacement $u(t)$
- α: reactivity coefficient
- β: fraction of delayed neutrons
- μ: decay constant for precursors
- τ: average time taken by a neutron to produce a neutron or precursor

The power output can be adjusted based on demand by inserting or retracting a neutron-absorbing control rod. Inserting the control rod absorbs neutrons, reducing the heat flux and power output, while retracting the rod has the opposite effect.

The objective is to change the neutron concentration $x(t)$ from an initial value x_0 to a stable value x_f at time t_f while minimizing the displacement of the control rod. This can be formulated as an optimal control problem, where the goal is to find the control function $u(t)$ that minimizes the objective functional:

$$I = \int_0^{t_\mathrm{f}} u^2(t)\,\mathrm{d}t$$

subject to the final conditions:

$$\begin{aligned} x(t_\mathrm{f}) &= x_\mathrm{f} \\ \dot{x}(t_\mathrm{f}) &= 0 \end{aligned}$$

and the constraint $|u(t)| \leq u_{\max}$

5.1.5 Chemotherapy

Chemotherapy uses drugs to kill cancer cells. However, these drugs can also have toxic effects on healthy cells in the body. To optimize the effectiveness of chemotherapy while minimizing its side effects, we can formulate an optimal control problem.

The drug concentration $y_1(t)$ and the number of immune cells $y_2(t)$, healthy cells $y_3(t)$, and cancer cells $y_4(t)$ in an organ at any time t during chemotherapy can be modeled using a system of ordinary differential equations:

$$\begin{aligned} \dot{y}_1(t) &= u(t) - \gamma_6 y_1(t) \\ \dot{y}_2(t) &= \dot{y}_{2,\mathrm{in}} + r_2 \frac{y_2(t) y_4(t)}{\beta_2 + y_4(t)} - \gamma_3 y_2(t) y_4(t) - \gamma_4 y_2(t) - \alpha_2 y_2(t) \left(1 - e^{-y_1(t)\lambda_2}\right) \\ \dot{y}_3(t) &= r_3 y_3(t)\left(1 - \beta_3 y_3(t)\right) - \gamma_5 y_3(t) y_4(t) - \alpha_3 y_3(t) \left(1 - e^{-y_1(t)\lambda_3}\right) \\ \dot{y}_4(t) &= r_1 y_4(t)\left(1 - \beta_1 y_4(t)\right) - \gamma_1 y_3(t) y_4(t) - \gamma_2 y_2(t) y_4(t) - \alpha_1 y_4(t) \left(1 - e^{-y_1(t)\lambda_1}\right) \end{aligned}$$

where:

- $y_1(t)$: drug concentration in the organ at time t
- $y_2(t)$: number of immune cells in the organ at time t
- $y_3(t)$: number of healthy cells in the organ at time t
- $y_4(t)$: number of cancer cells in the organ at time t

- $\dot{y}_{2,\text{in}}$: constant rate of immune cells entering the organ to fight cancer cells
- $u(t)$: rate of drug injection into the organ at time t
- r_i, β_i: constants in the growth terms
- α_i, λ_i: constants in the decay terms due to the action of the drug
- γ_i: constants in the remaining decay terms

The objective is to minimize the number of cancer cells $y_4(t)$ in a specified time t_f while using the minimum amount of drug to reduce its toxic effects. This can be formulated as an optimal control problem, where the goal is to find the control function $u(t)$ that minimizes the objective functional:

$$I = y_4(t_\text{f}) + \int_0^{t_\text{f}} u(t)\,\mathrm{d}t$$

subject to the system dynamics, initial conditions, and the constraint $u(t) \geq 0$.

Additional constraints may include:

- Maintaining a minimum number of healthy cells during treatment:

$$y_3(t) \geq y_{3,\min}$$

- Imposing an upper limit on the drug dosage:

$$u(t) \leq u_{\max}$$

5.1.6 Government Corruption

In this model from Feichtinger and Wirl (1994), we aim to understand the incentives for politicians to engage in corrupt activities or to combat corruption. The model considers a politician's popularity as a dynamic process that is influenced by the public's memory of recent and past corruption. The objective is to find conditions under which self-interested politicians would choose to be honest or dishonest.

The model introduces the following notation:

- $C(t)$: accumulated awareness (knowledge) of past corruption at time t
- $u(t)$: extent of corruption (politician's control variable) at time t
- δ: rate of forgetting past corruption
- $P(t)$: politician's popularity at time t
- $g(P)$: growth function of popularity; $g''(P) < 0$

- $f(C)$: function measuring the loss of popularity caused by C; $f'(C) > 0$, $f''(C) \geq 0$
- $U_1(P)$: benefits associated with being popular; $U_1'(P) > 0$, $U_1''(P) \leq 0$
- $U_2(u)$: benefits resulting from bribery and fraud; $U_2'(u) > 0$, $U_2''(u) < 0$
- r: discount rate

The dynamics of the public's memory of recent and past corruption $C(t)$ are modeled as:

$$\dot{C}(t) = u(t) - \delta C(t), \quad C(0) = C_0$$

The evolution of the politician's popularity $P(t)$ is governed by:

$$\dot{P}(t) = g(P(t)) - f(C(t)), \quad P(0) = P_0$$

The politician's objective is to maximize the following objective:

$$\int_0^\infty e^{-rt}[U_1(P(t)) + U_2(u(t))]\,\mathrm{d}t$$

subject to the dynamics of corruption awareness and popularity.
The optimal control problem can be formulated as follows:

$$\begin{aligned} \max_{u(\cdot)} \quad & \int_0^\infty e^{-rt}[U_1(P(t)) + U_2(u(t))]\,\mathrm{d}t \\ \text{s.t.} \quad & \dot{C}(t) = u(t) - \delta C(t), \quad C(0) = C_0 \\ & \dot{P}(t) = g(P(t)) - f(C(t)), \quad P(0) = P_0 \end{aligned}$$

The state variables are the accumulated awareness of past corruption $C(t)$ and the politician's popularity $P(t)$. The control variable is the extent of corruption $u(t)$. The objective functional represents the discounted stream of benefits coming from being honest (popularity) and from being dishonest (corruption).

5.2 Solving Initial Value Problems

An ODE is an implicit representation of a state-space trajectory: it tells us how the state changes in time but not precisely what the state is at any given time. To find out this information, we need to either solve the ODE analytically (for some special structure) or, as we're going to do, solve them numerically. This numerical procedure is meant to solve what is called an IVP (initial value problem) of the form:

$$\text{Find } x(t) \text{ given } \dot{x}(t) = f(x(t), t) \text{ and } x(t_0) = x_0 \tag{694}$$

5.2.1 Euler's Method

The algorithm to solve this problem is, in its simplest form, a for loop which closely resembles the updates encountered in gradient descent (in fact, gradient descent can be derived from the gradient flow ODE, but that's another discussion). The so-called explicit Euler's method can be implemented as follow:

Consider the following simple dynamical system of a ballistic motion model, neglecting air resistance. The state of the system is described by two variables: $y(t)$: vertical position at time t and $v(t)$, the vertical velocity at time t. The corresponding ODE is:

$$\begin{aligned} \frac{dy}{dt} &= v \\ \frac{dv}{dt} &= -g \end{aligned} \tag{695}$$

where $g \approx 9.81$ m/s^2 is the acceleration due to gravity. In our code, we use the initial conditions $y(0) = 0$ m and $v(0) = v_0$ m/s where v_0 is the initial velocity (in this case, $v_0 = 20$ m/s). The analytical solution to this system is:

$$\begin{aligned} y(t) &= v_0 t - \frac{1}{2} g t^2 \\ v(t) &= v_0 - gt \end{aligned} \tag{696}$$

This system models the vertical motion of an object launched upward, reaching a maximum height before falling back down due to gravity.

Euler's method can be obtained by taking the first-order Taylor expansion of $x(t)$ at t:

$$x(t+h) \approx x(t) + h\frac{dx}{dt}(t) = x(t) + hf(x(t), t) \tag{697}$$

Each step of the algorithm therefore involves approximating the function with a linear function of slope f over the given interval h.

Another way to understand Euler's method is through the fundamental theorem of calculus:

$$x(t+h) = x(t) + \int_t^{t+h} f(x(\tau), \tau) d\tau \tag{698}$$

We then approximate the integral term with a box of width h and height f, and therefore of area hf.

5.2.2 Implicit Euler's Method

An alternative approach is the Implicit Euler method, also known as the Backward Euler method. Instead of using the derivative at the current point to step forward, it uses the derivative at the end of the interval. This leads to the following update rule:

$$x_{new} = x + hf(x_{new}, t_{new}) \tag{699}$$

Note that x_{new} appears on both sides of the equation, making this an implicit method. The algorithm for the Implicit Euler method can be described as follows:

The main difference in the Implicit Euler method is step 4, where we need to solve a (potentially nonlinear) equation to find x_{new}. This is typically done using iterative methods such as fixed-point iteration or Newton's method.

Stiff ODEs While the Implicit Euler method requires more computation per step, it often allows for larger step sizes and can provide better stability for certain types of problems, especially stiff ODEs.

Stiff ODEs are differential equations for which certain numerical methods for solving the equation are numerically unstable, unless the step size is taken to be extremely small. These ODEs typically involve multiple processes occurring at widely different rates. In a stiff problem, the fastest-changing component of the solution can make the numerical method unstable unless the step size is extremely small. However, such a small step size may lead to an impractical amount of computation to traverse the entire interval of interest.

For example, consider a chemical reaction where some reactions occur very quickly while others occur much more slowly. The fast reactions quickly approach their equilibrium, but small perturbations in the slower reactions can cause rapid changes in the fast reactions.

A classic example of a stiff ODE is the Van der Pol oscillator with a large parameter. The Van der Pol equation is:

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0 \tag{700}$$

where μ is a scalar parameter. This second-order ODE can be transformed into a system of first-order ODEs by introducing a new variable $y = \frac{dx}{dt}$:

$$\begin{aligned} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= \mu(1 - x^2)y - x \end{aligned} \tag{701}$$

When μ is large (e.g., $\mu = 1000$), this system becomes stiff. The large μ causes rapid changes in y when x is near ± 1, but slower changes elsewhere.

This leads to a solution with sharp transitions followed by periods of gradual change.

5.2.3 Trapezoid Method

The trapezoid method, also known as the trapezoidal rule, offers improved accuracy and stability compared to the simple Euler method. The name "trapezoid method" comes from the idea of using a trapezoid to approximate the integral term in the fundamental theorem of calculus. This leads to the following update rule:

$$x_{new} = x + \frac{h}{2}[f(x,t) + f(x_{new}, t_{new})] \tag{702}$$

where $t_{new} = t+h$. Note that this formula involves x_{new} on both sides of the equation, making it an implicit method, similar to the implicit Euler method discussed earlier.

Algorithmically, the trapezoid method can be described as follows:

The trapezoid method can also be derived by averaging the forward Euler and backward Euler methods. Recall that:

1. **Forward Euler method:**

$$x_{n+1} = x_n + hf(x_n, t_n) \tag{703}$$

2. **Backward Euler method:**

$$x_{n+1} = x_n + hf(x_{n+1}, t_{n+1}) \tag{704}$$

Taking the average of these two methods yields:

$$\begin{aligned} x_{n+1} &= \frac{1}{2}\left(x_n + hf(x_n, t_n)\right) + \frac{1}{2}\left(x_n + hf(x_{n+1}, t_{n+1})\right) \\ &= x_n + \frac{h}{2}\left(f(x_n, t_n) + f(x_{n+1}, t_{n+1})\right) \end{aligned} \tag{705}$$

This gives us the update rule for the trapezoid method. Recall that the forward Euler method approximates the solution by extrapolating linearly using the slope at the beginning of the interval $[t_n, t_{n+1}]$. In contrast, the backward Euler method extrapolates linearly using the slope at the end of the interval. The trapezoid method, on the other hand, averages these two slopes. This averaging provides better approximation properties than either of the methods alone, offering both stability and accuracy. Note finally that unlike the forward or backward Euler methods, the trapezoid method is also symmetric in time. This means that if you were to reverse time and apply the method backward, you would get the same results (up to numerical precision).

5.2.4 Trapezoidal Predictor-Corrector

The trapezoid method can also be implemented under the so-called predictor-corrector framework. This interpretation reformulates the implicit trapezoid rule into an explicit two-step process:

1. **Predictor Step**:
   We make an initial guess for x_{n+1} using the forward Euler method:

   $$x_{n+1}^* = x_n + hf(x_n, t_n) \tag{706}$$

   This is our "predictor" step, where x_{n+1}^* is the predicted value of x_{n+1}.

2. **Corrector Step**:
   We then use this predicted value to estimate $f(x_{n+1}^*, t_{n+1})$ and apply the trapezoid formula:

   $$x_{n+1} = x_n + \frac{h}{2}\left[f(x_n, t_n) + f(x_{n+1}^*, t_{n+1})\right] \tag{707}$$

   This is our "corrector" step, where the initial guess x_{n+1}^* is corrected by taking into account the slope at (x_{n+1}^*, t_{n+1}).

This two-step process is similar to performing one iteration of Newton's method to solve the implicit trapezoid equation, starting from the Euler prediction. However, to fully solve the implicit equation, multiple iterations would be necessary until convergence is achieved.

5.2.5 Collocation Methods

The numerical integration methods we discussed earlier are inherently **sequential**: given an initial state, we step forward in time and approximate what happens over a short interval. The accuracy of this procedure depends on the chosen rule (Euler, trapezoid, Runge–Kutta) and on the information available locally. Each new state is obtained by evaluating a formula that approximates the derivative or integral over that small step.

Collocation methods provide an alternative viewpoint. Instead of advancing one step at a time, they approximate the entire trajectory with a finite set of basis functions and require the dynamics to hold at selected points. This replaces the original differential equation with a system of **algebraic equations**: relations among the coefficients of the basis functions that must all be satisfied simultaneously. Solving these equations fixes the whole trajectory in one computation.

Seen from this angle, integration rules, spline interpolation, quadrature, and collocation are all instances of the same principle: an infinite-dimensional problem is reduced to finitely many parameters linked by numerical rules. The difference is mainly in scope. Sequential integration advances the state forward one interval at a time, which makes it simple but prone to error accumulation.

Collocation belongs to the class of **simultaneous methods** already introduced for DOCPs: the entire trajectory is represented at once, the dynamics are imposed everywhere in the discretization, and approximation error is spread across the horizon rather than accumulating step by step.

This global enforcement comes at a computational cost since the resulting algebraic system is larger and denser. However, the benefit is precisely the structural one we saw in simultaneous methods earlier: by exposing the coupling between states, controls, and dynamics explicitly, collocation allows solvers to exploit sparsity and to enforce path constraints directly at the collocation points. This is why collocation is especially effective for challenging continuous-time optimal control problems where robustness and constraint satisfaction are central.

5.2.6 Quick Primer on Polynomials

Collocation methods are based on polynomial approximation theory. Therefore, the first step in developing collocation-based optimal control techniques is to review the fundamentals of polynomial functions.

Polynomials are typically introduced through their standard form:

$$p(t) = a_n t^n + a_{n-1} t^{n-1} + \cdots + a_1 t + a_0 \tag{708}$$

In this expression, the a_i are coefficients which linearly combine the powers of t to represent a function. The set of functions $\{1, t, t^2, t^3, \ldots, t^n\}$ used in the standard polynomial representation is called the **monomial basis**.

In linear algebra, a basis is a set of vectors in a vector space such that any vector in the space can be uniquely represented as a linear combination of these basis vectors. In the same way, a **polynomial basis** is such that any function $f(x)$ (within the function space) to be expressed as:

$$f(x) = \sum_{k=0}^{\infty} c_k p_k(x), \tag{709}$$

where the coefficients c_k are generally determined by solving a system of equation.

Just as vectors can be represented in different coordinate systems (bases), functions can also be expressed using various polynomial bases. However, the ability to apply a change of basis does not imply that all types of polynomials are equivalent from a practical standpoint. In practice, our choice of polynomial basis is dictated by considerations of efficiency, accuracy, and stability when approximating a function.

For instance, despite the monomial basis being easy to understand and implement, it often performs poorly in practice due to numerical instability. This instability arises as its coefficients take on large values: an ill-conditioning problem. The following kinds of polynomial often remedy this issues.

Orthogonal Polynomials An **orthogonal polynomial basis** is a set of polynomials that are orthogonal to each other and form a complete basis for a certain space of functions. This means that any function within that space can be represented as a linear combination of these polynomials.

More precisely, let $\{p_0(x), p_1(x), p_2(x), \dots\}$ be a sequence of polynomials where each $p_n(x)$ is a polynomial of degree n. We say that this set forms an orthogonal polynomial basis if any polynomial $q(x)$ of degree n or less can be uniquely expressed as a linear combination of $\{p_0(x), p_1(x), \dots, p_n(x)\}$. Furthermore, the orthogonality property means that for any $i \neq j$:

$$\langle p_i, p_j \rangle = \int_a^b p_i(x) p_j(x) w(x)\, dx = 0. \tag{710}$$

for some weight function $w(x)$ over a given interval of orthogonality $[a, b]$.

The orthogonality property allows to simplify the computation of the coefficients involved in the polynomial representation of a function. At a high level, what happens is that when taking the inner product of $f(x)$ with each basis polynomial, $p_k(x)$ isolates the corresponding coefficient c_k, which can be found to be:

$$c_k = \frac{\langle f, p_k \rangle}{\langle p_k, p_k \rangle} = \frac{\int_a^b f(x) p_k(x) w(x)\, dx}{\int_a^b p_k(x)^2 w(x)\, dx}. \tag{711}$$

Here are some examples of the most common orthogonal polynomials used in practice.

Legendre Polynomials Legendre polynomials $\{P_n(x)\}$ are defined on the interval $[-1, 1]$ and satisfy the orthogonality condition:

$$\int_{-1}^{1} P_n(x) P_m(x)\, dx = \begin{cases} 0 & \text{if } n \neq m, \\ \frac{2}{2n+1} & \text{if } n = m. \end{cases} \tag{712}$$

They can be generated using the recurrence relation:

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x), \tag{713}$$

with initial conditions:

$$P_0(x) = 1, \quad P_1(x) = x. \tag{714}$$

The first four Legendre polynomials resulting from this recurrence are the following:

Chebyshev Polynomials There are two types of Chebyshev polynomials: **Chebyshev polynomials of the first kind**, $\{T_n(x)\}$, and **Chebyshev polynomials of the second kind**, $\{U_n(x)\}$. We typically focus on the first kind. They are defined on the interval $[-1, 1]$ and satisfy the orthogonality condition:

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\,dx = \begin{cases} 0 & \text{if } n \neq m, \\ \frac{\pi}{2} & \text{if } n = m \neq 0, \\ \pi & \text{if } n = m = 0. \end{cases} \tag{715}$$

The Chebyshev polynomials of the first kind can be generated using the recurrence relation:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x), \tag{716}$$

with initial conditions:

$$T_0(x) = 1, \quad T_1(x) = x. \tag{717}$$

This recurrence relation also admits an explicit formula:

$$T_n(x) = \cos(n\cos^{-1}(x)). \tag{718}$$

Let's now implement it in Python:

Hermite Polynomials Hermite polynomials $\{H_n(x)\}$ are defined on the entire real line and are orthogonal with respect to the weight function $w(x) = e^{-x^2}$. They satisfy the orthogonality condition:

$$\int_{-\infty}^{\infty} H_n(x)H_m(x)e^{-x^2}\,dx = \begin{cases} 0 & \text{if } n \neq m, \\ 2^n n!\sqrt{\pi} & \text{if } n = m. \end{cases} \tag{719}$$

Hermite polynomials can be generated using the recurrence relation:

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x), \tag{720}$$

with initial conditions:

$$H_0(x) = 1, \quad H_1(x) = 2x. \tag{721}$$

The following code computes the coefficients of the first four Hermite polynomials:

5.2.7 Collocation Conditions

Consider a general ODE of the form:

$$\dot{y}(t) = f(y(t), t), \quad y(t_0) = y_0, \tag{722}$$

where $y(t) \in R^n$ is the state vector, and $f : R^n \times R \to R^n$ is a known function. The goal is to approximate the solution $y(t)$ over a given interval $[t_0, t_f]$. Collocation methods achieve this by:

1. **Choosing a basis** to approximate $y(t)$ using a finite sum of basis functions $\phi_i(t)$:

$$y(t) \approx \sum_{i=0}^{N} c_i \phi_i(t), \tag{723}$$

where $\{c_i\}$ are the coefficients to be determined.

2. **Selecting collocation points** $t_1, t_2, \ldots, t_N$ within the interval $[t_0, t_f]$. These are typically chosen to be the roots of certain orthogonal polynomials, like Legendre or Chebyshev polynomials, or can be spread equally across the interval.

3. **Enforcing the ODE at the collocation points** for each t_j:

$$\dot{y}(t_j) = f(y(t_j), t_j). \tag{724}$$

To implement this, we differentiate the approximate solution $y(t)$ with respect to time:

$$\dot{y}(t) \approx \sum_{i=0}^{N} c_i \dot{\phi}_i(t). \tag{725}$$

Substituting this into the ODE at the collocation points gives:

$$\sum_{i=0}^{N} c_i \dot{\phi}_i(t_j) = f\left(\sum_{i=0}^{N} c_i \phi_i(t_j), t_j\right), \quad j = 1, \ldots, N. \tag{726}$$

The collocation equations are formed by enforcing the ODE at all collocation points, leading to a system of nonlinear equations:

$$\sum_{i=0}^{N} c_i \dot{\phi}_i(t_j) - f\left(\sum_{i=0}^{N} c_i \phi_i(t_j), t_j\right) = 0, \quad j = 1, \ldots, N. \tag{727}$$

Furthermore when solving an initial value problem (IVP), we also need to incorporate the initial condition $y(t_0) = y_0$ as an additional constraint:

$$\sum_{i=0}^{N} c_i \phi_i(t_0) = y_0. \tag{728}$$

The collocation conditions and IVP condition are combined together to form a root-finding problem, which we can generically solve numerically using Newton's method.

5.2.8 Common Numerical Integration Techniques as Collocation Methods

Many common numerical integration techniques can be viewed as special cases of collocation methods. While the general collocation method we discussed earlier applies to the entire interval $[t_0, t_f]$, many numerical integration techniques can be viewed as collocation methods applied locally, step by step.

In practical numerical integration, we often divide the full interval $[t_0, t_f]$ into smaller subintervals or steps. In general, this allows us to user simpler basis functions thereby reducing computational complexity, and gives us more flexibility in dynamically ajusting the step size using local error estimates. When we apply collocation locally, we're essentially using the collocation method to "step" from t_n to t_{n+1}. As we did, earlier we still apply the following three steps:

1. We choose a basis function to approximate $y(t)$ over $[t_n, t_{n+1}]$.
2. We select collocation points within this interval.
3. We enforce the ODE at these points to determine the coefficients of our basis function.

We can make this idea clearer by re-deriving some of the numerical integration methods seen before using this perspective.

Explicit Euler Method For the Explicit Euler method, we use a linear basis function for each step:

$$\phi(t) = 1 + c(t - t_n) \tag{729}$$

Note that we use $(t - t_n)$ rather than just t because we're approximating the solution locally, relative to the start of each step. We then choose one collocation point at t_{n+1} where we have:

$$y'(t_{n+1}) = c = f(y_n, t_n) \tag{730}$$

Our local approximation is:

$$y(t) \approx y_n + c(t - t_n) \tag{731}$$

At $t = t_{n+1}$, this gives:

$$y_{n+1} = y_n + c(t_{n+1} - t_n) = y_n + hf(y_n, t_n) \tag{732}$$

where $h = t_{n+1} - t_n$. This is the classic Euler update formula.

Implicit Euler Method The Implicit Euler method uses the same linear basis function:

$$\phi(t) = 1 + c(t - t_n) \tag{733}$$

Again, we choose one collocation point at t_{n+1}. The main difference is that we enforce the ODE using y_{n+1}:

$$y'(t_{n+1}) = c = f(y_{n+1}, t_{n+1}) \tag{734}$$

Our approximation remains:

$$y(t) \approx y_n + c(t - t_n) \tag{735}$$

At $t = t_{n+1}$, this leads to the implicit equation:

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}) \tag{736}$$

Trapezoidal Method The Trapezoidal method uses a quadratic basis function:

$$\phi(t) = 1 + c(t - t_n) + a(t - t_n)^2 \tag{737}$$

We use two collocation points: t_n and t_{n+1}. Enforcing the ODE at these points gives:

- At t_n:

$$y'(t_n) = c = f(y_n, t_n) \tag{738}$$

- At t_{n+1}:

$$y'(t_{n+1}) = c + 2ah = f(y_n + ch + ah^2, t_{n+1}) \tag{739}$$

Our approximation is:

$$y(t) \approx y_n + c(t - t_n) + a(t - t_n)^2 \tag{740}$$

At $t = t_{n+1}$, this gives:

$$y_{n+1} = y_n + ch + ah^2 \tag{741}$$

Solving the system of equations leads to the trapezoidal update:

$$y_{n+1} = y_n + \frac{h}{2}[f(y_n, t_n) + f(y_{n+1}, t_{n+1})] \tag{742}$$

Runge-Kutta Methods Higher-order Runge-Kutta methods can also be interpreted as collocation methods. The RK4 method corresponds to a collocation method using a cubic polynomial basis:

$$\phi(t) = 1 + c_1(t - t_n) + c_2(t - t_n)^2 + c_3(t - t_n)^3 \tag{743}$$

Here, we're using a cubic polynomial to approximate the solution over each step, rather than the linear or quadratic approximations of the other methods above. For RK4, we use four collocation points:

1. t_n (the start of the step)
2. $t_n + h/2$
3. $t_n + h/2$
4. $t_n + h$ (the end of the step)

These points are called the "Gauss-Lobatto" points, scaled to our interval $[t_n, t_n + h]$. The RK4 method enforces the ODE at these collocation points, leading to four stages:

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \frac{1}{2}k_1, t_n + \frac{h}{2}) \\ k_3 &= hf(y_n + \frac{1}{2}k_2, t_n + \frac{h}{2}) \\ k_4 &= hf(y_n + k_3, t_n + h) \end{aligned} \tag{744}$$

The final update formula for RK4 can be derived by solving the system of equations resulting from enforcing the ODE at our collocation points:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{745}$$

5.2.9 Example: Solving a Simple ODE by Collocation

Consider a simple ODE:

$$\frac{dy}{dt} = -y, \quad y(0) = 1, \quad t \in [0, 2] \tag{746}$$

The analytical solution is $y(t) = e^{-t}$. We apply the collocation method with a monomial basis of order N:

$$\phi_i(t) = t^i, \quad i = 0, 1, \ldots, N \tag{747}$$

We select N equally spaced points $\{t_1, \ldots, t_N\}$ in $[0, 2]$ as collocation points.

5.3 Nonlinear Programming

Unless specific assumptions are made on the dynamics and cost structure, a DOCP is, in its most general form, a nonlinear mathematical program (commonly referred to as an NLP, not to be confused with Natural Language Processing). An NLP can be formulated as follows:

$$\begin{aligned} &\text{minimize } f(\mathbf{x}) \\ &\text{subject to } \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ &\qquad\qquad \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned} \tag{748}$$

Where:

- $f : R^n \to R$ is the objective function
- $\mathbf{g} : R^n \to R^m$ represents inequality constraints
- $\mathbf{h} : R^n \to R^\ell$ represents equality constraints

Unlike unconstrained optimization commonly used in deep learning, the optimality of a solution in constrained optimization must consider both the objective value and constraint feasibility. To illustrate this, consider the following problem, which includes both equality and inequality constraints:

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2) = (x_1 - 1)^2 + (x_2 - 2.5)^2 \\ \text{subject to} \quad & g(x_1, x_2) = (x_1 - 1)^2 + (x_2 - 1)^2 \leq 1.5, \\ & h(x_1, x_2) = x_2 - (0.5 \sin(2\pi x_1) + 1.5) = 0. \end{aligned}$$

In this example, the objective function $f(x_1, x_2)$ is quadratic, the inequality constraint $g(x_1, x_2)$ defines a circular feasible region centered at $(1, 1)$ with a radius of $\sqrt{1.5}$ and the equality constraint $h(x_1, x_2)$ requires x_2 to lie on a sine wave function. The following code demonstrates the difference between the unconstrained, and constrained solutions to this problem.

Karush-Kuhn-Tucker (KKT) conditions While this example is simple enough to convince ourselves visually of the solution to this particular problem, it falls short of providing us with actionable chracterization of what constitutes and optimal solution in general. The Karush-Kuhn-Tucker (KKT) conditions provide us with an answer to this problem by generalizing the first-order optimality conditions in unconstrained optimization to problems involving both equality and inequality constraints. This result relies on the construction of an auxiliary function called the Lagrangian, defined as:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \boldsymbol{\mu}^\top \mathbf{g}(\mathbf{x}) + \boldsymbol{\lambda}^\top \mathbf{h}(\mathbf{x}) \tag{749}$$

where $\boldsymbol{\mu} \in R^m$ and $\boldsymbol{\lambda} \in R^\ell$ are known as Lagrange multipliers. The first-order optimality conditions then state that if $\mathbf{x}^*$, then there must exist corresponding Lagrange multipliers $\boldsymbol{\mu}^*$ and $\boldsymbol{\lambda}^*$ such that:

Definition 5.1. *1. The gradient of the Lagrangian with respect to* $\mathbf{x}$ *must be zero at the optimal point (**stationarity**):*

$$\nabla_x \mathcal{L}(\mathbf{x}^*, \boldsymbol{\mu}^*, \boldsymbol{\lambda}^*) = \nabla f(\mathbf{x}^*) + \sum_{i=1}^{m} \mu_i^* \nabla g_i(\mathbf{x}^*) + \sum_{j=1}^{\ell} \lambda_j^* \nabla h_j(\mathbf{x}^*) = \mathbf{0} \quad (750)$$

In the case where we only have equality constraints, this means that the gradient of the objective and that of constraint are parallel to each other at the optimum but point in opposite directions.

2. *A valid solution of a NLP is one which satisfies all the constraints (**primal feasibility**)*

$$\mathbf{g}(\mathbf{x}^*) \leq \mathbf{0}, \quad \textit{and } \mathbf{h}(\mathbf{x}^*) \quad = \mathbf{0} \quad (751)$$

3. *Furthermore, the Lagrange multipliers for **inequality** constraints must be non-negative (**dual feasibility**)*

$$\boldsymbol{\mu}^* \geq \mathbf{0} \quad (752)$$

This condition stems from the fact that the inequality constraints can only push the solution in one direction.

4. *Finally, for each inequality constraint, either the constraint is active (equality holds) or its corresponding Lagrange multiplier is zero at an optimal solution (**complementary slackness**)*

$$\mu_i^* g_i(\mathbf{x}^*) = 0, \quad \forall i = 1, \ldots, m \quad (753)$$

Let's now solve our example problem above, this time using Ipopt via the Pyomo interface so that we can access the Lagrange multipliers found by the solver.

```
from pyomo.environ import *
from pyomo.opt import SolverFactory
import math

# Define the Pyomo model
model = ConcreteModel()

# Define the variables
model.x1 = Var(initialize=1.25)
model.x2 = Var(initialize=1.5)

# Define the objective function
```

```
def objective_rule(model):
    return (model.x1 - 1)**2 + (model.x2 - 2.5)**2
model.obj = Objective(rule=objective_rule, sense=minimize)

# Define the inequality constraint (circle)
def inequality_constraint_rule(model):
    return (model.x1 - 1)**2 + (model.x2 - 1)**2 <= 1.5
model.ineq_constraint = Constraint(rule=inequality_constraint_rule)

# Define the equality constraint (sine wave) using Pyomo's math functions
def equality_constraint_rule(model):
    return model.x2 == 0.5 * sin(2 * math.pi * model.x1) + 1.5
model.eq_constraint = Constraint(rule=equality_constraint_rule)

# Create a suffix component to capture dual values
model.dual = Suffix(direction=Suffix.IMPORT)

# Create a solver
solver=SolverFactory('ipopt')

# Solve the problem
results = solver.solve(model, tee=False)

# Check if the solver found an optimal solution
if (results.solver.status == SolverStatus.ok and
    results.solver.termination_condition == TerminationCondition.optimal):

    # Print the results
    print(f"x1: {value(model.x1)}")
    print(f"x2: {value(model.x2)}")

    # Print the objective value
    print(f"Objective value: {value(model.obj)}")

    # Print the Lagrange multipliers (dual values)
    print("\nLagrange multipliers:")
    ineq_lambda = None
    eq_lambda = None
    for c in model.component_objects(Constraint, active=True):
        for index in c:
            dual_val = model.dual[c[index]]
            print(f"{c.name}[{index}]: {dual_val}")
            if c.name == "ineq_constraint":
                ineq_lambda = dual_val
            elif c.name == "eq_constraint":
                eq_lambda = dual_val
```

```
else:
    print("Solver did not find an optimal solution.")
    print(f"Solver Status: {results.solver.status}")
    print(f"Termination Condition: {results.solver.termination_condition}")
```

After running the code above, we can observe the Lagrange multipliers. The Lagrange multiplier associated with the inequality constraint is very small (close to zero), suggesting that the inequality constraint is not active at the optimal solution—meaning that the solution point lies inside the circle defined by this constraint. This can be verified visually in the figure above. As for the equality constraint, its corresponding Lagrange multiplier is non-zero, indicating that this constraint is active at the optimal solution. In general, when we find a Lagrange multiplier close to zero (like the one for the inequality constraint), it means that constraint is not "binding"—the optimal solution does not lie on the boundary defined by this constraint. In contrast, a non-zero Lagrange multiplier, such as the one for the equality constraint, indicates that the constraint is active and that any relaxation would directly affect the objective function's value, as required by the stationarity condition.

Lagrange Multiplier Theorem The KKT conditions introduced above characterize the solution structure of constrained optimization problems with equality constraints. In this particular context, these conditions are referred to as the first-order optimality conditions, as part of the Lagrange multiplier theorem. Let's just re-state them in that simpler setting:

Definition 5.2 (Lagrange Multiplier Theorem)**.** *Consider the constrained optimization problem:*

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \textit{subject to} \quad & h_i(\mathbf{x}) = 0, \quad i = 1, \ldots, m \end{aligned} \tag{754}$$

where $\mathbf{x} \in R^n$*,* $f : R^n \to R$*, and* $h_i : R^n \to R$ *for* $i = 1, \ldots, m$*. Assume that:*

1. f *and* h_i *are continuously differentiable functions.*
2. *The gradients* $\nabla h_i(\mathbf{x}^*)$ *are linearly independent at the optimal point* $\mathbf{x}^*$*.*

Then, there exist unique Lagrange multipliers $\lambda_i^* \in R$*,* $i = 1, \ldots, m$*, such that the following first-order optimality conditions hold:*

1. *Stationarity:* $\nabla f(\mathbf{x}^*) + \sum_{i=1}^m \lambda_i^* \nabla h_i(\mathbf{x}^*) = \mathbf{0}$
2. *Primal feasibility:* $h_i(\mathbf{x}^*) = 0$*, for* $i = 1, \ldots, m$

Note that both the stationarity and primal feasibility statements are simply saying that the derivative of the Lagrangian in either the primal or dual variables must be zero at an optimal constrained solution. In other words:

$$\nabla_{\mathbf{x},\boldsymbol{\lambda}} L(\mathbf{x}^*, \boldsymbol{\lambda}^*) = \mathbf{0} \tag{755}$$

Letting $\mathbf{F}(\mathbf{x}, \boldsymbol{\lambda})$ stand for $\nabla_{\mathbf{x},\boldsymbol{\lambda}} L(\mathbf{x}, \boldsymbol{\lambda})$, the Lagrange multipliers theorem tells us that an optimal primal-dual pair is actually a zero of that function $\mathbf{F}$: the derivative of the Lagrangian. Therefore, we can use this observation to craft a solution method for solving equality constrained optimization using Newton's method, which is a numerical procedure for finding zeros of a nonlinear function.

Newton's Method Newton's method is a numerical procedure for solving root-finding problems. These are nonlinear systems of equations of the form:

Find $\mathbf{z}^* \in R^n$ such that $\mathbf{F}(\mathbf{z}^*) = \mathbf{0}$

where $\mathbf{F} : R^n \to R^n$ is a continuously differentiable function. Newton's method then consists in applying the following sequence of iterates:

$$\mathbf{z}^{k+1} = \mathbf{z}^k - [\nabla \mathbf{F}(\mathbf{z}^k)]^{-1} \mathbf{F}(\mathbf{z}^k) \tag{756}$$

where $\mathbf{z}^k$ is the k-th iterate, and $\nabla \mathbf{F}(\mathbf{z}^k)$ is the Jacobian matrix of $\mathbf{F}$ evaluated at $\mathbf{z}^k$.

Newton's method exhibits local quadratic convergence: if the initial guess $\mathbf{z}^0$ is sufficiently close to the true solution $\mathbf{z}^*$, and $\nabla \mathbf{F}(\mathbf{z}^*)$ is nonsingular, the method converges quadratically to $\mathbf{z}^*$ Ortega and Rheinboldt [1970]. However, the method is sensitive to the initial guess; if it's too far from the desired solution, Newton's method might fail to converge or converge to a different root. To mitigate this problem, a set of techniques known as numerical continuation methods Allgower and Georg [1990] have been developed. These methods effectively enlarge the basin of attraction of Newton's method by solving a sequence of related problems, progressing from an easy one to the target problem. This approach is reminiscent of several concepts in machine learning and statistical inference: curriculum learning in machine learning, where models are trained on increasingly complex data; tempering in Markov Chain Monte Carlo (MCMC) samplers, which gradually adjusts the target distribution to improve mixing; and modern diffusion models, which use a similar concept of gradually transforming noise into structured data.

Efficient Implementation of Newton's Method Note that each step of Newton's method involves computing the inverse of a Jacobian matrix. However, a cardinal rule in numerical linear algebra is to avoid computing matrix inverses explicitly: rarely, if ever, should there be a `np.lindex.inv` in your code. Instead, the numerically stable and computationally efficient approach is to solve a linear system of equations at each step. Given the Newton's method iterate:

$$\mathbf{z}^{k+1} = \mathbf{z}^k - [\nabla \mathbf{F}(\mathbf{z}^k)]^{-1} \mathbf{F}(\mathbf{z}^k) \tag{757}$$

We can reformulate this as a two-step procedure:

1. Solve the linear system: $\underbrace{[\nabla \mathbf{F}(\mathbf{z}^k)]}_{\mathbf{A}} \Delta \mathbf{z}^k = -\mathbf{F}(\mathbf{z}^k)$

2. Update: $\mathbf{z}^{k+1} = \mathbf{z}^k + \Delta \mathbf{z}^k$

The structure of the linear system in step 1 often allows for specialized solution methods. In the context of automatic differentiation, matrix-free linear solvers are particularly useful. These solvers can find a solution without explicitly forming the matrix A, requiring only the ability to evaluate matrix-vector or vector-matrix products. Typical examples of such methods include classical matrix-splitting methods (e.g., Richardson iteration) or conjugate gradient methods through `sparse.linalg.cg` for example. Another useful method is the Generalized Minimal Residual method (GMRES) implemented in SciPy via `sparse.linalg.gmres`, which is useful when facing non-symmetric and indefinite systems.

By inspecting the structure of matrix $\mathbf{A}$ in the specific application where the function $\mathbf{F}$ is the derivative of the Lagrangian, we will also uncover an important structure known as the KKT matrix. This structure will then allow us to derive a Quadratic Programming (QP) sub-problem as part of a larger iterative procedure for solving equality and inequality constrained problems via Sequential Quadratic Programming (SQP).

Solving Equality Constrained Programs with Newton's Method To solve equality-constrained optimization problems using Newton's method, we begin by recognizing that the problem reduces to finding a zero of the function $\mathbf{F}(\mathbf{z}) = \nabla_{\mathbf{x},\boldsymbol{\lambda}} L(\mathbf{x}, \boldsymbol{\lambda})$. Here, $\mathbf{F}$ represents the derivative of the Lagrangian function, and $\mathbf{z} = (\mathbf{x}, \boldsymbol{\lambda})$ combines both the primal variables $\mathbf{x}$ and the dual variables (Lagrange multipliers) $\boldsymbol{\lambda}$. Explicitly, we have:

$$\mathbf{F}(\mathbf{z}) = \begin{bmatrix} \nabla_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}) \\ \mathbf{h}(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \nabla f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla h_i(\mathbf{x}) \\ \mathbf{h}(\mathbf{x}) \end{bmatrix}. \tag{758}$$

Newton's method involves linearizing $\mathbf{F}(\mathbf{z})$ around the current iterate $\mathbf{z}^k = (\mathbf{x}^k, \boldsymbol{\lambda}^k)$ and then solving the resulting linear system. At each iteration k, Newton's method updates the current estimate by solving the linear system:

$$\mathbf{z}^{k+1} = \mathbf{z}^k - [\nabla \mathbf{F}(\mathbf{z}^k)]^{-1} \mathbf{F}(\mathbf{z}^k). \tag{759}$$

However, instead of explicitly inverting the Jacobian matrix $\nabla \mathbf{F}(\mathbf{z}^k)$, we solve the linear system:

$$\underbrace{\nabla \mathbf{F}(\mathbf{z}^k)}_{\mathbf{A}} \Delta \mathbf{z}^k = -\mathbf{F}(\mathbf{z}^k), \tag{760}$$

where $\Delta \mathbf{z}^k = (\Delta \mathbf{x}^k, \Delta \boldsymbol{\lambda}^k)$ represents the Newton step for the primal and dual variables. Substituting the expression for $\mathbf{F}(\mathbf{z})$ and its Jacobian, the system becomes:

$$\begin{bmatrix} \nabla^2_{\mathbf{xx}} L(\mathbf{x}^k, \boldsymbol{\lambda}^k) & \nabla \mathbf{h}(\mathbf{x}^k)^T \\ \nabla \mathbf{h}(\mathbf{x}^k) & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}^k \\ \Delta \boldsymbol{\lambda}^k \end{bmatrix} = - \begin{bmatrix} \nabla f(\mathbf{x}^k) + \nabla \mathbf{h}(\mathbf{x}^k)^T \boldsymbol{\lambda}^k \\ \mathbf{h}(\mathbf{x}^k) \end{bmatrix}. \tag{761}$$

The matrix on the left-hand side is known as the KKT matrix, as it stems from the Karush-Kuhn-Tucker conditions for this optimization problem The solution of this system provides the updates $\Delta \mathbf{x}^k$ and $\Delta \boldsymbol{\lambda}^k$, which are then used to update the primal and dual variables:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \Delta \mathbf{x}^k, \quad \boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k + \Delta \boldsymbol{\lambda}^k. \tag{762}$$

Demonstration The following code demonstates how we can implement this idea in Jax. In this demonstration, we are minimizing a quadratic objective function subject to a single equality constraint, a problem formally stated as follows:

$$\begin{aligned} \min_{x \in R^2} \quad & f(x) = (x_1 - 2)^2 + (x_2 - 1)^2 \\ \text{subject to} \quad & h(x) = x_1^2 + x_2^2 - 1 = 0 \end{aligned} \tag{763}$$

Geometrically speaking, the constraint $h(x)$ describes a unit circle centered at the origin. To solve this problem using the method of Lagrange multipliers, we form the Lagrangian:

$$L(x, \lambda) = f(x) + \lambda h(x) = (x_1 - 2)^2 + (x_2 - 1)^2 + \lambda(x_1^2 + x_2^2 - 1) \tag{764}$$

For this particular problem, it happens so that we can also find an analytical without even having to use Newton's method. From the first-order optimality conditions, we obtain the following linear system of equations:

$$\begin{aligned} 2(x_1 - 2) + 2\lambda x_1 &= 0 \\ 2(x_2 - 1) + 2\lambda x_2 &= 0 \\ x_1^2 + x_2^2 - 1 &= 0 \end{aligned}$$

From the first two equations, we then get:

$$x_1 = \frac{2}{1 + \lambda}, \quad x_2 = \frac{1}{1 + \lambda} \tag{765}$$

which we can substitute these into the 3rd constraint equation to obtain:

$$(\frac{2}{1 + \lambda})^2 + (\frac{1}{1 + \lambda})^2 = 1 \Leftrightarrow \lambda = \sqrt{5} - 1 \tag{766}$$

This value of the Lagrange multiplier can then be backsubstituted into the above equations to obtain $x_1 = \frac{2}{\sqrt{5}}$ and $x_2 = \frac{1}{\sqrt{5}}$. We can verify numerically (and visually on the following graph) that the point $(2/\sqrt{5}, 1/\sqrt{5})$ is indeed the point on the unit circle closest to $(2, 1)$.

5.3.1 The SQP Approach: Taylor Expansion and Quadratic Approximation

Sequential Quadratic Programming (SQP) tackles the problem of solving constrained programs by iteratively solving a sequence of simpler subproblems. Specifically, these subproblems are quadratic programs (QPs) that approximate the original problem around the current iterate by using a quadratic model of the objective function and a linear model of the constraints. Suppose we have the following optimization problem with equality constraints:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{h}(\mathbf{x}) = \mathbf{0}. \end{aligned} \tag{767}$$

At each iteration k, we approximate the objective function $f(\mathbf{x})$ using a second-order Taylor expansion around the current iterate $\mathbf{x}^k$. The standard Taylor expansion for f would be:

$$f(\mathbf{x}) \approx f(\mathbf{x}^k) + \nabla f(\mathbf{x}^k)^T(\mathbf{x} - \mathbf{x}^k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^k)^T \nabla^2 f(\mathbf{x}^k)(\mathbf{x} - \mathbf{x}^k).$$

This expansion uses the **Hessian of the objective function** $\nabla^2 f(\mathbf{x}^k)$ to capture the curvature of f. However, in the context of constrained optimization, we also need to account for the effect of the constraints on the local behavior of the solution. If we were to use only $\nabla^2 f(\mathbf{x}^k)$, we would not capture the influence of the constraints on the curvature of the feasible region. The resulting subproblem might then lead to steps that violate the constraints or are less effective in achieving convergence. The choice that we make instead is to use the Hessian of the Lagrangian, $\nabla^2_{\mathbf{xx}} L(\mathbf{x}^k, \boldsymbol{\lambda}^k)$, leading to the following quadratic model:

$$f(\mathbf{x}) \approx f(\mathbf{x}^k) + \nabla f(\mathbf{x}^k)^T(\mathbf{x} - \mathbf{x}^k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^k)^T \nabla^2_{\mathbf{xx}} L(\mathbf{x}^k, \boldsymbol{\lambda}^k)(\mathbf{x} - \mathbf{x}^k). \tag{768}$$

Similarly, the equality constraints $\mathbf{h}(\mathbf{x})$ are linearized around $\mathbf{x}^k$:

$$\mathbf{h}(\mathbf{x}) \approx \mathbf{h}(\mathbf{x}^k) + \nabla \mathbf{h}(\mathbf{x}^k)(\mathbf{x} - \mathbf{x}^k). \tag{769}$$

Combining these approximations, we obtain a Quadratic Programming (QP) subproblem, which approximates our original problem locally at $\mathbf{x}^k$ but is easier to solve:

$$\begin{aligned} \text{Minimize} \quad & \nabla f(\mathbf{x}^k)^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \nabla^2_{\mathbf{xx}} L(\mathbf{x}^k, \boldsymbol{\lambda}^k)\Delta\mathbf{x} \\ \text{subject to} \quad & \nabla \mathbf{h}(\mathbf{x}^k)\Delta\mathbf{x} + \mathbf{h}(\mathbf{x}^k) = \mathbf{0}, \end{aligned} \tag{770}$$

where $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}^k$. The QP subproblem solved at each iteration focuses on finding the optimal step direction $\Delta\mathbf{x}$ for the primal variables. While solving this

QP, we obtain not only the step $\Delta\mathbf{x}$ but also the associated Lagrange multipliers for the QP subproblem, which correspond to an updated dual variable vector $\boldsymbol{\lambda}^{k+1}$. More specifically, after solving the QP, we use $\Delta\mathbf{x}^k$ to update the primal variables:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \Delta\mathbf{x}^k.$$

Simultaneously, the Lagrange multipliers from the QP provide the updated dual variables $\boldsymbol{\lambda}^{k+1}$. We summarize the SQP algorithm in the following pseudocode:

Connection to Newton's Method in the Equality-Constrained Case The QP subproblem in SQP is directly related to applying Newton's method for equality-constrained optimization. To see this, note that the KKT matrix of the QP subproblem is:

$$\begin{bmatrix} \nabla^2_{\mathbf{xx}} L(\mathbf{x}^k, \boldsymbol{\lambda}^k) & \nabla\mathbf{h}(\mathbf{x}^k)^T \\ \nabla\mathbf{h}(\mathbf{x}^k) & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}^k \\ \Delta\boldsymbol{\lambda}^k \end{bmatrix} = - \begin{bmatrix} \nabla f(\mathbf{x}^k) + \nabla\mathbf{h}(\mathbf{x}^k)^T\boldsymbol{\lambda}^k \\ \mathbf{h}(\mathbf{x}^k) \end{bmatrix}$$

This is exactly the same linear system that have to solve when applying Newton's method to the KKT conditions of the original program! Thus, solving the QP subproblem at each iteration of SQP is equivalent to taking a Newton step on the KKT conditions of the original nonlinear problem.

5.3.2 SQP for Inequality-Constrained Optimization

So far, we've applied the ideas behind Sequential Quadratic Programming (SQP) to problems with only equality constraints. Now, let's extend this framework to handle optimization problems that also include inequality constraints. Consider a general nonlinear optimization problem that includes both equality and inequality constraints:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0}, \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0}. \end{aligned}$$

As we did earlier, we approximate this problem by constructing a quadratic approximation to the objective and a linearization of the constraints. QP subproblem at each iteration is then formulated as:

$$\begin{aligned} \text{Minimize} \quad & \nabla f(\mathbf{x}^k)^T\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T\nabla^2_{\mathbf{xx}} L(\mathbf{x}^k, \boldsymbol{\lambda}^k, \boldsymbol{\nu}^k)\Delta\mathbf{x} \\ \text{subject to} \quad & \nabla\mathbf{g}(\mathbf{x}^k)\Delta\mathbf{x} + \mathbf{g}(\mathbf{x}^k) \leq \mathbf{0}, \\ & \nabla\mathbf{h}(\mathbf{x}^k)\Delta\mathbf{x} + \mathbf{h}(\mathbf{x}^k) = \mathbf{0}, \end{aligned}$$

where $\Delta \mathbf{x} = \mathbf{x} - \mathbf{x}^k$ represents the step direction for the primal variables. The following pseudocode outlines the steps involved in applying SQP to a problem with both equality and inequality constraints:

Demonstration with JAX and CVXPy Consider the following equality and inequality-constrained problem:

$$\begin{aligned} \min_{x \in R^2} \quad & f(x) = (x_1 - 2)^2 + (x_2 - 1)^2 \\ \text{subject to} \quad & g(x) = x_1^2 - x_2 \leq 0 \\ & h(x) = x_1^2 + x_2^2 - 1 = 0 \end{aligned}$$

This example builds on our previous one but adds a parabola-shaped inequality constraint. We require our solution to lie not only on the circle defining our equality constraint but also below the parabola. To solve the QP subproblem, we will be using the CVXPY package. While the Lagrangian and derivatives could be computed easily by hand, we use JAX for generality:

5.3.3 The Arrow-Hurwicz-Uzawa algorithm

While the SQP method addresses constrained optimization problems by sequentially solving quadratic subproblems, an alternative approach emerges from viewing constrained optimization as a min-max problem. This perspective leads to a simpler algorithm, originally introduced by the Arrow-Hurwicz-Uzawa Arrow et al. [1958]. Consider the following general constrained optimization problem encompassing both equality and inequality constraints:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned} \tag{771}$$

Using the Lagrangian function $L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\mathbf{x}) + \boldsymbol{\mu}^T \mathbf{g}(\mathbf{x}) + \boldsymbol{\lambda}^T \mathbf{h}(\mathbf{x})$, we can reformulate this problem as the following min-max problem:

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}, \boldsymbol{\mu} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \tag{772}$$

The role of each component in this min-max structure can be understood as follows:

1. The outer minimization over $\mathbf{x}$ finds the feasible point that minimizes the objective function $f(\mathbf{x})$.
2. The maximization over $\boldsymbol{\mu} \geq 0$ ensures that inequality constraints $\mathbf{g}(\mathbf{x}) \leq \mathbf{0}$ are satisfied. If any inequality constraint is violated, the corresponding term in $\boldsymbol{\mu}^T \mathbf{g}(\mathbf{x})$ can be made arbitrarily large by choosing a large enough μ_i.

3. The maximization over $\boldsymbol{\lambda}$ ensures that equality constraints $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ are satisfied.

Using this observation, we can devise an algorithm which, like SQP, will update both the primal and dual variables at every step. But rather than using second-order optimization, we will simply use a first-order gradient update step: a descent step in the primal variable, and an ascent step in the dual one. The corresponding procedure, when implemented by gradient descent, is called Gradient Ascent Descent in the learning and optimization communities. In the case of equality constraints only, the algorithm looks like the following:

Now to account for the fact that the Lagrange multiplier needs to be non-negative for inequality constraints, we can use our previous idea from projected gradient descent for bound constraints and consider a projection, or clipping step to ensure that this condition is satisfied throughout. In this case, the algorithm looks like the following:

Here, $[\cdot]_+$ denotes the projection onto the non-negative orthant, ensuring that $\boldsymbol{\mu}$ remains non-negative.

However, as it is widely known from the lessons of GAN (Generative Adversarial Network) training Goodfellow et al. [2014], Gradient Descent Ascent (GDA) can fail to converge or suffer from instability. The Arrow-Hurwicz-Uzawa algorithm, also known as the first-order Lagrangian method, is known to converge only locally, in the vicinity of an optimal primal-dual pair.

5.3.4 Projected Gradient Descent

The Arrow-Hurwicz-Uzawa algorithm provided a way to handle constraints through dual variables and a primal-dual update scheme. Another commonly used approach for constrained optimization is **Projected Gradient Descent (PGD)**. The idea is simple: take a gradient descent step as if the problem were unconstrained, then project the result back onto the feasible set. Formally:

$$\mathbf{x}_{k+1} = \mathcal{P}_C\big(\mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)\big), \tag{773}$$

where $\mathcal{P}_C$ is the projection onto the feasible set C, α is the step size, and $f(\mathbf{x})$ is the objective function.

PGD is particularly effective when the projection is computationally cheap. A common example is **box constraints** (or bound constraints), where the feasible set is a hyperrectangle:

$$C = \{\mathbf{x} \mid \mathbf{x}_{\text{lb}} \leq \mathbf{x} \leq \mathbf{x}_{\text{ub}}\}. \tag{774}$$

In this case, the projection reduces to an element-wise clipping operation:

$$[\mathcal{P}_C(\mathbf{x})]_i = \max\big(\min([\mathbf{x}]_i, [\mathbf{x}_{\text{ub}}]_i), [\mathbf{x}_{\text{lb}}]_i\big). \tag{775}$$

For bound-constrained problems, PGD is almost as easy to implement as standard gradient descent because the projection step is just a clipping operation. For more general constraints, however, the projection may require solving

a separate optimization problem, which can be as hard as the original task. Here is the algorithm for a problem of the form:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x}_{\text{lb}} \le \mathbf{x} \le \mathbf{x}_{\text{ub}}. \end{aligned} \tag{776}$$

The clipping function is defined as:

$$\text{clip}(x, x_{\text{lb}}, x_{\text{ub}}) = \max\big(\min(x, x_{\text{ub}}), x_{\text{lb}}\big). \tag{777}$$

Under mild conditions such as Lipschitz continuity of the gradient, PGD converges to a stationary point of the constrained problem. Its simplicity and low cost make it a common choice whenever the projection can be computed efficiently.

6 References

6.1 Bibliography

References

C. P. Adams and V. V. Brantner. Spending on new drug development1. *Health Economics*, 19(2):130–141, 2 2009. ISSN 1099-1050. doi: 10.1002/hec.1454. URL http://dx.doi.org/10.1002/hec.1454.

E. L. Allgower and K. Georg. *Numerical Continuation Methods: An Introduction*, volume 13 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, Heidelberg, 1990.

K. J. Arrow, L. Hurwicz, and H. Uzawa. *Studies in linear and non-linear programming*. Stanford University Press, 1958.

J. T. Ash and R. P. Adams. Warm-starting and Amortization in Continual Learning. In *International Conference on Learning Representations (ICLR)*, 2020.

D. P. Bertsekas. Distributed asynchronous computation of fixed points. *Mathematical Programming*, 27(1):107–120, 9 1983. ISSN 1436-4646. doi: 10.1007/bf02591967. URL http://dx.doi.org/10.1007/BF02591967.

C3S. Era5 hourly data on single levels from 1940 to present, 2018. URL https://cds.climate.copernicus.eu/doi/10.24381/cds.adbb2d47.

M. Chang. *Monte Carlo Simulation for the Pharmaceutical Industry: Concepts, Algorithms, and Case Studies*. CRC Press, 9 2010. ISBN 9780429152382. doi: 10.1201/ebk1439835920. URL http://dx.doi.org/10.1201/EBK1439835920.

W. J. Cole, K. M. Powell, E. T. Hale, and T. F. Edgar. Reduced-order residential home modeling for model predictive control. *Energy and Buildings*, 74:69–77, 2014. ISSN 0378-7788. doi: https://doi.org/10.1016/j.enbuild.2014.01.033. URL https://www.sciencedirect.com/science/article/pii/S0378778814000711.

M. J. Conroy and J. T. Peterson. *Decision Making in Natural Resource Management: A Structured, Adaptive Approach: A Structured, Adaptive Approach*. Wiley, 1 2013. ISBN 9781118506196. doi: 10.1002/9781118506196. URL http://dx.doi.org/10.1002/9781118506196.

P. D'Oro, M. Schwarzer, E. Nikishin, P.-L. Bacon, M. G. Bellemare, and A. C. Courville. Sample-Efficient Reinforcement Learning by Breaking the Replay Ratio Barrier. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

D. Ernst, P. Geurts, and L. Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *J. Mach. Learn. Res.*, 6:503–556, 2005a. URL https://jmlr.org/papers/v6/ernst05a.html.

D. Ernst, P. Geurts, and L. Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6:503–556, 2005b.

S. Fujimoto, H. Hoof, and D. Meger. Addressing Function Approximation Error in Actor-Critic Methods. In *International Conference on Machine Learning (ICML)*, pages 1587–1596, 2018.

M. Gargiani, A. Zanelli, D. Liao-McPherson, T. H. Summers, and J. Lygeros. Dynamic Programming Through the Lens of Semismooth Newton-Type Methods. *IEEE Control Systems Letters*, 6:2996–3001, 2022. doi: 10.1109/LCSYS.2022.3181213.

M. Geist, B. Scherrer, and O. Pietquin. A Theory of Regularized Markov Decision Processes. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2160–2169. PMLR, jun 9 2019. URL https://proceedings.mlr.press/v97/geist19a.html.

P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 3 2006. ISSN 1573-0565. doi: 10.1007/s10994-006-6226-1. URL http://dx.doi.org/10.1007/s10994-006-6226-1.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, volume 27, 2014.

G. J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on International Conference on Machine Learning*, ICML'95, pages 261–268, Tahoe City, California, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1558603778.

G. J. Gordon. Approximate Solutions to Markov Decision Problems, 1999. Technical Report CMU-CS-99-111.

A. I. Grancharova and T. A. Johansen. *Explicit nonlinear model predictive control*. Lecture notes in control and information sciences. Springer, Berlin, Germany, 2012 edition, 3 2012.

J. Gravdahl and O. Egeland. Compressor surge control using a close-coupled valve and backstepping. In *Proceedings of the 1997 American Control Conference (Cat. No.97CH36041)*, pages 982–986 vol.2. IEEE, 1997. doi: 10.1109/acc.1997.609673. URL http://dx.doi.org/10.1109/ACC.1997.609673.

T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement Learning with Deep Energy-Based Policies. *Proceedings of the 34th International Conference on Machine Learning*, 70:1352–1361, 2017.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 1861–1870. PMLR, 2018.

R. Hafner and M. Riedmiller. Reinforcement learning in feedback control: Challenges and benchmarks from technical process control. *Machine Learning*, 84 (1-2):137–169, 2 2011. ISSN 1573-0565. doi: 10.1007/s10994-011-5235-x. URL http://dx.doi.org/10.1007/s10994-011-5235-x.

K. L. Judd. Projection methods for solving aggregate growth models. *Journal of Economic Theory*, 58(2):410–452, 1992.

K. L. Judd. *Approximation, perturbation, and projection methods in economic analysis*, volume 1, pages 509–585. Elsevier, 1996.

S. Kortum. Value Function Approximation in an Estimation Routine, 1992. Manuscript, Boston University.

S. Levine, A. Kumar, G. Tucker, and J. Fu. Reinforcement Learning as a Framework for Control: A Survey. *arXiv preprint arXiv:1806.04222*, 2018.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous Control with Deep Reinforcement Learning. *arXiv preprint arXiv:1509.02971*, 2015.

L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning, and teaching, 1992. Technical Report, CMU-CS-92-170.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, and others. Playing Atari with Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*, 2013.

D. Ormoneit and Ś. Sen. Kernel-Based Reinforcement Learning. *Machine Learning*, 49(2/3):161–178, 2002. ISSN 0885-6125. doi: 10.1023/a:1017928328829. URL http://dx.doi.org/10.1023/A:1017928328829.

J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Computer Science and Applied Mathematics. Academic Press, New York, 1970.

M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, 1994. ISBN 978-0-471-61977-3. First published in 1994.

M. Riedmiller. Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. In *Proceedings of the 16th European Conference on Machine Learning (ECML)*, pages 317–328, Berlin, Heidelberg, 2005a. Springer.

M. A. Riedmiller. Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method. In J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, editors, *Machine Learning: ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, volume 3720 of *Lecture Notes in Computer Science*, pages 317–328. Springer, 2005b. doi: 10.1007/11564096_32. URL https://doi.org/10.1007/11564096%5C_32.

J. Rust. Optimal Replacement of GMC Bus Engines: An Empirical Model of Harold Zurcher. *Econometrica*, 55(5):999–1033, 1987.

J. Rust. *Chapter 14 Numerical dynamic programming in economics*, pages 619–729. Elsevier, 1996. doi: 10.1016/s1574-0021(96)01016-7. URL http://dx.doi.org/10.1016/S1574-0021(96)01016-7.

M. S. Santos and J. Vigo-Aguiar. Analysis of a numerical dynamic programming algorithm applied to economic models. *Econometrica*, 66(2):409–426, 1998.

C. Savorgnan, C. Romani, A. Kozma, and M. Diehl. Multiple shooting for distributed systems with applications in hydro electricity production. *Journal of Process Control*, 21(5):738–745, 6 2011. ISSN 0959-1524. doi: 10.1016/j.jprocont.2011.01.011. URL http://dx.doi.org/10.1016/j.jprocont.2011.01.011.

Y. Sawaguchi, E. Furutani, G. Shirakami, M. Araki, and K. Fukuda. A Model-Predictive Hypnosis Control System Under Total Intravenous Anesthesia. *IEEE Transactions on Biomedical Engineering*, 55(3):874–887, 3 2008. ISSN 0018-9294. doi: 10.1109/tbme.2008.915670. URL http://dx.doi.org/10.1109/tbme.2008.915670.

J. Stachurski. *Economic Dynamics: Theory and Computation*. MIT Press, Cambridge, MA, 2009. ISBN 9780262012775.

J. Sun. Openap.top: Open Flight Trajectory Optimization for Air Transport and Sustainability Research. *Aerospace*, 9(7):383, 7 2022. ISSN 2226-4310. doi: 10.3390/aerospace9070383. URL http://dx.doi.org/10.3390/aerospace9070383.

H. Van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), 2016.

B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey. Maximum Entropy Inverse Reinforcement Learning. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1433–1438, 2008.