Machine Learning for Computational Economics

Dejanir H. Silva

November 26, 2025

Contents

Chapter 1

Introduction

Dynamic problems are ubiquitous in economics and finance, which explains why dynamic programming has become one of the cornerstones of modern economic analysis. Recursive methods provide the conceptual and computational foundation for studying intertemporal choice, strategic interaction, and equilibrium dynamics across a wide range of fields—from consumption and investment to firm behavior, asset pricing, public finance, and macroeconomics.

Yet, there remains a persistent tension between the dynamic problems we *can* currently solve and those we *would like* to solve in order to capture the richness of behavior observed in the data. It is now well documented that investors differ substantially in their beliefs, wealth, preferences, and financial sophistication. Assets differ in their characteristics, including risk, return, liquidity, and market completeness. Firms differ in productivity, technology, and financing constraints. Households are subject to heterogeneous income, wealth, and policy shocks. Capturing this diversity requires solving *high-dimensional* models with a large number of state variables and parameters.

The challenge is that the computational cost of traditional dynamic programming methods grows exponentially with the number of state variables—a phenomenon known as the *curse of dimensionality*. This curse limits our ability to model realistic heterogeneity, uncertainty, and interaction effects at the core of modern macro–finance questions. Overcoming this limitation requires new numerical tools that can scale to high dimensions while preserving stability and accuracy.

In recent years, advances in numerical optimization and machine learning—particularly in automatic differentiation, stochastic gradient methods, and neural-network-based function approximation—have opened the door to a new generation of computational techniques. These tools allow us to revisit long-standing economic problems with fresh computational power and to study models that were previously intractable.

The goal of these lecture notes is to build a unified toolbox that bridges traditional numerical methods in economics with modern machine learning. We begin by reviewing discrete- and continuous-time solution methods for dynamic programming, then show how these classical ideas connect naturally to optimization and learning algorithms, and finally apply deep-learning–based methods to solve and estimate large-scale macro–finance models.

1.1 The Three Curses of Dimensionality

Solving dynamic models in economics and finance often runs into a fundamental barrier: the so-called *curses of dimensionality*. As we enrich our models to capture heterogeneity—across households, firms, assets, or shocks—the number of state variables grows quickly, and three interrelated challenges emerge.

1. The curse of representation. The value and policy functions that describe agents' decisions must be represented on a grid or through basis functions. As the number of state variables increases, the number of grid points required grows exponentially, making traditional discretization methods infeasible.

2. The curse of optimization. Even if we can represent the value and policy functions, finding the optimal control at each point of a high-dimensional state space becomes computationally expensive. The search for optimal decisions can dominate the total cost of solving the model.

3. The curse of expectation. Dynamic problems require taking expectations over future states and shocks. As the number of risk factors increases, computing these expectations—either by integration or simulation—becomes prohibitively costly.

These three curses are the central computational obstacles faced in modern macro-finance modeling. They explain why many classical methods work well in low dimensions but struggle as we move toward richer, more realistic settings. The goal of this book is to show how ideas from modern machine learning—function approximation, stochastic optimization, and automatic differentiation—can be used to overcome each of these curses in turn.

1.2 Roadmap

The rest of these notes are organized into four chapters that progressively build from classical numerical foundations to modern machine-learning applications in dynamic macro–finance.

Chapter 2 — Discrete-Time Methods. We begin with the numerical solution of discrete-time dynamic programming problems. This chapter introduces the Bellman operator, contraction mappings, and value- and policy-function iteration. Students implement these methods in Julia, exploring convergence, monotonicity, and approximation via interpolation. The goal is to establish the recursive logic underlying all subsequent methods.

Chapter 3 — Continuous-Time Methods. We then transition from discrete to continuous time and derive the Hamilton–Jacobi–Bellman (HJB) equation as the continuous limit of the Bellman recursion. The chapter presents finite-difference and spectral (Chebyshev) methods for solving HJB equations, emphasizing numerical stability and boundary conditions. These methods connect naturally to the differential operators and optimization structures later exploited by machine learning.

Chapter 4 — Fundamentals of Machine Learning. This chapter introduces the main elements of modern machine learning from an economist's perspective. We discuss supervised learning, regularization, stochastic gradient descent, and automatic differentiation. Neural networks are presented as flexible nonlinear function approximators, and we show how automatic differentiation and backpropagation provide a computational bridge between traditional numerical optimization and deep learning.

Chapter 5 — Deep Policy Iteration and Applications. The final chapter develops and applies the *Deep Policy Iteration* (DPI) algorithm. We show how neural networks can represent value and policy functions in high-dimensional dynamic models and how stochastic optimization and automatic differentiation enable efficient training. Applications include asset-pricing models (the Lucas orchard), corporate-finance models (Hennessy–Whited), and high-dimensional portfolio-choice problems. These examples demonstrate how machine-learning tools can extend the frontier of solvable macro–finance models.

Together, these chapters aim to build both intuition and practical competence: how to translate economic structure into recursive problems, how to solve those problems numerically, and how to leverage modern computational methods to analyze models that were once beyond reach.

1.3 How to Use These Notes

These notes are designed to be both theoretical and computational. Each chapter is accompanied by interactive Pluto notebooks written in Julia, which reproduce the figures, examples, and algorithms discussed in the text. Students are encouraged to run, modify, and extend these notebooks to deepen their understanding of the methods. All examples rely on open-source Julia packages such as `Plots.jl`, `ForwardDiff.jl`, `Zygote.jl`, and `Lux.jl`, which allow concise and transparent implementation of numerical algorithms.

Although the presentation assumes basic familiarity with calculus, optimization, and dynamic models, the material is largely self-contained. The emphasis throughout is on intuition, reproducibility, and hands-on experimentation rather than black-box computation. Readers who wish to go further will find pointers to the original research papers, as well as suggestions for small projects that extend each module's material.

These notes can be read linearly as a compact course on machine learning for computational economics, or selectively, depending on one's interests. Instructors may adapt the content into shorter modules focused on specific topics such as dynamic programming, numerical PDEs, or neural-network-based optimization.

These notes accompany the course *Machine Learning for Computational Economics*. They blend analytical exposition with executable Julia code snippets, figures, and references.

1.4 Why Julia?

All computations and examples in these notes are implemented in the Julia programming language. Julia is a modern, high-level language designed for numerical and scientific

computing. It combines the ease of writing of high-level languages such as Python or MATLAB with the execution speed of low-level languages such as C or Fortran.

Julia's appeal for computational economics and finance lies in three key features:

1. Performance without sacrificing clarity. Julia is built on just-in-time (JIT) compilation via LLVM, which allows pure Julia code to run at speeds comparable to compiled languages. This makes it possible to prototype and scale numerical algorithms in the same language—no translation step between research code and production code is required. For instance, the same syntax used to define a simple function can be used to write high-performance solvers for dynamic models.

2. Multiple dispatch and composability. Julia's type system and multiple dispatch make it particularly suited for mathematical programming. Functions can be written generically—*"for all types that behave like numbers"*—and specialized automatically by the compiler. This feature underpins many of Julia's most powerful scientific libraries, including `ForwardDiff.jl`, `Zygote.jl`, and `Lux.jl`. It also means that user-defined types, such as the `DualNumber` or neural-network layers we will build later, interact seamlessly with existing packages.

3. Open source and reproducibility. Julia is entirely open source and designed for transparent, reproducible research workflows. Its package manager and project environments make it easy to share self-contained code and datasets. For example, a researcher can reproduce any example in these notes by running

```
Pkg.activate("."); Pkg.instantiate()
```

in the course folder, ensuring that the exact same versions of all packages are used.

4. Integration with modern tools. Julia interfaces naturally with Python, R, and C libraries, and it provides rich environments for literate programming such as Pluto.jl and Jupyter. In these notes, Pluto notebooks are used to bridge exposition and experimentation, letting students modify code cells and immediately visualize the results.

In short, Julia offers the transparency of mathematical pseudocode, the efficiency of compiled languages, and the interactivity of modern ML environments. It is a natural platform for studying the intersection of economics, dynamic programming, and machine learning.

Chapter 2

Discrete-Time Methods

Dynamic programming in discrete time provides the foundation for modern computational economics. This chapter introduces the classical numerical methods used to solve discrete-time models. We illustrate the use of these methods on a consumption-savings problem, which serves as a backbone of many applications in macroeconomics and finance. We use this benchmark problem to illustrate three core computational tasks: (i) representing value and policy functions on a grid, (ii) computing expectations in problems with uncertainty, and (iii) performing the maximization step efficiently.

We begin with the standard *value function iteration* (VFI) algorithm and then show how to iterate directly on policy functions using the *endogenous gridpoint method* (EGM) of Carroll (2006). We also discuss practical aspects such as grid design, interpolation, and Markov-chain discretization of shocks. Throughout, we illustrate the methods with concise Julia implementations.

2.1 A Consumption–Savings Problem

We start from a simple but fundamental setup—the income-fluctuation problem—which captures the core trade-offs faced by a household deciding how much to consume and save when income is uncertain. The household enters each period with cash-on-hand M_t, which consists of financial wealth W_t and current income y_t. The household earns a risk-free return R on its savings and receives a stochastic labor income Y_t. At each date, it chooses consumption c_t and carries wealth into the next period according to

$$M_{t+1} = R\,(M_t - c_t) + Y_{t+1}. \tag{2.1}$$

Let T denote the finite planning horizon and $\rho > 0$ the subjective discount rate. At the end of the planning horizon, the household receives a terminal payoff $V_0(M_T)$, which is a function of terminal cash-on-hand $M_T = W_T + Y_T$. This payoff may represent the utility from consuming terminal wealth or a warm-glow motive for leaving bequests.

The household's problem is to choose a consumption plan $\{c_t\}_{t=0}^{T-1}$ that maximizes expected lifetime utility:

$$V_T(M) = \max_{\{c_t\}_{t=0}^{T-1}} \mathbb{E}\left[\sum_{t=0}^{T-1} e^{-\rho t}\, u(c_t) + e^{-\rho T}\, V_0(M_T)\right], \tag{2.2}$$

subject to the transition equation (2.1) and the stochastic process for labor income,

$$\log Y_{t+1} \sim \mathcal{N}\left(-\tfrac{1}{2}\sigma_y^2,\, \sigma_y^2\right), \tag{2.3}$$

where the normalization ensures that $\mathbb{E}[Y_t] = 1$. For simplicity, labor income is assumed to be i.i.d. across time. We also assume that the household can borrow up to the natural borrowing limit, that is, there is a lower bound on savings, $M_t - c_t$, that ensures that the household can repay its debt next period in all future states of nature. As borrowing at the natural borrowing limit would imply zero future consumption, this constraint is never binding.

We assume preferences and the terminal payoff are of the constant relative risk aversion (CRRA) form:

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}, \qquad V_0(M) = A\frac{M^{1-\gamma}}{1-\gamma}, \tag{2.4}$$

where $\gamma > 0$ is the coefficient of relative risk aversion and $A > 0$ is a scaling parameter.

Equation (2.2) defines the *sequence representation* of the problem: the household must choose a full stochastic process for consumption $\{c_t\}$, contingent on the realizations of labor income up to date t. In practice, it is often more convenient to work with the equivalent *recursive representation*, in which the household's decision is summarized by a policy function $c_t(M)$—the optimal consumption given current cash-on-hand M—and the associated value function $V_t(M)$.

The recursive problem can be written as

$$V_t(M) = \max_c \left\{u(c) + e^{-\rho}\mathbb{E}[V_{t-1}(M')]\right\}, \tag{2.5}$$

subject to

$$M' = R(M-c) + Y', \qquad \log Y' \sim \mathcal{N}\left(-\tfrac{1}{2}\sigma_y^2,\, \sigma_y^2\right). \tag{2.6}$$

Given the value function $V_{t-1}(M)$, the first-order condition for the optimal policy $c_t(M)$ is

$$u'(c_t(M)) = e^{-\rho}\,\mathbb{E}\left[V'_{t-1}(R(M - c_t(M)) + Y')R\right]. \tag{2.7}$$

> **Note**
>
> The infinite-horizon problem, where $T \to \infty$, corresponds to the stationary solution of the Bellman equation (2.5). In this case, the value function satisfies the fixed-point condition $V_t(M) = V_{t-1}(M) \equiv V(M)$.

The Bellman equation can be solved by *backward induction*, starting from the terminal value function $V_0(M)$ and proceeding backward in time. Define the *action-value function*:

$$V_t(M, c) \equiv u(c) + e^{-\rho}\mathbb{E}[V_{t-1}(R(M-c) + Y')]. \tag{2.8}$$

At each step, the optimal consumption choice satisfies $c_t(M) = \arg\max_c V_t(M, c)$, and the value function updates according to $V_t(M) = V_t(M, c_t(M))$. This iterative procedure is known as *value function iteration* (VFI). To approximate the infinite-horizon problem numerically,

we iterate until the distance between successive value functions is sufficiently small. This procedure is summarized in Algorithm 1.

Algorithm 1: Value Function Iteration (VFI)

Input: Initial guess $V^{(0)}(M)$, tolerance `tol`
Output: Value $V(M)$, policy $c^*(M)$

initialize $t \leftarrow 0$
repeat
  Policy update: $c^{(t+1)}(M) \leftarrow \arg\max_c \left\{u(c) + e^{-\rho}\,\mathbb{E}[V^{(t)}(R(M-c)+Y')]\right\}$;
  Value update:
  $V^{(t+1)}(M) \leftarrow u(c^{(t+1)}(M)) + e^{-\rho}\,\mathbb{E}\left[V^{(t)}(R(M-c^{(t+1)}(M))+Y')\right]$;
  $t \leftarrow t+1.$;
until $\|V^{(t+1)} - V^{(t)}\|_\infty < tol$;
return $V^{(t)}$, $c^{(t)}$

2.2 Numerical Solution of the Discrete-Time Problem

Problem (2.2) allows us to illustrate a range of important issues that arise when numerically solving dynamic problems. These issues become particularly relevant once we turn to higher-dimensional problems in later chapters. We discuss three key components of the numerical solution: (i) how to represent the value function, (ii) how to compute expectations, and (iii) how to perform the maximization step.

2.2.1 Representing the Value Function

The first issue we must address is how to represent the value function $V_{t-1}(M)$ on a computer. For a one-dimensional state variable such as cash-on-hand, this is straightforward. We begin by constructing a finite grid for M:

$$M \in \mathcal{M} = \{M_1, M_2, \ldots, M_N\}.$$

At each point on this grid, we store the corresponding value function as a vector:

$$\mathbf{V}_t = (V_{t,1}, V_{t,2}, \ldots, V_{t,N})^\top, \qquad \text{where } V_{t,i} \equiv V_t(M_i).$$

The collection $\mathbf{V} = [\mathbf{V}_0, \mathbf{V}_1, \ldots, \mathbf{V}_{T-1}]$ contains the entire sequence of value functions across time. Similarly, the optimal consumption policy can be represented as a vector:

$$\mathbf{c}_t = (c_{t,1}, c_{t,2}, \ldots, c_{t,N})^\top, \qquad \text{where } c_{t,i} \equiv c_t(M_i).$$

Because the household's state variable M evolves continuously, we need to approximate the value function between grid points. A simple and effective approach is to use linear interpolation. For $M \in [M_{i-1}, M_i]$, we can approximate $V_t(M)$ as

$$V_t(M) \approx \frac{M - M_{i-1}}{M_i - M_{i-1}}\, V_{t,i} + \frac{M_i - M}{M_i - M_{i-1}}\, V_{t,i-1}. \tag{2.9}$$

Linear interpolation ensures that the value function is continuous and monotone between grid points, although its derivative is piecewise constant. In problems with higher curvature or where precision near the borrowing constraint is crucial, one can instead use higher-order interpolation (such as cubic splines) or a non-uniform grid that is denser near $M_{\min}$. We will return to this point below when discussing the numerical implementation of the endogenous gridpoint method.

2.2.2 Computing Expectations

The second issue we must address is how to compute expectations. In our setting, this amounts to integrating over the stochastic labor income Y', which follows the log-normal distribution specified in (2.3). The expectation in the first-order condition (2.7) can be written as

$$\mathbb{E}[V_t'(R(M - c_t(M)) + Y')] = \int_{-\infty}^{\infty} V_t'\Big(R(M - c_t(M)) + e^{y'}\Big)\,\phi\left(\frac{y' - \overline{y}}{\sigma_y}\right) dy', \tag{2.10}$$

where $\phi(\cdot)$ is the probability density function of a standard normal random variable and $\overline{y} \equiv -\frac{1}{2}\sigma_y^2$ ensures that $\mathbb{E}[Y_t] = 1$.

Computing this expectation numerically requires approximating the integral in (2.10). A common approach is to discretize the process for the log income $y_t \equiv \log Y_t$, replacing it by a finite-state Markov chain. This procedure is conceptually similar to evaluating the integral via a quadrature rule.[1]

Tauchen's (1986) method. Consider a random variable z_t that follows an AR(1) process:

$$z_{t+1} = \overline{z} + \rho_z(z_t - \overline{z}) + \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \sim \mathcal{N}(0, \sigma_\varepsilon^2), \tag{2.11}$$

where $|\rho_z| < 1$ and $\sigma_\varepsilon > 0$. In our application, labor income is i.i.d., corresponding to $\rho_z = 0$, but the method extends naturally to persistent processes.

The idea is to construct a discrete grid $\{z_1, z_2, \ldots, z_{N_z}\}$ that covers the relevant support of z_t and to compute transition probabilities $P_{ij} = \Pr(z_{t+1} = z_j \mid z_t = z_i)$ implied by (2.11). Let Δ denote the uniform grid spacing:

$$\Delta = \frac{z_{N_z} - z_1}{N_z - 1}.$$

A convenient choice for the endpoints is

$$z_1 = \overline{z} - m\,\frac{\sigma_z}{\sqrt{1 - \rho_z^2}}, \qquad z_{N_z} = \overline{z} + m\,\frac{\sigma_z}{\sqrt{1 - \rho_z^2}},$$

where m is typically set to 3, ensuring that the grid spans roughly ± 3 unconditional standard deviations of z_t.

[1] Indeed, the discretization method proposed by Tauchen and Hussey (1991) can be interpreted as a Gauss–Hermite quadrature scheme adapted to AR(1) processes. The original Tauchen (1986) method provides an approximation that becomes exact only in the limit as the grid becomes dense.

Conditional on $z_t = z_i$, the next period's value z_{t+1} is normally distributed with mean $\rho_z z_i$ and variance σ_ε^2. The transition probability from z_i to z_j equals the probability that z_{t+1} falls between the midpoints surrounding z_j:

$$P_{ij} = \begin{cases} \Phi\left(\dfrac{z_j - \overline{z} - \rho_z(z_i - \overline{z}) + \frac{\Delta}{2}}{\sigma_\varepsilon}\right), & j = 1, \\ \Phi\left(\dfrac{z_j - \overline{z} - \rho_z(z_i - \overline{z}) + \frac{\Delta}{2}}{\sigma_\varepsilon}\right) - \Phi\left(\dfrac{z_j - \overline{z} - \rho_z(z_i - \overline{z}) - \frac{\Delta}{2}}{\sigma_\varepsilon}\right), & 1 < j < N_z, \\ 1 - \Phi\left(\dfrac{z_j - \overline{z} - \rho_z(z_i - \overline{z}) - \frac{\Delta}{2}}{\sigma_\varepsilon}\right), & j = N_z, \end{cases} \tag{2.12}$$

where $\Phi(\cdot)$ denotes the standard normal cumulative distribution function.

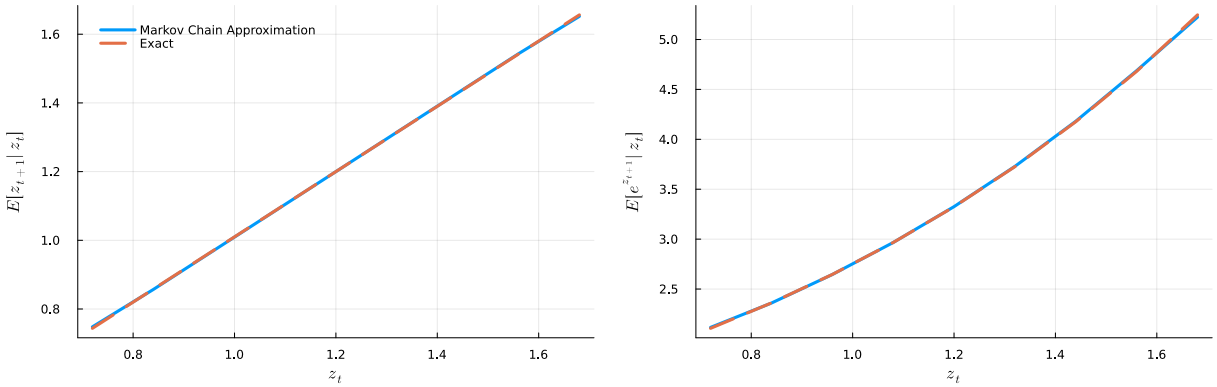
The resulting discrete Markov chain—with states $\{z_j\}$ and transition matrix P—preserves the moments and persistence of the continuous AR(1) process. Accuracy improves with the number of grid points N_z or the coverage parameter m, though at the cost of greater computational effort.

> **Important**
>
> Tauchen's method works particularly well for processes with moderate persistence. For highly persistent processes, the quadrature-based method of Tauchen and Hussey (1991) or the recursive scheme of Rouwenhorst (1995) provides greater accuracy.

Implementation in Julia. Listing 2.1 shows a complete implementation of Tauchen's discretization method in Julia. The function takes as inputs the number of grid points `M`, the persistence parameter ρ, and the innovation standard deviation σ_ε, and returns both the grid `z` and the transition matrix `P`.

```julia
"""
Tauchen (1986) discretization of the AR(1) process
    z_{t+1} = μ + ρ z_t + ε_{t+1},   ε ~ N(0, σε^2).
"""
function tauchen(M::Int, ρ::Real, σε::Real; μ::Real=0.0, m::Real=3.0)
    @assert M ≥ 2 "Need at least M=2 grid points."
    @assert abs(ρ) < 1 "Require |ρ|<1 for stationary AR(1)."
    σz = σε / sqrt(1 - ρ^2)                  # unconditional std. dev.
    z̄  = μ / (1 - ρ)                         # unconditional mean
    if σε == 0.0 return (; z = [z̄ ], P = [1.0]) end # degenerate case
    zmin, zmax = z̄  - m*σz, z̄  + m*σz
    Δ = (zmax - zmin) / (M - 1)
    z = collect(range(zmin, zmax, length=M))
    P = zeros(Float64, M, M)
    for i in 1:M
```

Figure 2.1: Conditional Moments: Exact vs. Markov Chain Approximation



Conditional moments for z_{t+1} (left panel) and $\exp(z_{t+1})$ (right panel) as functions of z_t. The solid blue line corresponds to the conditional moments computed using the Markov-chain approximation, while the dashed black line corresponds to the exact moments. Parameters: AR(1) process $z_{t+1} = \mu + \rho z_t + \varepsilon_{t+1}$ with $\rho = 0.95$, $\mu = 0.06$, $\sigma_\varepsilon = 0.05$; Tauchen grid size $N_z = 9$ and coverage $m = 3$.

```
        mean_next = μ + ρ*z[i]
        dist = Normal(mean_next, σϵ)
        # First bin: (-∞, midpoint_1]
        P[i, 1] = cdf(dist, z[1] + Δ/2)
        # Interior bins: (midpoint_{j-1}, midpoint_j]
        for j in 2:M-1
            upper = z[j] + Δ/2
            lower = z[j] - Δ/2
            P[i, j] = cdf(dist, upper) - cdf(dist, lower)
        end
        # Last bin: (midpoint_{N-1}, +∞)
        P[i, M] = 1 - cdf(dist, z[M] - Δ/2)
    end
    return (;z, P)
end
```

Listing 2.1: Tauchen (1986) discretization of AR(1) processes

Given the grid `z` and transition matrix `P`, conditional expectations of any function $g(z_{t+1})$ can be computed as $P\,g.(\texttt{z})$, where $g.(\texttt{z})$ denotes the vector of $g(z)$ values at the grid points.

Figure 2.1 compares the conditional moments of z_{t+1} and $\exp(z_{t+1})$ obtained from the Markov-chain approximation to their exact analytical counterparts. Even with a relatively coarse grid, the Markov-chain approximation closely matches the exact conditional moments.

2.2.3 Endogenous Gridpoint Method (EGM)

The third important component of the discrete-time problem concerns how to compute the optimal policy function. A direct approach is to specify a grid for the control variable $c \in \mathcal{C} = \{c_1, c_2, \ldots, c_{N_c}\}$ and, at each grid point for M, evaluate the action-value function

$$V_t(M, c) \equiv u(c) + e^{-\rho} \sum_{j=1}^{N_y} P_j \, V_{t-1}\big(R(M - c) + Y_j'\big) \,,$$

where $\{Y_j'\}$ are the possible realizations of next period's income and P_j their associated probabilities. The optimal policy $c_t(M)$ is then obtained as

$$c_t(M) = \arg\max_{c \in \mathcal{C}} V_t(M, c),$$

and the corresponding value function is given by $V_t(M) = \max_{c \in \mathcal{C}} V_t(M, c)$.

While conceptually simple, this brute-force maximization is computationally expensive, especially when the control grid is fine. Moreover, obtaining accurate derivatives of the value function—required to ensure theoretical properties such as concavity of the policy function holds—is difficult with this method.

A more efficient alternative is to solve the first-order condition directly. Given the value function $V_{t-1}(M')$, the first-order condition reads

$$u'(c_t(M)) = e^{-\rho} R \, \mathbb{E}\big[V_{t-1}'(R(M - c_t(M)) + Y')\big] \,. \tag{2.13}$$

Computing $c_t(M)$ from (2.13) for each grid point M requires a nonlinear root-finding step, which is costly to perform repeatedly for all M and all t.

The *endogenous gridpoint method* (EGM), proposed by Carroll (2006), provides an elegant and efficient solution. It eliminates the need for root-finding by inverting the first-order condition analytically and constructing a grid for next-period *assets* (or *savings*) instead of current cash-on-hand.

Step 1: Define a grid for end-of-period assets. Let the end-of-period assets be

$$a_t(M) \equiv M - c_t(M).$$

We construct a grid for assets, $\mathcal{A} = \{a_1, a_2, \ldots, a_{N_a}\}$, and treat $\{a_i\}$ as the endogenous state variable. While one can use a uniform grid for assets, where $a_j - a_{j-1} = \Delta_a$, it is often preferable to employ a non-uniform grid that is denser near the borrowing constraint, where the value and policy functions typically exhibit higher curvature.

> **Tip**
>
> A convenient choice for a non-uniform grid is the *double-exponential grid*, which clusters grid points near the lower bound. Let u^j denote a uniform grid on the unit interval $[0, 1]$.

The grid points are then given by

$$a_j = a_{\min} + (a_{\max} - a_{\min}) \frac{e^{\,e^{\alpha u^j} - 1} - 1}{e^{\,e^{\alpha} - 1} - 1},$$

where $\alpha > 0$ controls the degree of clustering. As $\alpha \to 0$, the grid becomes approximately uniform. This transformation provides fine resolution near the borrowing constraint while maintaining adequate coverage of the upper range.

Step 2: Use the first-order condition. Using the envelope condition $V'_{t-1}(M') = u'(c_{t-1}(M'))$, the Euler equation (2.13) becomes

$$u'(c_{t,i}) = e^{-\rho} R \sum_{j=1}^{N_y} P_j \, u'\big(c_{t-1}\big(Ra_i + Y'_j\big)\big)\,, \tag{2.14}$$

where the right-hand side is known given V_{t-1} (or equivalently c_{t-1}). The left-hand side depends only on $c_{t,i}$, so we can solve for it by inverting the marginal utility function:

$$c_{t,i} = u'^{-1}\left(e^{-\rho} R \sum_{j=1}^{N_y} P_j \, u'\big(c_{t-1}\big(Ra_i + Y'_j\big)\big)\right). \tag{2.15}$$

Step 3: Construct the endogenous grid. Once we have $c_{t,i}$ for each point s_i, we can compute the corresponding cash-on-hand:

$$M_{t,i} = a_i + c_{t,i}. \tag{2.16}$$

Each pair $(M_{t,i}, c_{t,i})$ defines one point on the *endogenous grid* for time t. The consumption policy $c_t(M)$ is then obtained by interpolating over these points.

Step 4: Update the value function. Given the policy function, the value function can be updated as

$$V_t(M_{t,i}) = u(c_{t,i}) + e^{-\rho} \sum_{j=1}^{N_y} P_j \, V_{t-1}\big(R(M_{t,i} - c_{t,i}) + Y'_j\big)\,. \tag{2.17}$$

The EGM's main advantage is that it converts a computationally heavy root-finding problem into a simple functional inversion, which is trivial when $u(c)$ has a known analytic form such as CRRA preferences. Because the method uses interpolation only once—after constructing the endogenous grid—it is both accurate and fast.

Important

Despite numerous extensions (see, e.g., the recent review by Fella 2025), it is important to emphasize that the EGM is not a general-purpose method for all dynamic optimization problems. It is not always possible to avoid the use of a root-finding step when solving the Euler equation. White (2015) provides formal conditions under which the method applies. Higher-dimensional problems present additional challenges, as discussed in Ludwig and Schoen (2018), since the endogenous grid becomes irregular and requires multidimensional interpolation.

Tip

There is a close connection between the EGM and continuous-time methods. In continuous time, the maximization step becomes particularly simple—often yielding closed-form expressions for the optimal control—even in settings where the application of the EGM would be computationally challenging.

2.3 Julia Implementation

We have seen how to represent the value function, how to compute expectations, and how to perform the maximization step without resorting to costly root finding. We are now ready to solve for the value and policy functions. To make the implementation modular and reusable, we encapsulate all parameters and grids in a Julia *struct*. This lets us pass the entire model cleanly between solvers, simulators, and calibration routines, and makes it easy to override the baseline calibration via keyword arguments.

```julia
Base.@kwdef struct ConsumptionSavingsDT
    γ::Float64 = 2.0        # CRRA coefficient
    ρ::Float64 = 0.05       # discount rate
    A::Float64 = 1.00       # terminal value function parameter
    R::Float64 = exp(ρ)     # interest rate
    σ::Float64 = 0.25       # standard deviation of log income
    Z::NamedTuple = tauchen(9, 0.0, σ) # income process
    Y::Vector{Float64} = exp.(Z.z)  # income levels
    N::Int64  = 11          # number of grid points
    α::Float64 = 0.0        # grid spacing parameter
    Mgrid::Vector{Float64} = make_grid(0.0, 2.5, N; α = α)
    agrid::Vector{Float64} = make_grid(0.0, 1.0, N; α = α)
end
```

Listing 2.2: Model object and default calibration

The model object (a Julia `struct`) stores parameters and grids and also defines a baseline calibration via the `@kwdef` constructor. Instantiating the model uses these defaults unless explicitly overridden by the user.

Once the model structure is defined, we can create instances using keyword arguments:

Julia REPL

```
julia> m0 = ConsumptionSavingsDT()
ConsumptionSavingsDT(...)

julia> m0.γ
2.0

julia> m1 = ConsumptionSavingsDT(γ = 1.5, ρ = 0.06)
ConsumptionSavingsDT(...)

julia> m1.γ, m1.ρ
(1.5, 0.06)
```

Listing 2.3: Parameter initialization in `ConsumptionSavingsDT`.

The model object internally calls the helper function `make_grid` to construct the grids for cash-on-hand and savings. The function returns a vector of grid points that is uniform when $\alpha = 0$ and increasingly dense near the lower bound as $\alpha > 0$ (double-exponential clustering).

```
function make_grid(zmin, zmax, Nz; α = 1.0)
    u = range(0.0, 1.0, length=Nz)
    double_exp = α == 0 ? u : @. (exp(exp(α * u) - 1.0) - 1.0) / (exp(exp(α) - 1.0) - 1.0)
    return @. zmin + (zmax - zmin) * double_exp
end
```

Listing 2.4: Grid construction helper `make_grid`

Tip

Encapsulating parameters and grids in a single `struct` promotes modularity and reproducibility. You can pass the model object directly to value-function solvers, EGM routines, and simulation code without tracking individual arguments, and you can switch calibrations by overriding only the relevant keyword parameters.

One-step value function iteration. We now solve for $V_1(M)$ and $c_1(M)$, given the terminal condition $V_0(M) = \frac{M^{1-\gamma}}{1-\gamma}$, using a single step of value function iteration. For each M on the state grid, we discretize the control space into N_c points and evaluate the action–value function, choosing the maximizer.

We first specify the admissible consumption set. The lower bound is naturally $c = 0$. For the upper bound, feasibility requires that next period's cash-on-hand be nonnegative

for *all* realizations of Y'. Let $Y_{\min}$ denote the lowest possible income realization under the (discretized) income process. Then the largest feasible consumption at M is

$$c_{\max}(M) \;=\; M + \frac{Y_{\min}}{R},$$

since choosing $c = c_{\max}(M)$ leaves $M' = R(M - c) + Y_{\min} = 0$ next period. Any $c > c_{\max}(M)$ would violate the borrowing constraint (default with positive probability).

Thus, for each M we take a uniform grid $c \in [0,\, c_{\max}(M)]$, evaluate

$$V_1(M, c) \;\equiv\; u(c) + e^{-\rho} \sum_{j=1}^{N_y} P_j \, V_0\Big(R\big(M - c\big) + Y_j\Big),$$

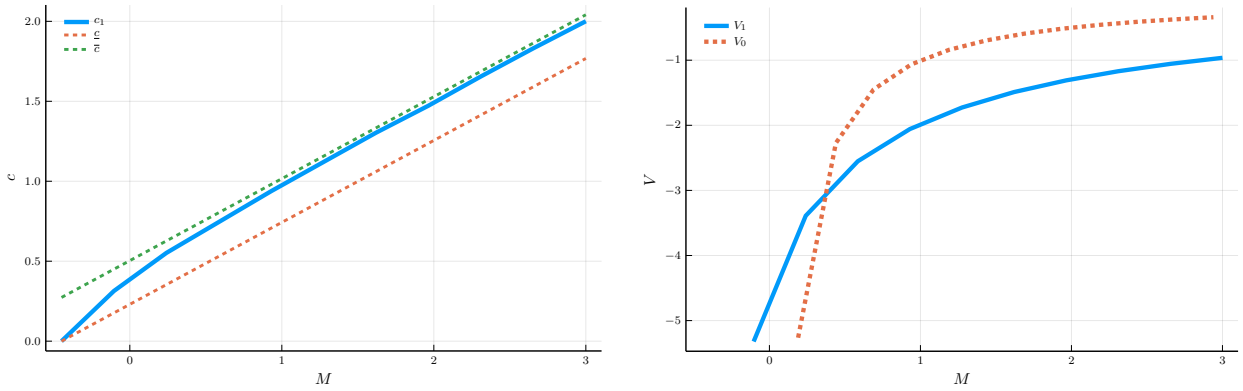
and set $V_1(M) = \max_c V_1(M, c)$ and $c_1(M) = \arg\max_c V_1(M, c)$. The listing below implements this one-step update.

```julia
function vf_iteration(m::ConsumptionSavingsDT, V0::Function;
            NM::Int64 = 11, Nc::Int64 = 101)
    (; Y, Z, R, γ, ρ) = m # unpack model parameters
    Mgrid = collect(range(-Y[1]/R, 3.0, length=NM)) # grid for M
    cgrid = [range(0.0, m + Y[1]/R, length=Nc)
            for m in Mgrid] # collection of grids for c
    # Action-value function
    V1(M, c) = c^(1-γ)/(1-γ) + exp(-ρ) * sum(Z.P[1,j] *
                V0(m,R * (M-c) + Y[j]+1e-12) for j in eachindex(Y))
    # Policy and value functions
    C = [cgrid[j][argmax([V1(Mgrid[j], c) for c in cgrid[j]])]
            for j in eachindex(Mgrid)]
    V = [V1(Mgrid[j], C[j]) for j in eachindex(Mgrid)]
    return (; C, V, Mgrid) # return a named tuple
end
```

Listing 2.5: One step of value function iteration

> **Tip**
>
> **Numerical guard.** With CRRA and $\gamma > 1$, $V_0(0)$ is $-\infty$. When evaluating $V_0(R(M - c) + Y_j)$ near zero, add a tiny ε (e.g., 10^{-12}) inside the argument or cap from below to avoid underflow.

Figure 2.2 shows the policy and value functions obtained from one step of value function iteration using the default calibration ($\gamma = 2$, $\rho = 0.05$, $\sigma_y = 0.25$). Panel (a) displays the policy function in solid blue. The function is increasing in M with a slope that decreases in M, consistent with the theoretical properties of the consumption function in this class of models (see, e.g., Carroll and Kimball 1996).



(a) Policy functions

(b) Value functions

Figure 2.2: Policy and value functions across the state grid

The figure also illustrates two analytical bounds for the policy function. The lower bound corresponds to the case in which the household receives the lowest income $Y_{\min}$ with certainty. Since there is no uncertainty, the optimal consumption function is affine:

$$\underline{c}(M) = \kappa \left[M + \frac{Y_{\min}}{R} \right], \tag{2.18}$$

where $\kappa \equiv \left[R^{\frac{1-\gamma}{\gamma}} e^{\frac{\rho}{\gamma}} + 1 \right]^{-1}$ is the *marginal propensity to consume* (MPC) under riskless income.[2]

The upper bound corresponds to a household that receives the *average* income $\overline{Y} \equiv \sum_{j=1}^{N_y} P_j Y_j$ with certainty:

$$\overline{c}(M) = \kappa \left[M + \frac{\overline{Y}}{R} \right]. \tag{2.19}$$

The fact that the computed consumption function lies below $\overline{c}(M)$ reflects *precautionary saving*: households save more than they would if income were riskless.

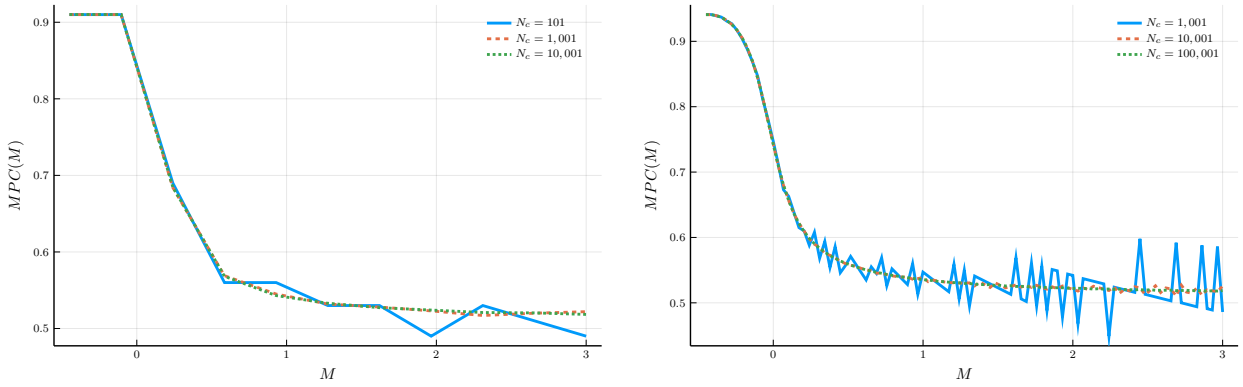
Panel (b) shows the value function (solid blue) and the terminal value function (dashed). The value function is increasing and concave in M, while the terminal value function is defined only for $M > 0$. By contrast, the one-period value function is well-defined for all $M > -Y_{\min}/R$, the natural borrowing limit implied by the solvency constraint.

Figure 2.3 plots the numerical MPCs for the baseline calibration. The MPC at each grid point M_i is approximated by

$$MPC(M_i) = \frac{c(M_{i+1}) - c(M_i)}{M_{i+1} - M_i}. \tag{2.20}$$

Panel (a) shows that for a coarse state grid ($N_M = 11$), the estimated MPCs are not strictly decreasing in M. For example, around $M = 2$, the estimated slope of $c(M)$ actually increases, contrary to theory. This reflects discretization error from using a finite grid for consumption and finite-difference approximations.

[2]This follows from solving the Euler equation $c_1(M)^{-\gamma} = e^{-\rho} R[R(M - c_1(M)) + Y]^{-\gamma}$.



(a) MPCs for $N_M = 11$

(b) MPCs for $N_M = 101$

Figure 2.3: Marginal propensities to consume for different grid sizes

Using a finer control grid ($N_c = 1001$) eliminates most of the non-monotonicity, but the MPCs is still not smooth. Panel (b) shows that increasing the number of state grid points ($N_M = 101$) can worsen the oscillations: as the state grid becomes finer, small numerical errors in $c(M)$ are magnified when computing derivatives. Even with $N_c = 10{,}001$, oscillations persist.

This example highlights a key limitation of brute-force value function iteration. While the level of the value function may be well approximated, its derivatives—and hence the implied policy slopes such as the MPC—may be numerically unstable. This motivates the use of alternative approaches, such as the *endogenous gridpoint method* (EGM), which directly iterates on the policy function and avoids the need for numerical differentiation.

Tip

Why the EGM helps. The endogenous gridpoint method eliminates the need to numerically differentiate the value function. Instead of computing $V'(M)$ using finite differences, it directly uses the analytical relationship between consumption and marginal utility from the Euler equation:

$$u'(c_t(M)) = e^{-\rho} R \, \mathbb{E}\left[V'_{t-1}(M')\right].$$

By inverting $u'(\cdot)$, we recover consumption *exactly* on an endogenous grid of points where the first-order condition holds. This approach produces smoother policy functions and more accurate marginal propensities to consume, even on coarse grids.

Solution with the endogenous gridpoint method. We now solve for the value and policy functions using the endogenous gridpoint method (EGM). A key step in applying the EGM is to define the appropriate grids for cash-on-hand and end-of-period assets. We focus on the case in which the household can borrow up to the *natural borrowing limit*—the largest debt that can still be repaid in all future states of nature. This limit depends on both the planning horizon and the income process.

For instance, when $T = 1$, end-of-period assets $a_1(M) \equiv M - c_1(M)$ must satisfy

$$a_1(M) \geq \underline{a}_1 \equiv -\frac{Y_{\min}}{R},$$

so that the household can repay its debt next period even if the lowest income $Y_{\min}$ is realized. If $T = 2$, solvency must hold for two consecutive low-income realizations, implying

$$\underline{a}_2 \equiv -\frac{Y_{\min}}{R} - \frac{Y_{\min}}{R^2}.$$

In general, for a T-period planning horizon, the natural borrowing limit is

$$\underline{a}_T \equiv -\sum_{t=1}^{T} \frac{Y_{\min}}{R^t}.$$

This definition ensures the household remains solvent under the worst possible income path.

> **Note**
>
> The natural borrowing limit expands with the horizon T, approaching $\underline{a}_\infty = -Y_{\min}/(R-1)$ as $T \to \infty$ in the stationary case. This value corresponds to the present discounted value of perpetual minimum income.

Because the borrowing limit depends on T, the corresponding grids for both savings and cash-on-hand also vary with the horizon. It is convenient to work with a shifted variable $\tilde{a}_t(M) \equiv a_t(M) - \underline{a}_t$, and to use a fixed grid for $\tilde{a}_t(M)$ across all periods. The same logic applies to cash-on-hand, $\tilde{M}_t \equiv M - \underline{M}_t$, where $\underline{M}_t = \underline{s}_t$ since $M = s + c$ and $c \geq 0$.

Listing 2.6 implements a single EGM iteration. After unpacking the model parameters, the grid for $\tilde{a}_t$ is shifted by the natural borrowing limit $\underline{a}_t$ to obtain the grid for a_t. We then use Equation (2.15) to solve for the consumption policy $c_t(M)$ at the endogenous grid points. The function returns the interpolated policy function $c_t(M)$ and the corresponding grid M_t as a named tuple.

```julia
function egm_step(m::ConsumptionSavingsDT, iter::Int, c0::Function)
    (; agrid, Z, Y, R, γ, ρ) = m # unpack model parameters
    agrid_shifted = -Y[1] * sum(R.^(-(1:iter))) .+ agrid
    # compute the consumption policy
    c1 = [sum(exp(-ρ) * Z.P[1,j] * R * c0(R * a + Y[j]+1e-12)^(-γ)
        for j in eachindex(Y))^(-1/γ) for a in agrid_shifted]
    M1 = agrid_shifted .+ c1 # compute the cash-on-hand
    return (; c = linear_interpolation(M1, c1;
        extrapolation_bc=Line()), M = M1)
end
```

Listing 2.6: Endogenous gridpoint method iteration step

Important

After obtaining the policy function at the endogenous grid points, we must interpolate to obtain a continuous function. We use the `linear_interpolation` function from the `Interpolations.jl` package, explicitly enabling extrapolation. By construction, extrapolation only occurs at the right boundary. If the upper bound of the grid is sufficiently large, the consumption function is approximately linear there, so linear extrapolation is accurate.

Since the function `egm_step` returns an interpolated policy function, we can pass it directly as input to the next iteration. Listing 2.7 implements the EGM iteration for horizons $T = 1, \ldots, 8$. We initialize the model with $N = 100$ grid points and non-uniform spacing ($\alpha = 1.5$), allocating more points near the natural borrowing limit. We start from $c_0(M) = M$, implying $V_0(M) = \frac{M^{1-\gamma}}{1-\gamma}$, and obtain $c_1(M)$ for $T = 1$. Subsequent iterations use $c_{t-1}(M)$ as input to compute $c_t(M)$, continuing until $T = 8$.

```
# Endogenous gridpoint method iteration
m = ConsumptionSavingsDT(N = 100, α = 1.5)
policies = [egm_step(m, 1, M->M)]
for i = 2:8
    push!(policies, egm_step(m, i, M->policies[i-1][1](M)))
end
```

Listing 2.7: Endogenous gridpoint method iteration

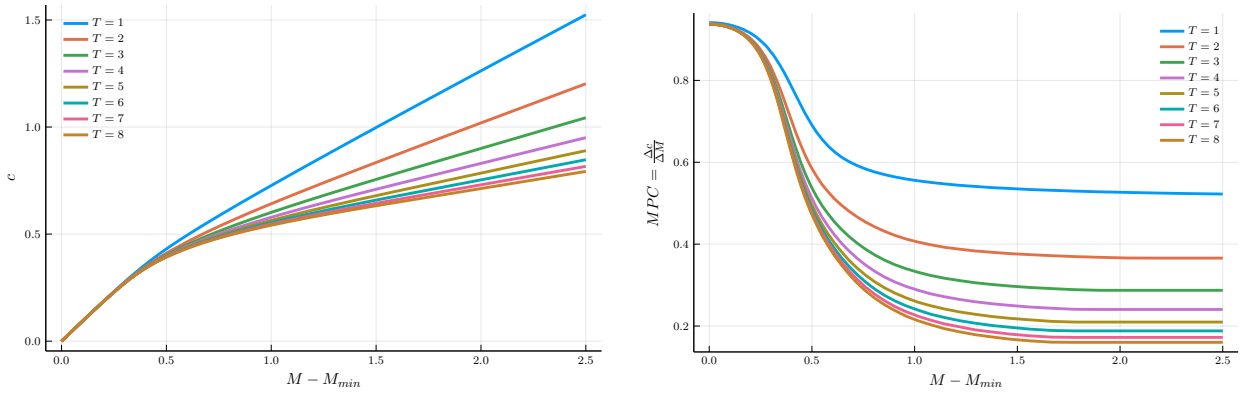
The left panel of Figure 2.4 displays the policy functions across planning horizons. The x-axis shows M shifted by the borrowing limit, $M - \underline{M}_t$. Each curve is increasing and concave, with curvature diminishing as T increases. The right panel shows the marginal propensities to consume (MPCs). The MPCs are smooth and strictly decreasing in M, approaching one near the borrowing limit and converging to the riskless MPC for wealthy households.

Note

Using a non-uniform grid is essential to obtain smooth MPCs with only $N = 100$ points. If $\alpha = 0.0$ (a uniform grid), achieving comparable smoothness would require nearly ten times as many points.

The derivatives are computed using the `fd_derivative` function, which applies a finite-difference approximation implemented via the `Interpolations.jl` package.

```
# Compute finite difference derivative
function fd_derivative(grid, x)
    x_interp = linear_interpolation(grid, x)
    return [Interpolations.gradient(x_interp, x)[1] for x in grid]
end
```



(a) Policy functions

(b) Marginal propensities to consume

Figure 2.4: Policy functions and MPCs obtained from the EGM for different planning horizons. The EGM solution yields smooth, monotone consumption functions and strictly decreasing MPCs.

Listing 2.8: Finite-difference derivative operator

Important

Numerical efficiency of the EGM. Compared with value function iteration (VFI), iterating over the policy function with the endogenous gridpoint method (EGM) is dramatically faster and more numerically stable. Each iteration involves a single pass over the state grid, with no root-finding and no interpolation of the value function. Convergence is typically achieved in a few iterations because the policy function mapping is much smoother than the value function mapping. Moreover, since the EGM computes the policy function directly from the Euler equation, it delivers more accurate marginal propensities to consume and smoother policy functions even on relatively coarse grids.

2.4 The Challenge of High-Dimensional Problems

The discrete-time methods introduced in this chapter form the backbone of modern computational economics. Value function iteration and the endogenous gridpoint method provide robust and transparent ways to approximate dynamic optimization problems. In one- or two-dimensional settings—such as the canonical consumption–savings model—these methods deliver accurate solutions at modest computational cost. However, extending them to higher-dimensional problems introduces severe computational challenges.

2.4.1 The Three Curses of Dimensionality

In Chapter 1, we introduced the idea that high-dimensional dynamic problems face three fundamental computational obstacles: the curses of *representation*, *optimization*, and *expectation*. We now formalize each of them and illustrate why the cost of traditional solution

methods explodes as dimensionality grows.

(i) Curse of representation. To approximate the value function, we discretize the state space and represent it on a grid. Let the value function depend on a vector of state variables $\mathbf{s} \in \mathcal{S} \subset \mathbb{R}^n$. If each dimension is discretized with S grid points, the total number of nodes is $|\mathcal{S}| = S^n$. This exponential growth quickly becomes infeasible as n increases.

It is easy to construct examples where this representation becomes impossible. In a regional real business cycle model for the U.S. economy with capital and productivity for each of 50 states, the state vector has $n = 100$ elements. If each variable takes only ten possible values, the grid would contain 10^{100} points—utterly beyond any conceivable computational capability.

(ii) Curse of optimization. At each grid point, the model requires finding the optimal control $\mathbf{a} \in \mathcal{A} \subset \mathbb{R}^p$. Brute-force maximization requires evaluating the objective at all grid points in $\mathcal{A}$. With A grid points per dimension, the total number of evaluations is $|\mathcal{A}| = A^p$. Alternatively, one can solve first-order conditions via root-finding methods, but these too scale poorly with p. Even the endogenous gridpoint method, which eliminates root-finding in one-dimensional problems, cannot fully circumvent this curse when the action space is multidimensional.

(iii) Curse of expectation. Dynamic models require computing the expected continuation value $\mathbb{E}[V(\mathbf{s}')]$, which involves integrating over future shocks. Suppose the state evolves as $\mathbf{s}' = f(\mathbf{s}, \mathbf{u})$, where $\mathbf{u} \in \mathcal{U} \subset \mathbb{R}^m$ is a vector of shocks with joint density $\phi(\mathbf{u})$. Then

$$\mathbb{E}[V(\mathbf{s}') \mid \mathbf{s}] = \int_{\mathcal{U}} V(f(\mathbf{s}, \mathbf{u}))\, \phi(\mathbf{u})\, d\mathbf{u} = \int_{u_1} \int_{u_2} \cdots \int_{u_m} V(f(\mathbf{s}, \mathbf{u}))\, \phi(\mathbf{u})\, du_1 \cdots du_m.$$

In the simple case $m = 1$, this integral can be approximated by quadrature or by a Markov discretization such as Tauchen (1986). If each shock takes U possible values, the number of evaluation points grows as $|\mathcal{U}| = U^m$, which again becomes infeasible when m is large.

2.4.2 The Way Forward

The three curses of dimensionality imply that the techniques discussed in this chapter—while powerful in low-dimensional settings—become computationally infeasible as models increase in complexity. Overcoming these limitations requires new tools that exploit structure, sparsity, and functional approximation.

In the chapters that follow, we explore two complementary strategies. First, continuous-time methods reformulate dynamic programming problems as partial differential equations (PDEs), replacing high-dimensional expectations with local derivatives that can be discretized efficiently. Second, machine-learning techniques—such as neural networks and stochastic optimization—provide flexible, scalable ways to represent value and policy functions without exponential growth in complexity.

Together, these approaches form the foundation for solving modern high-dimensional models in macroeconomics and finance.

Chapter 3

Continuous-Time Methods

Continuous-time formulations provide a powerful and elegant counterpart to discrete-time dynamic programming. Instead of expectations over future states, they express optimal behavior through *differential operators*, yielding compact analytical representations and computationally efficient numerical schemes. Uncertainty evolves continuously through diffusion or jump processes, and optimal decisions are characterized by a partial differential equation—the *Hamilton–Jacobi–Bellman (HJB)* equation—that replaces the recursive Bellman operator in discrete time.

In this chapter, we revisit the consumption–savings problem from Chapter 2 to illustrate the transition from discrete to continuous time. We begin by deriving the HJB equation as the limit of the discrete-time Bellman equation when the time step Δt tends to zero. We then analyze the stationary HJB, which characterizes long-run behavior in a time-invariant environment, and show how to solve it numerically using finite-difference schemes and Chebyshev collocation methods. These techniques transform the HJB into a system of algebraic equations that can be handled with standard linear-algebra tools, while the Barles–Souganidis theorem provides conditions guaranteeing convergence to the true (viscosity) solution. The chapter concludes by revisiting the curse of dimensionality in this continuous-time context and motivating the high-dimensional approximation and machine-learning methods developed in the next chapters.

3.1 From Discrete to Continuous Time

To derive the continuous-time analog of the Bellman equation, we start from the discrete-time consumption–savings problem developed in Chapter 2 and take the limit as the time step Δt tends to zero. We also extend the baseline model along two natural dimensions: (i) the household now decides how to allocate its wealth between a safe and a risky asset; and (ii) labor income follows a persistent Markov process rather than being i.i.d. over time. These extensions illustrate how to handle environments where state variables evolve through diffusion and jump processes, and set the stage for the general continuous-time framework developed in the rest of the chapter.

Asset returns. The gross return on the riskless asset over a small interval Δt is

$$R_f = 1 + r\,\Delta t,$$

where r is the risk-free rate per unit of time. The gross return on the risky asset over the same interval is

$$R_{r,t+\Delta t} = 1 + \mu_r\,\Delta t + \sigma_r\sqrt{\Delta t}\,\varepsilon_{r,t+\Delta t}, \tag{3.1}$$

where $\varepsilon_{r,t+\Delta t}$ is an i.i.d. random variable with mean zero and unit variance.

Note

This scaling ensures that both the mean and the variance of the risky return are of order Δt. Consequently, as the time step becomes small, risk per period vanishes, but cumulative risk over any fixed horizon does not. To see this, fix a horizon $T = n\,\Delta t$ and note that

$$\mathrm{Var}\left[\sum_{k=1}^{n} R_{r,t+k\Delta t}\right] = n\,\sigma_r^2\,\Delta t = \sigma_r^2\,T.$$

Hence, as we partition the time interval into finer and finer subperiods, the total amount of risk over $[t,\, t+T]$ remains unchanged.

Labor income. Labor income takes finitely many values,

$$Y_t \in \{Y_1, \ldots, Y_{N_y}\},$$

and follows a discrete-time Markov chain with transition probabilities $P_{ij} = \Pr(Y_{t+\Delta t} = Y_j \mid Y_t = Y_i)$. To obtain a well-defined continuous-time limit, we assume that these probabilities scale linearly with the time step:

$$P_{ij} = \lambda_{ij}\,\Delta t, \qquad j \neq i, \tag{3.2}$$

and $P_{ii} = 1 - \sum_{j\neq i} P_{ij}$. The matrix $\Lambda = (\lambda_{ij})$ defines the *intensity matrix* of the continuous-time income process.

Tip

The discrete-time transition matrix can be expressed as a short-time approximation

$$P = I + \Lambda\,\Delta t,$$

where Λ is the generator of the Markov chain. In the limit $\Delta t \to 0$, the income process converges to a continuous-time jump process with exponentially distributed waiting times between transitions.

Bellman equation with time step Δt. The Bellman equation for this discrete-time formulation is

$$V_t(W, Y) = \max_{c,\alpha} \left\{ u(c)\,\Delta t + e^{-\rho \Delta t}\, \mathbb{E}[V_{t-\Delta t}(W', Y')] \right\}, \tag{3.3}$$

subject to the law of motion for wealth,

$$W' = R_{p,t+\Delta t} \left(W + (Y - c)\,\Delta t \right), \qquad R_{p,t+\Delta t} = (1-\alpha)\, R_f + \alpha\, R_{r,t+\Delta t}, \tag{3.4}$$

and a borrowing limit $W \geq \underline{W}$, given a transition matrix $\Pr(Y' = Y_j \mid Y = Y_i) = P_{ij}$.

> **Note**
>
> When deriving the continuous-time limit, it is essential to distinguish between *flow* and *stock* variables. Flow variables—such as consumption and labor income—represent rates per unit of time and therefore enter the discrete-time problem multiplied by Δt. Stock variables—such as wealth—are defined at a point in time and are not scaled by Δt. The cash-on-hand variable used in the previous chapter,
>
> $$M = W + Y\,\Delta t,$$
>
> combines a stock (wealth) with a flow (labor income). As $\Delta t \to 0$, this distinction vanishes, and M and W coincide in the continuous-time limit.

State dynamics in continuous time. Consider next the limit of the law of motion for the state variables as $\Delta t \to 0$. The discrete-time wealth process can be expressed as

$$W_{t+\Delta t} - W_t = [(1-\alpha_t) r W_t + \alpha_t \mu_r W_t + Y_t - c_t]\,\Delta t + \alpha_t \sigma_r W_t \sqrt{\Delta t}\, \varepsilon_{r,t+\Delta t} + o(\Delta t), \tag{3.5}$$

where $o(\Delta t)$ collects higher-order terms that vanish faster than Δt.

Taking the limit as $\Delta t \to 0$, we obtain the continuous-time wealth dynamics:

$$dW_t = [r W_t + \alpha_t(\mu_r - r) W_t + Y_t - c_t]\, dt + \alpha_t \sigma_r W_t\, dB_t, \tag{3.6}$$

where B_t is a standard Brownian motion.

Labor income evolves according to the transition matrix introduced in (3.2). The expected change in income over a small interval Δt is

$$\mathbb{E}[Y_{t+\Delta t} - Y_t \mid Y_t = Y_i] = \sum_{j \neq i} \lambda_{ij} (Y_j - Y_i)\,\Delta t. \tag{3.7}$$

In continuous time, this implies the following jump process for income:

$$dY_t = \sum_j (Y_j - Y_i)\, dN_{ij,t}, \qquad \text{with } \mathbb{E}[dN_{ij,t}] = \lambda_{ij}\, dt, \tag{3.8}$$

where $N_{ij,t}$ is a Poisson process counting jumps from state i to j with intensity λ_{ij}.

The HJB equation. We now derive the continuous-time Bellman equation by taking the limit of the discrete-time recursion (3.3). Using $e^{-\rho\Delta t} = 1 - \rho\,\Delta t + O(\Delta t^2)$, we can rewrite the Bellman equation as

$$0 = \max_{c,\alpha} \left\{ u(c) + \frac{\mathbb{E}[V_{t-\Delta t}(W', Y') - V_t(W, Y) \,|\, W, Y]}{\Delta t} - \rho\, V_t(W, Y) \right\} + O(\Delta t). \tag{3.9}$$

Taking the limit as $\Delta t \to 0$, we obtain the continuous-time Bellman equation:

$$\rho V_t(W, Y)\, dt = \max_{c,\alpha} \left\{ u(c)\, dt + \mathbb{E}[dV_t(W, Y)] \right\}. \tag{3.10}$$

To evaluate the expected change in the value function, we apply a second-order Taylor expansion around (W, Y):[1]

$$\begin{aligned} \mathbb{E}[V_{t-\Delta t}(W', Y') - V_t(W, Y)] = &-\frac{\partial V_t}{\partial t}\,\Delta t + \frac{\partial V_t}{\partial W}\left[rW_t + \alpha_t(\mu_r - r)W_t + Y_t - c_t\right]\Delta t \\ &+ \tfrac{1}{2}\frac{\partial^2 V_t}{\partial W^2}\left[\alpha_t \sigma_r W_t\right]^2 \Delta t + \sum_{j \neq i} \lambda_{ij}\left[V_t(W, Y_j) - V_t(W, Y_i)\right]\Delta t + o(\Delta t). \end{aligned} \tag{3.11}$$

The limit of the expression above as $\Delta t \to 0$ corresponds to an application of Itô's lemma. Combining this with (3.10) yields the *Hamilton–Jacobi–Bellman (HJB) equation*:

$$\rho V_t(W, Y) = \max_{c,\alpha} \left\{ u(c) - \frac{\partial V_t}{\partial t} + \mathcal{D} V_t(W, Y) + \sum_{j \neq i} \lambda_{ij}\left[V_t(W, Y_j) - V_t(W, Y_i)\right] \right\}, \tag{3.12}$$

where $\mathcal{D}$ is the *Dynkin operator* associated with the diffusion part of the process:

$$\mathcal{D}V(W, Y) = V_W\left[rW + \alpha(\mu_r - r)W + Y - c_t\right] + \tfrac{1}{2} V_{WW}\left(\alpha\sigma_r W\right)^2. \tag{3.13}$$

Note

The Dynkin operator $\mathcal{D}$ summarizes the contribution of the *drift* and *diffusion* terms to the expected change in the value function. The first term represents deterministic wealth accumulation, while the second term—the "Itô correction"—captures the curvature correction arising from exposure to risk. The summation term in (3.12) adds the jump component arising from stochastic income transitions.

Tip

The presence of a borrowing limit in continuous time, requires that the drift term for wealth is non-negative, and the diffusion term for wealth is zero, at $W = \underline{W}$. This gives rise to the *state constraint*:

$$r\underline{W} + Y_i - c(\underline{W}, Y_i) \geq 0, \tag{3.14}$$

[1] Here, t indexes the planning horizon, not calendar time. This is why the derivative with respect to t appears with a negative sign, as the remaining horizon shrinks over time.

which ensures that the wealth process does not violate the borrowing limit.

The stationary HJB equation. In many applications, we are interested in the *stationary* or *time-independent* solution of the HJB equation. When the environment is deterministic and the parameters $(r, \mu_r, \sigma_r, \rho)$ are constant over time, the value function no longer depends explicitly on t:

$$\frac{\partial V_t(W,Y)}{\partial t} = 0, \qquad V_t(W,Y) \equiv V(W,Y).$$

In this case, the HJB equation (3.12) simplifies to

$$\rho V(W,Y_i) = \max_{c,\alpha} \Big\{ u(c) + \mathcal{D}V(W,Y_i) + \sum_{j \neq i} \lambda_{ij} \left[V(W,Y_j) - V(W,Y_i) \right] \Big\}. \tag{3.15}$$

The optimality condition for consumption is

$$u'(c(W,Y_i)) = V_W(W,Y_i) \quad \Rightarrow \quad c(W,Y_i) = u'^{-1}\big(V_W(W,Y_i)\big). \tag{3.16}$$

The optimal portfolio share satisfies

$$V_W(W,Y_i)(\mu_r - r)W = -V_{WW}(W,Y_i)\,(\sigma_r W)^2\,\alpha(W,Y_i) \quad \Rightarrow \quad \alpha(W,Y_i) = \frac{\mu_r - r}{\gamma(W,Y_i)\,\sigma_r^2}, \tag{3.17}$$

where $\gamma(W,Y_i) \equiv -\frac{W\,V_{WW}(W,Y_i)}{V_W(W,Y_i)}$ is the local coefficient of relative risk aversion. Because $V_{WW}(W,Y_i) < 0$, this implies that the optimal risky share is positive whenever $\mu_r > r$.

Plugging the optimality conditions back into the HJB equation yields a nonlinear partial differential equation (PDE) for $V(W,Y)$, which fully characterizes the value function in continuous time.

Important

A key advantage of continuous-time methods is that they replace the computation of expectations in the Bellman equation with derivatives that arise naturally from Itô's lemma. This transformation eliminates the need for numerical integration and will be particularly useful when we combine continuous-time methods with automatic differentiation in the following chapters.

3.2 Finite Differences

We now show how to solve the stationary HJB equation numerically using finite differences. As a warm-up exercise, we explain how to implement a finite difference scheme in the context of an option pricing problem in a Black–Scholes–Merton framework. This will provide a simpler setting where we can illustrate the main idea of the finite-difference method. We then apply the finite-difference method to solve the consumption-savings problem.

3.2.1 A Black–Scholes–Merton Example

We consider a simple example of an option pricing problem in a Black–Scholes–Merton framework. In particular, we consider an European call option on a stock with strike price K and maturity T.

The Black–Scholes–Merton PDE. The risk-free rate is constant and equal to r, and the stock price follows a geometric Brownian motion:

$$dS_t = \mu_S S_t \, dt + \sigma_S S_t \, dB_t, \tag{3.18}$$

where B_t is a standard Brownian motion.

We are interested in pricing the option using the principle of no-arbitrage. The absence of arbitrage opportunities implies the existence of a *stochastic discount factor* π_t, which evolves according to

$$d\pi_t = -r \, \pi_t \, dt - \eta \, \pi_t \, dB_t, \tag{3.19}$$

where $\eta \equiv \frac{\mu_S - r}{\sigma_S}$ is the market price of risk.

The value of the option is given by

$$V_T(S) = \mathbb{E}_0 \left[\frac{\pi_T}{\pi_0} \max(S_T - K, 0) \,\middle|\, S_0 = S \right]. \tag{3.20}$$

At maturity, the value of the option equals the call payoff:

$$V_0(S) = \max(S - K, 0).$$

The HJB equation for this problem at time $t \geq 0$ is

$$0 = \mathbb{E}_t \big[\, d\big(\pi_t \, V_{T-t}(S_t)\big) \big] \, . \tag{3.21}$$

Applying Itô's lemma to $V_{T-t}(S_t)$, and using the fact that $\eta \, \sigma_S = \mu_S - r$, yields the Black–Scholes–Merton PDE:

$$-r \, V_T(S) - \frac{\partial V_T(S)}{\partial T} + r \, S \, \frac{\partial V_T(S)}{\partial S} + \tfrac{1}{2} \sigma_S^2 S^2 \, \frac{\partial^2 V_T(S)}{\partial S^2} = 0, \tag{3.22}$$

with terminal condition $V_0(S) = \max(S - K, 0)$.

Analogously to the consumption-savings problem, the function takes time-to-maturity T rather than calendar time t as argument, which introduces the negative sign on the derivative with respect to T.

It is convenient to express the PDE in terms of the log stock price $s \equiv \log S$. Defining $v_T(s) \equiv V_T(e^s)$ gives

$$-r \, v_T(s) - \frac{\partial v_T(s)}{\partial T} + \overline{r} \, \frac{\partial v_T(s)}{\partial s} + \frac{1}{2} \sigma_S^2 \, \frac{\partial^2 v_T(s)}{\partial s^2} = 0, \tag{3.23}$$

with terminal condition $v_0(s) = \max(e^s - K, 0)$, where $\overline{r} \equiv r - \frac{1}{2}\sigma_S^2$ is the risk-adjusted drift.

Finite-difference approximations. We can solve this PDE by discretizing both time and space. Let the log-price take values on a uniform grid $s \in \{s_1, s_2, \ldots, s_N\}$, and the time-to-maturity on a grid $t \in \{t_1, t_2, \ldots, t_M\}$, with $t_1 = 0$ and $t_M = T$. We define $\Delta s = s_{i+1} - s_i$ and $\Delta t = t_{n+1} - t_n$ for all i, n, and write $v_{t_n}(s_i) = v_i^n$.

Spatial derivatives can be approximated, for example, by

$$\text{Forward: } \frac{\partial v_{t_n}(s_i)}{\partial s} = \frac{v_{i+1}^n - v_i^n}{\Delta s} + O(\Delta s), \tag{3.24}$$

$$\text{Backward: } \frac{\partial v_{t_n}(s_i)}{\partial s} = \frac{v_i^n - v_{i-1}^n}{\Delta s} + O(\Delta s), \tag{3.25}$$

$$\text{Centered: } \frac{\partial v_{t_n}(s_i)}{\partial s} = \frac{v_{i+1}^n - v_{i-1}^n}{2\Delta s} + O(\Delta s^2), \tag{3.26}$$

and the second derivative by a centered difference,

$$\frac{\partial^2 v_{t_n}(s_i)}{\partial s^2} = \frac{v_{i+1}^n - 2v_i^n + v_{i-1}^n}{\Delta s^2} + O(\Delta s^2). \tag{3.27}$$

Time derivatives are approximated by a forward Euler step,

$$\frac{\partial v_{t_n}(s_i)}{\partial T} = \frac{v_i^{n+1} - v_i^n}{\Delta t} + O(\Delta t). \tag{3.28}$$

Explicit scheme. We fix a forward difference in time and consider two versions for the spatial derivative: a forward difference and a backward difference. The forward-difference version is

$$r\, v_i^n = -\frac{v_i^{n+1} - v_i^n}{\Delta t} + \overline{r}\, \frac{v_{i+1}^n - v_i^n}{\Delta s} + \frac{\sigma_S^2}{2}\, \frac{v_{i+1}^n - 2v_i^n + v_{i-1}^n}{\Delta s^2}, \tag{3.29}$$

while the backward-difference version is

$$r\, v_i^n = -\frac{v_i^{n+1} - v_i^n}{\Delta t} + \overline{r}\, \frac{v_i^n - v_{i-1}^n}{\Delta s} + \frac{\sigma_S^2}{2}\, \frac{v_{i+1}^n - 2v_i^n + v_{i-1}^n}{\Delta s^2}. \tag{3.30}$$

Rearranging either form gives a common update rule:

$$v_i^{n+1} = -r\, \Delta t\, v_i^n + p_u\, v_{i+1}^n + p_s\, v_i^n + p_d\, v_{i-1}^n, \tag{3.31}$$

where, letting $\mathbf{1}_{\mathrm{F}}$ and $\mathbf{1}_{\mathrm{B}}$ denote the forward and backward difference indicators,

$$p_u = \frac{\overline{r}\, \Delta t}{\Delta s}\, \mathbf{1}_{\mathrm{F}} + \frac{\sigma_S^2\, \Delta t}{2\Delta s^2}, \tag{3.32}$$

$$p_s = 1 - \frac{\overline{r}\, \Delta t}{\Delta s}\, (\mathbf{1}_{\mathrm{F}} - \mathbf{1}_{\mathrm{B}}) - \frac{\sigma_S^2\, \Delta t}{\Delta s^2}, \tag{3.33}$$

$$p_d = -\frac{\overline{r}\, \Delta t}{\Delta s}\, \mathbf{1}_{\mathrm{B}} + \frac{\sigma_S^2\, \Delta t}{2\Delta s^2}. \tag{3.34}$$

Thus, using either a forward or backward difference for the spatial derivative, the update rule is given by Equation (3.31). The difference between the two versions lies only in the coefficients (p_u, p_s, p_d), which determine the numerical stability of the scheme.

> **Important**
>
> The right-hand side of Equation (3.31) can be evaluated explicitly given v_i^n. Hence this is an *explicit* (forward Euler) finite-difference method. In contrast, an *implicit* method would require solving a linear system at each step. The distinction between explicit and implicit methods is crucial for stability and convergence, as shown below.

Boundary conditions. The update rule(3.31) applies for $i = 2, \ldots, N-1$. We handle boundaries using *ghost nodes*, representing values outside the grid. At the right boundary, the option is deep in the money, and its value moves one-for-one with the underlying asset. Hence $\frac{\partial v_{t_n}(s_N)}{\partial s} = e^{s_N}$, which discretizes to

$$v_{N+1}^n = v_N^n + e^{s_N}\,\Delta s.$$

At the left boundary, the option is far out of the money, and its value is insensitive to the stock price, implying $\frac{\partial v_{t_n}(s_1)}{\partial s} = 0$, or equivalently,

$$v_0^n = v_1^n.$$

> **Note**
>
> Boundary conditions are typically classified as *Dirichlet* or *Neumann*, depending on whether they fix the value or its derivative. The conditions above are of Neumann type, as we specify the derivative of v with respect to s.

Matrix representation. Define $\mathbf{v}^n = [v_1^n, v_2^n, \ldots, v_N^n]^\top$ and the tridiagonal matrix of coefficients:

$$\mathbf{P} = \begin{pmatrix} p_s + p_d & p_u & 0 & \ldots & 0 \\ p_d & p_s & p_u & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & p_d & p_s & p_u \\ 0 & \ldots & 0 & p_d & p_s + p_u \end{pmatrix}. \tag{3.35}$$

Then the law of motion for $\mathbf{v}^n$ is

$$\mathbf{v}^{n+1} = \mathbf{P}^r\,\mathbf{v}^n + \mathbf{b}, \qquad \mathbf{P}^r \equiv \mathbf{P} - r\,\Delta t\,\mathbf{I}_N, \tag{3.36}$$

with initial condition $\mathbf{v}^0 = [v_0(s_1), v_0(s_2), \ldots, v_0(s_N)]^\top$, and adjustment vector $\mathbf{b} = [0, \ldots, 0,\ p_u\,e^{s_N}\,\Delta s]^\top$ accounting for the right Neumann boundary.

Julia implementation. We start by defining the model structure for the Black–Scholes–Merton problem in Listing 3.1.

```
@kwdef struct BlackScholesModel
    σS::Float64 = 0.20
```

```
    r::Float64 = 0.05
    K::Float64 = 1.0
    T::Float64 = 1.0
    Ns::Int64 = 150
    Nt::Int64 = 300
    sgrid::LinRange{Float64} = range(-1.5, 1.5, length=Ns)
end
```

Listing 3.1: Black–Scholes–Merton model struct

Listing 3.2 implements the finite-difference scheme.

```
function fd_scheme(m::BlackScholesModel; forward::Bool = true)
    (; σS, r, K, T, Ns, Nt, sgrid) = m # unpack model parameters
    Δs, Δt = sgrid[2] - sgrid[1], T / (Nt - 1) # spatial/time steps
    r̄  = r - 0.5 * σS^2 # risk-adjusted drift
    # Coefficients for the tridiagonal matrix
    pu = r̄  * Δt / Δs * forward + σS^2 * Δt / (2 * Δs^2)
    ps = 1 - r̄  * Δt / Δs * (2*forward-1) - σS^2 * Δt / (Δs^2)
    pd = -r̄  * Δt / Δs * (1-forward) + σS^2 * Δt / (2 * Δs^2)
    e  = zeros(Ns)
    e[1], e[end] = pd, pu # set boundary conditions
    P = Tridiagonal(pd*ones(Ns-1), ps*ones(Ns)+e, pu*ones(Ns-1))
    # Boundary conditions
    b = zeros(Ns)
    b[end] = pu * exp(sgrid[end]) * Δs
    # Initial condition
    v = @. max(0.0, exp(sgrid) - K) # terminal condition
    for n in 2:Nt
        v = (P - r * Δt * I) * v + b # update rule
    end
    return (; P, b, v)
end
```

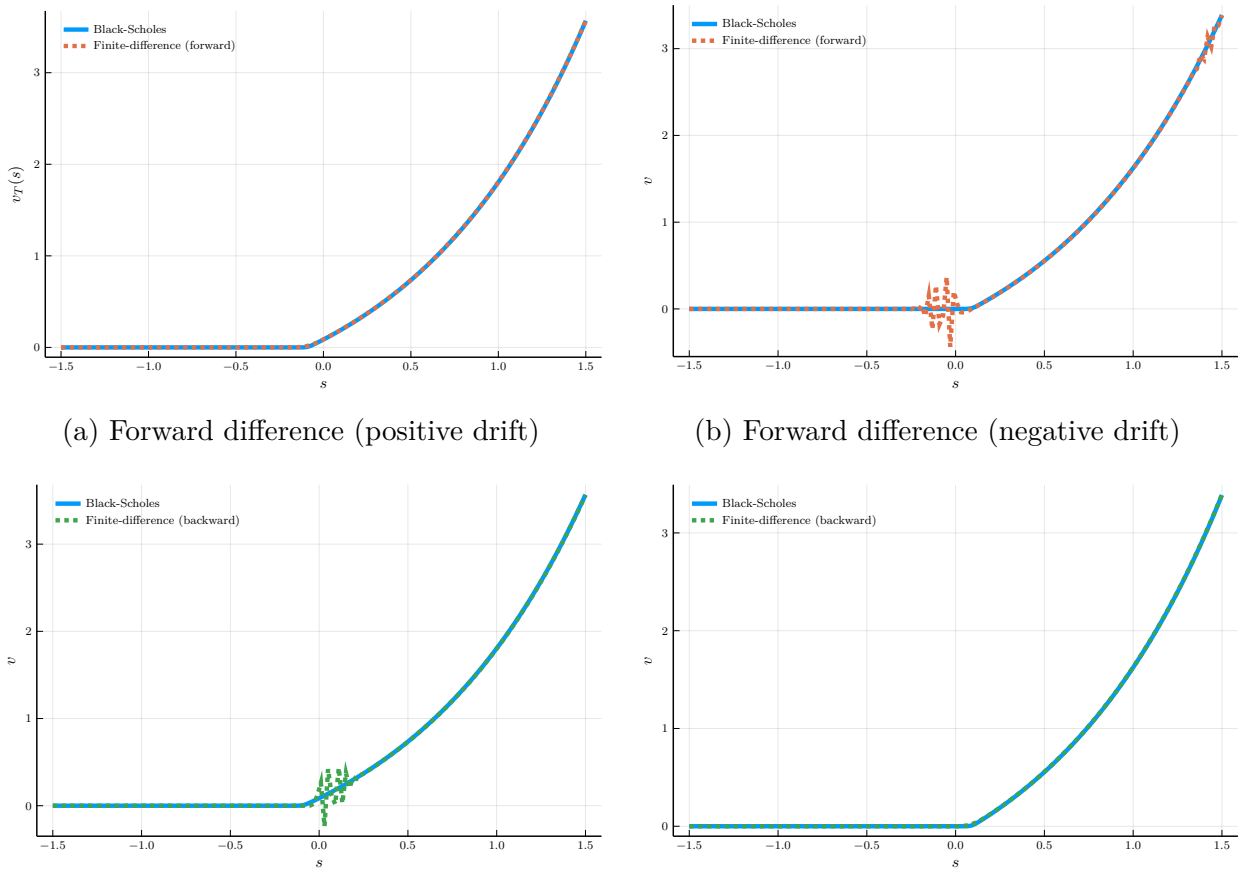
Listing 3.2: Finite-difference scheme for the Black–Scholes–Merton PDE

The code builds the tridiagonal matrix $\mathbf{P}$ using `LinearAlgebra.Tridiagonal` and iterates (3.36) to obtain prices at maturity T.

Note

Black–Scholes–Merton formula. This problem has a well-known analytical solution (Black and Scholes, 1973; Merton, 1973):

$$v_T(s) = e^s\, N(d_1) - K\, e^{-rT}\, N(d_2), \tag{3.37}$$



(a) Forward difference (positive drift)

(b) Forward difference (negative drift)

(c) Backward difference (positive drift)

(d) Backward difference (negative drift)

Figure 3.1: Finite-difference vs. analytical solutions.

Calibration: left column $r = 0.03$; right column $r = -0.03$. Common parameters: $\sigma_S = 0.01$, $T = 3.0$, $K = 1.0$, $N_t = 300$, $N_s = 150$.

where $d_1 = \dfrac{s - \log K + \left(r + \frac{1}{2}\sigma_S^2\right) T}{\sigma_S \sqrt{T}}$ and $d_2 = d_1 - \sigma_S \sqrt{T}$, with $N(\cdot)$ the standard normal CDF.

Figure 3.1 compares the finite-difference (dashed) and analytical (solid) solutions for forward and backward spatial differences under positive and negative drifts. We see that the forward (backward) difference scheme tracks the analytical solution when $\overline{r} > 0$ ($\overline{r} < 0$), but exhibits oscillations when used against the drift.

Important

Upwind scheme. The experiment above motivates the *upwind rule*: use a forward difference when the drift $\overline{r}$ is positive, and a backward difference when it is negative. This choice suppresses numerical oscillations and improves stability of the explicit scheme. To

implement upwinding in our setting, simply replace $\mathbf{1}_{\mathrm{F}}$ by $\mathbf{1}_{\{\overline{r}>0\}}$ and $\mathbf{1}_{\mathrm{B}}$ by $\mathbf{1}_{\{\overline{r}<0\}}$ in p_u, p_s, and p_d in (3.31).

A probabilistic interpretation. Why does upwinding work? A useful view comes from the Markov Chain Approximation (MCA) of Kushner and Dupuis (2001). Start from the discretized Bellman equation

$$v_T(s) = (1 - r\,\Delta t)\,\mathbb{E}[\,v_{T-\Delta t}(s')\,|\,s\,]\,, \tag{3.38}$$

under the risk–neutral log–price dynamics with drift $\overline{r}$ and variance σ_S^2:

$$\mathbb{E}[\,s' - s\mid s\,] = \overline{r}\,\Delta t, \qquad \mathrm{Var}[\,s'\mid s\,] = \sigma_S^2\,\Delta t.$$

Approximate s on the grid $\{s_i\}$ and let it move only to $\{s_{i-1}, s_i, s_{i+1}\}$ with probabilities (p_d, p_s, p_u) chosen to match the local mean and variance. A locally consistent choice is

$$p_u = \frac{\overline{r}\,\Delta t}{\Delta s}\,\mathbf{1}_{\{\overline{r}\geq 0\}} + \frac{\sigma_S^2\,\Delta t}{2\Delta s^2}, \quad p_s = 1 - \frac{|\overline{r}|\,\Delta t}{\Delta s} - \frac{\sigma_S^2\,\Delta t}{\Delta s^2}, \quad p_d = -\frac{\overline{r}\,\Delta t}{\Delta s}\,\mathbf{1}_{\{\overline{r}<0\}} + \frac{\sigma_S^2\,\Delta t}{2\Delta s^2}. \tag{3.39}$$

These satisfy $p_u + p_s + p_d = 1$ by construction. Upwinding ensures $p_u \geq 0$ and $p_d \geq 0$; nonnegativity of p_s is equivalent to the *Courant–Friedrichs–Lewy (CFL)* condition

$$1 - \frac{|\overline{r}|\,\Delta t}{\Delta s} - \frac{\sigma_S^2\,\Delta t}{\Delta s^2} \geq 0 \quad\Longleftrightarrow\quad \Delta t \leq \frac{\Delta s^2}{|\overline{r}|\,\Delta s + \sigma_S^2}. \tag{3.40}$$

Important

The CFL condition is crucial to ensure that the transition probabilities are nonnegative. Violating it produces instability and spurious oscillations in the numerical solution.

An important implication is that the time step Δt must shrink as the spatial step Δs shrinks to maintain stability. The next figure illustrates this point. Figure 3.2 reports the log root mean square error (RMSE) between the finite-difference solution and the analytical Black–Scholes–Merton formula for different combinations of (N_s, N_t). The figure shows that the log RMSE can *increase* dramatically if Δt is not small enough relative to Δs. A different discretization is required to use larger time steps when Δs is large.

Tip

We can derive the CFL condition directly for the explicit update rule (3.31), rather than for the Markov chain formulation (3.38), by interpreting the explicit scheme as an MCA with a "death" state of probability $r\,\Delta t$ (so the survival mass is $1 - r\,\Delta t$). In this case,

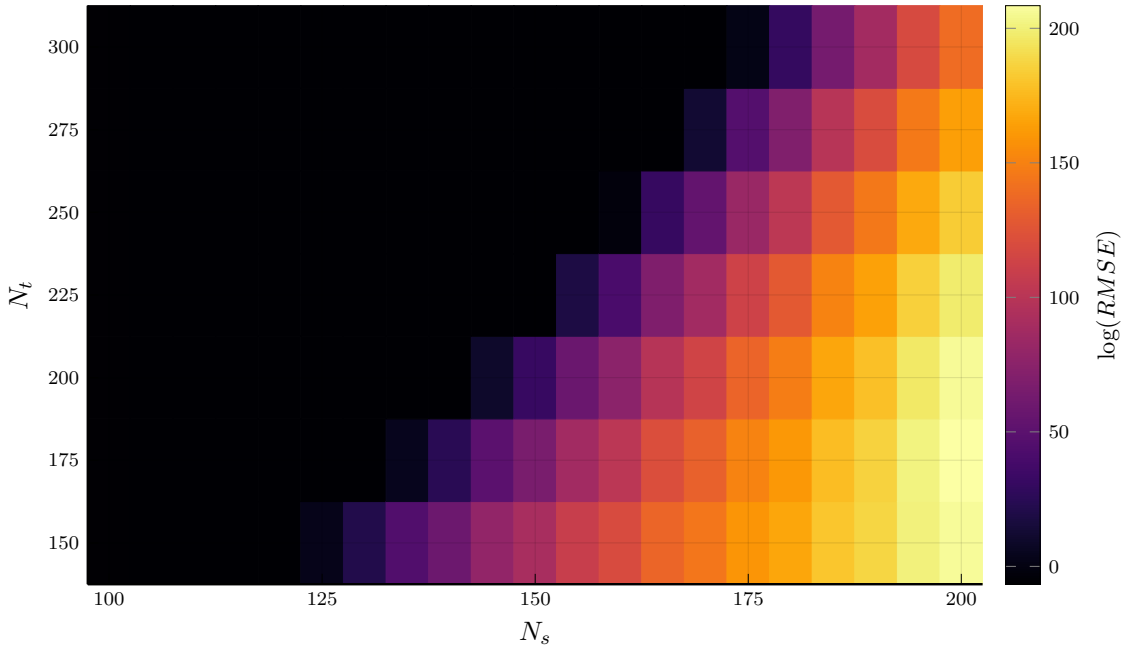


Figure 3.2: Discretization error heatmap (log RMSE) over grid sizes N_s and N_t.

the nonnegativity requirement becomes

$$1 - r\,\Delta t - \frac{|\overline{r}|\,\Delta t}{\Delta s} - \frac{\sigma_S^2\,\Delta t}{\Delta s^2} \geq 0 \quad \Longleftrightarrow \quad \Delta t \;\leq\; \frac{1}{r + \dfrac{|\overline{r}|}{\Delta s} + \dfrac{\sigma_S^2}{\Delta s^2}}. \tag{3.41}$$

The discussion above highlights the importance of upwinding and satisfying the CFL condition. The MCA approach provides a probabilistic interpretation of the finite-difference scheme and a direct route to the CFL bound: ensure the Markov chain's transition probabilities are nonnegative. We discuss next an alternative route to derive stability conditions.

Note

Upwinding, combined with the CFL condition, is *sufficient* to ensure nonnegativity of the transition probabilities and monotonicity. It is not always necessary: when diffusion dominates advection (i.e., the volatility term dominates the drift term), both forward and backward first–derivative stencils can be stable. Upwinding becomes critical in advection-dominated regimes.

3.2.2 The Barles–Souganidis Conditions for Convergence

The analysis above provided a probabilistic intuition for the stability of the finite-difference scheme. A complementary and more general approach is given by the Barles and Souganidis (1991) theorem, which provides sufficient conditions for the convergence of numerical schemes

for nonlinear partial differential equations (PDEs) such as the Hamilton–Jacobi–Bellman equation.

The main result. Consider a numerical scheme of the general form

$$F^h(x, V(x), V(\cdot)) = 0, \tag{3.42}$$

where h collects the discretization parameters (e.g., Δt, Δs), and V is the numerical approximation to the value function. Barles and Souganidis (1991) show that if the scheme satisfies three key properties, then V converges uniformly to the *viscosity solution* of the continuous-time HJB equation as $h \to 0$.

Important

Barles–Souganidis theorem. Suppose that the continuous-time problem admits a unique bounded viscosity solution. If a numerical scheme is:

1. **Monotone:** The numerical operator F^h is non-decreasing in V.
2. **Stable:** The sequence of numerical approximations V^h remains uniformly bounded.
3. **Consistent:** As $h \to 0$, the discrete operator F^h converges to the continuous operator F defining the PDE.

Then the scheme converges locally uniformly to the viscosity solution of the HJB equation.

Interpretation. The monotonicity condition ensures that the discrete scheme preserves the comparison principle of the continuous problem: if $V_1 \leq V_2$ initially, then this ordering is preserved under iteration. Stability guarantees that the numerical solution does not explode as we refine the grid. Consistency ensures that the discretized equations approximate the true differential operator as $\Delta s, \Delta t \to 0$.

Connection to the explicit upwind scheme. Consider our explicit upwind scheme:

$$v_i^{n+1} = p_u^r \, v_{i+1}^n + p_s^r \, v_i^n + p_d^r \, v_{i-1}^n + b_i^n,$$

with coefficients $p_u^r = p_u, p_s^r = p_s - r\,\Delta t, p_d^r = p_d$. The scheme is therefore:

- **Monotone** if $p_u^r, p_s^r, p_d^r \geq 0$ — that is, when upwinding and the CFL condition hold (3.41);
- **Stable** if $\|v^n\|$ remains bounded, which again follows from $p_u^r + p_s^r + p_d^r = 1 - r\,\Delta t \leq 1$ and nonnegativity;
- **Consistent** because the finite-difference approximations of first and second derivatives converge to their continuous counterparts as $\Delta s, \Delta t \to 0$.

Tip

The Barles–Souganidis theorem thus provides a powerful formal justification for the combination of upwinding and CFL conditions. When these hold, the finite-difference scheme converges to the viscosity solution of the continuous-time HJB equation. A viscosity solution is a generalized notion of solution to nonlinear PDEs like the HJB equation, which applies to problems where the value function may exhibit kinks or corners, as is typical in dynamic optimization problems with borrowing constraints or nonconvexities.

Note

Intuitively, the Barles–Souganidis result tells us that convergence of a numerical scheme does not depend on its algebraic form but on its qualitative properties. Monotonicity guarantees that the discrete operator respects the ordering of the true solution, stability prevents numerical blow-up, and consistency ensures that the discrete dynamics mimic the continuous generator as the grid is refined. When these three conditions hold, the numerical approximation inherits the comparison principle of the HJB equation and therefore converges to the economically meaningful (viscosity) solution.

Implicit schemes. We will use the Barles–Souganidis theorem to analyze the stability of *implicit* schemes. The implicit version of the discretization of the Black–Scholes–Merton PDE is

$$r\, v_i^{n+1} = -\frac{v_i^{n+1} - v_i^n}{\Delta t} + \overline{r}\left[\mathbf{1}_{\{\overline{r}\geq 0\}}\,\frac{v_{i+1}^{n+1} - v_i^{n+1}}{\Delta s} + \mathbf{1}_{\{\overline{r}<0\}}\,\frac{v_i^{n+1} - v_{i-1}^{n+1}}{\Delta s}\right] + \frac{\sigma_S^2}{2}\,\frac{v_{i+1}^{n+1} - 2v_i^{n+1} + v_{i-1}^{n+1}}{\Delta s^2}, \tag{3.43}$$

In contrast to the explicit scheme, we evaluate v and its spatial derivatives at the new time step $n+1$ instead of n. We can rewrite the equation above as

$$v_i^{n+1} = v_i^n + p_u^r\, v_{i+1}^{n+1} - \delta^r\, v_i^{n+1} + p_d^r\, v_{i-1}^{n+1}, \tag{3.44}$$

where $\delta^r \equiv p_u^r + p_d^r + r\,\Delta t$.

In matrix form, we have

$$\mathbf{A}\mathbf{v}^{n+1} = \mathbf{v}^n + \mathbf{b}, \tag{3.45}$$

where $\mathbf{A}$ is a tridiagonal matrix of coefficients:

$$\mathbf{A} = \begin{pmatrix} 1+\delta^r - p_d^r & -p_u^r & 0 & \dots & 0 \\ -p_d^r & 1+\delta^r & -p_u^r & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & -p_d^r & 1+\delta^r & -p_u^r \\ 0 & \dots & 0 & -p_d^r & 1+\delta^r - p_u^r \end{pmatrix}. \tag{3.46}$$

Let's check that the scheme is monotone. Monotonicity requires that, if $\mathbf{v}^n$ and $\mathbf{u}^n$ satisfy the recursion (3.45), and $\mathbf{v}^n \geq \mathbf{u}^n$, then $\mathbf{v}^{n+1} \geq \mathbf{u}^{n+1}$. This is equivalent to saying that

$\mathbf{A}$ is a *M-matrix*. A sufficient condition for this is that the off-diagonal elements of $\mathbf{A}$ are non-positive and the diagonal is strictly dominant: $\mathbf{A}_{ii} > \sum_{j \neq i} |\mathbf{A}_{ij}|$. Upwinding ensures that the off-diagonal elements, p_u^r and p_d^r, are non-positive. Strict diagonal dominance requires:

$$1 + \delta^r > |p_d^r| + |p_u^r| \iff 1 + r\Delta t > 0, \tag{3.47}$$

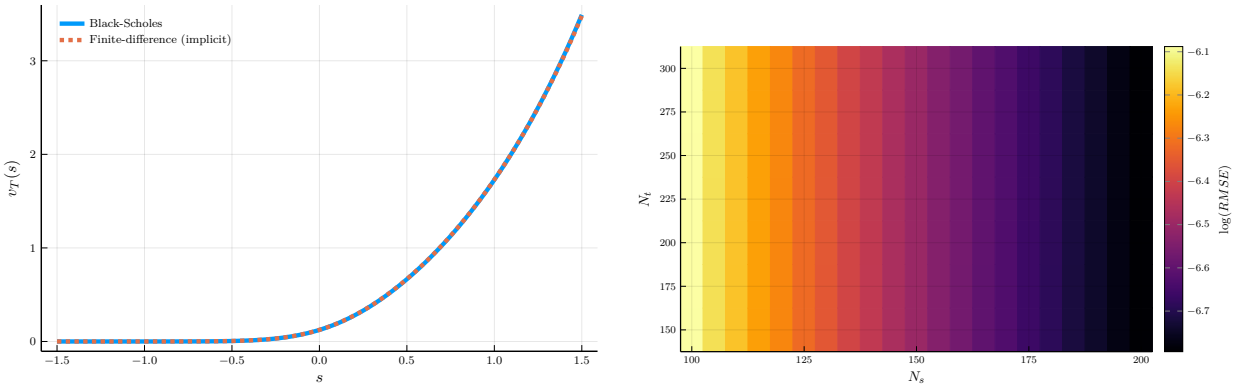
which holds for *any* $\Delta t > 0$ provided $r > 0$.

> **Important**
>
> Condition (3.47) plays the same role as the CFL condition in ensuring monotonicity and stability, but unlike in the explicit scheme, it imposes no restriction on Δt. In contrast, the explicit scheme requires Δt to be sufficiently small for the scheme to be stable. This property is often referred to as *unconditional stability*. The trade-off is computational: each time step requires solving the linear system (3.45), but the gain in stability usually outweighs this cost in diffusion-dominated problems.

Julia implementation. Listing 3.3 shows the implementation of the implicit finite-difference scheme for the Black–Scholes–Merton PDE.

```julia
function fd_implicit(m::BlackScholesModel)
    (; σS, r, K, T, Ns, Nt, sgrid) = m # unpack model parameters
    Δs, Δt = sgrid[2] - sgrid[1], T / (Nt - 1) # spatial/time steps
    r̄  = r - 0.5 * σS^2 # risk-adjusted drift
    # Coefficients for the tridiagonal matrix
    forward = r̄  > 0 ? 1 : 0
    pu = r̄  * Δt / Δs * forward + σS^2 * Δt / (2 * Δs^2)
    ps = 1 - r̄  * Δt / Δs * (2*forward-1) - σS^2 * Δt / (Δs^2)
    pd = -r̄  * Δt / Δs * (1-forward) + σS^2 * Δt / (2 * Δs^2)
    e  = zeros(Ns)
    e[1], e[end] = pd, pu # set boundary conditions
    P = Tridiagonal(pd*ones(Ns-1), ps*ones(Ns)+e, pu*ones(Ns-1))
    A = (2+r * Δt) * I - P
    # Boundary conditions
    b = zeros(Ns)
    b[end] = pu * exp(sgrid[end]) * Δs
    # Initial condition
    v = @. max(0.0, exp(sgrid) - K) # terminal condition
    for n in 2:Nt
        v = A \ (v + b) # update rule
    end
    return (; P, b, v)
end
```

Listing 3.3: Finite-difference implicit scheme for the Black–Scholes–Merton PDE



(a) Solution from the implicit scheme. (b) Error heatmap (log RMSE) across grid sizes.

Figure 3.3: Performance of the implicit finite-difference scheme for the Black–Scholes–Merton PDE.

Calibration: $\sigma_S = 0.30$, $r = 0.01$, $T = 1.0$, $K = 1.0$. The left panel uses $N_t = 25$ and $N_s = 100$. The right panel confirms unconditional stability: increasing N_s reduces the error even for small N_t.

The implementation follows closely the structure of the code for the explicit scheme. The main difference is that, after assembling the tridiagonal matrix $\mathbf{A}$, we solve the linear system (3.45), $\mathbf{A}\,\mathbf{v}^{n+1} = \mathbf{v}^n + \mathbf{b}$, for $\mathbf{v}^{n+1}$ using the backslash operator `A \ (v + b)`.

The left panel of Figure 3.3 shows the solution of the Black–Scholes–Merton PDE using the implicit scheme. The solution is highly accurate, with a discretization error of less than 10^{-6} for $N_t = 25$ and $N_s = 100$. The solution would diverge with such a small number of time steps for the explicit scheme.

The right panel of Figure 3.3 shows the error heatmap for the implicit scheme. The error heatmap shows that the implicit scheme is stable for a wide range of grid sizes. In contrast to the explicit scheme, increasing the number of grid points in the spatial dimension, for a fixed number of time steps, now *decreases* the discretization error, while we saw this could cause the explicit scheme to diverge.

3.3 Income Fluctuations in Continuous Time

We now solve the income–fluctuations problem from Section 3.1 using finite differences. Relative to the Black–Scholes–Merton example, we must handle (i) the optimal consumption choice and (ii) the borrowing constraint. For simplicity, we consider the case without portfolio choice, $\mu_r = r$ and $\sigma_r = 0$, and a two-state income process, $N_y = 2$, following Achdou et al. (2022). We solve the HJB *backward* in time, discretizing both wealth and time.

3.3.1 Finite-Difference Scheme

The HJB equation. Let $\lambda_{12} = \lambda_1$, $\lambda_{21} = \lambda_2$, and denote $V_{j,t}(W) \equiv V_t(W, Y_j)$ for $j \in \{1, 2\}$. The HJB reads

$$\rho\, V_{j,t}(W) \;=\; u\big(c_{j,t}(W)\big) - \frac{\partial V_{j,t}}{\partial t}(W) + \frac{\partial V_{j,t}}{\partial W}(W)\,\big[rW + Y_j - c_{j,t}(W)\big] + \lambda_j\big[V_{-j,t}(W) - V_{j,t}(W)\big], \tag{3.48}$$

where $c_{j,t}(W) = u'^{-1}\big(V'_{j,t}(W)\big)$ and $\lambda_j \in \{\lambda_1, \lambda_2\}$ is the outgoing intensity from state j.

Let wealth lie on a uniform grid $W \in \{W_1, \dots, W_N\}$ with $W_1 = \underline{W}$, $W_N = W_{\max}$, and time-to-horizon on $t \in \{t_1, \dots, t_M\}$ with $t_1 = 0$, $t_M = T$. Write $\Delta W = W_{i+1} - W_i$, $\Delta t = t_{n+1} - t_n$, and $V_{j,t_n}(W_i) = v^n_{i,j}$.

Following Achdou et al. (2022), we adopt an upwind *semi-implicit* finite-difference scheme. Because there is no diffusion term, upwinding is crucial for stability.

Upwinding and the state constraint. Define forward and backward differences for the wealth derivative:

$$v^n_{i,j,F} \;=\; \frac{v^n_{i+1,j} - v^n_{i,j}}{\Delta W}, \qquad v^n_{i,j,B} \;=\; \frac{v^n_{i,j} - v^n_{i-1,j}}{\Delta W}. \tag{3.49}$$

The corresponding wealth drifts are

$$\mu^n_{i,j,F} \;=\; rW_i + Y_j - u'^{-1}\big(v^n_{i,j,F}\big), \qquad \mu^n_{i,j,B} \;=\; rW_i + Y_j - u'^{-1}\big(v^n_{i,j,B}\big). \tag{3.50}$$

Thus $\mu^n_{i,j,\cdot}$ represents $\dot W = rW + Y_j - c$ evaluated with the local forward/backward slope.

We now define the upwinded derivative used in the drift:

$$v^n_{i,j,W} \;=\; v^n_{i,j,F}\,\mathbf{1}_{\{\mu^n_{i,j,F} > 0\}} \;+\; v^n_{i,j,B}\,\mathbf{1}_{\{\mu^n_{i,j,B} < 0\}} \;+\; \overline{v}_{i,j}\,\mathbf{1}_{\{\mu^n_{i,j,F} \le 0 \le \mu^n_{i,j,B}\}}, \tag{3.51}$$

where $\overline{v}_{i,j} \equiv u'(rW_i + Y_j)$ enforces $\dot W = 0$ when the forward slope implies borrowing (negative savings) while the backward slope implies saving.

To impose the no-borrowing condition at the lower boundary $W = \underline{W}$, set

$$v^n_{1,j,B} \;=\; u'(rW_1 + Y_j),$$

which enforces $\dot W \ge 0$ at W_1. Similarly, at the upper boundary we may set $v^n_{N,j,F} = u'(rW_N + Y_j)$ to prevent spurious outflow.

Semi-implicit upwind scheme. Let $c^n_{i,j} \equiv u'^{-1}\big(v^n_{i,j,W}\big)$, $u^n_{i,j} \equiv u(c^n_{i,j})$, and denote $[x]^+ = \max\{x, 0\}$, $[x]^- = \min\{x, 0\}$. The semi-implicit scheme is

$$\rho\, v^{n+1}_{i,j} \;=\; u^n_{i,j} - \frac{v^{n+1}_{i,j} - v^n_{i,j}}{\Delta t} + \frac{v^{n+1}_{i+1,j} - v^{n+1}_{i,j}}{\Delta W}\,[\mu^n_{i,j,F}]^+ + \frac{v^{n+1}_{i,j} - v^{n+1}_{i-1,j}}{\Delta W}\,[\mu^n_{i,j,B}]^- + \lambda_j\big(v^{n+1}_{i,-j} - v^{n+1}_{i,j}\big)\,. \tag{3.52}$$

Rearranging (3.52) yields, for interior $i = 2, \dots, N-1$,

$$-\ell^n_{i,j}\, v^{n+1}_{i-1,j} \;+\; s^n_{i,j}\, v^{n+1}_{i,j} \;-\; r^n_{i,j}\, v^{n+1}_{i+1,j} \;-\; \lambda_j\, v^{n+1}_{i,-j} \;=\; \frac{1}{\Delta t}\, v^n_{i,j} \;+\; u^n_{i,j}, \tag{3.53}$$

with nonnegative coefficients

$$\ell_{i,j}^n \equiv -\frac{[\mu_{i,j,B}^n]^-}{\Delta W} \geq 0, \tag{3.54}$$

$$r_{i,j}^n \equiv \frac{[\mu_{i,j,F}^n]^+}{\Delta W} \geq 0, \tag{3.55}$$

$$s_{i,j}^n \equiv \frac{1}{\Delta t} + \ell_{i,j}^n + r_{i,j}^n + \rho + \lambda_j. \tag{3.56}$$

By construction, $\ell_{1,j}^n = r_{N,j}^n = 0$, so ghost nodes are never used.

Block matrix form. Stack the two income states as

$$\mathbf{v}_j^n = \left(v_{1,j}^n, \ldots, v_{N,j}^n\right)^\top, \quad \mathbf{v}^n = \left((\mathbf{v}_1^n)^\top, (\mathbf{v}_2^n)^\top\right)^\top, \quad \mathbf{u}^n = \left(u(c_{1,1}^n), \ldots, u(c_{N,2}^n)\right)^\top.$$

Then

$$\mathbf{A}^n \mathbf{v}^{n+1} = \frac{1}{\Delta t}\mathbf{v}^n + \mathbf{u}^n, \tag{3.57}$$

with the $2N \times 2N$ block matrix

$$\mathbf{A}^n = \begin{pmatrix} \mathbf{A}_1^n & -\lambda_1 \mathbf{I}_N \\ -\lambda_2 \mathbf{I}_N & \mathbf{A}_2^n \end{pmatrix},$$

where each $\mathbf{A}_j^n$ is tridiagonal with

$$(\mathbf{A}_j^n)_{i,i-1} = -\ell_{i,j}^n \leq 0, \quad (\mathbf{A}_j^n)_{i,i} = s_{i,j}^n, \quad (\mathbf{A}_j^n)_{i,i+1} = -r_{i,j}^n \leq 0.$$

> **Important**
>
> **Unconditional stability.** Off-diagonals of $\mathbf{A}^n$ are nonpositive and, for each row, $(\text{diag}) - \sum_{\text{offdiag}} |\cdot| = \left[\frac{1}{\Delta t} + \rho + \lambda_j + \ell + r\right] - (\ell + r + \lambda_j) = \frac{1}{\Delta t} + \rho > 0$. Thus $\mathbf{A}^n$ is an M-matrix, the scheme is monotone and (by Barles–Souganidis) convergent, and it is *unconditionally stable* (no CFL restriction on Δt).

Stationary solution. As the scheme is unconditionally stable, we can take arbitrarily large time steps, and the solution will still be stable. In particular, when interested in the stationary solution, it is convenient to take the limit as $\Delta t \to \infty$.

3.3.2 Julia Implementation

Listing 3.4 shows the Julia `struct` that stores the model parameters.

```
@kwdef struct IncomeFluctuationsModel
    ρ::Float64 = 0.05
    r::Float64 = 0.03
    γ::Float64 = 2.0
```

```
    Y::Vector{Float64} = [0.1, 0.2]
    λ::Vector{Float64} = [0.02, 0.03]
    Wmin::Float64 = -0.02
    Wmax::Float64 = 2.0
    N::Int64 = 500
    Wgrid::LinRange{Float64} = range(Wmin, Wmax, length = N)
end
```

Listing 3.4: Income fluctuations model `struct`

Listing 3.5 presents the finite-difference implementation of the semi-implicit upwind scheme for the income-fluctuations problem.

```
function fd_implicit(m::IncomeFluctuationsModel; Δt::Float64 = Inf,
    tol::Float64 = 1e-6, max_iter::Int64 = 100, print_residual::Bool
↪   = true)
    (; ρ, r, γ, Y, λ, Wgrid, Wmin, Wmax, N) = m # unpack parameters
    ΔW = Wgrid[2] - Wgrid[1]
    # Initial guess
    v = [1/ρ * (y + r * w)^(1-γ) / (1-γ) for w in Wgrid, y in Y]
    c, vW, residual = similar(v), similar(v), 0.0 # pre-allocation
    for i = 1:max_iter
        # Compute derivatives
        Dv = (v[2:end,:] - v[1:end-1,:]) / ΔW
        vB = [(@. (r * Wmin + Y)^(-γ))'; Dv] # backward difference
        vF = [Dv; (@. (r * Wmax + Y)^(-γ))'] # forward difference
        v̄  = (r * Wgrid .+ Y').^(-γ) # zero-savings case
        μB = r * Wgrid .+ Y' - vB.^(-1/γ) # backward drift
        μF = r * Wgrid .+ Y' - vF.^(-1/γ) # forward drift
        vW = ifelse.(μF .> 0.0, vF, ifelse.(μB .< 0.0, vB, v̄ ))
        # Assemble matrix
        c  = vW.^(-1/γ)          # consumption
        u  = c.^(1-γ) / (1-γ)    # utility
        L = -min.(μB, 0) / ΔW    # subdiagonal
        R = max.(μF, 0) / ΔW     # superdiagonal
        S = @. 1/Δt + L + R + ρ + λ' # diagonal
        Aj = [Tridiagonal(-L[2:end,j], S[:,j], -R[1:end-1,j])
                for j in eachindex(Y)] # tridiagonal matrices
        A = [sparse(Aj[1]) -λ[1] * I
            -λ[2] * I sparse(Aj[2])] # block matrix
        # Update
        vp = A \ (u + v/Δt)[:]
        residual = sqrt(mean((vp - v[:]).^2))
        v = reshape(vp, N, length(Y)) # reshape vector to matrix
        if print_residual println("Iteration $i, residual =
↪   $residual") end
```

```
        if residual < tol
            break
        end
    end
    s = r * Wgrid .+ Y' - c   # savings
    return (; v, vW, c, s, residual) # return solution
end
```

Listing 3.5: Finite-difference semi-implicit scheme for the income-fluctuations problem

Notice that the function `fd_implicit` has the same name as the one used in the Black–Scholes–Merton example. Julia allows overloading functions with the same name provided their argument types differ. This is an example of *multiple dispatch*, a core feature of Julia that promotes code reuse, modularity, and clarity.

The code in Listing 3.5 starts by unpacking the model parameters and initializing the value function v with a simple guess where consumption equals labor plus interest income. The value function `v` is stored as a matrix of size $N \times 2$, where each column corresponds to one of the two income states—the first column to the low-income state and the second to the high-income state.

The algorithm iterates until convergence, defined as the root-mean-square difference between successive iterations falling below a tolerance level. Because the linear system solver in Julia expects a vector, the value function `v` is flattened using the `[:]` operator before solving, and reshaped back into an $N \times 2$ matrix afterward. The solution is obtained by repeatedly solving the semi-implicit update equation until the fixed point of the HJB operator is reached.

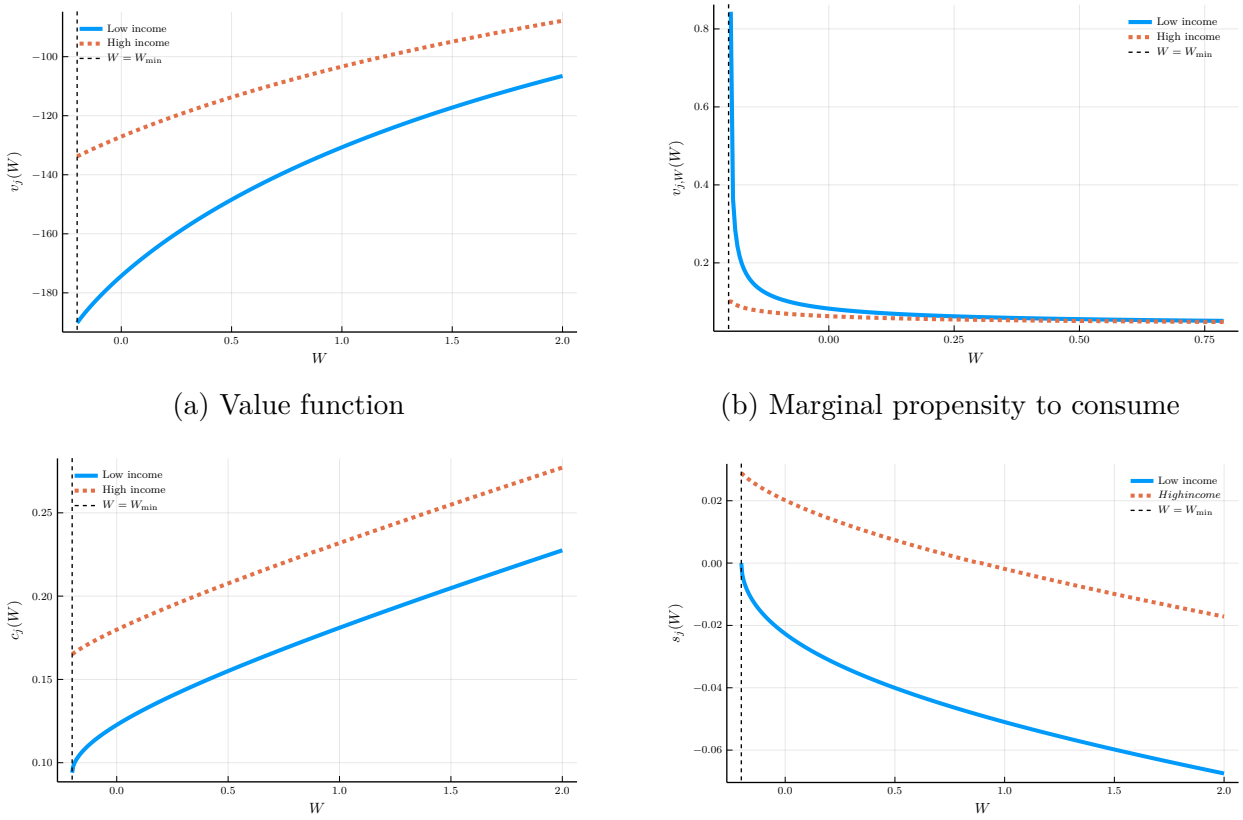
Figure 3.4 displays the solution of the income-fluctuations problem. Panel (a) shows the value function for the two income states, with the vertical line marking the borrowing limit $W = \underline{W}$. Panel (b) reports the marginal propensity to consume (MPC), which declines monotonically with wealth. Panel (c) shows the concave consumption policy function, consistent with the discrete-time version of the model in Chapter 2. Panel (d) plots the savings policy. In the low-income state, households borrow up to the constraint; in the high-income state, they exhibit a *target wealth* behavior—saving when wealth is below the target and dissaving when wealth is above it. These patterns are derived analytically by Achdou et al. (2022) and confirmed by our numerical solution.

3.3.3 Finite Differences as Policy Function Iteration

We have seen that the finite-difference method can be interpreted as an application of the Markov Chain Approximation (MCA) method of Kushner and Dupuis (2001). This reduces the problem to a discrete-time dynamic programming problem with Bellman equation

$$\mathbf{v} = \max_{\mathbf{c}} \left\{ u(\mathbf{c})\,\Delta t + (1 - \rho\Delta t)\,\mathbf{P}(\mathbf{c})\,\mathbf{v} \right\} \;\equiv\; \mathbf{T}\mathbf{v}, \tag{3.58}$$

where the right-hand side defines the operator $\mathbf{T}$ that maps an initial value function into an updated one. The stationary solution is the fixed point of $\mathbf{T}$, that is, $\mathbf{v} = \mathbf{T}\mathbf{v}$. In this



(a) Value function

(b) Marginal propensity to consume

(c) Consumption policy

(d) Savings policy

Figure 3.4: Income-fluctuations model: value, marginal value, consumption, and savings.

formulation, the state takes values on the wealth grid $W \in \{W_1, \ldots, W_N\}$. The control $\mathbf{c}$ affects the *transition probabilities* of the Markov chain. For instance, under upwinding, wealth can move up or stay the same if the drift is positive, and move down or stay the same if the drift is negative. As seen in Section 3.2.1, these transition probabilities depend on the drift of wealth, which in turn depends on the control.

One can solve this problem using any method for discrete-time dynamic programming. As pointed out by Phelan and Eslami (2022), the (semi-implicit) finite-difference scheme of Achdou et al. (2022) is closely related to *policy function iteration* (PFI), summarized here in Algorithm 2.

Algorithm 2: Policy Function Iteration (PFI)

Input: Initial policy $\mathbf{c}^{(0)}$, tolerance `tol`
Output: Value $\mathbf{v}$, policy $\mathbf{c}$

initialize $n \leftarrow 0$
repeat
  Policy evaluation: Solve $[I - (1 - \rho\Delta t)P(\mathbf{c}^{(n)})]\mathbf{v}^{(n+1)} = \Delta t\, u(\mathbf{c}^{(n)})$.;
  Policy improvement: $\mathbf{c}^{(n+1)} \leftarrow \arg\max_{\mathbf{c}}\{\Delta t\, u(\mathbf{c}) + (1 - \rho\Delta t)P(\mathbf{c})\mathbf{v}^{(n)}\}$.;
  $n \leftarrow n + 1$.;
until $\|\mathbf{c}^{(n+1)} - \mathbf{c}^{(n)}\| < \mathtt{tol}$;
return $\mathbf{v}^{(n)}$, $\mathbf{c}^{(n)}$

Policy function iteration consists of two alternating steps: (i) a *policy evaluation step*, where we compute the value function implied by the current policy; and (ii) a *policy improvement step*, where we update the policy given the current value function.

Tip

PFI can be interpreted as a version of Newton's method to solve for the zero of $\mathbf{Bv} = 0$, given $\mathbf{B} \equiv \mathbf{T} - \mathbf{I}$. Using the expression given in the policy evaluation step to eliminate $\Delta t u(\mathbf{c}^{n+1})$ from the policy improvement step, we obtain the Newton update:

$$\mathbf{v}^{n+1} = \mathbf{v}^n - ((1 - \rho\Delta t)\mathbf{P}(\mathbf{c}^{n+1}) - \mathbf{I})^{-1}(\mathbf{T}\mathbf{v}^n - \mathbf{v}^n), \tag{3.59}$$

which is analogous to the Newton update for the zero of a function $f(x)$ given by $x^{n+1} = x^n - f'(x^n)^{-1}f(x^n)$.

The finite-difference method of Section 3.3 is conceptually equivalent. The policy evaluation step corresponds to solving the linear system of equations in (3.57), in the limiting case where the time step $\Delta t \to \infty$. The policy improvement step corresponds to choosing the control that maximizes the right-hand side of (3.53), obtained by inverting the first-order condition for consumption.

Important

Continuous time vs. EGM. The policy improvement step in continuous time is closely related to the Endogenous Gridpoint Method (EGM) used in Chapter 2. In the discrete-time formulation, computing the optimal control typically requires either a root-finding procedure or the EGM, which constructs a grid for end-of-period assets and interpolates the consumption policy. In the continuous-time model (or its MCA discretization), the policy improvement step does not require a new grid or interpolation, provided the control can be expressed analytically as a function of the value function's derivative. This makes the method especially convenient when dealing with multiple control variables.

3.4 Spectral Methods

We have seen how to use finite differences to solve the HJB equation. The procedure consists of using a *local approximation* of derivatives, based on function values at neighboring points. In this section, we will see how to use *spectral methods*, which instead provide a *global approximation* of derivatives based on information from the entire domain.

3.4.1 From Local to Global Approximations of Derivatives

Finite differences as local polynomial approximations. Suppose we are interested in computing the derivative of a function $v(x)$ that we observe at discrete points x_i, $i = 1, \ldots, N$. Let $v_i = v(x_i)$ denote the function values at these grid points. For simplicity, assume the grid is uniform, $x_{i+1} - x_i = \Delta x$. A natural approximation for the derivative is the forward difference

$$v'(x_i) \approx \frac{v_{i+1} - v_i}{\Delta x}, \qquad \text{error: } O(\Delta x). \tag{3.60}$$

The $O(\Delta x)$ error term follows directly from a Taylor expansion around x_i.

This approximation is equivalent to replacing $v(x)$ locally by a *linear interpolant* between x_i and x_{i+1}:

$$\tilde{v}(x) = v_i + Dv_i(x - x_i), \qquad Dv_i = \frac{v_{i+1} - v_i}{\Delta x}. \tag{3.61}$$

Differentiating the interpolant yields $\tilde{v}'(x_i) = Dv_i$, i.e., the forward difference.

Hence, finite differences can be viewed as the derivatives of local polynomial interpolants. The central difference formula, for example,

$$v'(x_i) = \frac{v_{i+1} - v_{i-1}}{2\Delta x} + O(\Delta x^2), \tag{3.62}$$

corresponds to using a quadratic interpolant that matches $v(x)$ at the points (x_{i-1}, x_i, x_{i+1}):

$$\tilde{v}(x) = \ell_{i-1}(x)v_{i-1} + \ell_i(x)v_i + \ell_{i+1}(x)v_{i+1}, \tag{3.63}$$

where the basis polynomials are

$$\ell_{i-1}(x) = \frac{(x - x_i)(x - x_{i+1})}{2\Delta x^2}, \tag{3.64}$$

$$\ell_i(x) = -\frac{(x - x_i + \Delta x)(x - x_i - \Delta x)}{\Delta x^2}, \tag{3.65}$$

$$\ell_{i+1}(x) = \frac{(x - x_i + \Delta x)(x - x_i)}{2\Delta x^2}. \tag{3.66}$$

Differentiating and evaluating at x_i gives

$$\tilde{v}'(x_i) = \frac{v_{i+1} - v_{i-1}}{2\Delta x}, \qquad \tilde{v}''(x_i) = \frac{v_{i+1} - 2v_i + v_{i-1}}{\Delta x^2},$$

that is, the standard second-order central difference formulas.

Tip

Lagrange interpolation. The interpolant $\tilde{v}(x)$ is a special case of the *Lagrange interpolant*,

$$\tilde{v}(x) = \sum_{i=0}^{N} \ell_i(x)\, v_i,$$

where each Lagrange basis polynomial is defined as

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N} \frac{x - x_j}{x_i - x_j}.$$

The Lagrange form provides a unified way to construct both local (finite-difference) and global polynomial approximations.

Higher-order finite differences. We can obtain higher-order finite-difference formulas by fitting higher-degree local polynomials and differentiating them. For instance, a quartic interpolant yields a fourth-order approximation to $v'(x_i)$. Listing 3.6 illustrates how to compute such higher-order approximations in Julia using the `Polynomials.jl` package.

```julia
"""
    finite_difference(f, x0, accuracy; scheme=:forward, Δx=0.01)

Approximate f'(x0) by differentiating a local interpolating
polynomial built on a stencil that achieves the requested accuracy.
"""
function finite_difference(f::Function, x0::Float64, accuracy::Int;
                    scheme::Symbol = :forward, Δx::Float64 = 0.01)
    @assert accuracy > 0 "accuracy order must be positive"
    steps = zeros(Int, accuracy+1)
    if iseven(accuracy)
        m = accuracy ÷ 2
        steps = collect(-m:m)
    else
        q = accuracy
        steps = scheme === :forward ? collect(0:q) : collect(-q:0)
    end
    x_points = x0 .+ Δx .* steps
    p    = Polynomials.fit(x_points, f.(x_points), length(x_points)
    - 1)
    return Polynomials.derivative(p)(x0)
end
```

Listing 3.6: Higher-order finite-difference approximation of derivatives.

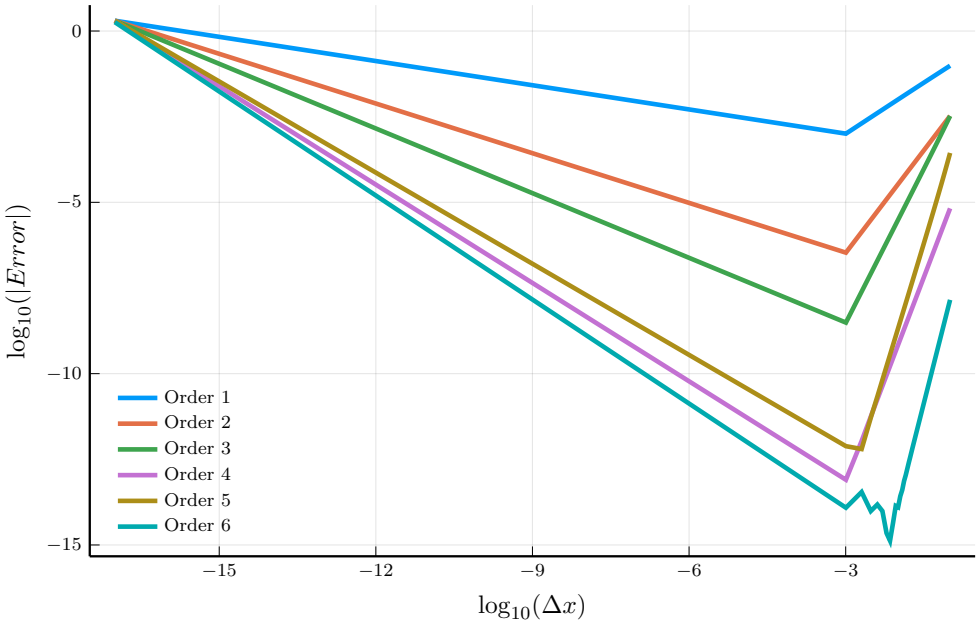


Figure 3.5: Finite-difference approximation error as a function of grid size.

Figure 3.5 shows the error of the finite-difference approximation for $f(x) = e^{x^2} + 2\sin x$ evaluated at $x = 0$. For moderate step sizes, we obtain the expected pattern: the error decreases as the grid is refined ($\Delta x \to 0$) or as the order of the approximation increases. For example, a sixth-order approximation reaches machine precision around $\Delta x \approx 10^{-2}$, while a first-order formula is accurate to only about two decimal places.

Interestingly, the error eventually increases again as Δx becomes very small. This behavior is due to *roundoff error*, which dominates when Δx is too small. In floating-point arithmetic, the subtraction $v_{i+1} - v_i$ involves nearly equal numbers, producing a loss of significant digits; dividing this by the tiny step size Δx amplifies the numerical noise. Hence, while truncation error decreases with smaller Δx, roundoff error increases, and the total error is minimized at an intermediate step size.

Note

Finite vs. spectral accuracy. Higher-order finite differences improve accuracy algebraically, with error $O(\Delta x^p)$ for a p-th order scheme. In contrast, spectral methods achieve *exponential* (or *spectral*) convergence for smooth functions, as we will see next.

From local to global polynomials. Higher-order finite differences improve accuracy by adding more points to the stencil. We are, in effect, bringing more information about the function into a local polynomial fit. An alternative is to use a *global polynomial approximation*, which uses all information from the domain to approximate the function. This can, in principle, achieve much higher accuracy with far fewer points. However, such global approximations require care in how interpolation nodes are chosen.

Global interpolation. Suppose we approximate a smooth function $v(x)$ on $[-1, 1]$ by a single polynomial of degree N:

$$v(x) \approx \tilde{v}_N(x) = \sum_{k=0}^{N} a_k x^k, \tag{3.67}$$

where the coefficients a_k are chosen so that $\tilde{v}_N(x_i) = v(x_i)$ at a set of interpolation nodes $\{x_i\}_{i=0}^N$.

> **Tip**
>
> **Change of domain.** When approximating a function $V(z)$ on a bounded domain $[a, b]$, we can always map it to the interval $[-1, 1]$ via
>
> $$v(x) \equiv V\left(\tfrac{(b-a)x+(a+b)}{2}\right), \qquad x \in [-1, 1].$$
>
> For a discussion of approximations on unbounded domains, see Chapter 17 of Boyd (2001).

This global approach uses information from the entire domain to approximate derivatives or other operators. However, if the interpolation nodes x_i are *equally spaced*, the interpolant oscillates violently near the boundaries as N increases—a pathology known as the *Runge phenomenon*.

The Runge phenomenon. Even for a simple function such as $v(x) = 1/(1 + 25x^2)$, interpolation on a uniform grid fails as we add more points. Figure 3.6 shows how the interpolant develops large oscillations near $x = \pm 1$ as the degree N increases. This instability, known as the *Runge phenomenon*, arises not from numerical error but from the mathematical properties of polynomial interpolation on uniformly spaced nodes. Near the boundaries, the interpolation error grows rapidly because the spacing of points does not capture the curvature of the function effectively.

The interpolation error theorem. To understand how to improve interpolation, it helps to characterize its error precisely. Cauchy's interpolation error theorem provides a clean decomposition of the interpolation error.

Theorem 3.1 (Cauchy's interpolation error theorem)**.** *Let $v(x)$ have at least $N+1$ derivatives on $[-1, 1]$ and let $\tilde{v}_N(x)$ be the degree-N interpolant of $v(x)$ at nodes x_i, $i = 0, \ldots, N$. Then,*

$$v(x) - \tilde{v}_N(x) = \frac{v^{(N+1)}(\xi)}{(N+1)!}\,\mathcal{P}_{N+1}(x), \qquad \mathcal{P}_{N+1}(x) = \prod_{i=0}^{N}(x - x_i), \tag{3.68}$$

for some $\xi \in [-1, 1]$.

The error therefore depends on two components:

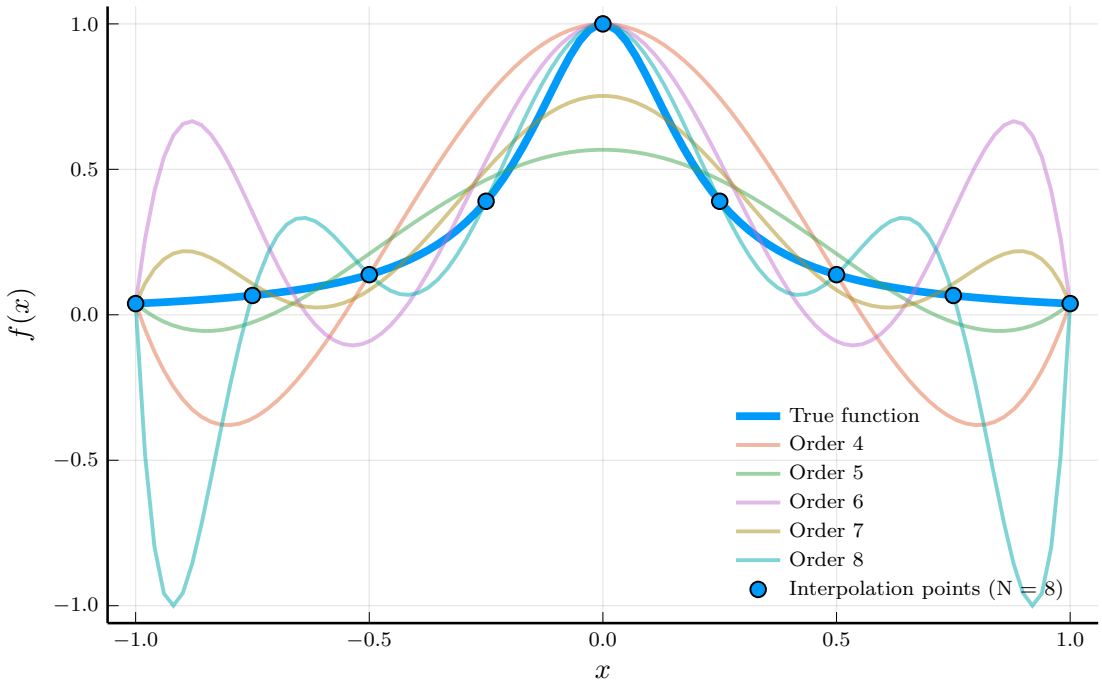


Figure 3.6: Runge phenomenon: polynomial interpolation of $f(x) = 1/(1+25x^2)$ on a uniform grid. Increasing N leads to oscillations near the boundaries.

1. The $(N+1)$-st derivative $v^{(N+1)}(\xi)$, which depends only on the smoothness of v; and
2. The *node polynomial* $\mathcal{P}_{N+1}(x)$, which depends solely on the choice of interpolation nodes.

We cannot control the smoothness of v, but we *can* control the second term. The worst-case (uniform) interpolation error is bounded by

$$|v(x) - \tilde{v}_N(x)| \leq \frac{1}{(N+1)!} \max_{x\in[-1,1]} |v^{(N+1)}(x)| \, \|\mathcal{P}_{N+1}\|_\infty, \qquad \|\mathcal{P}_{N+1}\|_\infty \equiv \max_{x\in[-1,1]} |\mathcal{P}_{N+1}(x)|. \tag{3.69}$$

Hence, the goal is clear: choose interpolation nodes $\{x_i\}$ that minimize $\|\mathcal{P}_{N+1}\|_\infty$. This leads directly to the *Chebyshev minimal amplitude theorem.*

3.4.2 Chebyshev Polynomials and the Minimax Property

Theorem 3.2 (Chebyshev minimal amplitude theorem)**.** *Define the Chebyshev nodes as*

$$\hat{x}_i = \cos\left(\frac{(2i+1)\pi}{2(N+1)}\right), \qquad i = 0, \ldots, N. \tag{3.70}$$

Then, among all possible node sets $\{x_i\}_{i=0}^N$, the polynomial $\mathcal{P}_{N+1}(x) = \prod_{i=0}^N (x - x_i)$ attains its smallest maximum amplitude on $[-1, 1]$ when the x_i are the Chebyshev nodes:

$$\|\mathcal{P}_{N+1}\|_\infty = \max_{x\in[-1,1]} |\mathcal{P}_{N+1}(x)| \geq \max_{x\in[-1,1]} \left| \prod_{i=0}^N (x - \hat{x}_i) \right|. \tag{3.71}$$

The Chebyshev nodes are the roots of the Chebyshev polynomial of the first kind,

$$T_{N+1}(x) = \cos\big((N+1)\arccos x\big),$$

which oscillates between -1 and 1 exactly $N+2$ times on $[-1,1]$. This *equal oscillation* property makes the Chebyshev nodes optimal: the interpolation error alternates between positive and negative peaks of equal magnitude, distributing the error uniformly across the interval.

> **Tip**
>
> The Chebyshev polynomial minimizes the worst-case (uniform) interpolation error among all monic polynomials of degree $N+1$. This *minimax property* is the key reason Chebyshev polynomials underpin spectral methods: they yield the smallest possible interpolation error for a given degree N.

Orthogonality of Chebyshev polynomials. Beyond optimal node placement, Chebyshev polynomials enjoy a fundamental orthogonality relation with respect to the weight function $w(x) = 1/\sqrt{1-x^2}$:

$$\int_{-1}^{1} T_m(x)\, T_n(x)\, w(x)\, dx = 0, \qquad m \neq n. \tag{3.72}$$

This orthogonality implies that the Chebyshev basis is well-conditioned for numerical work: unlike the monomial basis $\{1, x, x^2, \dots\}$, which produces highly ill-conditioned Vandermonde matrices, the Chebyshev basis yields numerically stable polynomial expansions. In summary, Chebyshev polynomials combine two key advantages—near-minimax interpolation error and orthogonality—making them the canonical choice for global interpolation and spectral differentiation.

3.4.3 Spectral differentiation with Chebyshev polynomials

We have seen that Chebyshev polynomials provide an optimal set of nodes and basis functions for global interpolation. We now show how to use this interpolation to compute derivatives with spectral accuracy.

From interpolation to differentiation. Let $v(x)$ be a smooth function on $[-1,1]$. We approximate $v(x)$ by its degree-N Chebyshev interpolant:

$$\tilde{v}_N(x) = \sum_{k=0}^{N} a_k T_k(x),$$

where the coefficients a_k are chosen so that $\tilde{v}_N(x_j) = v(x_j)$ at a set of interpolation nodes $\{x_j\}_{j=0}^{N}$.

Tip

For spectral differentiation, it is convenient to use the *Chebyshev–Lobatto nodes*, which include the endpoints:

$$x_j = \cos\left(\frac{j\pi}{N}\right), \qquad j = 0, \dots, N.$$

These correspond to the extrema of $T_N(x)$ rather than its roots, ensuring that the grid includes $x = \pm 1$ and that derivative boundary values are well-defined.

Define the vector of function values at the nodes:

$$\mathbf{v} = \begin{bmatrix} v(x_0) \\ v(x_1) \\ \vdots \\ v(x_N) \end{bmatrix}.$$

The derivative of the interpolant $\tilde{v}_N'(x)$ can be evaluated exactly at the same nodes:

$$\mathbf{v}' = \begin{bmatrix} \tilde{v}_N'(x_0) \\ \tilde{v}_N'(x_1) \\ \vdots \\ \tilde{v}_N'(x_N) \end{bmatrix} = \mathbf{D}\,\mathbf{v},$$

where $\mathbf{D}$ is the *Chebyshev differentiation matrix*.

Explicit formula for the differentiation matrix. Let $x_j = \cos(j\pi/N)$ for $j = 0, \dots, N$, and define

$$c_j = \begin{cases} 2(-1)^j, & j = 0 \text{ or } j = N, \\ (-1)^j, & \text{otherwise.} \end{cases}$$

Then the entries of $\mathbf{D}$ are given by

$$D_{ij} = \begin{cases} \dfrac{c_i}{c_j}\,\dfrac{1}{x_i - x_j}, & i \neq j, \\ -\dfrac{x_j}{2(1 - x_j^2)}, & 1 \leq j \leq N-1, \\ \dfrac{2N^2 + 1}{6}, & i = j = 0, \\ -\dfrac{2N^2 + 1}{6}, & i = j = N. \end{cases} \tag{3.73}$$

This closed-form expression yields the exact derivative of the interpolating polynomial at the Chebyshev–Lobatto nodes.

Accuracy. Because the Chebyshev interpolation error decays exponentially for smooth functions, the same holds for the derivative approximation:

$$\|\tilde{v}_N'(x) - v'(x)\|_\infty = O(e^{-\alpha N}),$$

for some $\alpha > 0$ depending on the analyticity of v. This *spectral accuracy* is what distinguishes Chebyshev differentiation from finite differences.

Julia implementation. Listing 3.7 shows the Julia implementation of the Chebyshev differentiation matrix. The function `cheb_diff_matrix(N)` returns both the nodes and the differentiation matrix.

```julia
function cheb_diff_matrix(N::Int64)
    x = cos.(pi .* (0:N) ./ N) # Chebyshev-Lobatto nodes
    c = [2; ones(N-1); 2] .* (-1).^(0:N) # c_j weights
    D = zeros(N+1, N+1) # Chebyshev differentiation matrix
    for i in 1:N+1, j in 1:N+1
        if i != j
            D[i,j] = (c[i] / c[j]) / (x[i] - x[j]) # off-diagonals
        end
    end
    D[1,1], D[end,end] =  (2N^2 + 1) / 6, -(2N^2 + 1) / 6
    for j in 2:N
        D[j,j] = -x[j] / (2*(1 - x[j]^2)) # diagonals
    end
    return x, D
end
```

Listing 3.7: Chebyshev differentiation matrix in Julia.

The function `cheb_diff_matrix(N)` returns the differentiation matrix for a function defined on the canonical interval $[-1, 1]$. Using Julia's multiple dispatch, we can extend the definition of the differentiation matrix to functions defined on a general interval $[a, b]$.

```julia
function cheb_diff_matrix(N::Integer, a::Real, b::Real)
    x, D = cheb_diff_matrix(N)
    z = (a + b)/2 .+ (b - a)/2 .* x   # map [-1,1] -> [a,b]
    Dz = (2/(b - a)) .* D
    return z, Dz
end
```

Listing 3.8: Chebyshev differentiation matrix in Julia for a general interval $[a, b]$.

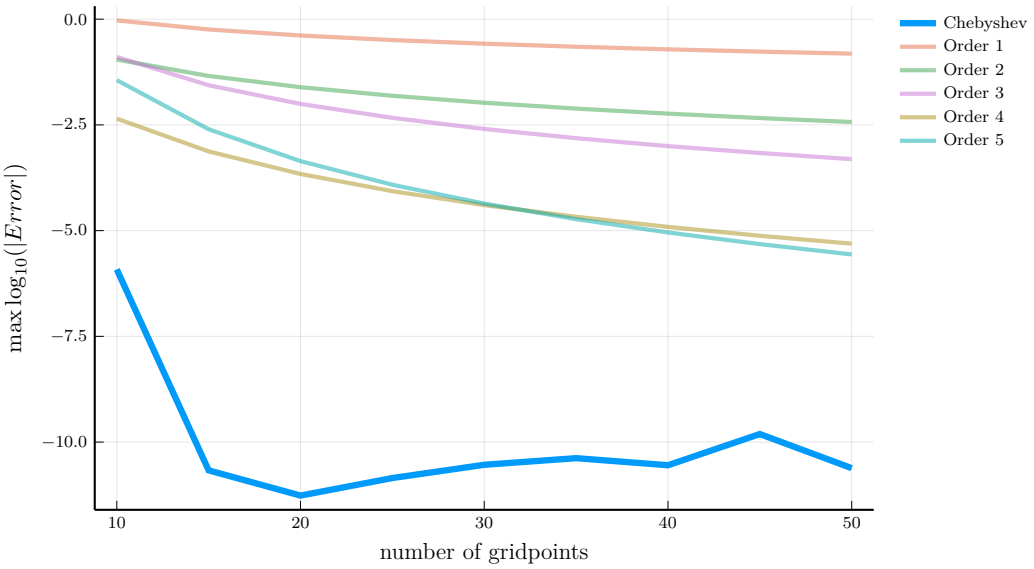


Figure 3.7: Derivative approximation error: Chebyshev differentiation versus finite differences

Spectral accuracy in practice. Figure 3.7 compares the derivative approximation error obtained using the Chebyshev differentiation matrix and a finite-difference approximation for the smooth function $v(x) = e^{x^2} + 2\sin(x)$. For smooth functions, the spectral method reaches machine precision with a modest number of nodes (around $N = 20$), while the finite-difference method converges only algebraically in N. This exponential accuracy, combined with the stability of the Chebyshev basis, makes spectral methods particularly powerful for solving dynamic programming and PDE problems with smooth value functions.

3.4.4 Chebyshev Collocation

Two-trees model. To illustrate how to use Chebyshev polynomials to solve a dynamic programming problem, we consider the two-trees model of Cochrane et al. (2008). They study an economy with two Lucas trees with dividends following a geometric Brownian motion:

$$\frac{dD_{i,t}}{D_{i,t}} = \mu dt + \sigma dB_{i,t}, \qquad i = 1, 2, \tag{3.74}$$

where $dB_{i,t}$ are independent standard Brownian motions.

Aggregate consumption equals the sum of dividends from the two trees: $C_t = D_{1,t} + D_{2,t}$. The representative household has log utility, so the SDF for this economy can be written as $\pi_t = e^{-\rho t}/C_t$, where ρ is the subjective discount rate. This implies that the price of the first tree is given by:

$$P_t = \mathbb{E}_t \left[\int_0^\infty e^{-\rho s} \frac{C_t}{C_{t+s}} D_{1,t+s} ds \right]. \tag{3.75}$$

Let $s_t \equiv \frac{D_{1,t}}{D_{1,t}+D_{2,t}}$ be the dividend share of the first tree. Itô's lemma implies that:

$$ds_t = -2\sigma^2 s_t(1 - s_t)(s_t - 1/2)dt + \sigma s_t(1 - s_t)(dB_{1,t} - dB_{2,t}). \tag{3.76}$$

Define the price-consumption ratio $v_t \equiv \frac{P_t}{C_t}$. Then, v_t satisfies the pricing condition:

$$v_t = \mathbb{E}_t \left[\int_0^\infty e^{-\rho s} s_{t+s} ds \right]. \tag{3.77}$$

As s_t is a Markov process, we can write the price-consumption ratio as a function of s_t: $v_t = v(s_t)$. Because s_t is Markov, $v_t = v(s_t)$ satisfies the stationary HJB equation:

$$\rho v = s - v_s \, 2\sigma^2 s(1-s)\big(s - \tfrac{1}{2}\big) + \frac{1}{2} v_{ss} \left(2\sigma^2 s^2 (1-s)^2\right), \tag{3.78}$$

with boundary conditions $v(0) = 0$ and $v(1) = 1/\rho$. The first boundary condition corresponds to the case where the first tree's dividend share is zero, and the second corresponds to the limiting case of a single-tree economy.

Chebyshev collocation. We can solve this boundary value problem using Chebyshev collocation. In particular, we approximate $v(s)$ by a series of Chebyshev polynomials:

$$v(s) = \sum_{n=0}^{\infty} a_n \tilde{T}_n(s) \approx \sum_{n=0}^{N} a_n \tilde{T}_n(s), \qquad \tilde{T}_n(s) = T_n(2s-1). \tag{3.79}$$

> **Tip**
>
> The mapping $s \mapsto x = 2s - 1$ rescales the domain from $[0, 1]$ to $[-1, 1]$, allowing the use of standard Chebyshev polynomials. Derivatives with respect to s follow from the chain rule, yielding $\tilde{T}_n'(s) = 2T_n'(2s-1)$ and $\tilde{T}_n''(s) = 4T_n''(2s-1)$.

Plugging the expansion into the HJB equation and evaluating at the grid points, we obtain:

$$\mathbf{La} = \mathbf{b}, \tag{3.80}$$

where $\mathbf{L}$ is a $(N+1) \times (N+1)$ matrix and $\mathbf{b}$ is a $(N+1)$ vector.

For $i = 2, \ldots, N$ and $j = 1, \ldots, N+1$, we have:

$$\mathbf{L}_{i,j} \;=\; \rho \tilde{T}_{j-1}(s_i) \;+\; 2\sigma^2 s_i(1-s_i)\big(s_i - \tfrac{1}{2}\big)\, \tilde{T}_{j-1}'(s_i) \;-\; \sigma^2 s_i^2 (1-s_i)^2\, \tilde{T}_{j-1}''(s_i). \tag{3.81}$$

The first and last rows accommodate the boundary conditions:

$$\mathbf{L}_{1,j} = \tilde{T}_{j-1}(0), \qquad \mathbf{L}_{N+1,j} = \tilde{T}_{j-1}(1). \tag{3.82}$$

Similarly, for the vector $\mathbf{b}$:

$$b_i = s_i \quad \text{for } i = 2, \ldots, N, \qquad b_1 = 0, \quad b_{N+1} = 1/\rho. \tag{3.83}$$

> **Note**
>
> Collocation methods, also known as pseudospectral methods, are a class of methods that approximate the solution of a differential equation by a series evaluated at a set of collocation points. In this case, we approximate the solution of the HJB equation by a series of Chebyshev polynomials evaluated at the Chebyshev-Lobatto grid.

Julia implementation. Listing 3.9 shows the Julia implementation of the Two-trees model struct.

```julia
@kwdef struct TwoTrees
    ρ::Float64 = 0.04
    σ::Float64 = sqrt(0.04)
    μ::Float64 = 0.02
    N::Int = 7
end
```

Listing 3.9: Two-trees model struct in Julia.

To construct the matrix $\mathbf{L}$, we need to evaluate the Chebyshev polynomials and their derivatives at the Chebyshev-Lobatto grid. Listing 3.11 uses the implementation of Chebyshev polynomials and their derivatives from the `Polynomials.jl` package. The function also performs the mapping from the interval $[z_{\min}, z_{\max}]$, and uses the chain rule to obtain the derivatives of $\tilde{T}_n(\cdot)$.

```julia
function chebyshev_derivatives(n::Int, z::Real;
        zmin::Real = -1.0, zmax::Real = 1.0)
    a, b = 2 / (zmax - zmin), -(zmin + zmax) / (zmax - zmin)
    x = a * z + b # Map to [-1,1]
    p = ChebyshevT([zeros(n);1.0]) # Degree n Chebyshev polynomial
    d1p = derivative(p) # First derivative
    d2p = derivative(d1p) # Second derivative
    return p(x), d1p(x) * a, d2p(x) * a^2
end
```

Listing 3.10: Chebyshev polynomials and their derivatives in Julia.

The function `chebyshev_solver` solves for the coefficients a_n of the Chebyshev expansion of the price-consumption ratio $v(s)$. We start by constructing the Chebyshev-Lobatto grid and the mapping to the interval $[z_{\min}, z_{\max}] = [0, 1]$. Then, we construct the matrix $\mathbf{L}$ and the right-hand side $\mathbf{b}$, and solve for the vector of coefficients $\mathbf{a}$. The function returns a NamedTuple with the function $v(s)$ and the grid s_i. Notice that the function $v(s)$ can be evaluated at any point in the interval $[z_{\min}, z_{\max}]$, not just the grid points.

```julia
function chebyshev_solver(m::TwoTrees)
    (; ρ, σ, N) = m
    # Chebyshev grid and mapping to [0,1]
    x = reverse(cos.(pi .* (0:N) ./ N))
    s = (x .+ 1) ./ 2
    # Assemble the linear operator
    L, b = zeros(N+1, N+1), copy(s)
    for i in 1:N+1, j in 1:N+1
        T̃, dT, d2T = chebyshev_derivatives(j-1, s[i]; zmin = 0)
        if i == 1 || i == N+1
            L[i,j] = T̃  # Boundary points
        else
            μs = -2 * σ^2 * s[i] * (1 - s[i]) * (s[i] - 1/2)
            σs = sqrt(2) * σ * s[i] * (1 - s[i])
            L[i,j] = ρ * T̃  - dT * μs - d2T * σs^2 / 2
        end
    end
    b[end] = 1/ρ # Boundary condition at s = 1
    a = L \ b  # Solve for the coefficients
    return (; v = z -> ChebyshevT(a)(2 * z - 1), s = s)
end
```

Listing 3.11: Chebyshev solver in Julia.

Figure 3.8 shows the solution of the two-trees model for the price-consumption ratio $v(s)$ and compares with the exact analytical solution provided by Cochrane et al. (2008). Notice that even with a modest number of nodes ($N = 7$), the Chebyshev collocation method approximates the exact solution very well.

Figure 3.9 compares the accuracy of the Chebyshev collocation method and a finite-difference scheme for the two-trees model using the same number of grid points. The Chebyshev collocation method achieves errors several orders of magnitude smaller for the same N. In fact, the accuracy with only $N = 10$ Chebyshev nodes is comparable to that of a finite-difference scheme with $N = 100$, highlighting the spectral (exponential) convergence of the Chebyshev method compared to the algebraic convergence of finite differences.

3.5 The Challenge of High-Dimensional Problems Redux

We have seen how to use finite-difference and Chebyshev collocation methods to solve dynamic programming problems in continuous time. These techniques extend naturally from the discrete-time setting and are extremely powerful in one or two dimensions. However, they suffer from the same three *curses of dimensionality* introduced in Chapters 1 and 2: representation, optimization, and expectation. Here we revisit how these challenges manifest

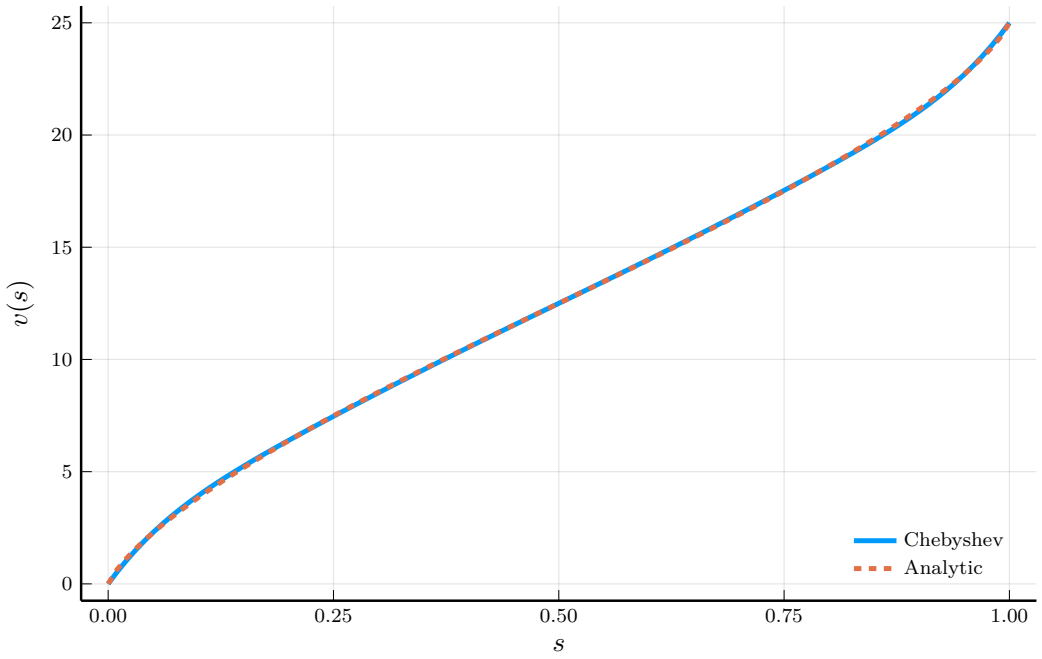


Figure 3.8: Two-trees model: Chebyshev collocation solution for the price–consumption ratio of the first tree $v(s)$.

in the context of continuous-time PDE methods.

Finite-difference schemes in higher dimensions. Because finite-difference schemes represent the state space on a grid, they inherit the first curse of dimensionality—the curse of *representation*. The number of grid points grows exponentially with the number of state variables, making the approach computationally infeasible beyond two or three dimensions with uniform grids.[2]

A second difficulty is maintaining the *monotonicity* of the scheme. As shown in Section 3.2.2, monotonicity is essential for ensuring the stability and convergence of the scheme. It is straightforward to guarantee monotonicity in one dimension—by combining upwinding for the first derivative with a central difference for the second. In higher dimensions, however, cross-derivative terms create complications. Consider an HJB equation with two state variables $\mathbf{s} = (s_1, s_2)$:

$$\rho v = u(\mathbf{s}) + v_{s_1}\,\mu_{s_1}(\mathbf{s}) + v_{s_2}\,\mu_{s_2}(\mathbf{s}) + \tfrac{1}{2}v_{s_1 s_1}\,\sigma^2_{s_1}(\mathbf{s}) + \tfrac{1}{2}v_{s_2 s_2}\,\sigma^2_{s_2}(\mathbf{s}) + v_{s_1 s_2}\,\sigma_{s_1 s_2}(\mathbf{s}). \tag{3.84}$$

A central-difference approximation for the mixed derivative,

$$v_{s_1 s_2} \approx \frac{v_{i+1,j+1} - v_{i-1,j-1} - v_{i+1,j-1} + v_{i-1,j+1}}{4\Delta s_1 \Delta s_2}, \tag{3.85}$$

produces nonnegative and nonmonotone off-diagonal elements in the discretization matrix, violating the M-matrix conditions required for stability. Bonnans et al. (2004) and d'Avernas

[2]For a discussion of finite differences with sparse grids, see Schaab and Zhang (2022).

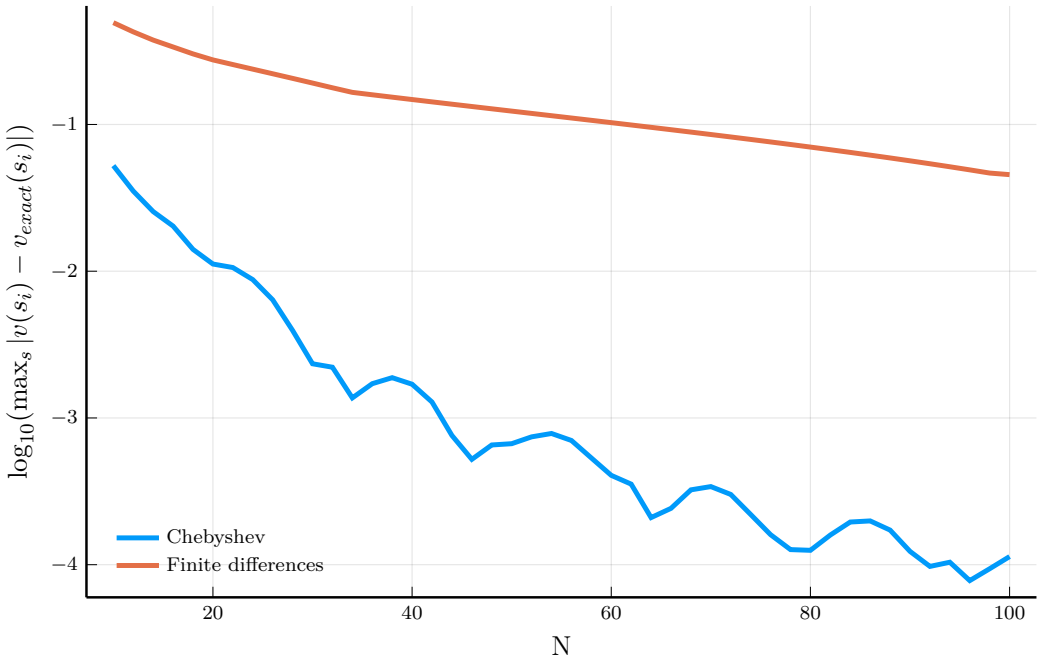


Figure 3.9: Spectral accuracy for the HJB: convergence of the Chebyshev collocation error as the number of nodes increases.

et al. (2024) propose a stencil-decomposition approach that restores monotonicity in two dimensions. Unfortunately, this method becomes combinatorially expensive in higher dimensions, as constructing nonnegative weights for all cross-directions grows exponentially with the number of state variables.

Chebyshev collocation in higher dimensions. Chebyshev collocation methods share the same first curse. In one dimension, they achieve exponential accuracy with relatively few nodes, as the Chebyshev basis captures smooth variation efficiently. In d dimensions, however, the tensor product of one-dimensional bases yields $(N+1)^d$ nodes, and the size of the differentiation and collocation matrices grows as $\mathcal{O}(N^{2d})$, rapidly exhausting computational memory and time.

Several strategies mitigate this explosion. *Smolyak sparse grids* and *anisotropic tensor-product bases* (see Judd et al. (2014)) dramatically reduce the number of collocation points when the value function is sufficiently smooth or when only a subset of dimensions exhibits strong nonlinearity. Yet even these approaches face steep scaling once d exceeds five to ten dimensions.

The need for new methods. Both finite-difference and collocation methods ultimately confront the same exponential barriers as their discrete-time counterparts. In the chapters that follow, we explore more modern machine-learning-based approaches that circumvent these limitations. By using neural networks to represent value and policy functions, automatic differentiation to compute derivatives, and stochastic optimization to approximate expectations, we can design algorithms—such as the *Deep Policy Iteration* method—that remain

feasible in truly high-dimensional settings. Before turning to those applications, however, the next chapter develops the necessary machine-learning foundations.

Chapter 4

Fundamentals of Machine Learning

In this chapter, we introduce the fundamental concepts of machine learning (ML). Our goal is not to provide a comprehensive review of the field—for that, see the excellent textbooks by Hastie et al. (2009), Goodfellow et al. (2016), and Prince (2023)—but rather to develop the core tools needed to study and approximate high-dimensional economic models in later chapters.

We proceed in three steps. First, we show how to represent functions using neural networks. Second, we explain how to estimate their parameters using stochastic gradient descent. Finally, we describe how automatic differentiation enables efficient computation of the gradients required for training.

4.1 Supervised Learning and Neural Networks

In this section, we introduce the fundamental concepts of supervised learning and neural networks. We begin with a brief review of supervised learning, the branch of machine learning concerned with predicting outputs from inputs based on labeled examples. We then introduce *shallow neural networks*, which generalize linear regression by combining linear transformations with nonlinear activation functions. Finally, we turn to *deep neural networks*, which extend this architecture through multiple layers and can represent complex functions more efficiently.

4.1.1 Supervised Learning

Supervised learning concerns the task of inferring a mapping from inputs to outputs using labeled data. Given examples where both the inputs and desired outputs are observed, the goal is to learn a function that generalizes well to new, unseen data. This framework includes familiar models such as linear regression but extends naturally to settings where the relationships are high-dimensional and nonlinear.

Formally, supervised learning seeks a function f that maps inputs $\mathbf{x} \in \mathbb{R}^d$ to outputs $\mathbf{y} \in \mathbb{R}^p$, given a dataset of labeled pairs $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^I$. In essence, this is a regression problem—but one that often involves extremely large input spaces and complex nonlinear dependencies. Such problems arise routinely in modern applications, from image and speech recognition to

(a) Original (b) Red channel (c) Green channel (d) Blue channel

Figure 4.1: An example of an ImageNet image (gazelle) and its RGB channels.

text understanding, where the dimension d can reach hundreds of thousands.

Example: Image classification. The ImageNet dataset contains over 15 million labeled images spanning more than 22,000 categories. Each image is a $224 \times 224 \times 3$ array of pixels, where each pixel takes values between 0 and 255. Each of the three 224×224 layers corresponds to a color channel (red, green, or blue), as illustrated in Figure 4.1. We can represent such an image as a vector of dimension $d = 224 \times 224 \times 3 = 150{,}528$. Its label is a 22,000-dimensional one-hot vector indicating the correct category. This example highlights how high-dimensional inputs naturally arise in supervised learning tasks.

A linear model could, in principle, be used to predict the category of an image. However, such a model would treat each pixel independently—implying that the effect of changing one pixel's intensity is unrelated to its neighbors. Images, by contrast, contain strong spatial structure and nonlinear relationships between pixels across channels. Capturing these relationships requires nonlinear models.

In groundbreaking work, Krizhevsky et al. (2012)—widely known as AlexNet—introduced a deep neural network architecture that achieved dramatic improvements in classification accuracy on ImageNet. Their model had eight layers and over 60 million parameters, exceptionally large at the time, and demonstrated that deep neural networks could solve complex visual tasks at scale.

By today's standards, AlexNet is relatively small: for comparison, GPT-4 contains roughly 175 billion parameters. In what follows, we will unpack the main building blocks behind such high-dimensional, nonlinear models—starting with simple linear regression and gradually building toward deep neural networks.

4.1.2 Linear Regression

To illustrate the key concepts in supervised learning, it is useful to start with a familiar model: linear regression. Suppose we are given a dataset of I observations of the input $\mathbf{x}_i \in \mathbb{R}^d$ and output $y_i \in \mathbb{R}$ for $i = 1, \ldots, I$. Given the input $\mathbf{x}_i$, the model prediction $\hat{y}_i$ is given by the function $\hat{y}_i = f(\mathbf{x}_i, \boldsymbol{\theta})$, where $\boldsymbol{\theta} \in \mathbb{R}^d$ is a vector of parameters. In a linear regression model, the function f is a linear combination of the inputs:

$$f(\mathbf{x}_i, \boldsymbol{\theta}) = \mathbf{w}^\top \mathbf{x}_i + b \tag{4.1}$$

where $\mathbf{w}$ are the weights and b is the bias.

The function in Equation 4.1 defines a family of linear functions parameterized by $\boldsymbol{\theta} = (\mathbf{w}^\top, b)^\top$, where $\mathbf{w} \in \mathbb{R}^{d-1}$ is the weight vector and $b \in \mathbb{R}$ is the bias.[1] Panel (a) of Figure 4.2 illustrates the linear functions for three different parameter values.

Our goal is to find the parameter values that best fit the data. To do this, we define a loss function that measures how well the model fits. A common choice is the mean squared error (MSE):

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2I} \sum_{i=1}^{I} (y_i - f(\mathbf{x}_i, \boldsymbol{\theta}))^2. \tag{4.2}$$

Panel (b) of Figure 4.2 shows the MSE loss surface. Darker regions indicate lower loss values, and the contour lines correspond to level sets of constant loss. Our goal is to find the parameter vector $\boldsymbol{\theta}$ that minimizes $\mathcal{L}(\boldsymbol{\theta})$.

The least squares estimator. This problem is simple enough that we can solve it analytically. Differentiating the loss function with respect to the parameters and setting the derivatives to zero yields:

$$\frac{1}{I} \sum_{i=1}^{I} (y_i - \mathbf{w}^\top \mathbf{x}_i - b)\mathbf{x}_i = \mathbf{0}_{d-1}, \qquad \frac{1}{I} \sum_{i=1}^{I} (y_i - \mathbf{w}^\top \mathbf{x}_i - b) = 0. \tag{4.3}$$

It is convenient to rewrite this in matrix form. Define the design matrix and output vector as:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top & 1 \\ \mathbf{x}_2^\top & 1 \\ \vdots & \vdots \\ \mathbf{x}_I^\top & 1 \end{bmatrix} \in \mathbb{R}^{I \times d}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_I \end{bmatrix} \in \mathbb{R}^I. \tag{4.4}$$

The gradient of the loss function with respect to the parameters can then be written compactly as:

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \\ \frac{\partial \mathcal{L}}{\partial b} \end{bmatrix} = \frac{1}{I} \mathbf{X}^\top (\mathbf{X}\boldsymbol{\theta} - \mathbf{y}). \tag{4.5}$$

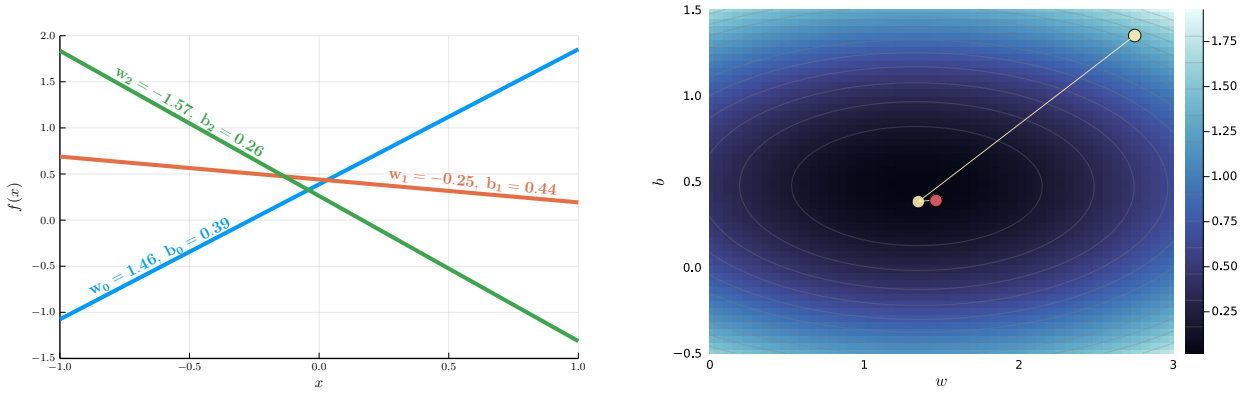
> **Tip**
>
> We denote the gradient of a function $f(\mathbf{x}) \in \mathbb{R}$ with respect to $\mathbf{x} \in \mathbb{R}^d$ as $\nabla f(\mathbf{x}) \in \mathbb{R}^d$. The gradient is a column vector, so its differential is written as $df(\mathbf{x}) = \nabla f(\mathbf{x})^\top d\mathbf{x}$.

Setting $\nabla \mathcal{L}(\boldsymbol{\theta}) = 0$ and solving yields the least squares estimator:

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{4.6}$$

Given $\hat{\boldsymbol{\theta}}$, predictions for new inputs $\mathbf{x}_*$ are obtained as $\hat{y}_* = f(\mathbf{x}_*, \hat{\boldsymbol{\theta}})$.

[1] Formally, $f(\mathbf{x}_i, \boldsymbol{\theta})$ is an affine function of the inputs when the bias is nonzero. We follow common practice and abstract from this distinction, referring to $f(\mathbf{x}_i, \boldsymbol{\theta})$ as a *linear function*. Functions that do not take this form are called *nonlinear*.



(a) Linear functions. (b) Loss surface.

Figure 4.2: Linear model visualization and its loss landscape.

Gradient descent. When $f(\mathbf{x}_i, \boldsymbol{\theta})$ is nonlinear in the parameters, the analytical solution typically no longer exists. In such cases, we minimize the loss function iteratively. Starting from an initial guess $\boldsymbol{\theta}^{(0)}$, we update $\boldsymbol{\theta}$ in the direction opposite to the gradient of the loss. To see why, note that the change in the loss function is approximately given by:

$$\mathcal{L}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) - \mathcal{L}(\boldsymbol{\theta}) \approx \nabla\mathcal{L}(\boldsymbol{\theta})^\top \Delta\boldsymbol{\theta}. \tag{4.7}$$

By the Cauchy–Schwarz inequality, $|\nabla\mathcal{L}(\boldsymbol{\theta})^\top \Delta\boldsymbol{\theta}| \leq \|\nabla\mathcal{L}(\boldsymbol{\theta})\|\,\|\Delta\boldsymbol{\theta}\|$, so the largest reduction in the loss function is achieved when $\Delta\boldsymbol{\theta}$ is proportional to the negative gradient. This leads to the iterative update rule:

Algorithm 3: Gradient Descent

Input: Initial guess $\boldsymbol{\theta}^{(0)}$, learning rate η
Output: Parameter estimate $\boldsymbol{\theta}$

initialize $\boldsymbol{\theta}^{(0)}$ **repeat**
  $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})$
until $\|\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})\| < \epsilon$;
return $\boldsymbol{\theta}^{(t)}$

Algorithm 3 iteratively updates $\boldsymbol{\theta}$ by moving against the gradient, with step size η (the learning rate). Listing 4.1 provides a Julia implementation of the least-squares gradient descent algorithm for $d = 2$.

```julia
function ls_grad_descent(y::AbstractVector{<:Real},
    x::AbstractVector{<:Real}, θ₀::AbstractVector{<:Real};
    learning_rate::Real=0.01, max_iter::Integer=100, ε::Real = 1e-4)
    @assert length(x) == length(y)
    I = length(x)
    X = hcat(x, ones(I)) # design matrix
    ∇f(θ) = X' * (X * θ - y) / I # gradient
    w_path, b_path =Float64[θ₀[1]], Float64[θ₀[2]] # initial values
```

```
    for _ in 1:max_iter
        g = ∇f([w_path[end], b_path[end]])
        push!(w_path, w_path[end] - learning_rate * g[1])
        push!(b_path, b_path[end] - learning_rate * g[2])
        if norm(g) < ε
            break
        end
    end
    return (;w = w_path, b = b_path)
end
```

Listing 4.1: Least squares gradient descent.

In Panel (b) of Figure 4.2, the yellow dots trace the path of the gradient descent iterations, and the orange dot marks the true parameter values used to simulate the data. Starting from $\boldsymbol{\theta}^{(0)} = (2.75, 1.35)$, the algorithm converges rapidly to the minimum at $\boldsymbol{\theta} = (1.46, 0.39)$. The estimated parameters are nearly identical to the true values, confirming the accuracy of gradient descent in this simple case.

Important

Computational cost. The gradient of the loss function with respect to the parameters is given by:

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{I} \sum_{i=1}^{I} \big(f(\mathbf{x}_i, \boldsymbol{\theta}) - y_i \big) \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_i, \boldsymbol{\theta}). \tag{4.8}$$

Each iteration of gradient descent therefore requires evaluating $\nabla_{\boldsymbol{\theta}} f(\mathbf{x}_i, \boldsymbol{\theta})$ for all I observations. For large datasets, this can be computationally expensive—a limitation addressed by *stochastic gradient descent*, discussed in the next section.

Training and testing. In the discussion above, we used all available data to estimate the parameters. In practice, we typically split the data into a *training set* and a *test set*. The training set is used to estimate the parameters, and the test set to evaluate how well the model generalizes to new data. This distinction becomes crucial when comparing models of different complexity.

As an example, suppose the data are generated by a noisy version of the Runge function,

$$y_i = \frac{1}{1 + 25 z_i^2} + \epsilon_i, \qquad \epsilon_i \sim \text{i.i.d. with } \mathbb{E}[\epsilon_i] = 0,$$

with $z_i \in [-1, 1]$. Consider a polynomial regression with regressors $\mathbf{x}_i = [z_i, z_i^2, \ldots, z_i^{d-1}]$. As the degree d increases, the model becomes more flexible and fits the training data increasingly well. When $d = I$, the model reaches the *interpolation threshold*, perfectly fitting the training observations.

From Section 3.4, we know that high-degree polynomial interpolants of the Runge function oscillate violently between nodes, deviating substantially from the true function away from

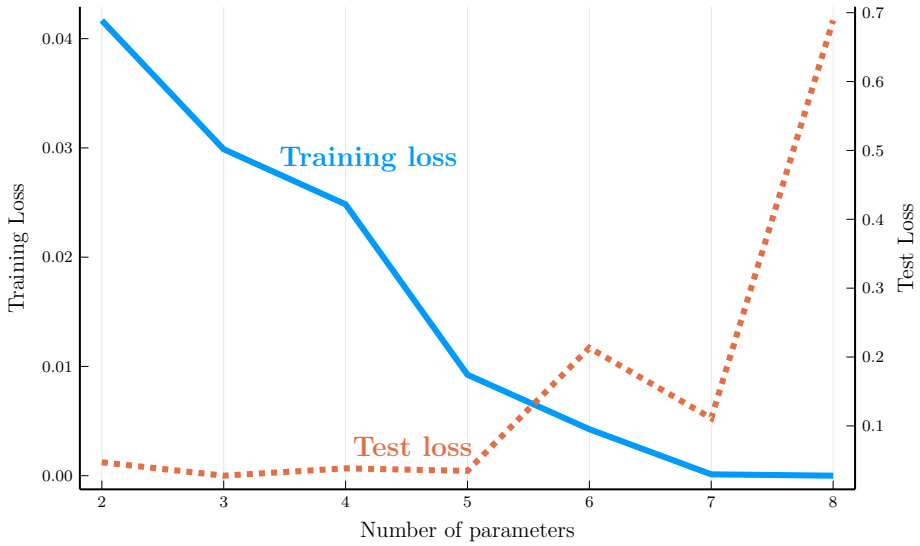


Figure 4.3: Training and test loss as the number of parameters increases in polynomial regression.

interpolation points. Consequently, the model performs poorly on the test set. Figure 4.3 illustrates this pattern: as d increases, the training loss declines steadily, but the test loss eventually rises. This is the hallmark of *overfitting*—the model learns not only the underlying signal but also the noise in the training data.

4.1.3 Shallow Neural Networks

We now move from linear to nonlinear models by introducing the *shallow neural network* (SNN). An SNN can be viewed as a generalization of the linear regression model, where a series of intermediate transformations combines a linear function with a nonlinear activation.

Consider a collection of nonlinear transformations of the inputs known as *hidden units*:

$$h_j(\mathbf{x}_i, \boldsymbol{\theta}_j) = \sigma\left(\mathbf{w}_j^\top \mathbf{x}_i + b_j\right), \qquad j = 0, \ldots, n-1, \tag{4.9}$$

where each hidden unit has parameters $\boldsymbol{\theta}_j = (\mathbf{w}_j^\top, b_j)^\top$. Stacking these hidden units in parallel and linearly combining their outputs yields the shallow neural network.

The nonlinear transformation $\sigma(\cdot)$ is known as the *activation function*. Throughout this section, we focus on the simplest and most widely used choice—the *rectified linear unit* (ReLU):

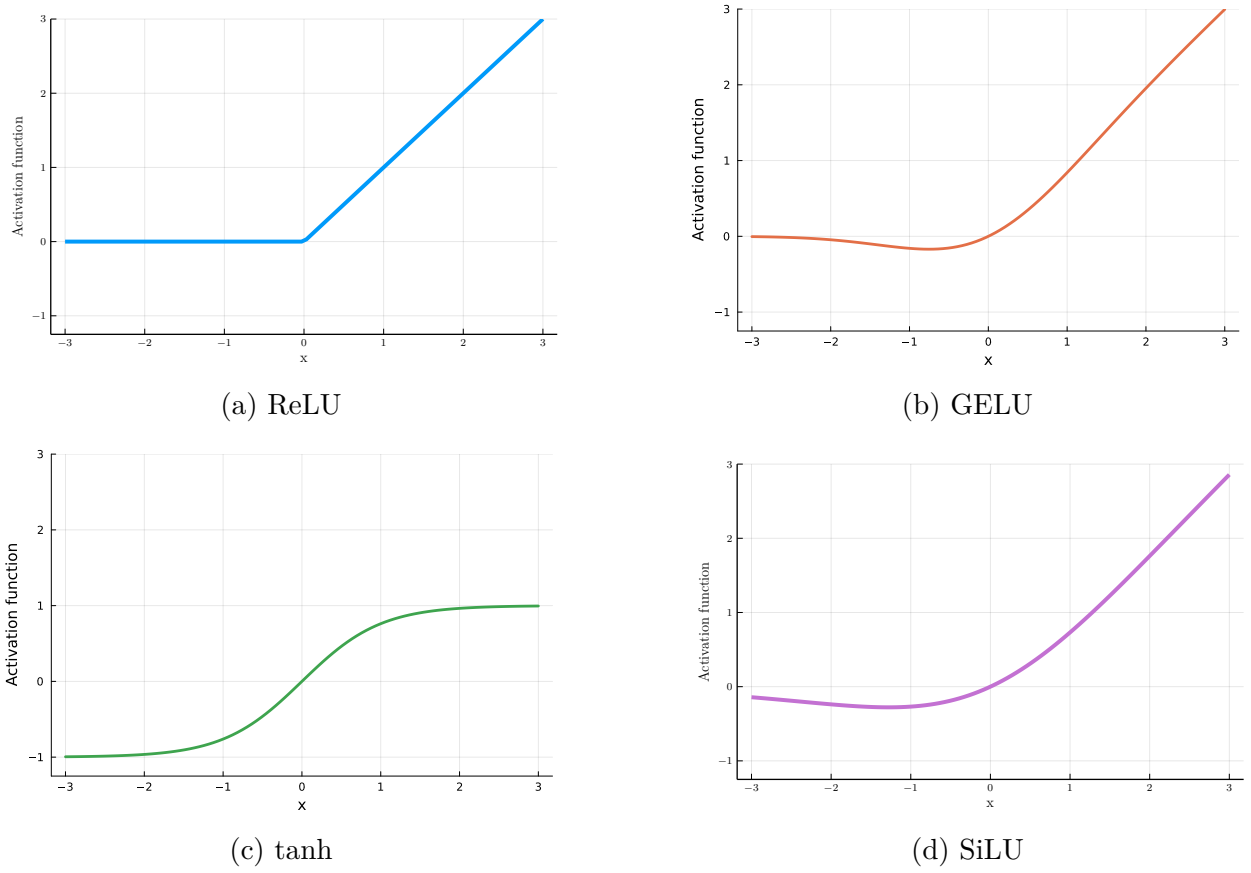
$$\sigma(x) = \max(0, x). \tag{4.10}$$

Other popular choices are shown in Figure 4.4.

The SNN can then be written as

$$f(\mathbf{x}_i, \boldsymbol{\theta}) = \mathbf{w}_n^\top \mathbf{h}(\mathbf{x}_i, \boldsymbol{\theta}) + b_n, \tag{4.11}$$

where $\boldsymbol{\theta} = (\boldsymbol{\theta}_0^\top, \ldots, \boldsymbol{\theta}_n^\top)^\top$, $\boldsymbol{\theta}_n = (\mathbf{w}_n^\top, b_n)^\top$, and $\mathbf{h}(\mathbf{x}_i, \boldsymbol{\theta}) = (h_0(\mathbf{x}_i, \boldsymbol{\theta}_0), \ldots, h_{n-1}(\mathbf{x}_i, \boldsymbol{\theta}_{n-1}))^\top$ is the vector of hidden-unit outputs.



(a) ReLU (b) GELU

(c) tanh (d) SiLU

Figure 4.4: Common activation functions.
Note: the four activation functions are defined as follows: (i) Rectified Linear Unit (ReLU): $\sigma(x) = \max(0, x)$; (ii) Gaussian Error Linear Unit (GELU): $\sigma(x) = x\Phi(x)$, where $\Phi(x)$ is the standard normal cdf; see Hendrycks and Gimpel (2016); (iii) Hyperbolic tangent (tanh): $\sigma(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$; (iv) Sigmoid Linear Unit (SiLU): $\sigma(x) = \frac{x}{1+\exp(-x)}$.

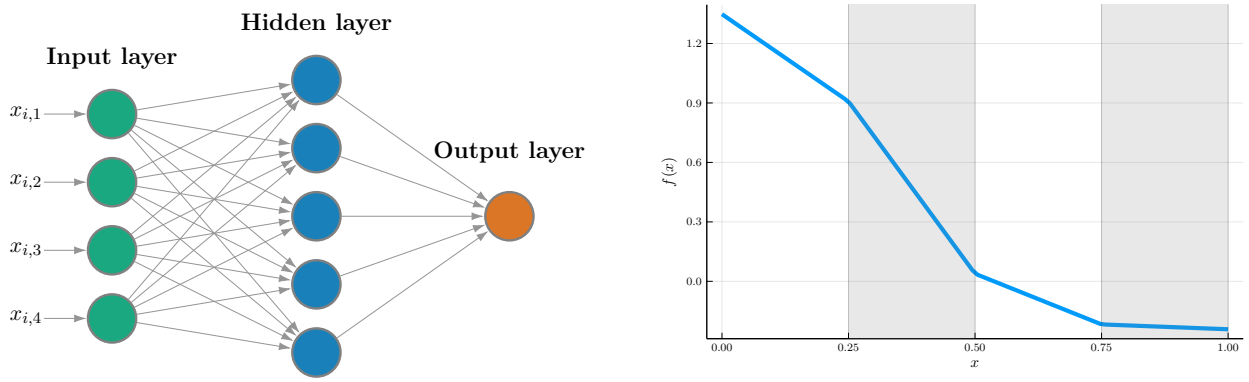
It is convenient to express the SNN in matrix form:

$$f(\mathbf{x}_i, \boldsymbol{\theta}) = \mathbf{w}_n^\top \sigma(\mathbf{W}\mathbf{x}_i + \mathbf{b}) + b_n, \tag{4.12}$$

where $\mathbf{W} = (\mathbf{w}_0, \ldots, \mathbf{w}_{n-1})^\top$ and $\mathbf{b} = (b_0, \ldots, b_{n-1})^\top$, with $\sigma(\cdot)$ applied elementwise.

Panel (a) of Figure 4.5 illustrates the architecture of an SNN. The green nodes correspond to inputs $\mathbf{x}_i$ (*input layer*), the blue nodes correspond to the nonlinear transformations (*hidden layer*), and the orange node represents the final linear combination (*output layer*). Arrows indicate the flow of information, with their associated weights determining the strength of each connection.

Julia implementation. Although Julia provides a high-quality package for neural networks, `Lux.jl`, it is instructive to implement a shallow neural network manually to better understand its building blocks. Listing 4.2 presents a minimal implementation that mirrors the mathematical formulation in Equations (4.9)–(4.11).



(a) Architecture (b) Output $f(x)$

Figure 4.5: Shallow neural network: architecture (left) and realized function (right).

Note: The left panel shows the architecture of a shallow neural network. The green nodes correspond to inputs $\mathbf{x}_i$, the blue nodes are hidden units (each applying an activation function), and the red node represents the output $f(\mathbf{x}_i, \boldsymbol{\theta})$. The parameters $\boldsymbol{\theta}$ include all weights and biases. The right panel shows the realized function $f(x, \boldsymbol{\theta})$ for a given parameter vector $\boldsymbol{\theta}$. The grey areas indicate when the second and fourth hidden units are active.

The function `shallow_nn` takes as input: (i) a vector of inputs $\mathbf{x}_i$, (ii) a matrix of weights $\mathbf{W}$ for the hidden layer, (iii) a vector of hidden-unit biases $\mathbf{b}$, (iv) a vector of output weights $\mathbf{w}_n$, and (v) an output bias b_n. The activation function $\sigma(\cdot)$ is passed as an optional argument. The function returns the output of the network for a given input $\mathbf{x}_i$.

```julia
function shallow_nn(x::AbstractVector{<:Real},
    W::AbstractMatrix{<:Real}, b::AbstractVector{<:Real},
    wₙ::AbstractVector{<:Real}, bₙ::Real; σ::Function = x->max(0,x))
    @assert size(W,2) == length(x)  # ncols of W = length of x
    @assert size(W,1) == length(wₙ) # nrows of W = length of wₙ
    @assert length(b) == size(W,1)  # biases for the hidden units
    return wₙ' * σ.(W * x .+ b)+ bₙ
end
# Convenience: scalar input (d = 1); w is the column of W
function shallow_nn(x::Real,
    w::AbstractVector{<:Real}, b::AbstractVector{<:Real},
    wₙ::AbstractVector{<:Real}, bₙ::Real; σ::Function = x->max(0,x))
    return shallow_nn([x], reshape(w,length(w),1), b, wₙ, bₙ, σ = σ)
end
```

Listing 4.2: Manual implementation of a shallow neural network in Julia.

Below is a Julia REPL demonstration of the shallow network in action. We can verify that the network behaves nonlinearly by evaluating it at two proportional input vectors. Notice that doubling the inputs more than doubles the output.

Julia REPL

```
julia> shallow_nn([0.2, 0.3], ones(4,2), -[0.2,0.4,0.6,0.8],
                  ones(4), 0.0)
0.4  # output for smaller inputs

julia> shallow_nn([0.4, 0.6], ones(4,2), -[0.2,0.4,0.6,0.8],
                  ones(4), 0.0)
2.0  # more than double – nonlinear response
```

Listing 4.3: Nonlinear response in a shallow neural network (Julia REPL example).

This explicit implementation clarifies how the different parameters interact and how the activation function is applied elementwise to the hidden layer. In later sections, we will use `Lux.jl` to construct and train neural networks more efficiently, while retaining the same functional structure.

The role of the activation function. The activation function introduces nonlinearity into the network. If we replace the activation function by the identity, $\sigma(x) = x$, the network collapses to

$$f(\mathbf{x}_i, \boldsymbol{\theta}) = \left(\sum_{j=0}^{n-1} w_{n,j} \mathbf{w}_j \right)^\top \mathbf{x}_i + \left(\sum_{j=0}^{n-1} w_{n,j} b_j + b_n \right),$$

which is just a linear regression model. Thus, without a nonlinear activation, stacking layers provides no additional expressive power.

SNNs as piecewise linear functions. To illustrate the structure of a shallow neural network, consider a one-dimensional input $x_i \in [0, 1]$ and a ReLU activation function. Assume, for simplicity, that all hidden-unit weights are $w_j = 1$. The network can then be written as

$$f(x, \boldsymbol{\theta}) = \sum_{j=0}^{n-1} w_{n,j} \max(0, x - \hat{x}_j) + b_n, \tag{4.13}$$

where $\hat{x}_j \equiv -b_j$ represents a convenient reparametrization of the biases. Ordering the breakpoints as $0 = \hat{x}_0 < \hat{x}_1 < \cdots < \hat{x}_{n-1} < 1 \equiv \hat{x}_n$, we obtain

$$f(x, \boldsymbol{\theta}) = f(\hat{x}_j, \boldsymbol{\theta}) + w_{n,j}(x - \hat{x}_j), \qquad x \in (\hat{x}_j, \hat{x}_{j+1}], \tag{4.14}$$

with $f(\hat{x}_0, \boldsymbol{\theta}) = b_n$.

Equation (4.14) shows that $f(x, \boldsymbol{\theta})$ is piecewise linear, with the breakpoints $\hat{x}_j$ learned by the model. Panel (b) of Figure 4.5 illustrates the behavior of $f(x, \boldsymbol{\theta})$ for $\hat{x}_j \in \{0, 0.25, 0.5, 0.75\}$ and randomly chosen output-layer weights $w_{n,j}$ and bias b_n. The slope of the function changes at each breakpoint, highlighting how each ReLU neuron contributes one "kink" to the fitted function.

When training the SNN, the model must simultaneously learn both the breakpoints (through the biases b_j) and the output-layer parameters $(w_{n,j}, b_n)$. If we fix the breakpoints to be equally spaced, the network collapses to the locally linear interpolant from Section 3.4. Hence, an SNN can be interpreted as a finite-difference method with an *adaptive grid* that learns where to place the nodes.

This adaptive placement of kinks closely resembles mesh refinement in numerical PDE methods. In fact, a shallow neural network with ReLU activation can be interpreted as a special case of a finite element method with a learned, nonuniform mesh.

Important

The SNN is also closely related to a *finite element method* (FEM) with a linear basis. In FEM, the domain is divided into a mesh of elements, and the function is approximated by a linear combination of tent functions:

$$\phi_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{\Delta x}, & x \in [x_{i-1}, x_i], \\ \dfrac{x_{i+1} - x}{\Delta x}, & x \in [x_i, x_{i+1}], \\ 0, & \text{otherwise}, \end{cases} \tag{4.15}$$

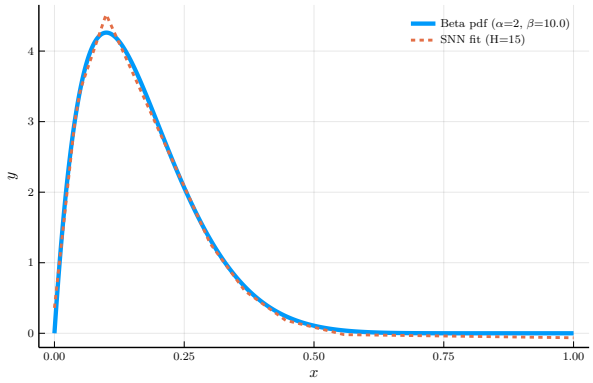
where $\Delta x = x_{i+1} - x_{i-1}$ is the mesh size. Each tent function can be written as a linear combination of ReLU functions:

$$\phi_i(x) = \frac{\sigma(x - x_{i-1}) - 2\sigma(x - x_i) + \sigma(x - x_{i+1})}{\Delta x}. \tag{4.16}$$

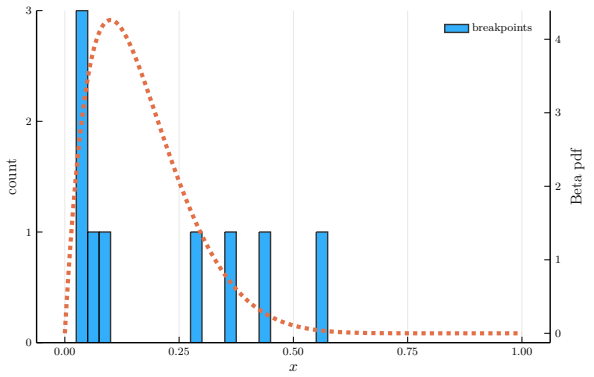
In financial terms, a tent function corresponds to a *butterfly spread*, a combination of calls and puts. For a given set of breakpoints, linear combinations of ReLU functions span the same space as these tent functions. Importantly, the breakpoints are learned by the SNN rather than fixed in advance. Hence, an SNN can be viewed as a FEM with a linear basis and automatic generation of the element grid. For an application of FEM to the stochastic growth model, see McGrattan (1996); for theory and foundations, see Brenner and Scott (2008).

The adaptive choice of breakpoints in action. A key property of ReLU networks is that they *adaptively place their breakpoints* in regions where the target function exhibits strong nonlinearity. To illustrate this, we fit a one–hidden–layer ReLU network to the density of a Beta distribution with parameters $\alpha = 2$ and $\beta = 10$, whose curvature is concentrated at low x.

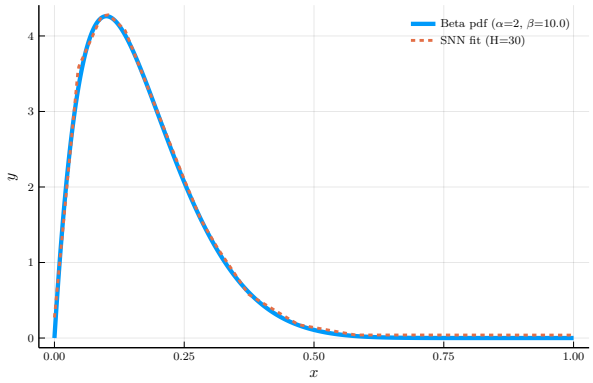
Figure 4.6 summarizes the results for networks with 15, 30, and 100 hidden units. The left column shows the model fit (true function in solid blue; network fit in dashed orange), while the right column displays the corresponding distributions of learned breakpoints $\hat{x}_j \equiv -b_j/w_j$ (restricted to $[0, 1]$). As the number of hidden units increases, the fit improves and the piecewise–linear structure becomes less visible. The breakpoints cluster near the left tail—precisely where the Beta pdf bends most sharply—confirming the network's adaptive



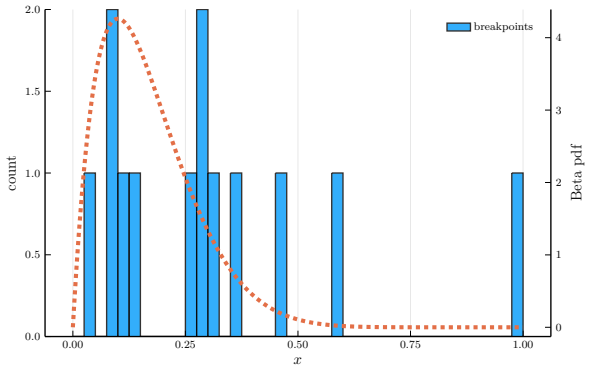
(a) Model fit (15 hidden units)



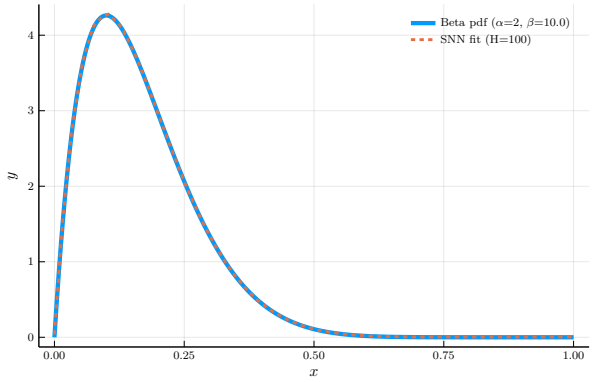
(b) Breakpoint histogram (15 hidden units)



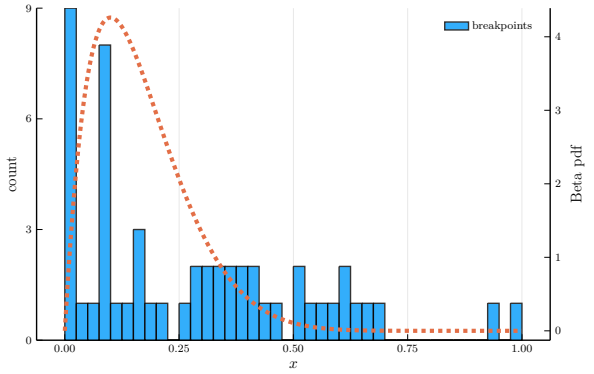
(c) Model fit (30 hidden units)



(d) Breakpoint histogram (30 hidden units)



(e) Model fit (100 hidden units)



(f) Breakpoint histogram (100 hidden units)

Figure 4.6: Beta density fit (left column) and learned breakpoint distributions (right column) for 15, 30, and 100 hidden units.

refinement.

Note

Dying ReLU. Even though the first network has 15 hidden units, only 9 breakpoints appear in the right panel because some learned $\hat{x}_j$ fall outside $[0, 1]$ and those neurons are inactive on the domain. When a ReLU neuron receives negative preactivations for all training points, its output and gradient are both zero and the parameters stop updating—the well-known *dying ReLU* problem. Popular mitigations include using, for example, the GELU (Hendrycks and Gimpel, 2016) and leaky ReLU (Maas et al., 2013) activation functions.

This example shows how a shallow ReLU network learns to concentrate breakpoints in regions of high curvature, effectively creating an *adaptive grid*.

The universal approximation theorem. We have seen that the piecewise–linear structure of a shallow neural network (SNN) enables it to approximate nonlinear functions by adjusting the location and slope of its linear segments. A natural question is: how expressive is this model? What class of functions can an SNN represent or approximate?

The *Universal Approximation Theorem* provides a definitive answer.

Theorem 4.1 (Universal Approximation Theorem)**.** *Any continuous function on a compact subset of $\mathbb{R}^n$ can be approximated to arbitrary precision by a shallow neural network with any non-polynomial activation function.*

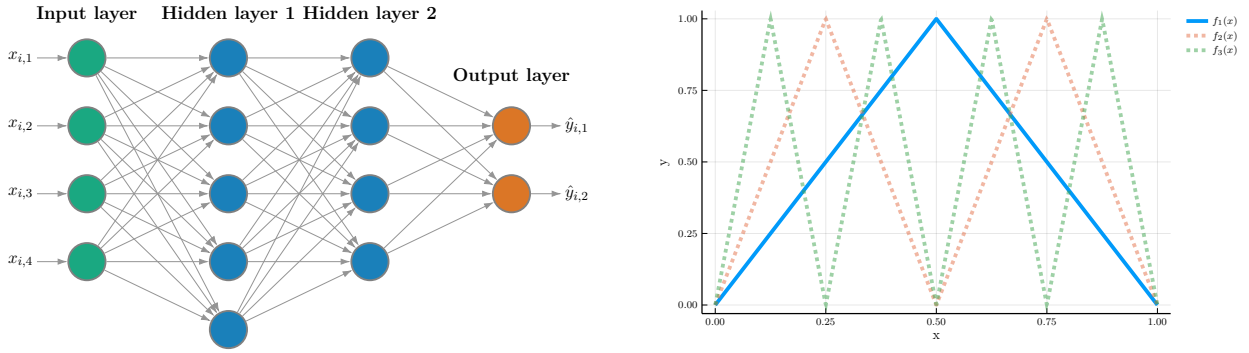
Proof. See Cybenko (1989); Hornik (1991); Leshno et al. (1993). □

This result implies that, with a sufficiently large number of hidden units, an SNN can approximate any continuous function on a compact domain as closely as desired. The version with the ReLU activation function is particularly familiar to financial economists. It is closely related to the *Options Spanning Theorem* of Ross (1976), which states that any payoff can be replicated by a portfolio of call and put options. Each ReLU unit, with its kinked $\max(0, x - \hat{x}_j)$ shape, plays the same role as an option payoff, and an SNN corresponds to a portfolio of these contracts. Hence, a shallow ReLU network provides a *computational analogue* of an options portfolio. From the Option Spanning Theorem, we know options portfolios can replicate any continuous payoff function, so a ReLU SNN is an universal approximator.

While the Universal Approximation Theorem establishes that shallow networks are theoretically sufficient, it does not imply that they are the most efficient way to approximate a function. Achieving high accuracy may require an exponentially large number of hidden units. We therefore turn next to a richer class of models—*deep neural networks*—which can approximate complex functions more efficiently through hierarchical composition.

4.1.4 Deep Neural Networks

A shallow neural network contains a single hidden layer between the input and output layers. *Deep neural networks* (DNNs) extend this architecture by stacking multiple hidden layers



(a) Architecture (b) Output $f(x)$

Figure 4.7: Deep neural network: architecture (left) and realized function (right).

Note: The left panel shows the architecture of a deep neural network. The green nodes correspond to inputs $\mathbf{x}_i$, the blue nodes are hidden units (each applying an activation function), and the orange nodes represent the outputs $f(\mathbf{x}_i, \boldsymbol{\theta})$. The parameters $\boldsymbol{\theta}$ include all weights and biases. The right panel shows the realized functions $f_k(x)$ for $k = 1, 2, 3$.

on top of each other. Each layer applies a linear transformation followed by a nonlinear activation, enabling the network to build progressively richer representations of the input.

Formally, let the n_l-dimensional vector of hidden units at layer l be defined recursively as

$$\mathbf{h}_l(\mathbf{h}_{l-1}, \boldsymbol{\theta}_l) = \sigma(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{b}_l)\,, \qquad l = 1, \ldots, L-1, \tag{4.17}$$

where $\sigma(\cdot)$ is applied elementwise and $\boldsymbol{\theta}_l = (\mathrm{vec}(\mathbf{W}_l)^\top, \mathbf{b}_l^\top)^\top$ collects all parameters of layer l. Here $\mathbf{W}_l$ is an $n_l \times n_{l-1}$ weight matrix, $\mathbf{b}_l$ a vector of biases, and $\mathbf{h}_{l-1}$ the output vector from the previous layer.

The first hidden layer operates directly on the input:

$$\mathbf{h}_0(\mathbf{x}, \boldsymbol{\theta}_0) = \sigma(\mathbf{W}_0 \mathbf{x} + \mathbf{b}_0)\,, \tag{4.18}$$

and the output layer is given by

$$f(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{W}_L \mathbf{h}_{L-1} + \mathbf{b}_L, \tag{4.19}$$

where $\boldsymbol{\theta} = (\boldsymbol{\theta}_0^\top, \ldots, \boldsymbol{\theta}_L^\top)^\top$.

Panel (a) of Figure 4.7 illustrates a DNN with two hidden layers. The depth of the network corresponds to the number of hidden layers, and the width refers to the number of hidden units per layer. Panel (b) shows an example of the functions generated by recursive compositions of such layers.

Example: composition of functions. To understand the expressive power of deep neural networks, consider composing a simple shallow ReLU network with itself. Let

$$f_1(x) = 2\sigma(x) - 4\sigma(x - 0.5), \tag{4.20}$$

which produces a triangular function with a single kink at $x = 0.5$. Composing this function with itself yields

$$f_2(x) = 2\sigma\big(f_1(x)\big) - 4\sigma\big(f_1(x) - 0.5\big), \tag{4.21}$$

and further iterations generate

$$f_k(x) = 2\sigma\big(f_{k-1}(x)\big) - 4\sigma\big(f_{k-1}(x) - 0.5\big), \tag{4.22}$$

each corresponding to a deeper network obtained by composing f_1 with itself k times.

Figure 4.7(b) plots $f_k(x)$ for $k = 1, 2, 3$, showing that the number of linear regions grows exponentially: $f_k(x)$ has $2^k - 1$ kinks in the region $[0, 1]$.

We can represent $f_k(x)$ as a deep neural network with k hidden layers, where layer l is given by

$$\mathbf{h}_l(x) = \begin{bmatrix} \sigma\big(f_{l-1}(x)\big) \\ \sigma\big(f_{l-1}(x) - 0.5\big) \end{bmatrix}.$$

The number of parameters required to represent $f_k(x)$ increases linearly with k. For instance, when $k = 2$, the model uses 4 parameters for the first hidden layer, 6 for the second, and 3 for the output layer, totaling 13 parameters. In general, for k hidden layers, the total parameter count is

$$4 + (k - 1) \times 6 + 3 = 6k + 1.$$

The same function can also be represented as a shallow neural network, but it would require 2^k hidden units to capture all linear regions. Hence, a deep representation achieves the same expressiveness with only a linear number of parameters, while a shallow one requires exponentially more. Depth thus provides an *exponential gain in expressive efficiency*.

Universal approximation theorem revisited. Shallow neural networks are universal approximators: with enough width, they can approximate any continuous function on a compact domain. A complementary result establishes that depth alone can also achieve universal approximation.

Theorem 4.2 (Universal Approximation Theorem for ReLU Networks)**.** *For any Lebesgue-integrable function $f \in L^1(\mathbb{R}^n)$, there exists a ReLU deep neural network with width at most $n + 4$ in every hidden layer that approximates f to arbitrary precision.*

Proof. See Lu et al. (2017). □

While the width theorem (Section 4.1.3) shows that a single wide layer suffices for universal approximation, this depth theorem demonstrates that even networks of modest width can achieve the same expressive power when sufficiently deep. Depth therefore offers an alternative, more parameter-efficient, route to functional richness.

Julia implementation. We now implement a deep neural network in Julia using `Lux.jl`—a high-level, modular package for building and training neural networks. Following the example of $f_1(x)$ from the previous section, we begin by defining a network architecture that maps a one-dimensional input to a single output through two hidden units.

```
# Load packages
using Lux, Random

```

```julia
# Define the network architecture
model = Chain(
    Dense(1 ⇒ 2, Lux.relu), # first hidden layer
    Dense(2 ⇒ 1, identity)  # output layer
)

# Initialize the parameters
rng = Random.Xoshiro(123)
parameters, state = Lux.setup(rng, model)
```

Listing 4.4: Definition of a deep neural network in Julia using Lux.jl.

The command `Chain` defines a sequence of layers. Each `Dense` layer applies an affine transformation followed by an activation function. Because $f_1(x)$ depends on a single input and two hidden units, the first layer is defined as `Dense(1 ⇒ 2, Lux.relu)`. The output layer is a linear combination of these hidden units, implemented as `Dense(2 ⇒ 1, identity)`.

The `model` object stores the network structure and provides the parameter count, as shown in the Julia REPL example below.

Julia REPL

```
julia> model
Chain(
  layer_1 = Dense(1 => 2, relu),                # 4 parameters
  layer_2 = Dense(2 => 1),                      # 3 parameters
)         # Total: 7 parameters,
          #        plus 0 states.
```

Listing 4.5: Lux model definition (Julia REPL example).

We initialize the parameters of the network using a random number generator. In Listing 4.4, we use `Random.Xoshiro(123)` for reproducibility. The function `Lux.setup` initializes both the model parameters and the layer state. The latter is empty in this example but will become relevant later when using normalization or momentum.

Lux stores parameters explicitly as a *nested NamedTuple*, also known as a *parameter tree*. Each layer has its own sub-tuple, or *leaf*, containing its `weight` matrix and `bias` vector. This explicit separation of model definition and parameter storage makes Lux highly composable and compatible with other Julia packages.

Julia REPL

```
julia> parameters
(layer_1 = (weight = Float32[-3.12; 0.15;;], bias =
 ↪  Float32[-0.33, 0.17]),
 layer_2 = (weight = Float32[-1.20 0.96], bias = Float32[0.57]))
```

Listing 4.6: Lux model parameters (Julia REPL example).

Tip

Parameter tree. This tree structure for the model parameters is common across the Julia ecosystem. In particular, it is used not only by the `Lux.jl` package for defining neural networks, but also is also used by the `Optimisers.jl` package for optimization and by the `Zygote.jl` package for automatic differentiation. A similar structure is used in the commonly used Python package `JAX`.

To reproduce the function $f_1(x)$, we can manually assign new values to the network parameters. Lux allows direct updates through a named tuple with the same structure as the initialized parameters:

```
# Defining the parameters
parameters = (layer_1 = (weight = [1.0;1.0;;], bias = [0.0,-0.5]),
              layer_2 = (weight = [2.0 -4.0], bias = [0.0]))

# Checking the model prediction
y_pred, state = model([0.0,0.5,1.0]', parameters, state) # returns
 ↪  [0.0, 1.0, 0.0]
```

Listing 4.7: Updating the parameters of the network.

Defining deeper networks is equally straightforward. Listing 4.8 shows how to extend the model to three hidden layers.

```
# Define the network architecture
model = Chain(
    Dense(1 => 2, Lux.relu), # first hidden layer
    Dense(2 => 2, Lux.relu), # second hidden layer
    Dense(2 => 2, Lux.relu), # third hidden layer
    Dense(2 => 1, identity)  # output layer
)
```

Listing 4.8: Defining a neural network with three hidden layers.

As before, the `model` object summarizes the architecture and parameter count:

Julia REPL

```
julia> model
Chain(
  layer_1 = Dense(1 => 2, relu),                # 4 parameters
```

```
    layer_(2-3) = Dense(2 => 2, relu),          # 12 (6 × 2)
     ↪  parameters
    layer_4 = Dense(2 => 1),                    # 3 parameters
)         # Total: 19 parameters,
          #        plus 0 states.
```

Listing 4.9: Lux model definition with three hidden layers (Julia REPL example).

Having seen how to define and parameterize a network, we next turn to the process of training it.

4.2 Gradient Descent and Its Variants

Having defined a neural network and its parameters, the next step is to *train* the model—that is, to find the parameter values that minimize the loss function on the training data. Because the loss depends on potentially thousands (or millions) of parameters, closed-form solutions are infeasible. Instead, neural networks are trained using iterative, gradient-based optimization methods. Among these, *gradient descent* and its stochastic variants form the foundation of nearly all modern training algorithms.

We begin by revisiting the basic gradient descent algorithm and then introduce its stochastic extensions, which make large-scale learning computationally feasible and can also improve exploration in non-convex loss landscapes.

4.2.1 Stochastic Gradient Descent

In our discussion of linear regression in Section 4.1.2, we saw how to use gradient descent to minimize a loss function. The method updates parameters in the direction opposite to the gradient of the loss, which for the mean squared error (MSE) loss function is given by

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{I} \sum_{i=1}^{I} \big(f(\mathbf{x}_i, \boldsymbol{\theta}) - y_i \big) \, \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_i, \boldsymbol{\theta}). \tag{4.23}$$

Computing this full gradient requires evaluating all I data points at each iteration, which becomes prohibitive for large datasets.

A simple and powerful idea is to approximate the gradient using a random subset of the data. Let $\mathcal{B} \subset \{1, \ldots, I\}$ denote a randomly drawn *mini-batch* of size $B \ll I$. The stochastic gradient descent (SGD) update replaces the full gradient with

$$\nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}; \mathcal{B}) = \frac{1}{B} \sum_{i \in \mathcal{B}} \big(f(\mathbf{x}_i, \boldsymbol{\theta}) - y_i \big) \, \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_i, \boldsymbol{\theta}), \tag{4.24}$$

yielding an unbiased but noisier estimate of the population gradient. Both $\nabla \mathcal{L}(\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}; \mathcal{B})$ estimate $\mathbb{E}[(f(\mathbf{x}_i, \boldsymbol{\theta}) - y_i) \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_i, \boldsymbol{\theta})]$, but the variance of the stochastic estimator is higher. This stochasticity introduces randomness into the updates, which, as we shall see, can be both a cost (increased variance) and a benefit (improved exploration).

A simple example. Consider a toy model where $y_i = \bar{\theta} + \epsilon_i$, with ϵ_i a mean-zero disturbance and $f(\mathbf{x}_i, \boldsymbol{\theta}) = \theta$. In this case, the full-batch and stochastic gradients simplify to

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{I} \sum_{i=1}^{I} (\theta - y_i), \qquad \nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}; \mathcal{B}) = \frac{1}{B} \sum_{i \in \mathcal{B}} (\theta - y_i). \tag{4.25}$$

The following listing implements this example in Julia.

```julia
# True parameter
rng = Random.MersenneTwister(123)
θ_true, sample_size, batch_size = 2.0, 100_000, 32
noisy_sample = θ_true .+ 0.5 .* randn(rng, sample_size)

# Gradient functions: function and mini-batch version
grad_full(θ) = 2 * (θ - mean(noisy_sample))
grad_sgd(θ, B) = 2 * (θ -
    mean(noisy_sample[rand(rng, 1:sample_size, B)]))

# Training loop
η = 0.05
θ_full, θ_sgd = 0.0, 0.0
θ_path_full, θ_path_sgd = Float64[], Float64[]
for t in 1:200
    θ_full -= η * grad_full(θ_full)
    θ_sgd  -= η * grad_sgd(θ_sgd, batch_size)
    push!(θ_path_full, θ_full)
    push!(θ_path_sgd, θ_sgd)
end
```

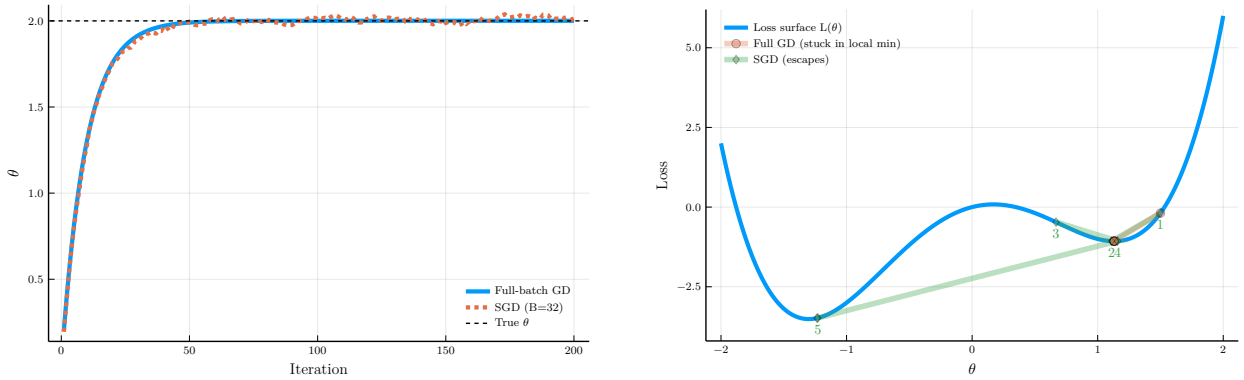
Listing 4.10: Example of stochastic gradient descent in Julia.

Panel (a) of Figure 4.8 compares the trajectories of SGD and full-batch gradient descent on this mean-estimation task. While the stochastic updates fluctuate around the full-batch path, they converge to the same true parameter value $\bar{\theta}$. The advantage of SGD lies in its computational efficiency: each iteration processes only a small mini-batch. In this example, with a sample size of $I = 100{,}000$ and a mini-batch of $B = 32$, each SGD update is roughly $I/B = 3{,}125$ times cheaper to compute.

SGD in a non-convex landscape. The previous example involved a convex loss, where both methods are guaranteed to reach the global minimum. A further advantage of SGD emerges in non-convex problems: its inherent randomness can help escape local minima.

To illustrate, consider the non-convex objective

$$L(\theta) = \theta^4 - 3\theta^2 + \theta, \tag{4.26}$$



(a) SGD vs full-batch GD

(b) SGD in a non-convex landscape

Figure 4.8: Illustrations of stochastic gradient descent: variance in updates (left) and exploration in non-convex objectives (right).

plotted in Panel (b) of Figure 4.8. Both algorithms start from $\theta_0 = 1.5$. The full-batch gradient descent path (orange markers) becomes trapped in a local minimum, whereas the noisy updates of SGD (green markers) allow it to escape and reach the global optimum. This property is especially valuable in neural-network training, where loss landscapes are typically high-dimensional and non-convex.

4.2.2 Momentum, RMSProp, and Adam

Momentum. Gradient descent—and its stochastic variant, SGD—can struggle when the loss surface is *anisotropic*, that is, steep in some directions and flat in others. In such cases, the algorithm may oscillate along steep directions while making slow progress along shallow ones.

To see this, consider a quadratic loss function

$$L(\boldsymbol{\theta}) = \tfrac{1}{2}\boldsymbol{\theta}^\top \Lambda \boldsymbol{\theta}, \qquad \nabla L(\boldsymbol{\theta}) = \Lambda \boldsymbol{\theta}, \tag{4.27}$$

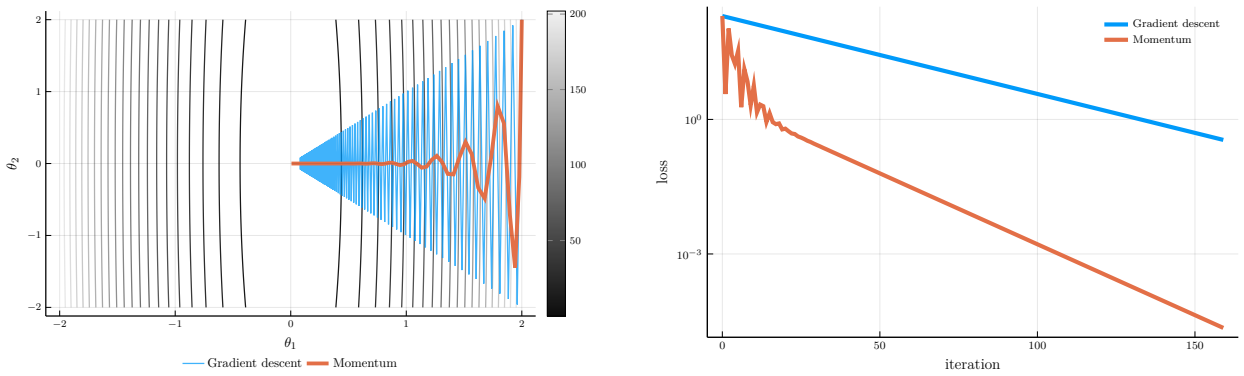
where Λ is diagonal with eigenvalues λ_1 and λ_2. Suppose $0 < \lambda_1 \ll \lambda_2$, so the loss surface is highly elongated. The update rule for gradient descent is then

$$\theta_{n+1,j} = (1 - \eta\lambda_j)\,\theta_{n,j}. \tag{4.28}$$

Convergence requires $|1 - \eta\lambda_j| < 1$, i.e., $0 < \eta < 2/\lambda_j$. If one eigenvalue is large, η must be small to ensure stability—causing very slow convergence along the direction associated with the small eigenvalue. Larger learning rates accelerate convergence but introduce oscillations. This trade-off is one of the classic limitations of basic gradient descent.

A simple yet powerful modification is to introduce *momentum*, which accumulates past gradients to build a smoother update direction. We define the velocity vector $\mathbf{m}$ as an exponentially weighted moving average of past gradients:

$$\mathbf{m}_{n+1} = \beta \mathbf{m}_n + (1 - \beta)\nabla L(\boldsymbol{\theta}_n), \tag{4.29}$$



(a) Trajectories (b) Loss per iteration

Figure 4.9: Momentum vs. gradient descent: parameter trajectories (left) and loss paths (right).

Note: The left panel shows contour lines for the loss function and the trajectories of gradient descent and momentum on a quadratic loss surface. The right panel shows the loss per iteration for both methods. The parameters are initialized at $\boldsymbol{\theta}_0 = (1.0, 1.0)$, with learning rates $\eta_{\text{GD}} = 0.02$ and $\eta_{\text{MB}} = 0.033$, chosen optimally for each method, and momentum coefficient $\beta = 0.67$. The eigenvalues are $\lambda_1 = 1$ and $\lambda_2 = 100$.

where $0 \leq \beta < 1$ is the momentum coefficient. The parameter update becomes

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta\, \mathbf{m}_{n+1}. \tag{4.30}$$

When $\beta = 0$, we recover standard gradient descent; as β increases, recent gradients have more influence, allowing the algorithm to accelerate in directions of consistent descent and to smooth oscillations caused by steep curvature.

The convergence rate now depends jointly on (η, β), and for quadratic objectives it is possible to derive optimal values analytically. Figure 4.9 illustrates the dynamics using a quadratic loss with eigenvalues $\lambda_1 = 1$ and $\lambda_2 = 100$, an extreme case of anisotropy. Panel (a) shows the parameter trajectories for gradient descent and momentum, and Panel (b) plots the corresponding loss per iteration. Momentum dampens oscillations and accelerates convergence, effectively learning the fast direction (θ_2) while maintaining stability along the slow one (θ_1).

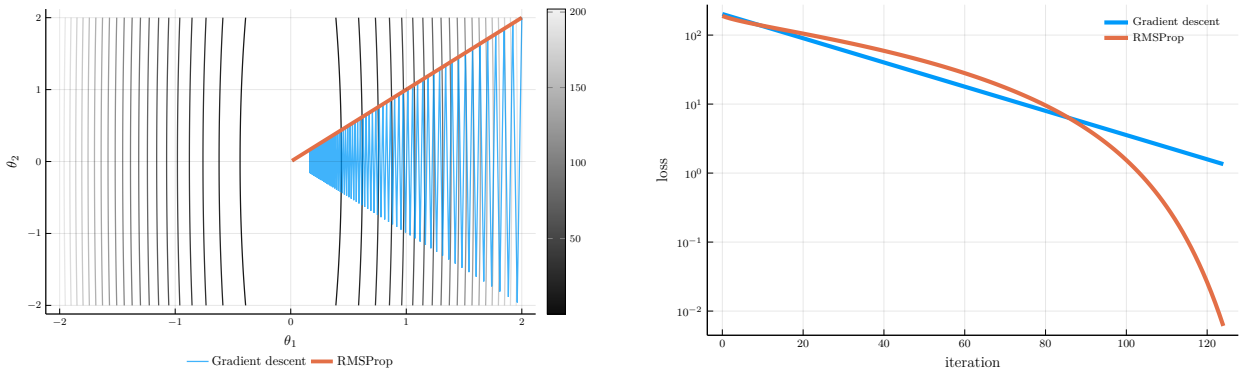
Tip

Nesterov momentum. A common variant, *Nesterov accelerated gradient* (NAG), uses the gradient evaluated at the anticipated next position rather than the current one:

$$\mathbf{m}_{n+1} = \beta \mathbf{m}_n + (1 - \beta)\, \nabla L(\boldsymbol{\theta}_n + \beta \mathbf{m}_n). \tag{4.31}$$

This "look-ahead" correction provides a more accurate update direction and often improves convergence, particularly in highly curved loss landscapes.

RMSProp. While momentum accelerates convergence by accumulating gradients over time, it does not adapt the learning rate to the geometry of the loss surface. When the curvature



(a) Trajectories

(b) Loss per iteration

Figure 4.10: RMSProp vs. gradient descent: parameter trajectories (left) and loss paths (right).

Note: The left panel shows contour lines for the loss function and the trajectories of gradient descent and RMSProp on a quadratic loss surface. The right panel shows the loss per iteration for both methods. The parameters are initialized at $\boldsymbol{\theta}_0 = (2.0, 2.0)$, with learning rates $\eta_{\text{GD}} = 0.02$ and $\eta_{\text{RMSProp}} = 0.02$, and RMSProp decay coefficient $\rho = 0.9$. The eigenvalues are $\lambda_1 = 1$ and $\lambda_2 = 100$.

of the loss function varies across parameters, using a single learning rate η for all directions can be inefficient. Adaptive gradient methods address this issue by scaling the learning rate according to the magnitude of recent gradients.

The *Root Mean Square Propagation* (RMSProp) algorithm keeps an exponentially decaying average of the squared gradients:

$$\mathbf{v}_{n+1} = \rho\, \mathbf{v}_n + (1-\rho)\, (\nabla L(\boldsymbol{\theta}_n))^{\odot 2}, \tag{4.32}$$

where $\rho \in [0, 1)$ controls the decay rate, and $\odot$ denotes elementwise multiplication. The parameter update rule is

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \frac{\nabla L(\boldsymbol{\theta}_n)}{\sqrt{\mathbf{v}_{n+1}} + \epsilon}, \tag{4.33}$$

where ϵ is a small constant to prevent division by zero.

Intuitively, parameters that experience consistently large gradients receive smaller effective learning rates, while those with small gradients receive larger updates.

Figure 4.10 illustrates the behavior of RMSProp in comparison with standard gradient descent on a quadratic loss function. As in the previous example, the loss surface is highly anisotropic with eigenvalues $\lambda_1 = 1$ and $\lambda_2 = 100$. Panel (a) shows that gradient descent oscillates along the steep direction, while RMSProp adapts its step sizes and follows a smoother trajectory toward the minimum. Panel (b) shows the corresponding loss per iteration: RMSProp converges substantially faster and with fewer oscillations.

Tip

Intuition. RMSProp can be viewed as a form of adaptive preconditioning: it rescales each parameter update by the recent variance of its gradient. This makes it particularly

effective in problems with heterogeneous curvature or sparse gradients, such as deep neural network training.

Note

AdaGrad. The RMSProp algorithm builds upon an earlier method known as *AdaGrad* (Duchi et al., 2011). AdaGrad maintains a cumulative sum of squared gradients (without decay):

$$\mathbf{s}_{n+1} = \mathbf{s}_n + (\nabla L(\boldsymbol{\theta}_n))^{\odot 2},$$

and updates parameters according to

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \frac{\nabla L(\boldsymbol{\theta}_n)}{\sqrt{\mathbf{s}_{n+1}} + \epsilon}.$$

Over time, the accumulated squared gradients cause learning rates to shrink monotonically, which may eventually halt learning. RMSProp modifies this idea by using an *exponential moving average*

$$\mathbf{v}_{n+1} = \rho\, \mathbf{v}_n + (1-\rho)\, (\nabla L(\boldsymbol{\theta}_n))^{\odot 2},$$

preventing the effective learning rate from decaying indefinitely and improving performance in long training runs.

Adam. The ideas behind momentum and RMSProp can be combined to yield one of the most widely used optimization algorithms in modern deep learning: the *adaptive moment estimation* (**Adam**) algorithm (Kingma and Ba, 2014). Adam maintains exponentially decaying averages of both the first moment (the mean) and the second moment (the uncentered variance) of the gradients, thereby incorporating the benefits of both momentum and adaptive learning rates.

Let $\mathbf{m}_n$ and $\mathbf{v}_n$ denote the first and second moment estimates of the gradient at iteration n. Their recursive updates are:

$$\mathbf{m}_{n+1} = \beta_1 \mathbf{m}_n + (1-\beta_1)\, \nabla L(\boldsymbol{\theta}_n), \tag{4.34}$$

$$\mathbf{v}_{n+1} = \beta_2 \mathbf{v}_n + (1-\beta_2)\, (\nabla L(\boldsymbol{\theta}_n))^{\odot 2}, \tag{4.35}$$

where β_1 and β_2 are the exponential decay rates for the first and second moments, respectively, and $\odot$ denotes elementwise multiplication. Both moment estimates are initialized at zero, $\mathbf{m}_0 = \mathbf{v}_0 = \mathbf{0}$.

Since these moving averages are biased toward zero at the beginning of training, Adam applies bias-correction terms:

$$\hat{\mathbf{m}}_n = \frac{\mathbf{m}_n}{1-\beta_1^n}, \qquad \hat{\mathbf{v}}_n = \frac{\mathbf{v}_n}{1-\beta_2^n}. \tag{4.36}$$

The final parameter update combines both corrections:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \frac{\hat{\mathbf{m}}_{n+1}}{\sqrt{\hat{\mathbf{v}}_{n+1}} + \epsilon}, \tag{4.37}$$

where ϵ is a small constant to prevent division by zero. Typical default values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

Adam adapts the learning rate of each parameter individually based on both the mean and variance of its past gradients. Parameters with consistently large gradients receive smaller effective learning rates, while those with small or noisy gradients receive larger updates. This makes Adam robust, fast-converging, and highly effective in large-scale or ill-conditioned problems.

Note

AdamW. In practice, weight regularization is often applied together with Adam. A naïve approach adds an L^2 penalty directly to the gradient, which scales the regularization term by $1/\sqrt{\hat{\mathbf{v}}_{n+1}}$, causing parameter-dependent shrinkage. The *AdamW* variant (Loshchilov and Hutter, 2019) decouples weight decay from the adaptive rescaling:

$$\boldsymbol{\theta}_{n+1} = (1 - \eta\lambda)\boldsymbol{\theta}_n - \eta \frac{\hat{\mathbf{m}}_{n+1}}{\sqrt{\hat{\mathbf{v}}_{n+1}} + \epsilon}, \tag{4.38}$$

where λ is the weight decay coefficient. This decoupled formulation has become the standard default in modern deep learning frameworks.

4.2.3 Julia Implementation

We now illustrate how to use the optimisers discussed above in Julia. All examples use the `Optimisers.jl` library, which provides efficient implementations of the core algorithms described in this section. Table 4.1 summarizes the main optimisers available in `Optimisers.jl`, together with their update rules, internal state variables, and key hyperparameters.

Anisotropic non-convex loss function. As an illustrative example, we consider a high-dimensional, anisotropic, and non-convex loss function. Although simple, this example highlights some of the challenges faced in deep-learning optimisation.

We begin with the fourth-order polynomial from Section 4.2.1:

$$\ell(\theta) = \theta^4 - 3\theta^2 + \theta. \tag{4.39}$$

Given a parameter vector $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_n)^\top$, the overall loss function is defined as

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} w_i \, \ell(\theta_i), \tag{4.40}$$

where $w_i \geq 0$ are weights satisfying $\sum_{i=1}^{n} w_i = 1$.

Listing 4.11 shows the Julia implementation of this loss function.

```
ℓ(θ) = θ^4 - 3θ^2 + θ   # fourth-order polynomial
```

Table 4.1: Core Optimisers in `Optimisers.jl`

Optimiser	**Update (per step)**	**State**	**Key hyperparameters (typical)**
`Descent` (SGD)	$\theta \leftarrow \theta - \eta g$	—	η
Baseline; good generalisation but sensitive to conditioning.			
`Momentum`	$v \leftarrow \beta v + (1-\beta) g;\ \theta \leftarrow \theta - \eta v$	v	$\eta,\ \beta \approx 0.9$
Smooths noisy gradients; accelerates in flat directions; may oscillate if under-damped.			
`Nesterov`	look-ahead momentum variant	v	$\eta,\ \beta$
Improves damping and acceleration relative to momentum.			
`AdaGrad`	$h \leftarrow h + g^{\odot 2};\ \theta \leftarrow \theta - \eta\, g./\sqrt{h+\epsilon}$	h	$\eta,\ \epsilon$
Strong early adaptivity; step size decays monotonically (may stall). Suitable for sparse features.			
`RMSProp`	$v \leftarrow \rho v + (1-\rho) g^{\odot 2};\ \theta \leftarrow \theta - \eta\, g./\sqrt{v+\epsilon}$	v	$\eta,\ \rho \approx 0.9,\ \epsilon$
Adaptive per-coordinate step size; robust for heteroskedastic gradients.			
`Adam`	$m \leftarrow \beta_1 m + (1-\beta_1) g;\ v \leftarrow \beta_2 v + (1-\beta_2) g^{\odot 2};$ $\hat{m} = m/(1-\beta_1^t);\ \hat{v} = v/(1-\beta_2^t);$ $\theta \leftarrow \theta - \eta\, \hat{m}./(\sqrt{\hat{v}} + \epsilon)$	$m,\ v,\ t$	$\eta \approx 10^{-3}$ $(\beta_1, \beta_2) = (0.9, 0.999)$ ϵ
Momentum + RMSProp with bias correction; strong general-purpose default; may overfit relative to SGD.			
`AdamW`*	Decoupled decay: $\theta \leftarrow (1-\eta\lambda)\theta$; then Adam step as above	$m,\ v,\ t$	$\eta,\ (\beta_1, \beta_2),\ \lambda,\ \epsilon$
Adam with separate weight-decay term; better generalisation; common recommended default.			

* "Decoupled" means the shrinkage $(1-\eta\lambda)\theta$ is applied *separately* from the adaptive step; unlike coupled L_2 regularisation, it is not scaled by $\sqrt{v}$.

```
weights = range(1,100, length = 10)  # weights
loss(θ) = dot(weights, ℓ.(θ))/sum(weights) # loss function
```

Listing 4.11: Loss function.

Gradient descent. We now use gradient descent to minimise the loss function. Listing 4.12 shows a generic implementation of a loss optimiser in Julia that can accommodate any optimisation algorithm from the `Optimisers.jl` library.

```
# Loss optimisation function
function loss_optimiser(loss, θ0, opt; steps=1_000, tol=1e-8)
    θ = deepcopy(θ0) # make a copy to avoid in-place modifications
    st  = Optimisers.setup(opt, θ)  # builds a "state tree"
    losses = Float64[]; gnorms = Float64[]
    for _ in 1:steps
        ℓ, back = Zygote.pullback(loss, θ) # pullback of the loss
        g = first(back(1.0)) # compute gradient
        push!(losses, ℓ); push!(gnorms, norm(g))
        if gnorms[end] ≤ tol; break; end
        st, θ = Optimisers.update(st, θ, g) # update state/params
    end
    return θ, (losses=losses, grad_norms=gnorms)
end
```

Listing 4.12: Generic loss optimisation routine in Julia.

The function `loss_optimiser` takes as inputs a loss function `loss`, an initial parameter vector $\boldsymbol{\theta}_0$, and an optimiser object `opt`. Inside the routine, we first create a copy of the initial parameters to avoid in-place modifications. We then construct the optimiser's *state tree* using `Optimisers.setup`, which stores all internal variables associated with the optimiser. This structure is empty for simple algorithms like gradient descent, but non-empty for methods such as Momentum, RMSProp, and Adam, which keep running averages of past gradients.

The main optimisation loop proceeds as follows: at each iteration, we evaluate the loss and its gradient using `Zygote.pullback`, store the loss and gradient norm, and update the parameters via `Optimisers.update`. The loop terminates once the gradient norm falls below a specified tolerance or a maximum number of steps is reached. For illustration, we can instantiate gradient descent as follows:

```
opt = Optimisers.Descent(η)
θ_opt, stats = loss_optimiser(loss, θ0, opt, steps = 1000)
```

This example uses the `Descent` optimiser with learning rate η. Because the optimiser is passed as an argument, we can easily replace it with any other algorithm–such as Momentum, RMSProp, or Adam–without modifying the `loss_optimiser` function itself.

Changing optimisers. It is straightforward to switch between different optimisers. Because the `loss_optimiser` function takes the optimiser as an argument, we only need to modify the optimiser definition itself. For example, to use Adam instead of gradient descent, we simply replace:

```
opt = Optimisers.Adam(η, (β1, β2), ϵ)
```

Listing 4.13: Adam optimiser.

Adam introduces three additional hyperparameters: β_1 and β_2, which are the exponential decay rates for the first and second moments of the gradients, and ϵ, a small constant used for numerical stability. Unlike gradient descent, Adam's state tree is non-empty. The Julia REPL example below shows the structure of the state tree after one iteration.

Julia REPL

```
julia> st = Optimisers.setup(opt, θ)
Leaf(Adam(eta=0.01, beta=(0.9, 0.999), epsilon=1.0e-8),
     ([-0.0176, -0.211, -0.405, -0.599, -0.793, -0.987, -1.181,
      ↪ -1.375, -1.569, -1.762],
      [3.1e-5, 0.0045, 0.0164, 0.0359, 0.0629, 0.0974, 0.1394,
       ↪  0.1890, 0.2460, 0.3106],
      (0.81, 0.998001)))
```

Listing 4.14: Adam state tree (Julia REPL example).

The state tree for Adam contains three elements: (i) the optimiser itself, (ii) a tuple with the first and second moment estimates $(\mathbf{m}_t, \mathbf{v}_t)$, and (iii) a tuple with the accumulated decay rates (β_1^t, β_2^t).

Comparison of optimisers. We now compare the performance of different optimisers on the same objective function. The initial condition is set to $\boldsymbol{\theta}_0 = (-3.0, -3.0, \ldots, -3.0)$, and the optimal solution is $\boldsymbol{\theta}^* = (-1.30, -1.30, \ldots, -1.30)$.

Figure 4.11 summarises the results. Gradient descent struggles to update parameters along the low-curvature direction—similar to the behaviour observed in Figure 4.9. Momentum and Nesterov momentum tend to overshoot in the high-curvature coordinates and can get pulled toward the positive local minimum. By contrast, RMSProp, Adam, and AdamW adapt step sizes per coordinate and converge reliably to the global minimum across all parameters.

Fitting a DNN. As a final example, we train a deep neural network (DNN) to approximate a nonlinear, high-dimensional function. The goal is to illustrate how a DNN can learn a complex mapping and generalize beyond the training data.

We begin with a one-dimensional polynomial of degree 5,

$$p(x) = \prod_{i=1}^{5} (x - r_i), \tag{4.41}$$

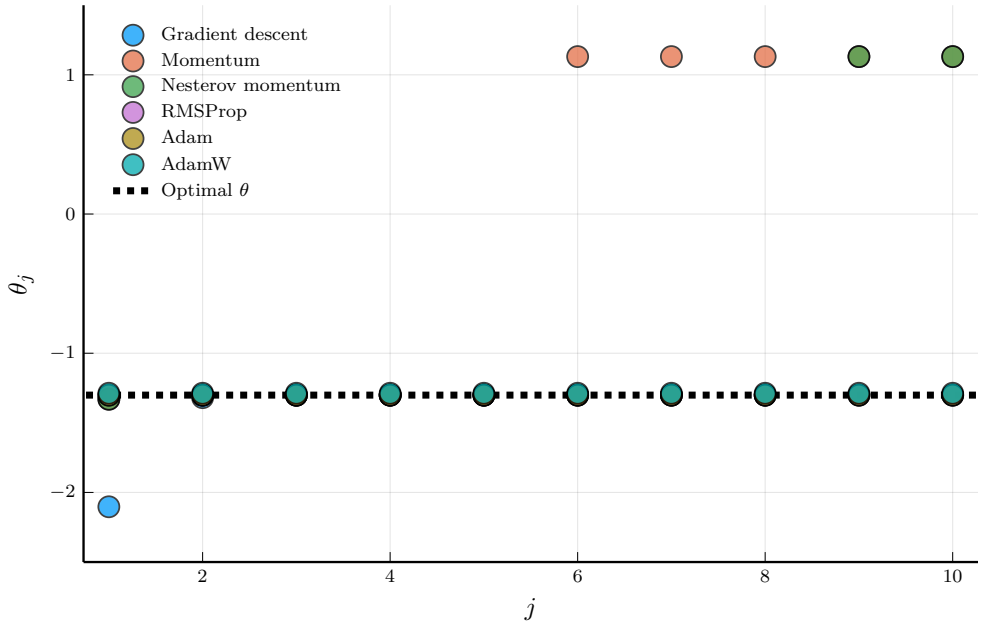


Figure 4.11: Comparison of optimisation algorithms.

where the roots r_i are drawn from a standard normal distribution. We then define a multivariate extension as the average of the univariate polynomials evaluated componentwise:

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} p(x_i), \tag{4.42}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^\top$. Listing 4.15 shows the Julia implementation of the multivariate polynomial.

```julia
# Constructing function f
rng  = Xoshiro(0)           # pseudo random number generator
roots = randn(rng, 5)       # polynomial roots
p(x) = prod(x .- roots)     # univariate polynomial
f(x) = mean(p.(x))          # multivariate version

# Random samples
n_states, sample_size = 10, 100_000
x_samples = rand(rng, Uniform(-1,1), (n_states,sample_size))
y_samples = [f(x_samples[:,i]) for i = 1:sample_size]'
```

Listing 4.15: Multivariate polynomial function.

We fix $n = 10$ and draw 100,000 sample pairs $(\mathbf{x}_i, y_i)$, where $y_i = f(\mathbf{x}_i)$. The input array `x_samples` is a $10 \times 100{,}000$ matrix, with each column representing one observation. Following Lux's convention, columns correspond to samples and rows to features. The output array `y_samples` is a $1 \times 100{,}000$ matrix containing the target values.

We next define a DNN with two hidden layers of 32 units each and GELU activations. The first layer maps the 10-dimensional input to a 32-dimensional hidden representation. Listing 4.16 shows the Julia implementation.

```
# Archictecture
layers = [n_states, 32, 32, 1]
model = Chain(
    Dense(layers[1] ⇒ layers[2], Lux.gelu),
    Dense(layers[2] ⇒ layers[3], Lux.gelu),
    Dense(layers[3] ⇒ layers[4], identity)
)
```

Listing 4.16: DNN architecture.

The model summary is:

Julia REPL

```
julia> model
Chain(
  layer_1 = Dense(10 => 32, gelu_tanh),       # 352 parameters
  layer_2 = Dense(32 => 32, gelu_tanh),       # 1_056 parameters
  layer_3 = Dense(32 => 1),                   # 33 parameters
)         # Total: 1_441 parameters,
          #        plus 0 states.
```

Listing 4.17: DNN architecture (Julia REPL example).

We initialize the parameters and choose the optimisation algorithm, as shown in Listing 4.18.

```
# Initialize the parameters
parameters, layer_states = Lux.setup(rng, model)

# Use Adam for optimization
learning_rate = 1e-3
opt = Adam(learning_rate)
opt_state = Optimisers.setup(opt, parameters)
```

Listing 4.18: DNN initialization.

The optimisation state stores quantities needed by the optimiser (e.g., first and second moment estimates for Adam). This state is a nested named tuple with the same structure as the model:

Julia REPL

```
julia> keys(parameters), keys(layer_states)
((:layer_1, :layer_2, :layer_3), (:layer_1, :layer_2, :layer_3))
```

Listing 4.19: DNN parameters and layer states (Julia REPL example).

We now define the loss function (Listing 4.20). Each call samples a random mini-batch of size 128, computes the network's predictions, and returns the mean squared error (with the usual $\frac{1}{2}$ factor) together with updated layer states.

```
# Loss function
function loss_fn(p, ls; batch_size = 128)
    ind = rand(rng,1:sample_size,batch_size)
    y_prediction, new_ls = model(x_samples[:,ind], p, ls)
    loss = 0.5 * mean( (y_prediction-y_samples[:,ind]).^2 )
    return loss, new_ls
end
```

Listing 4.20: DNN loss function.

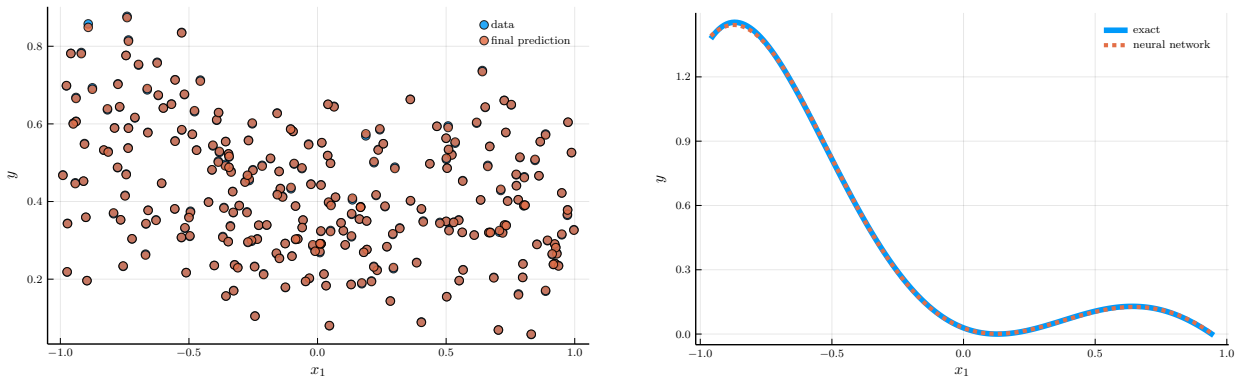
We are now ready to train the network. Listing 4.21 shows the training loop.

```
# train loop
loss_history = []
n_steps = 300_000
for _ in 1:n_steps
    loss, layer_states = loss_fn(parameters, layer_states)
    grad = gradient(p->loss_fn(p, layer_states)[1], parameters)[1]
    opt_state, parameters = Optimisers.update(opt_state, parameters,
 ↪  grad)
    push!(loss_history, loss)
    if epoch % 5000 == 0
        println("Epoch: $epoch, Loss: $loss")
    end
end
```

Listing 4.21: DNN training loop.

Note

Epochs and iterations. In machine learning, an *epoch* refers to one complete pass through the training dataset. With mini-batches, the optimiser performs multiple updates per epoch—one per mini-batch. An *iteration* is a single parameter update. Thus, if the



(a) Sample scatter

(b) Target polynomial

Figure 4.12: Data and target function for the DNN example.

dataset is split into I/B mini-batches of size B, one epoch consists of I/B iterations. In this example, instead of looping over the full dataset, we draw a fresh random mini-batch at each iteration (pure stochastic training).

Figure 4.12 presents the results. Panel (a) shows a random subsample of 256 data points (blue) and the corresponding DNN predictions (red); the two are nearly indistinguishable, indicating an excellent in-sample fit.

A natural concern is that the network might simply be *memorizing* the training data. To test this, we evaluate the network in the special case $x_1 = x_2 = \cdots = x_n = x$, for which $f(\mathbf{x})$ reduces to $p(x)$—a configuration that never appears in the random training sample. Panel (b) compares the true function (solid blue) with the DNN prediction (dashed red). Even without seeing this zero-probability case during training, the network's predictions closely match $p(x)$, demonstrating strong generalization beyond the training domain.

4.3 Automatic Differentiation and Backpropagation

In the previous section, we introduced several optimisation algorithms for training neural networks. All these methods are *gradient-based*—they rely on the derivatives of the loss function with respect to the model parameters to update the network. Hence, the efficient and accurate computation of gradients is a central ingredient in modern machine learning.

Automatic differentiation (AD) provides a systematic and numerically stable way to compute these gradients. Unlike symbolic differentiation, which manipulates algebraic expressions, or numerical differentiation, which approximates derivatives using finite differences, AD applies the chain rule algorithmically on a sequence of elementary operations. This makes it both exact (up to machine precision) and computationally efficient.

There are two main modes of automatic differentiation:

1. **Forward mode AD**, in which derivatives are propagated *forward* through the computational graph from inputs to outputs; and

2. **Reverse mode AD**, in which derivatives are propagated *backward* from outputs to inputs.

Reverse mode AD is also known as *backpropagation* and forms the foundation of training algorithms for neural networks. We begin with forward mode AD, which is conceptually simpler and helps to build intuition.

4.3.1 Forward-Mode Automatic Differentiation

To build intuition for forward-mode AD, consider the first-order approximation of a function $f(x)$ around a point x_0:

$$f(x) = f(x_0) + f'(x_0)\,\epsilon + \mathcal{O}(\epsilon^2), \tag{4.43}$$

where $x = x_0 + \epsilon$ for a small perturbation ϵ, and $\mathcal{O}(\epsilon^2)$ collects higher-order terms.

A first-order approximation is therefore characterized by two quantities: (i) the function value $f(x_0)$ and (ii) its derivative $f'(x_0)$. Our goal is to teach the computer how to propagate these two pieces of information through any composition of functions.

Example: The product rule. Suppose we know the linear approximations of two functions,

$$f(x) = f(x_0) + f'(x_0)\,\epsilon + \mathcal{O}(\epsilon^2), \tag{4.44}$$

$$g(x) = g(x_0) + g'(x_0)\,\epsilon + \mathcal{O}(\epsilon^2). \tag{4.45}$$

We want to find the linear approximation of their product $h(x) = f(x)g(x)$. Multiplying the two expansions and ignoring higher-order terms gives:

$$h(x) = \underbrace{f(x_0)g(x_0)}_{h(x_0)} + \underbrace{\big[f'(x_0)g(x_0) + f(x_0)g'(x_0)\big]}_{h'(x_0)}\,\epsilon + \mathcal{O}(\epsilon^2). \tag{4.46}$$

The term in brackets is precisely the *product rule* of derivatives. Thus, knowing the pairs $(f(x_0), f'(x_0))$ and $(g(x_0), g'(x_0))$ allows us to compute the pair $(h(x_0), h'(x_0))$.

Example: the chain rule. Now consider a composition $h(x) = g(f(x))$. Substituting the linear approximation of $f(x)$ into $g(\cdot)$ gives:

$$h(x) = g(f(x_0) + f'(x_0)\,\epsilon + \mathcal{O}(\epsilon^2)) \tag{4.47}$$

$$= \underbrace{g(f(x_0))}_{h(x_0)} + \underbrace{g'(f(x_0))f'(x_0)}_{h'(x_0)}\,\epsilon + \mathcal{O}(\epsilon^2), \tag{4.48}$$

which recovers the familiar *chain rule*. Given $(f(x_0), f'(x_0))$ and $(g(f(x_0)), g'(f(x_0)))$, we can again compute $(h(x_0), h'(x_0))$.

Example: linear combinations. For completeness, a linear combination $h(x) = af(x) + bg(x)$ expands as:

$$h(x) = \big[af(x_0) + bg(x_0)\big] + \big[af'(x_0) + bg'(x_0)\big]\epsilon + \mathcal{O}(\epsilon^2). \tag{4.49}$$

This recovers the linear combination rule of derivatives. Given $(f(x_0), f'(x_0))$ and $(g(x_0), g'(x_0))$, we can compute $(h(x_0), h'(x_0))$.

Dual numbers. We have seen that by combining a few simple rules—product, chain, and linear combination—we can compute the linear approximation of arbitrarily complex functions. This idea can be implemented in a computer using the concept of *dual numbers*.

Analogous to complex numbers, we represent a dual number as

$$u = a + b\,\epsilon, \tag{4.50}$$

where the *primal part* a stores the function value and the *dual part* b stores its derivative. The defining property of the dual unit ϵ is

$$\epsilon^2 = 0,$$

which encodes the fact that we neglect all higher-order terms when computing a first-order approximation.

Using this property, the product of two dual numbers, $u = a + b\epsilon$ and $v = c + d\epsilon$, is

$$u \times v = (a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\,\epsilon, \tag{4.51}$$

which mirrors the product rule of derivatives.

Similarly, given a real function $g(x)$, we can extend it to dual numbers by defining

$$g(a + b\epsilon) = g(a) + g'(a)\,b\,\epsilon. \tag{4.52}$$

This extension automatically implements the chain rule. Linear combinations and quotients follow analogously.

In essence, forward-mode AD works by performing ordinary arithmetic on dual numbers rather than on real numbers. Each operation propagates both the *value* and its *derivative*, yielding exact first-order derivatives up to machine precision.

Julia implementation. We can implement a basic automatic differentiation engine in Julia with only a few lines of code. Listing 4.22 defines the dual number type. At this stage, it simply stores the *primal* and *dual* parts of a function—its value and derivative at a given point.

```julia
# Define the dual number type
struct D <: Number
    v::Real # primal part (value of the function)
    d::Real # dual part (derivative of the function)
end
```

Listing 4.22: Dual number type.

Every struct in Julia has an associated *constructor* function that creates an instance of that type. Listing 4.23 shows how to use the constructor to instantiate a dual number.

Julia REPL

```
julia> u = DualNumber(1.0, 1.0)
DualNumber(1.0, 1.0)
```

Listing 4.23: Dual number constructor (Julia REPL example).

Given this new number type, we can now define the basic operations on dual numbers. Listing 4.24 shows the Julia implementation of addition, subtraction, multiplication, and division, as well as elementary functions such as exponentials and trigonometric functions.

```
1  # How to perform basic operations on dual numbers
2  import Base: +,-,*,/, convert, promote_rule, exp, sin, cos, log
3  +(x::D, y::D)  = D(x.v + y.v, x.d + y.d)
4  -(x::D, y::D)  = D(x.v - y.v, x.d - y.d)
5  *(x::D, y::D)  = D(x.v * y.v, x.d * y.v + x.v * y.d)
6  /(x::D, y::D)  = D(x.v / y.v, (x.v * y.d - y.v * x.d) / y.v^2)
7  exp(x::D)      = D(exp(x.v), exp(x.v) * x.d)
8  log(x::D)      = D(log(x.v), 1.0 / x.v * x.d)
9  sin(x::D)      = D(sin(x.v), cos(x.v) * x.d)
10 cos(x::D)      = D(cos(x.v), -sin(x.v) * x.d)
11 promote_rule(::Type{D}, ::Type{<:Number}) = D
12 Base.show(io::IO,x::D) = print(io,x.v," + ",x.d," ϵ")
```

Listing 4.24: Basic operations on dual numbers.

The first line in Listing 4.24 imports Julia's `Base` module, which defines the built-in arithmetic for standard number types. We then use Julia's *multiple dispatch* to extend these operations to dual numbers. Julia automatically selects the correct method depending on the argument types—real, complex, or dual. Lines 3–6 define addition, subtraction, multiplication, and division. Lines 7–10 extend elementary functions such as exponentials, logarithms, sines, and cosines. Line 11 specifies the promotion rule so that operations between reals and duals behave naturally. Finally, line 12 customizes how dual numbers are displayed to match our mathematical notation for first-order approximations.

At this point, we already have a working automatic differentiation engine written in fewer than twenty lines of code. We can test it on two example functions, $f(x) = \exp(x^2)$ and $g(x) = \cos(x^3)$:

```
# Define f and g functions and their exact derivatives
f(x) = exp(x^2)
g(x) = cos(x^3)
fprime_exact(x) = exp(x^2) * 2 * x
gprime_exact(x) = -sin(x^3) * 3 * x^2
```

Listing 4.25: Test functions.

These are ordinary Julia functions—no modifications are needed for them to work with dual numbers. We can now evaluate them at the dual inputs $u = 1.0 + 1.0\epsilon$ and $v = 2.0 + 1.0\epsilon$:

Julia REPL

```
julia> u = D(1.0, 1.0)
1.0 + 1.0 ϵ
julia> v = D(2.0, 1.0)
2.0 + 1.0 ϵ
```

Listing 4.26: Dual number instantiation (Julia REPL example).

These correspond to the first-order approximations of the identity function $h(x) = x_0 + 1.0\,\epsilon$ at $x_0 = 1.0$ and $x_0 = 2.0$. Evaluating the functions at these points gives:

Julia REPL

```
julia> f(u)
2.718281828459045 + 5.43656365691809 ϵ
julia> fprime_exact(1.0)
5.43656365691809
julia> g(v)
0.5403023058681398 + -11.872298959480581 ϵ
julia> gprime_exact(2.0)
-11.872298959480581
```

Listing 4.27: Test functions evaluation (Julia REPL example).

Notice that Julia evaluates these functions on dual numbers without any explicit modification of the function definitions—multiple dispatch ensures that each operation uses the appropriate rule. This is precisely how forward-mode automatic differentiation works: every arithmetic operation propagates both the value and its derivative automatically.

To compute a derivative at a given point, we simply construct the dual number $u = x_0 + 1.0\epsilon$, evaluate $f(u)$, and extract the dual part. It is convenient to wrap this logic in a helper function so the dual arithmetic remains hidden from the user. Listing 4.28 implements this helper.

```
# Compute derivative
function derivative(f::Function, x::Real)
    u = D(x, 1.0) # construct the dual number u = x_0 + 1.0 * ϵ
    return f(u).d # extract the dual part of the result
end
```

Listing 4.28: Derivative helper function.

With this function, computing derivatives is immediate:

Julia REPL

```
julia> derivative(f, 1.0)
5.43656365691809
julia> derivative(g, 2.0)
-11.872298959480581
```

Listing 4.29: Derivative computation (Julia REPL example).

In a single evaluation, the function returns both its value and derivative—capturing the essence of forward-mode automatic differentiation.

Tip

The ForwardDiff.jl package. `ForwardDiff.jl` provides a high-performance forward-mode AD engine in Julia. It implements the dual-number ideas from this section and extends them to vector-valued functions and multiple outputs.

Multiple dimensions. Suppose $\mathbf{y} = \mathbf{f}(\mathbf{x}) \in \mathbb{R}^m$ with $\mathbf{x} \in \mathbb{R}^n$. Introduce a vector dual number

$$\mathbf{x} = \mathbf{x}_0 + \epsilon\, \mathbf{v}, \tag{4.53}$$

where $\mathbf{x}_0 \in \mathbb{R}^n$ is the primal part and $\mathbf{v} \in \mathbb{R}^n$ is the *tangent direction*.

The first-order approximation of $\mathbf{f}$ is

$$\mathbf{f}(\mathbf{x}) = \underbrace{\mathbf{f}(\mathbf{x}_0)}_{\text{primal}} + \underbrace{\mathbf{J}\mathbf{f}(\mathbf{x}_0)\,\mathbf{v}}_{\text{Jacobian–vector product (JVP)}} \epsilon + \mathcal{O}(\epsilon^2), \tag{4.54}$$

where $\mathbf{J}\mathbf{f}(\mathbf{x}_0) \in \mathbb{R}^{m\times n}$ has (i,j) entry $\partial f_i/\partial x_j(\mathbf{x}_0)$. Thus, applying $\mathbf{f}$ to $(\mathbf{x}_0, \mathbf{v})$ returns both the value $\mathbf{f}(\mathbf{x}_0)$ and the directional derivative $\mathbf{J}\mathbf{f}(\mathbf{x}_0)\mathbf{v}$.

Cost: JVP vs. full Jacobian. For $f : \mathbb{R}^n \to \mathbb{R}^m$, a JVP needs only a *single* forward pass with the tangent $\mathbf{v}$:

$$\text{cost(JVP)} \sim \mathcal{O}(\text{cost}(f)).$$

Forming the full Jacobian by forward mode requires n passes (one per input direction):

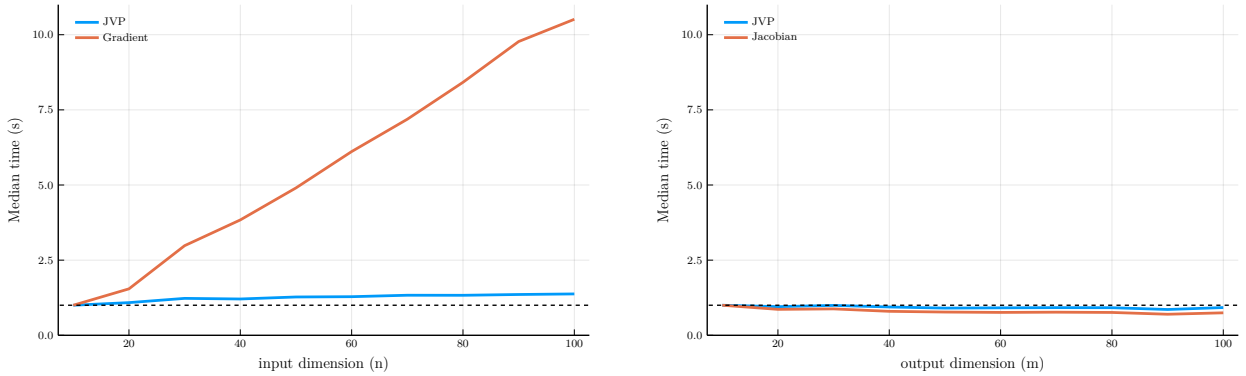
$$\text{cost(Jacobian via forward mode)} \sim \mathcal{O}(n\,\text{cost}(f))\,.$$

This gap is pronounced when n is large.

To illustrate, we compare JVPs to full Jacobians on the test function

$$f_i(\mathbf{x}) = \exp\left(-\tfrac{1}{n}\sum_{i=1}^{n}\sqrt{x_i}\right) + i, \qquad \mathbf{f}(\mathbf{x}) = \big[f_1(\mathbf{x}), \ldots, f_m(\mathbf{x})\big]^\top \in \mathbb{R}^m. \tag{4.55}$$

Listing 4.30 shows the `ForwardDiff.jl` code to compute JVPs and full Jacobians.



(a) Input dimension scaling

(b) Output dimension scaling

Figure 4.13: Computational efficiency of JVPs vs. full Jacobian/gradient in `ForwardDiff.jl`.

Left: median time vs. input dimension n for $f : \mathbb{R}^n \to \mathbb{R}$. The gradient (full Jacobian transpose) scales with n, while a single JVP stays near the cost of one evaluation (dashed baseline). Right: median time vs. output dimension m for $\mathbf{g} : \mathbb{R} \to \mathbb{R}^m$. With $n=1$, the full Jacobian and a JVP have similar cost. Times are normalized by function evaluation time.

```julia
# Test function
f(x; n = 1) = [exp(-mean(sqrt.(x)))+i for i = 1:n]

# Compute JVP
x, v = [1.0, 2.0, 3.0], [0.1, 0.2, 0.3]
xdual = ForwardDiff.Dual{Float64}.(x, v) # vector of dual numbers
ydual = f(xdual; n = 2) # evaluate function at dual numbers
jvp = ForwardDiff.partials.(ydual) # jvp

# Compute Jacobian
jac = ForwardDiff.jacobian(x->f(x; n = 2), x)
```

Listing 4.30: JVP and full Jacobian computation with `ForwardDiff.jl`.

After defining the test function, we create dual inputs and tangents using `ForwardDiff.Dual`; `ForwardDiff.partials` extracts the dual part, i.e., the JVP. To obtain the full Jacobian, we use `ForwardDiff.jacobian`.

Figure 4.13 confirms these scaling properties. For scalar outputs, the gradient via forward mode grows with n, whereas a JVP remains near one evaluation. For $\mathbf{g} : \mathbb{R} \to \mathbb{R}^m$, forward-mode cost is insensitive to m and comparable to a JVP. Forward-mode AD thus excels at computing Jacobian–vector products but scales poorly with input dimension n when a full gradient is needed. To address this limitation, we now turn to the *reverse-mode* approach, which computes gradients efficiently by propagating sensitivities backward.

4.3.2 Reverse-Mode Automatic Differentiation

The order of operations matters. Consider the composition of functions

$$\mathbf{F}(\mathbf{x}) = \mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x}))),$$

where $\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^p$, $\mathbf{g} : \mathbb{R}^p \to \mathbb{R}^q$, and $\mathbf{f} : \mathbb{R}^q \to \mathbb{R}^m$. The chain rule gives

$$\frac{\partial \mathbf{F}}{\partial \mathbf{x}^\top} = \underbrace{\frac{\partial \mathbf{f}}{\partial \mathbf{g}^\top}}_{m\times q} \underbrace{\frac{\partial \mathbf{g}}{\partial \mathbf{h}^\top}}_{q\times p} \underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{x}^\top}}_{p\times n}. \tag{4.56}$$

Thus, the Jacobian of $\mathbf{F}$ is the product of the Jacobians of $\mathbf{f}$, $\mathbf{g}$, and $\mathbf{h}$.

Two computational strategies. There are two natural ways to evaluate the product in (4.56).

(i) **Right-to-left (forward mode):**

$$\frac{\partial \mathbf{F}}{\partial \mathbf{x}^\top} = \frac{\partial \mathbf{f}}{\partial \mathbf{g}^\top} \left(\frac{\partial \mathbf{g}}{\partial \mathbf{h}^\top} \frac{\partial \mathbf{h}}{\partial \mathbf{x}^\top} \right).$$

We first propagate a perturbation in $\mathbf{x}$ forward through each layer. This corresponds to the *forward mode* of automatic differentiation, which computes *Jacobian–vector products (JVPs)*.

(ii) **Left-to-right (reverse mode):**

$$\frac{\partial \mathbf{F}}{\partial \mathbf{x}^\top} = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{g}^\top} \frac{\partial \mathbf{g}}{\partial \mathbf{h}^\top} \right) \frac{\partial \mathbf{h}}{\partial \mathbf{x}^\top}.$$

We instead start from the output and propagate sensitivities backward. This corresponds to the *reverse mode* of automatic differentiation, which computes *vector–Jacobian products (VJPs)*.

Computational efficiency. To see the difference, consider a loss function for a deep neural network with two hidden layers of r units each. This corresponds to the special case $p = q = r$ and $m = 1$. We are interested in the high-dimensional setting with large input dimension n and many hidden units r.

- **Forward mode:** The first term computed (in parentheses) costs $\mathcal{O}(r^2 n)$ operations, and the second step adds another $\mathcal{O}(rn)$, for a total cost of $\mathcal{O}(r^2 n)$. Hence, forward mode scales poorly with the number of hidden units.
- **Reverse mode:** The first term (in parentheses) costs only $\mathcal{O}(r)$, and the second step costs $\mathcal{O}(rn)$, for a total cost of $\mathcal{O}(rn)$. This is dramatically more efficient when r is large.

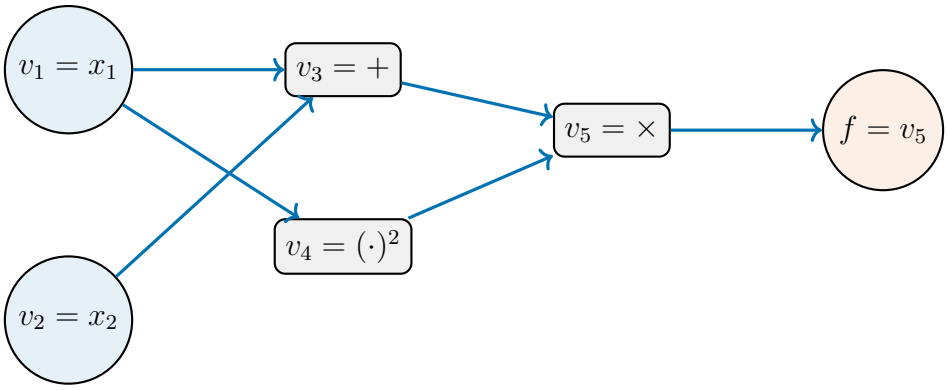


Figure 4.14: Computational graph for $f(\mathbf{x}) = (x_1 + x_2)x_1^2$. Blue arrows indicate the *forward pass* (evaluation of node values).

Interpretation. Forward mode propagates *derivatives of inputs* (tangent directions) forward through the computational graph, whereas reverse mode propagates *sensitivities of outputs* (cotangent directions) backward. When the function has many inputs (n large) and few outputs (m small, e.g., a scalar loss), reverse mode is far more efficient—this is precisely the situation in deep learning. This backward propagation of gradients is known as *backpropagation*.

The computational graph. To understand how reverse mode works, it is useful to represent the function as a *computational graph*. Consider:

$$f(\mathbf{x}) = (x_1 + x_2)\, x_1^2. \tag{4.57}$$

Each operation in the function can be viewed as a node in a directed acyclic graph (DAG), as shown in Figure 4.14. Reverse mode proceeds in two stages:

(i) a *forward pass* to compute and store the value at each node; and

(ii) a *backward pass* to accumulate gradients of the output with respect to each node.

We now compute the gradient of f step by step using reverse mode.

(i) **Forward pass.** Define the intermediate variables

$$v_1 = x_1, \qquad v_2 = x_2, \qquad v_3 = v_1 + v_2, \qquad v_4 = v_1^2, \qquad v_5 = v_3 v_4. \tag{4.58}$$

For $x_1 = 1.0$ and $x_2 = 2.0$, we obtain

$$v_3 = 3.0, \quad v_4 = 1.0, \quad v_5 = 3.0.$$

(ii) **Backward pass.** We seek $\nabla_{\mathbf{x}} f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right)$. Starting from the output v_5, we initialize its adjoint as

$$\overline{v}_5 = \frac{\partial v_5}{\partial v_5} = 1.$$

The local derivatives at the last node are

$$\frac{\partial v_5}{\partial v_3} = v_4, \qquad \frac{\partial v_5}{\partial v_4} = v_3. \tag{4.59}$$

Hence the adjoints of the predecessors are

$$\overline{v}_3 = \overline{v}_5 \frac{\partial v_5}{\partial v_3} = 1 \cdot 1 = 1, \qquad \overline{v}_4 = \overline{v}_5 \frac{\partial v_5}{\partial v_4} = 1 \cdot 3 = 3. \tag{4.60}$$

Next, we move one step further back. The local derivatives are

$$\frac{\partial v_3}{\partial v_1} = 1, \quad \frac{\partial v_3}{\partial v_2} = 1, \quad \frac{\partial v_4}{\partial v_1} = 2v_1, \quad \frac{\partial v_4}{\partial v_2} = 0. \tag{4.61}$$

Since v_1 influences both v_3 and v_4, its adjoint collects contributions from two branches:

$$\overline{v}_1 += \overline{v}_3 \frac{\partial v_3}{\partial v_1} = 1 \cdot 1 = 1, \tag{4.62}$$

$$\overline{v}_1 += \overline{v}_4 \frac{\partial v_4}{\partial v_1} = 3 \cdot 2v_1 = 6. \tag{4.63}$$

(The **+=** operator indicates that adjoints are *accumulated* whenever a node contributes through multiple paths.) Variable v_2 affects only v_3, so

$$\overline{v}_2 = \overline{v}_3 \frac{\partial v_3}{\partial v_2} = 1 \cdot 1 = 1. \tag{4.64}$$

Finally, we obtain the gradient:

$$\frac{\partial f}{\partial x_1} = \overline{v}_1 = 7, \qquad \frac{\partial f}{\partial x_2} = \overline{v}_2 = 1. \tag{4.65}$$

This is consistent with the analytical derivatives:

$$\nabla f = (3x_1^2 + 2x_1x_2, x_1^2) = (7, 1). \tag{4.66}$$

Tip

Forward vs. Reverse Pass. In forward mode, each node carries both its value and its tangent during evaluation. In reverse mode, each node stores its value during the forward pass and later accumulates an *adjoint*—its contribution to the final gradient—during the backward pass. Backpropagation is simply reverse–mode AD applied to a scalar output or loss function.

Julia implementation. Reverse-mode automatic differentiation is implemented in the `Zygote.jl` package. Listing 4.31 shows how to use `Zygote.jl` to compute the gradient of a scalar function.

```
using Zygote
f(x₁, x₂) = (x₁ + x₂) * x₁^2
y, back = Zygote.pullback(f, 1.0, 2.0)
```

Listing 4.31: Reverse-mode automatic differentiation in `Zygote.jl`.

The function `pullback` returns both the value of the function and a new function that maps an adjoint on the output to the corresponding adjoint on the inputs. Evaluating this returned function at `1.0` computes the gradient:

Julia REPL

```
julia> back(1.0)
(7.0, 1.0)
```

Listing 4.32: Evaluating the pullback function in `Zygote.jl`.

Internally, Zygote constructs the computational graph at compile time, then performs the backward pass to propagate adjoints through the graph. Effectively, Zygote generates Julia code that mirrors the manual reverse-mode procedure derived earlier:

```
function pullback_manual(x1, x2)
    v1, v2 = x1, x2
    v3 = v1 + v2
    v4 = v1^2
    v5 = v3 * v4
    function back(v̄5)
        v̄3 = v̄5 * v4
        v̄4 = v̄5 * v3
        v̄1 = v̄3 * 1 + v̄4 * (2v1)
        v̄2 = v̄3 * 1
        return (v̄1, v̄2)
    end
    return v5, back
end
```

Listing 4.33: Manual implementation of reverse-mode automatic differentiation.

Multi-output functions and VJPs. Why does `Zygote.pullback` return a function rather than the gradient directly? This design becomes essential when dealing with *vector-valued* functions. Consider

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} (x_1 + x_2)x_1^2 \\ x_1 x_2 \end{pmatrix}, \tag{4.67}$$

whose Jacobian is

$$\mathbf{J}\mathbf{f}(\mathbf{x}) = \begin{pmatrix} 3x_1^2 + 2x_1x_2 & x_1^2 \\ x_2 & x_1 \end{pmatrix}. \tag{4.68}$$

We can again use `Zygote.pullback` to compute gradients with respect to $\mathbf{x}$:

```
f(x₁, x₂) = [(x₁ + x₂) * x₁^2, x₁ * x₂]
y, back = Zygote.pullback(f, 1.0, 2.0)
```

Listing 4.34: Multi-output function in `Zygote.jl`.

In this case, the function `back` takes a vector of output adjoints and returns the *vector–Jacobian product* (VJP):

$$\texttt{back}(\overline{\mathbf{v}}) = \overline{\mathbf{v}}^\top \mathbf{J}\mathbf{f}(\mathbf{x}).$$

To recover the full Jacobian, we evaluate `back` on the standard basis vectors:

Julia REPL

```
julia> back([1.0, 0.0])
(7.0, 1.0)
julia> back([0.0, 1.0])
(2.0, 1.0)
```

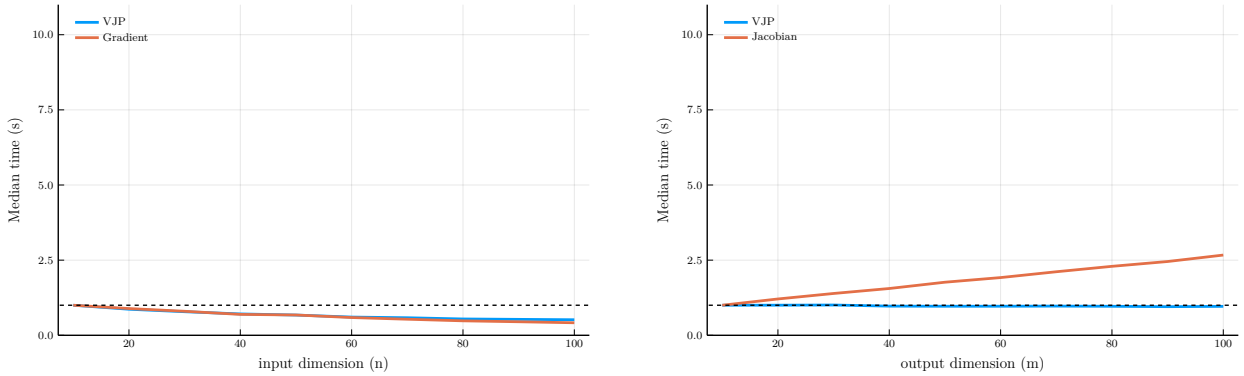
Listing 4.35: Evaluating vector–Jacobian products (VJPs) with `Zygote.jl`.

Here, `back([1.0, 0.0])` corresponds to the first row of the Jacobian (the gradient of the first output), and `back([0.0, 1.0])` corresponds to the second row. This demonstrates that reverse-mode AD computes VJPs efficiently with cost $\mathcal{O}(\text{cost}(f))$—independent of the output dimension m.

Performance comparison. We now illustrate the scaling properties of reverse-mode AD using `Zygote.jl`. We employ the same test functions as in Section 4.3.1 (Equation (4.55)).

Figure 4.15 highlights the complementary scaling of reverse-mode AD. For scalar-valued functions ($m = 1$), a single VJP efficiently computes the entire gradient, with cost nearly independent of the input dimension n. For vector-valued functions, the cost of one VJP is independent of the output dimension m, while computing the full Jacobian requires one backward pass per output dimension.

Interestingly, the total runtime of the full Jacobian grows slightly sublinearly with m. This occurs because each backward pass reuses the same forward evaluation of the function, so the forward computation acts as a fixed cost. For instance, increasing m from 10 to 100 raises the total cost by only about 2.5×, not by a factor of 10.



(a) Input dimension scaling

(b) Output dimension scaling

Figure 4.15: Computational efficiency of VJPs vs. full gradient/Jacobian in `Zygote.jl`.

Left: median time vs. input dimension n for $f : \mathbb{R}^n \to \mathbb{R}$. The gradient (full Jacobian transpose) scales linearly with n, whereas a single VJP remains near the cost of one function evaluation (dashed baseline). Right: median time vs. output dimension m for $\mathbf{g} : \mathbb{R}^{10} \to \mathbb{R}^m$. The full Jacobian scales approximately linearly with m, while a single VJP remains constant. Times are normalized by the cost of one forward evaluation.

Taking stock. Automatic differentiation (AD) provides an efficient and reliable way to compute exact derivatives of functions expressed as computer programs. Forward-mode AD excels when the number of outputs is large but the number of inputs is small, whereas reverse-mode AD is ideal when many inputs map to a few outputs—especially a single scalar loss. This explains why reverse-mode AD, better known as *backpropagation*, became the workhorse of modern machine learning: it efficiently computes gradients of high-dimensional parameter vectors with respect to a single scalar objective. These same techniques will be central to the next chapter, where we apply deep learning architectures and backpropagation-based optimization to solve high-dimensional dynamic macro–finance models.

Chapter 5

The Deep Policy Iteration Method

In this chapter, we show how to use the machine-learning tools developed in Chapter 4 to solve high-dimensional dynamic programming problems in continuous time. We introduce the *Deep Policy Iteration* (DPI) method, a powerful technique for solving such problems by combining stochastic optimization, automatic differentiation, and neural-network function approximation. We will see how DPI can be applied to a wide range of economic and financial problems, including asset pricing, corporate finance, and portfolio choice.

5.1 The Dynamic Programming Problem

We consider a continuous-time optimal control problem in which an infinitely lived agent faces a Markovian decision process. The system's state is represented by a vector $\mathbf{s}_t \in \mathbb{R}^n$, and the control by a vector $\mathbf{c}_t \in \Gamma(\mathbf{s}_t) \subset \mathbb{R}^p$, where $\Gamma(\mathbf{s}_t)$ denotes the set of admissible controls at time t given state $\mathbf{s}_t$. Importantly, the feasible control set depends on the state. For instance, in the consumption–savings problem discussed in Chapter 3, the maximum feasible consumption at time t depends on the household's wealth, which is one of the state variables.

The agent's objective is to maximize the expected discounted value of the reward function $u(\mathbf{c}_t)$:

$$V(\mathbf{s}) = \max_{\{\mathbf{c}_t\}_{t=0}^{\infty}} \mathbb{E}\left[\int_0^{\infty} e^{-\rho t} u(\mathbf{c}_t)\, dt \;\middle|\; \mathbf{s}_0 = \mathbf{s} \right], \tag{5.1}$$

subject to the stochastic law of motion

$$d\mathbf{s}_t = \mathbf{f}(\mathbf{s}_t, \mathbf{c}_t)\, dt + \mathbf{g}(\mathbf{s}_t, \mathbf{c}_t)\, d\mathbf{B}_t, \tag{5.2}$$

$$\mathbf{c}_t \in \Gamma(\mathbf{s}_t), \quad \forall t \geq 0, \tag{5.3}$$

where $\mathbf{B}_t$ is an $m \times 1$ vector of independent Brownian motions, $\mathbf{f}(\mathbf{s}_t, \mathbf{c}_t) \in \mathbb{R}^n$ is the drift, and $\mathbf{g}(\mathbf{s}_t, \mathbf{c}_t) \in \mathbb{R}^{n \times m}$ is the diffusion matrix. Both the drift and the diffusion can depend on the control variables.

For most of this chapter, we focus on diffusion processes without jumps; extensions to jump processes are discussed later. The machine-learning tools introduced in Chapter 4 are particularly useful in this setting.

Interpretation. This general formulation encompasses a broad class of problems. It could describe a household maximizing lifetime utility, a firm choosing investment and financing policies, a regulator designing optimal policy, or a central bank managing welfare over time. As seen in Chapter 3, even derivative pricing problems can often be cast as optimal control problems in continuous time. The following sections present a unified approach for solving all such models.

The HJB Equation. As shown in Chapter 3, the optimal policy $\mathbf{c}(\mathbf{s})$ satisfies the Hamilton–Jacobi–Bellman (HJB) equation:

$$0 = \max_{\mathbf{c}\in\Gamma(\mathbf{s})} \Big[u(\mathbf{c}) + \nabla_{\mathbf{s}} V(\mathbf{s})^\top \mathbf{f}(\mathbf{s},\mathbf{c}) + \tfrac{1}{2} \operatorname{Tr}\big(\mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}_{\mathbf{s}} V(\mathbf{s})\, \mathbf{g}(\mathbf{s},\mathbf{c})\big) \Big], \tag{5.4}$$

where $\nabla_{\mathbf{s}} V(\mathbf{s})$ and $\mathbf{H}_{\mathbf{s}} V(\mathbf{s})$ denote, respectively, the gradient and Hessian of the value function. This equation follows from the multivariate version of Itô's lemma, applied to the stochastic process $V(\mathbf{s}_t)$. For completeness, the derivation of the multivariate Itô formula is provided in Appendix A.1.

Dealing with the Three Curses. As discussed in Chapters 1–3, dynamic programming methods face three intertwined computational obstacles– the so-called *three curses of dimensionality*: representation, optimization, and expectation. In low-dimensional settings, these challenges are manageable using the grid-based or collocation schemes introduced earlier. In higher dimensions, however, the same bottlenecks that limited discrete- and continuous-time methods reappear with full force.

(i) **Curse of representation:** as the number of state variables grows, storing and interpolating the value and policy functions on a grid becomes infeasible;

(ii) **Curse of optimization:** evaluating or solving the maximization step $\max_{\mathbf{c}}\{u(\mathbf{c}) + \dots\}$ at each state point becomes increasingly expensive as the control space expands;

(iii) **Curse of expectation:** computing the expected continuation value $\mathbb{E}[V(\mathbf{s}')]$ involves integration over many stochastic dimensions, which scales exponentially in the number of shocks.

The remainder of this chapter shows how the *Deep Policy Iteration (DPI)* algorithm addresses each of these curses using modern machine-learning tools:

- Neural networks provide compact, differentiable representations of value and policy functions, circumventing the curse of representation;
- Automatic differentiation enables efficient computation of gradients and drift terms needed for the HJB, mitigating the curse of optimization;
- Stochastic optimization and simulation-based training replace explicit high-dimensional integration, alleviating the curse of expectation.

Together, these components allow DPI to solve high-dimensional continuous-time models that were previously computationally intractable.

5.2 Overcoming the Curse of Expectation: Itô's Lemma and Automatic Differentiation

One of the central challenges in solving high-dimensional dynamic programming problems is the computation of the expected continuation value. In discrete time, this requires evaluating high-dimensional integrals. In continuous time, these expectations enter the Hamilton–Jacobi–Bellman (HJB) equation through the infinitesimal generator of the value function. The key insight of continuous-time methods—and one that lies at the heart of the Deep Policy Iteration algorithm—is that these expectations can be replaced by derivatives via Itô's lemma. This allows us to transform the *curse of expectation* into a problem of efficient differentiation.

From integration to differentiation. In discrete time, the Bellman equation involves the expectation of the continuation value,

$$\mathbb{E}[V(\mathbf{s}')] = \int \cdots \int V(s + f(\mathbf{s},\mathbf{c})\,\Delta t + g(\mathbf{s},\mathbf{c})\,\sqrt{\Delta t}\,\mathbf{Z})\,\phi(\mathbf{Z})\,d\mathbf{Z}_1 \cdots d\mathbf{Z}_m,$$

where $\phi(\mathbf{Z})$ is the joint density of the shocks. In continuous time, the expected change in the value function can be written as

$$\mathbb{E}[dV(\mathbf{s})] = \nabla_{\mathbf{s}} V(\mathbf{s})^\top \mathbf{f}(\mathbf{s},\mathbf{c}) + \tfrac{1}{2}\,\mathrm{Tr}\big[\mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}_{\mathbf{s}} V(\mathbf{s}) \mathbf{g}(\mathbf{s},\mathbf{c})\big]\,. \tag{5.5}$$

Hence, instead of computing high-dimensional integrals, we only need to compute derivatives of the value function.

Computational challenge. One could use finite differences to compute these derivatives, but this quickly becomes infeasible in higher dimensions. Suppose $n = 10$ and we use a grid of 100 points for each state variable. Then, just to store the grid in memory, we would need 10^{17} terabytes of RAM.

As discussed in Chapter 4, automatic differentiation (AD) provides an efficient way to compute derivatives by propagating local gradients through a computational graph. However, a naive use of AD can also be computationally expensive here. Computing the drift of $V(\mathbf{s})$ requires both the gradient and the Hessian matrix of $V(\mathbf{s})$, and the cost of forming the full Hessian increases rapidly with the number of state variables. Nested calls to an AD library to compute these second derivatives can be prohibitively slow.

Illustration. To see this, consider the simple example:

$$V(\mathbf{s}) = \sum_{i=1}^{n} s_i^2,$$

where s_i is the i-th state variable. Suppose there is a single shock ($m = 1$) and $n = 100$ state variables. We evaluate the drift at the point $\mathbf{s} = \mathbf{f}(\mathbf{s}) = \mathbf{g}(\mathbf{s}) = \mathbf{1}_{n\times 1}$, abstracting from controls for simplicity.

Table 5.1: Computational Cost of Numerical Derivatives

Method	FLOPs	Memory	Error
1. Finite differences	9,190,800	112,442,048	1.58%
2. Naive autodiff	2,100,501	25,673,640	0.00%
3. Analytical	20,501	44,428	0.00%
4. Hyper-dual Itô	599	6,044	0.00%

Notes: The table shows the computational cost for computing the drift of $V(s) = \sum_{i=1}^{100} s_i^2$, assuming $s_i = \mu_i = \sigma_i = 1$ for $i = 1, \ldots, 100$, using four different methods: 1) finite differences (with $h = 0.001$), 2) a naive use of automatic differentiation (where the Hessian is computed by nested calls to the Jacobian function), 3) using the analytical partial derivatives, and 4) the hyper-dual Itô's lemma method described in Proposition 5.1 combined with forward-mode automatic differentiation. The column FLOPs shows the number of floating point operations required by each approach. The column Memory is measured as bytes accessed. The column Error measures the absolute value of the relative error of each method in percentage terms.

Table 5.1 compares the number of floating-point operations (FLOPs) required to compute the drift of $V(\mathbf{s})$ using different methods. Finite differences are both memory-intensive and inaccurate, while a naive AD implementation—computing the Hessian via nested Jacobian calls—offers only limited improvement. We next introduce a more efficient method.

A hyper-dual approach to Itô's lemma. One of the key insights from Chapter 4 is that computing a *Jacobian–vector product* (JVP) is much more efficient than forming the full Jacobian. In particular, the cost of a JVP is independent of the number of state variables. In the absence of shocks, computing the drift of $V(\mathbf{s})$ amounts to evaluating a JVP:

$$\mathbb{E}[dV(\mathbf{s})] = \nabla_{\mathbf{s}} V(\mathbf{s})^\top \mathbf{f}(\mathbf{s})\, dt,$$

which can be efficiently computed using forward-mode AD.

However, when stochastic shocks are present, the drift also depends on quadratic forms involving the Hessian of $V(\mathbf{s})$. In this case, a JVP is no longer sufficient.

The idea of the *hyper-dual approach* is to extend dual numbers—the building block of forward-mode AD—to include not only the function value and tangent direction, but also the matrix of risk exposures associated with the diffusion term. We can represent a hyper-dual number as a triplet containing the value of the function, its tangent vector, and the diffusion matrix. The following proposition formalizes this idea.

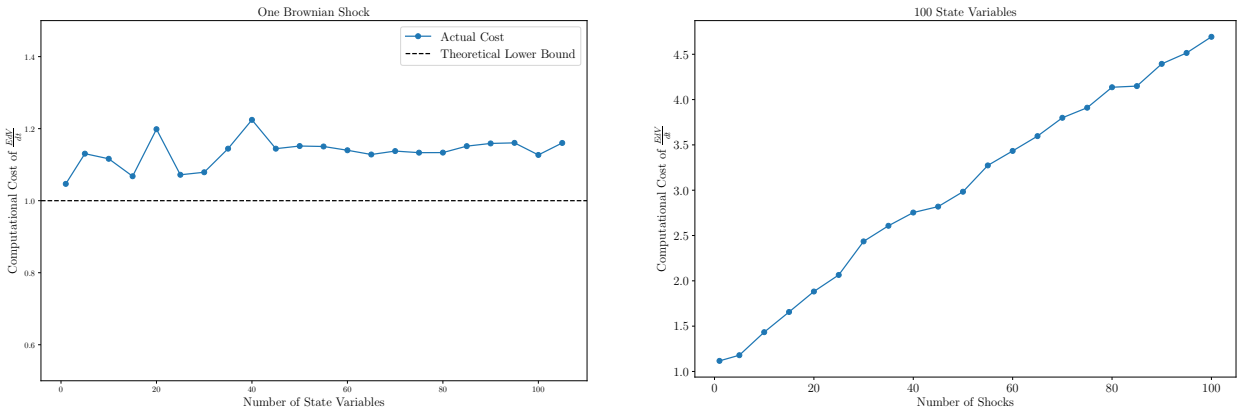
Proposition 5.1 (Hyper-dual Itô's lemma)**.** *For a given* $\mathbf{s}$*, define the auxiliary functions* $F_i : \mathbb{R} \to \mathbb{R}$ *as*

$$F_i(\epsilon; \mathbf{s}) = V\left(\mathbf{s} + \tfrac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\,\epsilon + \tfrac{\mathbf{f}(\mathbf{s})}{2m}\,\epsilon^2\right),$$

where $\mathbf{f}(\mathbf{s})$ *is the drift of* $\mathbf{s}$ *and* $\mathbf{g}_i(\mathbf{s})$ *is the* i*-th column of the diffusion matrix* $\mathbf{g}(\mathbf{s})$*. Then:*

1. ***Diffusion:*** *The* $1 \times m$ *diffusion matrix of* $V(\mathbf{s})$ *is*

$$\nabla V(\mathbf{s})^\top \mathbf{g}(\mathbf{s}) = \sqrt{2}\,[F_1'(0; \mathbf{s}), F_2'(0; \mathbf{s}), \ldots, F_m'(0; \mathbf{s})].$$



(a) (b)

Figure 5.1: Itô's lemma computational cost.

Notes: This figure shows how the cost of computing the drift of a function V scales with the number of state variables and Brownian shocks. Cost is measured as the execution time of $\frac{\mathbb{E}dV}{dt}(\mathbf{s})$ relative to that of $V(\mathbf{s})$. The left panel fixes $m = 1$ and varies n from 1 to 100; the right panel fixes $n = 100$ and varies m from 1 to 100. In this example, V is a two-layer neural network, and execution times are averaged over 10,000 runs on a mini-batch of 512 states.

2. ***Drift:*** *The drift of* $V(\mathbf{s})$ *is*

$$\mathcal{D}V(\mathbf{s}) = F''(0; \mathbf{s}),$$

where $F(\epsilon; \mathbf{s}) = \sum_{i=1}^{m} F_i(\epsilon; \mathbf{s})$.

Note

Automatic Itô calculus. In the same way automatic differentiation teaches the computer how to apply the rules of classical calculus, the hyper-dual approach effectively teaches it the rules of *stochastic calculus*. This enables the automatic computation of drifts and diffusions in continuous-time models. Appendix A.2.2 provides a detailed proof of Proposition 5.1.

The result in Proposition 5.1 reduces the problem of computing the drift of $V(\mathbf{s})$—which normally involves computing both a gradient and a Hessian—to evaluating the second derivative of a univariate function. Consistent with our discussion in Chapter 4, the cost of computing the drift is the same as evaluating $F(\epsilon)$, which involves evaluating $V(\mathbf{s})$ repeated m times (once per Brownian shock). Hence the computational complexity is

$$\mathcal{O}(m \times \text{cost}(V(\mathbf{s}))),$$

which is independent of the number of state variables n.

Julia implementation. It is straightforward to implement the hyper-dual approach to Itô's lemma in Julia. Listing 5.1 shows how to compute the drift of $V(\mathbf{s})$ using the `ForwardDiff.jl` package to compute the second derivative of the auxiliary function $F(\epsilon)$.

```julia
using ForwardDiff, LinearAlgebra
V(s) = sum(s.^2) # example function
n, m  = 100, 1 # number of state variables and shocks
s0, f, g = ones(n), ones(n), ones(n,m) # example values

# Analytical drift
∇f, H = 2*s0, Matrix(2.0*I, n,n) # gradient and Hessian
drift_analytical = ∇f'*f + 0.5*tr(g'*H*g) # analytical drift

# Hyper-dual approach
F(ϵ) = sum([V(s0 + g[:,i]*ϵ/sqrt(2) + f/(2m)*ϵ^2) for i = 1:m])
drift_hyper = ForwardDiff.derivative(ϵ ->
        ForwardDiff.derivative(F, ϵ), 0.0) # scalar 2nd derivative
```

Listing 5.1: Hyper-dual Itô's lemma implementation.

To verify correctness, we can compare the analytical and hyper-dual drifts:

Julia REPL

```
julia> drift_analytical, drift_hyper
(300.0, 300.0)
```

Listing 5.2: Comparison of analytical and hyper-dual drift.

These results confirm that the hyper-dual approach computes the drift exactly, matching the analytical benchmark.

Benchmark. We can benchmark the performance of the hyper-dual approach to Itô's lemma by comparing the execution time of alternative methods. Table 5.1 shows that finite differences are both memory-intensive and inaccurate, while a naive AD implementation—computing the Hessian via nested Jacobian calls—offers only limited improvement. The third row reports results using analytical partial derivatives, while the fourth row uses the hyper-dual Itô's lemma method. The hyper-dual approach proves to be the most efficient, even outperforming the case with analytical derivatives in both speed and memory usage.

Figure 5.1 illustrates how the execution time of computing the drift scales with the number of state variables and Brownian shocks. Panel (a) fixes the number of shocks at one and varies the number of state variables from 1 to 100, while Panel (b) fixes the number of state variables at 100 and varies the number of shocks from 1 to 100. In this example, V is a two-layer neural network, and execution times are averaged over 10,000 runs on a mini-batch of 512 states.

The figure shows that the execution time of the hyper-dual Itô's lemma method is roughly independent of the number of state variables, and increases linearly with the number of Brownian shocks. This stands in sharp contrast to the discrete-time approach, where the

expected continuation value must be computed via numerical quadrature, whose cost grows exponentially with the number of shocks. The fact that computational cost is independent of the number of state variables and increases only linearly with the number of shocks demonstrates how this method overcomes the *curse of expectation*.

5.3 The Deep Policy Iteration Algorithm

In this section, we introduce the Deep Policy Iteration (DPI) algorithm for solving dynamic programming problems in continuous time. Our objective is to compute the value function $V(\mathbf{s})$ and policy function $\mathbf{c}(\mathbf{s})$ satisfying the coupled functional equations:

$$0 = HJB(\mathbf{s}, \mathbf{c}(\mathbf{s}), V(\mathbf{s})), \qquad \mathbf{c}(\mathbf{s}) = \arg\max_{\mathbf{c}\in\Gamma(\mathbf{s})} HJB(\mathbf{s}, \mathbf{c}, V(\mathbf{s})), \tag{5.6}$$

where

$$HJB(\mathbf{s}, \mathbf{c}, V) = u(\mathbf{c}) - \rho V + \underbrace{(\nabla_{\mathbf{s}} V)^\top \mathbf{f}(\mathbf{s}, \mathbf{c}) + \tfrac{1}{2}\operatorname{Tr}\big[\mathbf{g}(\mathbf{s}, \mathbf{c})^\top \mathbf{H}_{\mathbf{s}} V(\mathbf{s})\, \mathbf{g}(\mathbf{s}, \mathbf{c})\big]}_{F''(0;\mathbf{s},\mathbf{c})}. \tag{5.7}$$

The drift term in the HJB can be computed efficiently using the auxiliary function $F(\cdot)$ defined in Proposition 5.1.

Overcoming the curse of representation. To solve for $V(\mathbf{s})$ and $\mathbf{c}(\mathbf{s})$ numerically, we must represent them on a computer. A traditional approach is to discretize the state space and interpolate between grid points, leading to a piecewise-linear approximation. This corresponds to a parametric representation with parameters $\boldsymbol{\theta}_V$ and $\boldsymbol{\theta}_C$ for the value and policy functions, respectively: $V(\mathbf{s}_i; \boldsymbol{\theta}_V) = \boldsymbol{\theta}_{V,i}$ and $\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C) = \boldsymbol{\theta}_{C,i}$. However, as the dimensionality of the state space grows, the number of grid points explodes exponentially—an expression of the first curse of dimensionality. We observed a similar limitation for spectral methods in Chapter 3, where the number of basis coefficients also grows rapidly in multiple dimensions.

In Chapter 4, we showed that such grid-based approximations can be viewed as shallow neural networks with fixed breakpoints. Neural networks generalize this idea by learning flexible breakpoints and nonlinear combinations of basis functions. A deep neural network (DNN) can approximate complex value and policy functions with relatively few parameters, making it an effective representation even in high-dimensional settings. We therefore represent $V(\mathbf{s})$ and $\mathbf{c}(\mathbf{s})$ using DNNs parameterized by $\boldsymbol{\theta}_V$ and $\boldsymbol{\theta}_C$, respectively.

Overcoming the curse of optimization. We now turn to the challenge of training the DNNs to satisfy the functional equations above. A key difficulty lies in performing the maximization step efficiently, without resorting to costly root-finding procedures at every state point. Our approach combines *generalized policy iteration* (see, e.g., Sutton and Barto 2018) with deep function approximation, alternating between policy evaluation and policy improvement. This leads to the *Deep Policy Iteration (DPI)* algorithm.

We describe the algorithm in three stages: (i) sampling, (ii) policy improvement, and (iii) policy evaluation.

Tip

Simplifying assumptions. For clarity, we make several simplifying assumptions that can be relaxed in practice. First, we adopt plain stochastic gradient descent (SGD) for parameter updates, although any of the optimizers discussed in Chapter 4 (e.g., Adam, RMSProp) could be used instead. Second, we perform exactly one iteration of policy evaluation and policy improvement at each update. Third, we use a quadratic loss function for the policy evaluation step.

Step 1: Sampling. We begin by sampling a mini-batch of states $\{\mathbf{s}_i\}_{i=1}^I$ from the state space. This batch can be drawn from a uniform distribution within plausible state-space bounds, or from an estimated ergodic distribution based on previous iterations.

Step 2: Policy Improvement. The policy improvement step, represented by the second equation in Eq. (5.6), involves solving an optimization problem for every state in the mini-batch. This step can be computationally demanding and lies at the heart of the *curse of optimization*.

Note

Generalized policy iteration. In Chapter 3, we introduced the policy function iteration (PFI) algorithm, which alternates between two steps: *policy evaluation* and *policy improvement*. In the policy evaluation step, we solve for the new value function $V_{n+1}(\mathbf{s})$ given the policy $\mathbf{c}_n(\mathbf{s})$. In the policy improvement step, we solve for the new policy $\mathbf{c}_{n+1}(\mathbf{s})$ given the current value function $V_{n+1}(\mathbf{s})$. We repeat this process until convergence.
However, when the initial guess for V is far from optimal, fully solving the maximization problem at each iteration is inefficient. This motivates an *approximate policy improvement* step that performs only a single gradient-based update in the direction of improvement.

To implement this approximate step, for each state $\mathbf{s}_i$ in the mini-batch we start from the current policy estimate $\mathbf{c}_{0,i} \equiv \mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C^{j-1})$ and perform one gradient ascent step on $HJB(\mathbf{s}_i, \mathbf{c}, \boldsymbol{\theta}_V^{j-1})$:

$$\mathbf{c}_{1,i} = \mathbf{c}_{0,i} + \nabla_{\mathbf{c}} HJB(\mathbf{s}_i; \mathbf{c}_{0,i}, \boldsymbol{\theta}_V^{j-1}). \tag{5.8}$$

(The learning rate is normalized to 1 without loss of generality; see discussion below.)

We then treat $(\mathbf{s}_i, \mathbf{c}_{1,i})_{i=1}^I$ as a mini-batch of training data, and update the policy network so that its output $\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C)$ matches these improved controls. The corresponding loss function is a quadratic penalty:

$$\mathcal{L}(\boldsymbol{\theta}_C) = \frac{1}{2I} \sum_{i=1}^{I} \|\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C) - \mathbf{c}_{1,i}\|^2. \tag{5.9}$$

Differentiating with respect to the network parameters yields

$$\nabla_{\boldsymbol{\theta}_C} \mathcal{L}(\boldsymbol{\theta}_C) = \frac{1}{I} \sum_{i=1}^{I} \mathbf{J}_{\boldsymbol{\theta}_C} \mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C)^\top \big(\mathbf{c}(\mathbf{s}_i; \boldsymbol{\theta}_C) - \mathbf{c}_{1,i}\big), \tag{5.10}$$

where $\mathbf{J}_{\boldsymbol{\theta}_C}\mathbf{c}(\mathbf{s}_i;\boldsymbol{\theta}_C)$ is the Jacobian of the policy network with respect to its parameters.

Evaluating the gradient at $\boldsymbol{\theta}_C^{j-1}$ and using $\mathbf{c}_{0,i} - \mathbf{c}_{1,i} = -\nabla_{\mathbf{c}} HJB(\mathbf{s}_i; \mathbf{c}_{0,i}, \boldsymbol{\theta}_V^{j-1})$, we obtain

$$\nabla_{\boldsymbol{\theta}_C}\mathcal{L}(\boldsymbol{\theta}_C^{j-1}) = -\frac{1}{I}\sum_{i=1}^{I} \mathbf{J}_{\boldsymbol{\theta}_C}\mathbf{c}(\mathbf{s}_i;\boldsymbol{\theta}_C^{j-1})^\top \nabla_{\mathbf{c}} HJB(\mathbf{s}_i, \mathbf{c}_{0,i}, \boldsymbol{\theta}_V^{j-1}) \tag{5.11}$$

$$= -\frac{1}{I}\sum_{i=1}^{I} \nabla_{\boldsymbol{\theta}_C} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j-1}, \boldsymbol{\theta}_V^{j-1}). \tag{5.12}$$

This leads to the following policy update:

Policy improvement step.

$$\boldsymbol{\theta}_C^{j} = \boldsymbol{\theta}_C^{j-1} + \eta_C \frac{1}{I}\sum_{i=1}^{I} \nabla_{\boldsymbol{\theta}_C} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^{j-1}, \boldsymbol{\theta}_V^{j-1}), \tag{5.13}$$

where η_C is the learning rate controlling the step size in parameter space. Equation (5.13) corresponds to a single gradient-ascent step on the HJB objective with respect to $\boldsymbol{\theta}_C$.

Note

Normalization. If we had introduced a learning rate η' in Eq. (5.8), its effect would simply multiply η_C in Eq. (5.13). Because only the product $\eta_C\eta'$ matters for the update, we can normalize $\eta' = 1$ without loss of generality.

Step 3: Policy Evaluation. We now update the value function given the new policy parameters $\boldsymbol{\theta}_C^j$. We present two alternative update rules, each with distinct trade-offs.

The first rule mirrors the iterative policy evaluation procedure in Algorithm 2. Analogous to the explicit finite-difference method in Chapter 3, we consider a *false-transient* formulation that iterates the value function backward in (pseudo-)time:

$$\frac{V(\mathbf{s};\boldsymbol{\theta}_V^j) - V(\mathbf{s}_i;\boldsymbol{\theta}_V^{j-1})}{\Delta t} = HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V^{j-1}). \tag{5.14}$$

Instead of discretizing the spatial derivatives as in finite differences, we obtain them analytically through automatic differentiation of the neural network $V(\mathbf{s};\boldsymbol{\theta}_V)$, potentially using the hyper-dual Itô method from Proposition 5.1.

Rather than iterating until convergence, we interpret this expression as defining a *target* for the next value update:

$$V(\mathbf{s};\boldsymbol{\theta}_V^j) = V(\mathbf{s}_i;\boldsymbol{\theta}_V^{j-1}) + HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V^{j-1})\,\Delta t, \tag{5.15}$$

and train the value network by minimizing the quadratic loss

$$\mathcal{L}(\boldsymbol{\theta}_V) = \frac{1}{2I}\sum_{i=1}^{I}\big(V(\mathbf{s}_i;\boldsymbol{\theta}_V) - V(\mathbf{s}_i;\boldsymbol{\theta}_V^{j-1}) - HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V^{j-1})\Delta t\big)^2. \tag{5.16}$$

Evaluating the gradient at $\boldsymbol{\theta}_V^{j-1}$ yields

$$\nabla_{\boldsymbol{\theta}_V}\mathcal{L}(\boldsymbol{\theta}_V^{j-1}) = -\frac{\Delta t}{I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V^{j-1})\nabla_{\boldsymbol{\theta}_V} V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1}), \tag{5.17}$$

and the corresponding update rule is:

Policy evaluation step 1.

$$\boldsymbol{\theta}_V^j = \boldsymbol{\theta}_V^{j-1} + \eta_V \frac{\Delta t}{I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V^{j-1})\nabla_{\boldsymbol{\theta}_V} V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1}). \tag{5.18}$$

Alternatively, we can proceed as in the implicit finite-difference method, and evaluate the HJB equation at the new value function parameters $\boldsymbol{\theta}_V^j$:

$$\frac{V(\mathbf{s}; \boldsymbol{\theta}_V^j) - V(\mathbf{s}_i; \boldsymbol{\theta}_V^{j-1})}{\Delta t} = HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V^j). \tag{5.19}$$

We can define our loss function as minimizing the mean-squared error of the equation above. As in the implicit finite-difference method in Chapter 3, we can take the limit as $\Delta t \to 0$, so the loss function becomes simply the mean-squared error of the HJB residuals:

$$\mathcal{L}(\boldsymbol{\theta}_V) = \frac{1}{2I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V)^2, \tag{5.20}$$

whose gradient is

$$\nabla_{\boldsymbol{\theta}_V}\mathcal{L}(\boldsymbol{\theta}_V) = \frac{1}{I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V)\nabla_{\boldsymbol{\theta}_V} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V). \tag{5.21}$$

Evaluating at $\boldsymbol{\theta}_V^{j-1}$ gives:

Policy evaluation step 2.

$$\boldsymbol{\theta}_V^j = \boldsymbol{\theta}_V^{j-1} - \eta_V \frac{1}{I}\sum_{i=1}^{I} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V^{j-1})\nabla_{\boldsymbol{\theta}_V} HJB(\mathbf{s}_i, \boldsymbol{\theta}_C^j, \boldsymbol{\theta}_V^{j-1}). \tag{5.22}$$

Tip

The trade-off between the two update rules. In the reinforcement learning literature, methods that minimize the Bellman residual directly are known to converge more stably but often more slowly than iterative policy evaluation. Moreover, Eq. (5.22) requires relatively costly third-order derivatives, since $\nabla_{\boldsymbol{\theta}_V} HJB$ involves the Hessian of V. As a rule of thumb, it is preferable to start with Eq. (5.18) for speed, and switch to

Eq. (5.22) if instability or divergence is observed.

We can now summarize the complete algorithm:

Algorithm 4: Deep Policy Iteration (DPI)

Initialize parameters $\boldsymbol{\theta}_V^0$ and $\boldsymbol{\theta}_C^0$;
for $j = 1, 2, \ldots$ **do**
  Sample a mini-batch of states $\{\mathbf{s}_i\}_{i=1}^I$;
  `// Policy improvement (actor update)`
  Update $\boldsymbol{\theta}_C$ using Eq. (5.13);
  `// Policy evaluation (critic update)`
  Update $\boldsymbol{\theta}_V$ using Eq. (5.18) or Eq. (5.22);

Note

Actor–Critic Interpretation. The Deep Policy Iteration (DPI) algorithm mirrors the *actor–critic* architecture from reinforcement learning. The **actor** corresponds to the policy network $\mathbf{c}(\mathbf{s}; \boldsymbol{\theta}_C)$, which proposes actions (controls) given the current state. The **critic** corresponds to the value network $V(\mathbf{s}; \boldsymbol{\theta}_V)$, which evaluates those actions by estimating the value function through the HJB residual. In each iteration:

- the **actor update** (policy improvement step, Eq. (5.13)) adjusts $\boldsymbol{\theta}_C$ to increase the value estimated by the critic;
- the **critic update** (policy evaluation step, Eq. (5.18) or Eq. (5.22)) refines $\boldsymbol{\theta}_V$ so that the critic better approximates the value implied by the current policy.

This alternating structure allows DPI to learn both the optimal policy and its associated value function jointly, just as actor–critic methods do in modern reinforcement learning, but here grounded in continuous-time economic dynamic programming.

5.4 Applications

In this section, we apply the Deep Policy Iteration (DPI) algorithm to solve a variety of economic and financial problems. We consider three canonical domains of finance—asset pricing, corporate finance, and portfolio choice—to demonstrate how the DPI algorithm can be adapted to different environments. The different applications illustrates how the DPI algorithm can be applied to solve a variety of economic and financial problems

5.4.1 Asset Pricing: The Two-Trees Model

We start with a classic asset-pricing problem—the two-trees model from Chapter 3. Although this model can be solved analytically, it provides a transparent benchmark to illustrate how the DPI algorithm operates in practice and how each of its components fits together. We

then extend the model to the multi-asset setting, where the dimensionality of the problem grows rapidly.

The two-trees model. As shown in Chapter 3, the solution of the two-trees model boils down to solving a boundary-value problem for the price–consumption ratio v_t. The pricing condition for a log investor implies

$$v_t = \mathbb{E}_t \left[\int_0^\infty e^{-\rho s} s_{t+s} \, ds \right], \tag{5.23}$$

where the relative share process s_t evolves as

$$ds_t = -2\sigma^2 s_t(1-s_t)\left(s_t - \tfrac{1}{2}\right) dt + \sigma s_t(1-s_t)(dB_{1,t} - dB_{2,t}). \tag{5.24}$$

Since s_t is Markov, we can write $v_t = v(s_t)$. The stationary HJB equation for $v(s)$ is

$$\rho v = s - v_s \, 2\sigma^2 s(1-s)\left(s - \tfrac{1}{2}\right) + \frac{1}{2} v_{ss} \left(2\sigma^2 s^2 (1-s)^2\right), \tag{5.25}$$

subject to the boundary conditions $v(0) = 0$ and $v(1) = 1/\rho$.

Although this one-dimensional problem is straightforward to solve using finite differences or collocation methods, we solve it here using the DPI framework to illustrate the workflow.

Model setup. We start by defining a model struct that contains both the parameters and the functions for the drift and diffusion of the state variable s.

```
@kwdef struct TwoTrees
  ρ::Float64 = 0.04
  σ::Float64 = sqrt(0.04)
  μ::Float64 = 0.02
  μₛ::Function = s -> @. -2 * σ^2 * s * (1-s) * (s-0.5)
  σₛ::Function = s -> @. sqrt(2) * σ * s * (1-s)
end;
```

Listing 5.3: Model struct for the two-trees model.

The drift and diffusion functions use Julia's broadcast operator `@.` to apply the transformation elementwise to batched inputs. This allows the code to handle multiple state draws simultaneously, which will be useful during training.

Hyper-dual Itô's lemma. We next implement the hyper-dual approach to Itô's lemma to compute the drift of the value function efficiently.

```
# Hyper-dual approach to Ito's lemma
function drift_hyper(V::Function, s::AbstractMatrix, m::TwoTrees)
  F(ϵ) = V(s + m.σₛ(s)/sqrt(2)*ϵ + m.μₛ(s)/2*ϵ^2)
  ForwardDiff.derivative(ϵ->ForwardDiff.derivative(F, ϵ), 0.0)
end;
```

Listing 5.4: Hyper-dual approach to Itô's lemma.

The implementation in Listing 5.4 is remarkably compact: two lines of code suffice to compute the drift of any function $V(\mathbf{s})$ using Proposition 5.1. To validate the implementation, we test it against the analytical drift; the results match to machine precision.

```
# Small test: exact vs. automatic differentiation
rng = Xoshiro(0)
s   = rand(rng, 1, 1000)
# Exact drift for test function
V_test(s) = sum(s.^2, dims = 1)
drifts_exact = map(1:size(s, 2)) do i
    ∇V, H = 2 * s[:,i], 2 * Matrix(I,length(s[:,i]),length(s[:,i]))
    ∇V' * m.μₛ(s[:,i]) + 0.5 * tr(m.σₛ(s[:,i])' * H * m.σₛ(s[:,i]))
end'
drifts_hyper = drift_hyper(V_test, s, m)
errors = maximum(abs.(drifts_exact - drifts_hyper))
```

Listing 5.5: Test of the hyper-dual approach to Itô's lemma.

Neural-network representation. We represent the value function with a neural network.

```
### Defining the neural net
model = Chain(
  Dense(1 ⇒ 25, Lux.gelu),
  Dense(25 ⇒ 25, Lux.gelu),
  Dense(25 ⇒ 1)
)
```

Listing 5.6: Neural network for the value function.

The model summary confirms the network's structure:

Julia REPL

```
julia> model
Chain(
  layer_1 = Dense(1 => 25, gelu_tanh),          # 50 parameters
  layer_2 = Dense(25 => 25, gelu_tanh),         # 650 parameters
  layer_3 = Dense(25 => 1),                     # 26 parameters
)  # Total: 726 parameters, plus 0 states.
```

Training setup. We initialize the parameters and optimizer state using the Adam optimizer with a learning rate of 0.001.

```julia
# Initialization
ps, ls = Lux.setup(rng, model) |> f64   # parameters/layer states
opt = Adam(1e-3)                        # optimizer
os = Optimisers.setup(opt, ps)          # optimizer state

# Loss function
function loss_fn(ps, ls, s, target)
  return mean(abs2, model(s, ps, ls)[1] - target)
end

# Target
function target(v, s, m; Δt = 0.2)
  hjb = s + drift_hyper(v, s, m) - m.ρ * v(s)
  return v(s) + hjb * Δt
end
```

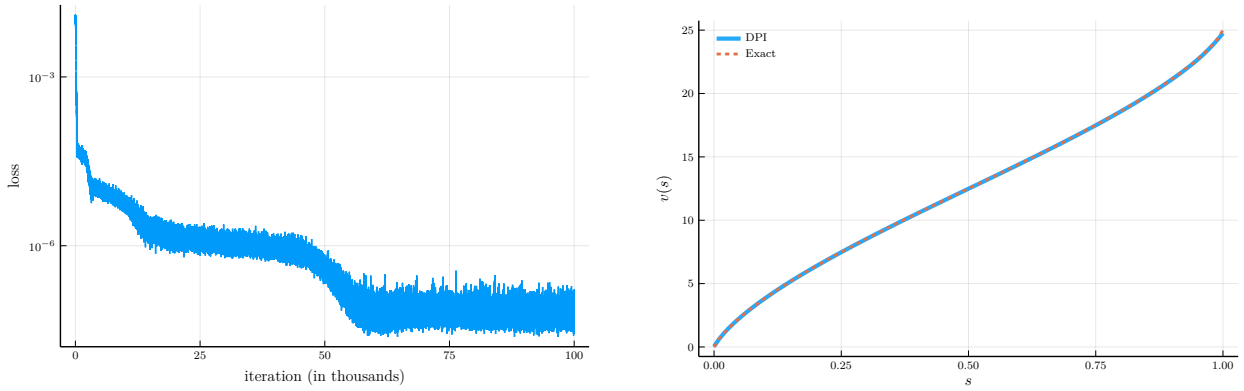
Listing 5.7: Initialization of parameters and optimizer state.

We adopt the first update rule for the policy evaluation step (Eq. (5.18)), which avoids the need for mixed-mode automatic differentiation.

> **Tip**
>
> **Mixed-mode automatic differentiation.** Implementing the second update rule (Eq. (5.22)) typically requires mixed-mode AD: forward-mode for the second derivative of the auxiliary function in Proposition 5.1, and reverse-mode for the parameter gradients ∇_{θ_V}. While `Zygote.jl` does not differentiate through dual-number arithmetic, a mixed approach using `Reactant.jl` can achieve this. See the documentation of `Lux.jl` for details on how to use mixed-mode AD. In the next section, we discuss an alternative approach to sidesteps the need for mixed-mode AD.

Training the network. We now train the neural network by sampling batches of states, computing the HJB residual, and minimizing the corresponding quadratic loss.

```julia
# # Training loop
loss_history = Float64[]
for i = 1:40_000
    s_batch = rand(rng, 1,  128)
    tgt     = target(s-> model(s, ps, ls)[1], s_batch, m, Δt = 1.0)
    loss    = loss_fn(ps, ls, s_batch, tgt)
    grad    = gradient(p -> loss_fn(p, ls, s_batch, tgt), ps)[1]
```



(a) Training loss over iterations.

(b) DPI prediction vs. analytical solution.

Figure 5.2: Training progress and fitted price–consumption ratio for the two-trees model.

```
    os, ps  = Optimisers.update(os,ps, grad)
    push!(loss_history, loss)
end
```

Listing 5.8: Training the neural net.

Results. Figure 5.2 shows the training loss and the fitted price–consumption ratio. The network fits the analytical solution with high precision: the loss decreases smoothly over iterations, and the fitted function tracks the true $v(s)$ almost perfectly.

Important

Handling boundary conditions. Notice that we did not explicitly impose the boundary conditions during training. In this model, the PDE for the value function is *degenerate* at the boundaries: the drift and/or diffusion vanish at $s = 0$ and $s = 1$. As a result, the boundary values are endogenously pinned by the PDE itself. This is a common feature of models with heterogeneity. The DPI algorithm handles this automatically, provided that boundary states are properly represented in the training samples.

Discussion. This example demonstrates the DPI workflow in its simplest form. We used the hyper-dual Itô method to compute the drift efficiently, represented the value function with a neural network, and trained it using gradient descent. Even though the two-trees model is one-dimensional, the same structure generalizes seamlessly to high-dimensional settings with multiple states and shocks. We next illustrate this by extending the model to a multi-tree (Lucas orchard) economy.

5.4.2 Asset Pricing: Lucas Orchard

We now extend the two-trees model to a multi-tree economy, known as the *Lucas orchard model* (Martin 2013). By varying the number of trees, we can examine how the DPI algorithm scales with the dimensionality of the state space and compare its performance with existing numerical methods in the literature.

The model. Consider a representative investor with log utility who can invest in a riskless asset and N risky assets. Each risky asset i pays a continuous dividend stream $D_{i,t}$ that follows a geometric Brownian motion:

$$\frac{dD_{i,t}}{D_{i,t}} = \mu_i\, dt + \sigma_i\, dB_{i,t}, \qquad i = 1, 2, \ldots, N, \tag{5.26}$$

where each $B_{i,t}$ is a Brownian motion satisfying $dB_{i,t}\, dB_{j,t} = 0$ for $i \neq j$.

As in the two-trees model, it is convenient to express the system in terms of the *dividend shares*

$$s_{i,t} = \frac{D_{i,t}}{C_t}, \qquad C_t = \sum_{i=1}^{N} D_{i,t},$$

where C_t is aggregate consumption. Appendix A.3 derives the law of motion for the vector of shares $\mathbf{s}_t = (s_{1,t}, \ldots, s_{N,t})^\top$:

$$d\mathbf{s}_t = \boldsymbol{\mu}_s(\mathbf{s}_t)\, dt + \boldsymbol{\sigma}_s(\mathbf{s}_t)\, d\mathbf{B}_t, \tag{5.27}$$

where $\boldsymbol{\mu}_s(\mathbf{s}_t)$ and $\boldsymbol{\sigma}_s(\mathbf{s}_t)$ are the drift and diffusion of the vector of dividend shares, respectively.

Valuation. Let $v_{i,t} \equiv P_{i,t}/C_t$ denote the price–consumption ratio of asset i. The pricing condition for a log investor is analogous to the two-trees case:

$$v_{i,t} = \mathbb{E}_t\left[\int_0^\infty e^{-\rho s} s_{i,t+s}\, ds\right]. \tag{5.28}$$

Because the process for each share $s_{i,t}$ depends on the entire vector $\mathbf{s}_t$, the price–consumption ratio for asset i must be a function of all dividend shares, $v_{i,t} = v_i(\mathbf{s}_t)$.

HJB equation. The stationary HJB equation for $v_i(\mathbf{s})$ is

$$\rho\, v_i(\mathbf{s}) = s_i + \nabla_{\mathbf{s}} v_i(\mathbf{s})^\top \boldsymbol{\mu}_s(\mathbf{s}) + \frac{1}{2}\operatorname{Tr}\left[\boldsymbol{\sigma}_s(\mathbf{s})^\top \mathbf{H}_{\mathbf{s}} v_i(\mathbf{s}) \boldsymbol{\sigma}_s(\mathbf{s})\right], \tag{5.29}$$

subject to the boundary conditions $v_i(0) = 0$ and $v_i(1) = 1/\rho$.

Dimensionality of the state space. The state vector $\mathbf{s}$ lies in the $(N-1)$-dimensional simplex:

$$\mathcal{S} = \left\{ \mathbf{s} \in [0,1]^N \;:\; \sum_{i=1}^{N} s_i = 1 \right\}.$$

In principle, one could eliminate one of the shares—for instance, $s_1 = 1 - \sum_{i=2}^{N} s_i$—to reduce the number of state variables to $N - 1$. In practice, however, the DPI algorithm can handle high-dimensional state spaces directly, since the neural-network representation of $v_i(\mathbf{s})$ does not rely on a tensor grid. Hence, it is often simpler and computationally convenient to retain all N shares as independent state variables. The resulting redundancy does not affect numerical stability and has negligible cost.

Julia implementation. We now implement the Lucas orchard model in Julia. The workflow of the DPI algorithm for the Lucas orchard model is virtually identical to that for the simple two-trees model.

As usual, we start by defining a model struct that contains both the parameters and the functions for the drift and diffusion of the state variable s.

```
@kwdef struct LucasOrchard
    ρ::Float64 = 0.04
    N::Int = 10
    σ::Vector{Float64} = sqrt(0.04) * ones(N)
    μ::Vector{Float64} = 0.02 * ones(N)
    μc::Function = s -> μ' * s
    σc::Function = s -> [s[i,:]' * σ[i] for i in 1:N]
    μₛ::Function = s ->  s .* (μ .- μc(s)- s.*σ.^2 .+
        sum(σc(s)[i].^2 for i in 1:N)) # drift of s
    σₛ::Function = s -> [s .* ([j == i ? σ[i] : 0 for j in 1:N] .-
        σc(s)[i]) for i in 1:N] # diffusion of s
end;
```

Listing 5.9: Model struct for the Lucas orchard model.

The drift and diffusion functions are written using Julia's broadcast operator `@.`, which efficiently applies operations elementwise over batched inputs. For a mini-batch of size B, the input $\mathbf{s}$ is an $N \times B$ matrix, where N is the number of assets and each column corresponds to a draw from the Dirichlet distribution. The drift function returns an $N \times B$ matrix of the same shape, while the diffusion function returns a vector of length N, where each element is an $N \times B$ matrix representing the exposures to the corresponding Brownian motion.

Listing 5.10 shows how to instantiate the model.

```
# Instantiate the model
m          = LucasOrchard(N = 10) # number of assets
rng, d     = MersenneTwister(0), Dirichlet(ones(m.N))
s_samples  = rand(rng, d, 1_000) # N x 1_000 matrix
vcat(m.μₛ(s_samples), m.σₛ(s_samples)...) # N*(N+1) x 1_000 matrix
```

Listing 5.10: Instantiation of the Lucas orchard model.

The vector of state variables $\mathbf{s}$ is sampled from the Dirichlet distribution, as $\mathbf{s}$ is non-negative and sums to one. Line 5 stacks the drift and diffusion of the state variable s into a single matrix of size $N(N+1) \times B$, allowing us to visually inspect the resulting dynamics.

Tip

Dirichlet distribution. The Dirichlet distribution is a probability distribution over the simplex, that is, the set of all vectors $\mathbf{x}$ of non-negative real numbers that sum to one. It generalizes the beta distribution to multiple dimensions, with pdf

$$f(\mathbf{x}; \boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{i=1}^{N} x_i^{\alpha_i - 1},$$

where $B(\boldsymbol{\alpha})$ is the multivariate beta function. By varying the concentration parameters $\boldsymbol{\alpha}$, we can control how dispersed or concentrated the draws are across the simplex. In the Lucas orchard model, using a symmetric Dirichlet with $\alpha_i = 1$ generates uniform coverage of the state space.

Hyper-dual Itô's lemma. We next implement the hyper-dual approach to Itô's lemma to compute the drift of the value function.

```
# Hyper-dual approach to Ito's lemma
function drift_hyper(V::Function, s::AbstractMatrix,
    m::LucasOrchard)
  N, σs, μs = m.N, m.σₛ(s), m.μₛ(s) # Preallocations
    F(ϵ) = sum(V(s .+ σs[i] .* (ϵ / sqrt(2)) .+
        μs .* (ϵ^2 / (2 * N))) for i in 1:N)
  return ForwardDiff.derivative(ϵ ->
        ForwardDiff.derivative(F, ϵ), 0.0)
end
```

Listing 5.11: Hyper-dual approach to Itô's lemma.

The implementation in Listing 5.11 for the Lucas orchard model is virtually identical to the implementation in Listing 5.4 for the two-trees model, but now we are dealing with the case of multiple state variables and Brownian motions.

Tip

The importance of preallocations. Relative to the two-trees model, we now preallocate arrays for the drift and diffusion of the state variable $\mathbf{s}$, rather than constructing them inside loops over $i = 1, \ldots, N$. In a low-dimensional example, this difference is negligible, but in higher-dimensional problems (large N or large batch size B), preallocations avoid repeated memory allocation and garbage collection, which can otherwise dominate runtime in tight training loops.

Having implemented the drift computation efficiently, we are now ready to embed it within the DPI algorithm to train the neural networks that approximate the value functions $v_i(\mathbf{s})$ for each asset.

Neural-network representation. Next, we represent the value function with a neural network.

```
### Defining the neural net
model = Chain(
  Dense(10 ⇒ 25, Lux.gelu),
  Dense(25 ⇒ 25, Lux.gelu),
  Dense(25 ⇒ 1)
)
```

Listing 5.12: Neural network for the value function.

We use essentially the same architecture as in the two-trees model, but now the input is the 10-dimensional vector of state variables $\mathbf{s}$ representing the dividend shares of each asset. The network has two hidden layers with 25 units each and GELU activations. The output is a scalar corresponding to the value of the first tree.

The model summary confirms the network's structure:

Julia REPL

```
julia> model
Chain(
    layer_1 = Dense(10 => 25, gelu_tanh),      # 275 parameters
    layer_2 = Dense(25 => 25, gelu_tanh),      # 650 parameters
    layer_3 = Dense(25 => 1),                  # 26 parameters
)         # Total: 951 parameters,
          #        plus 0 states.
```

Training setup. We initialize the parameters and optimizer state using the Adam optimizer with a learning rate of 10^{-3}. The `loss_fn` function computes the mean squared deviation between the model's prediction and the target, while the `target` function constructs the target value according to the false-transient update rule from Equation (5.18). Because we are focusing on the value of the first tree, the first term in line 14 selects the first component of the state vector $\mathbf{s}$.

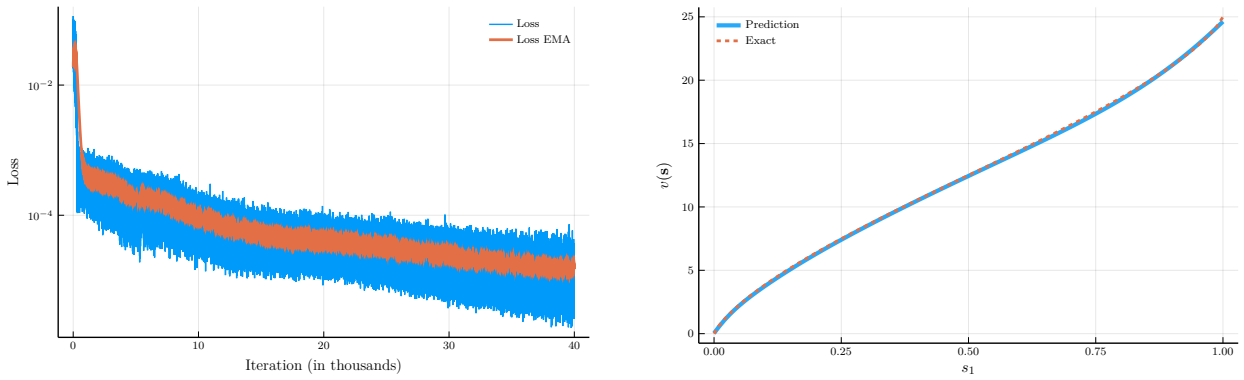
```
# Initializaion
ps, ls = Lux.setup(rng, model) ▷ f64
opt = Adam(1e-3)
os = Optimisers.setup(opt, ps)

```

```julia
# Loss function
function loss_fn(ps, ls, s, target)
  return mean(abs2, model(s, ps, ls)[1] - target)
end

# Target
function target(v, s, m; Δt = 0.2)
    v̄  = v(s)
    hjb = s[1,:]' + drift_hyper(v, s, m) - m.ρ * v̄
    return v̄  + hjb * Δt, mean(abs2, hjb)
end
```

Listing 5.13: Initialization of parameters and optimizer state.

Training the network. We now train the neural network by sampling batches of states, computing the HJB residual, and minimizing the corresponding quadratic loss.

```julia
# Training parameters
max_iter, Δt = 40_000, 1.0
# Sampling interior and boundary states
d_int  = Dirichlet(ones(m.N))            # Interior region
d_edge = Dirichlet(0.05 .* ones(m.N))  # Boundary region
# Loss history and exponential moving average loss
loss_history, loss_ema_history, α_ema = Float64[], Float64[], 0.99
# Training loop
p = Progress(max_iter; desc="Training...", dt=1.0) #progress bar
for i = 1:max_iter
    if rand(rng) < 0.50
        s_batch = rand(rng, d_int, 128)
    else
        s_batch = rand(rng, d_edge, 128)
    end
    v(s)       = model(s, ps, ls)[1] # define value function
    tgt, hjb_res = target(v, s_batch, m, Δt = Δt) #target/residual
    loss, back = Zygote.pullback(p -> loss_fn(p,ls,s_batch,tgt), ps)
    grad       = first(back(1.0)) # gradient
    os, ps     = Optimisers.update(os,ps, grad) # update parameters
    loss_ema   = i==1 ? loss : α_ema*loss_ema + (1.0-α_ema)*loss
    push!(loss_history, loss)
    push!(loss_ema_history, loss_ema)
    next!(p, showvalues = [(:iter, i),("Loss", loss),
        ("Loss EMA", loss_ema), ("HJB residual", hjb_res)])
end
```



(a) Training loss.

(b) Two-trees prediction.

Figure 5.3: Lucas orchard training and two-trees special case.

Listing 5.14: Training the neural net.

After defining the training parameters, we specify the state distributions from which we sample the training data. We use two Dirichlet distributions: one for the interior region and one for the boundary region of the simplex. For the interior region, we use a Dirichlet distribution with all parameters equal to one, which corresponds to the uniform distribution over the simplex. For the boundary region, we use a Dirichlet distribution with all parameters equal to 0.05, which is highly concentrated near the edges of the simplex. At each iteration, we sample from the interior region with probability 0.5 and from the boundary region with probability 0.5. This ensures that the model learns both the interior dynamics and the boundary behavior, which are critical for stability in degenerate PDEs.

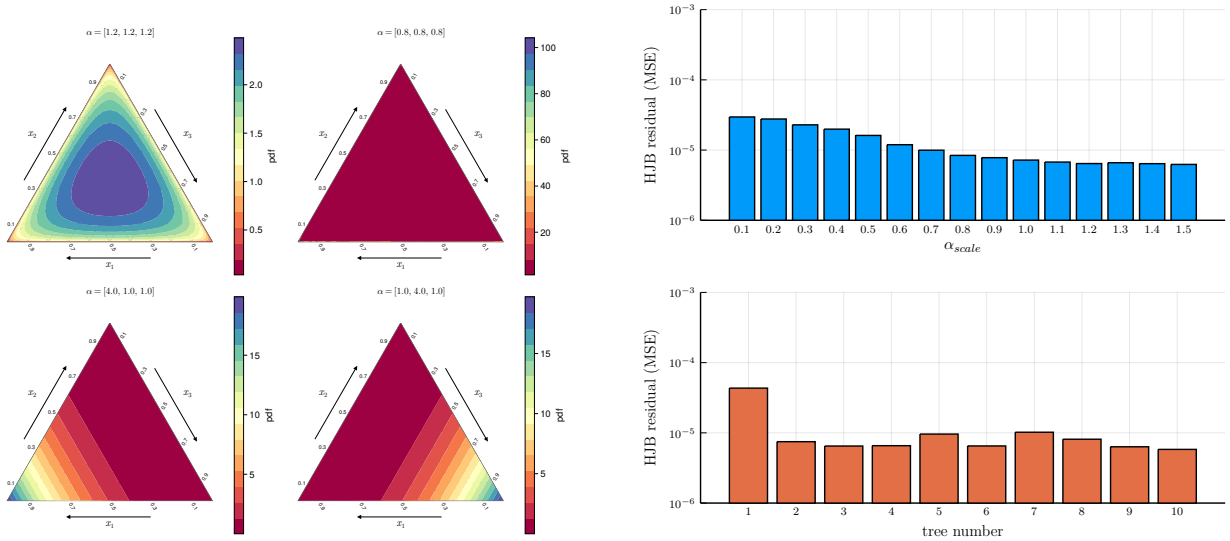
To monitor progress, we use the `ProgressMeter.jl` package to display a live progress bar. The progress bar reports the iteration count, the instantaneous loss, its exponential moving average, and the HJB residual. Apart from the sampling and progress display, the overall training loop is nearly identical to the one used for the two-trees model in Section 5.4.1.

Panel (a) of Figure 5.3 shows the evolution of the training loss and its exponential moving average. The loss decreases smoothly over iterations, while the moving average tracks it closely, confirming stable convergence. The loss in Figure 5.3 is computed on the training set. To evaluate generalization, we next assess model performance on held-out test sets, and analyze how the computational cost and accuracy scale with the number of trees N.

Test sets. We next evaluate the model's performance on out-of-sample test sets.

Our first test set is the two-trees special case. This corresponds to an extremely asymmetric configuration of the state vector $\mathbf{s} = (s_1, 1 - s_1, 0, \ldots, 0)$, which lies outside the region used for training. Although the Lucas orchard model in general has no closed-form analytical solution, for this specific configuration the model should reproduce the price–consumption ratio from the two-trees model. Panel (b) of Figure 5.3 confirms that the network's prediction coincides almost perfectly with the analytical benchmark, illustrating the model's ability to generalize beyond the training data.

Our second test set draws states from a symmetric Dirichlet distribution with parameters $\boldsymbol{\alpha} = \alpha_{\text{scale}}(1, 1, \ldots, 1)$, where α_{scale} controls the concentration of points within the simplex.



(a) Dirichlet densities. (b) Orchard residuals.

Figure 5.4: Dirichlet simplex visualizations and Lucas orchard residual diagnostics.

Panel (a) of Figure 5.4 illustrates the corresponding Dirichlet densities for different values of α_{scale}. When $\alpha_{\text{scale}} > 1$, samples are concentrated near the center of the simplex; when $\alpha_{\text{scale}} < 1$, samples are concentrated near the edges. The top panel of Figure 5.4(b) shows the mean-squared error (MSE) of the HJB residuals as α_{scale} ranges from 0.1 to 1.5. The residuals remain uniformly small across all regions of the simplex, with slightly better performance when points cluster near the center.

Our final test set draws from an asymmetric Dirichlet distribution where the j-th element of $\boldsymbol{\alpha}_j$ equals 4.0 and the remaining elements equal 1.0. This shifts mass toward one vertex of the simplex, allowing us to test the network's performance near highly skewed states. The bottom two plots of Figure 5.4(a) display the resulting densities for the first and second tree, respectively, while the bottom panel of Figure 5.4(b) reports the HJB residual MSE for $j = 1, \ldots, 10$. Residuals remain low for all trees, confirming that the network generalizes well to asymmetric configurations of the state space.

Overall, these tests demonstrate that the DPI-trained neural network achieves excellent accuracy and generalization across a wide range of states.

Comparison with other methods. We next compare the DPI algorithm with classical numerical methods for solving high-dimensional models, using the Lucas orchard economy as a benchmark. In particular, we focus on the performance of each method as we increase the number of trees—that is, the dimensionality of the state space.

Finite-difference schemes become computationally infeasible beyond a few dimensions, and Chebyshev collocation on full tensor-product grids also suffers from exponential growth in cost. We therefore compare the DPI algorithm to a sparse-grid version of the Chebyshev collocation method, known as the *Smolyak method* (Smolyak 1963).

Note

The Smolyak Sparse Grid Method. The Smolyak method is a sparse-grid technique for approximating multivariate functions with high accuracy while mitigating the exponential growth in computational cost that arises with tensor-product grids. Originally proposed by Smolyak (1963), it combines one-dimensional interpolation or quadrature formulas of varying precision to construct a multi-dimensional approximation that is both adaptive and efficient.
The key idea is to build a d-dimensional interpolant not on the full tensor grid (which scales as S^d for S points per dimension) but on a carefully selected subset of grid points:

$$\mathcal{A}(q,d) = \sum_{|\boldsymbol{i}|\leq q+d-1} \left(\Delta_{i_1} \otimes \cdots \otimes \Delta_{i_d}\right),$$

where q controls the order (or "level") of the approximation and Δ_{i_j} denotes the incremental contribution of the i_j-th one-dimensional rule. As q increases, the approximation becomes more accurate, but the number of grid points grows only polynomially in d rather than exponentially.
In economics and finance, the Smolyak method has become one of the workhorses for solving high-dimensional dynamic models (Judd et al. 2014; Brumm and Scheidegger 2017). It is commonly used with Chebyshev or Clenshaw–Curtis nodes to approximate value or policy functions, and with quadrature rules to approximate expectations.
Despite its efficiency relative to full tensor grids, the Smolyak method still faces practical limitations:

- The number of grid points grows rapidly with the approximation order q and the dimension d;
- The resulting system of equations can become ill-conditioned, especially for high-order polynomials;
- Sparse-grid interpolation can be difficult to parallelize efficiently in very high-dimensional settings.

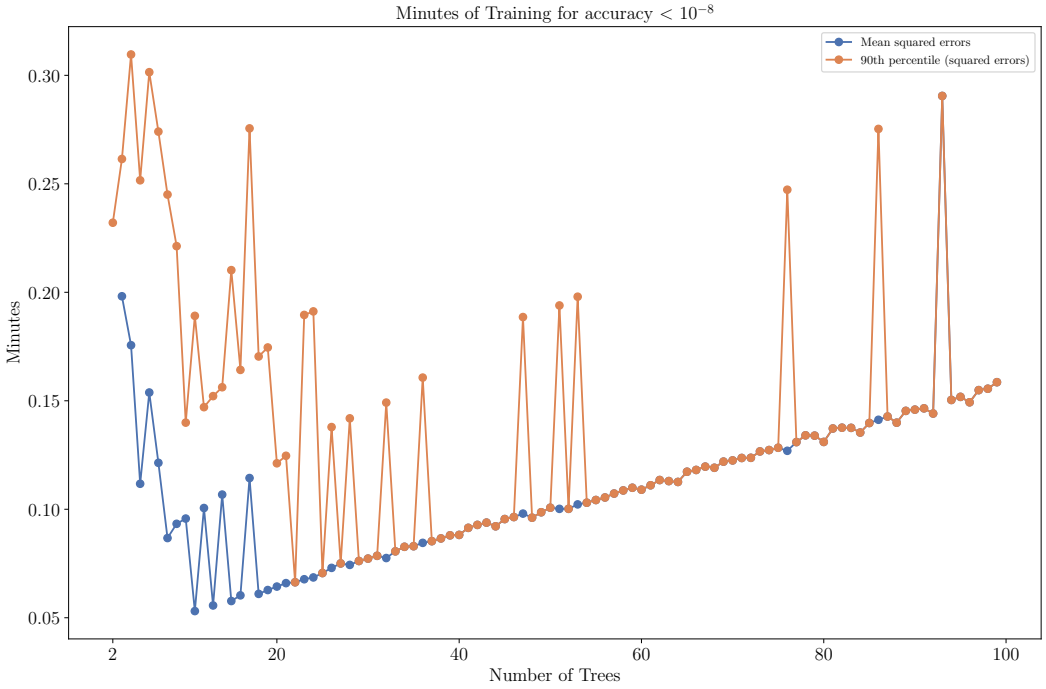
By contrast, the DPI algorithm replaces the global polynomial approximation with a neural-network representation that adapts its capacity to the local complexity of the value function and scales linearly with the number of parameters rather than exponentially with the dimension of the state space.

To assess performance, we solve the Lucas orchard model for an increasing number of trees, beginning with two and progressively raising N. For each economy, we measure the time-to-solution until the mean-squared error (MSE) of the HJB residuals falls below 10^{-8}. As a stricter criterion, we also record the time required for the 90th percentile of the squared errors to fall below 10^{-8}.

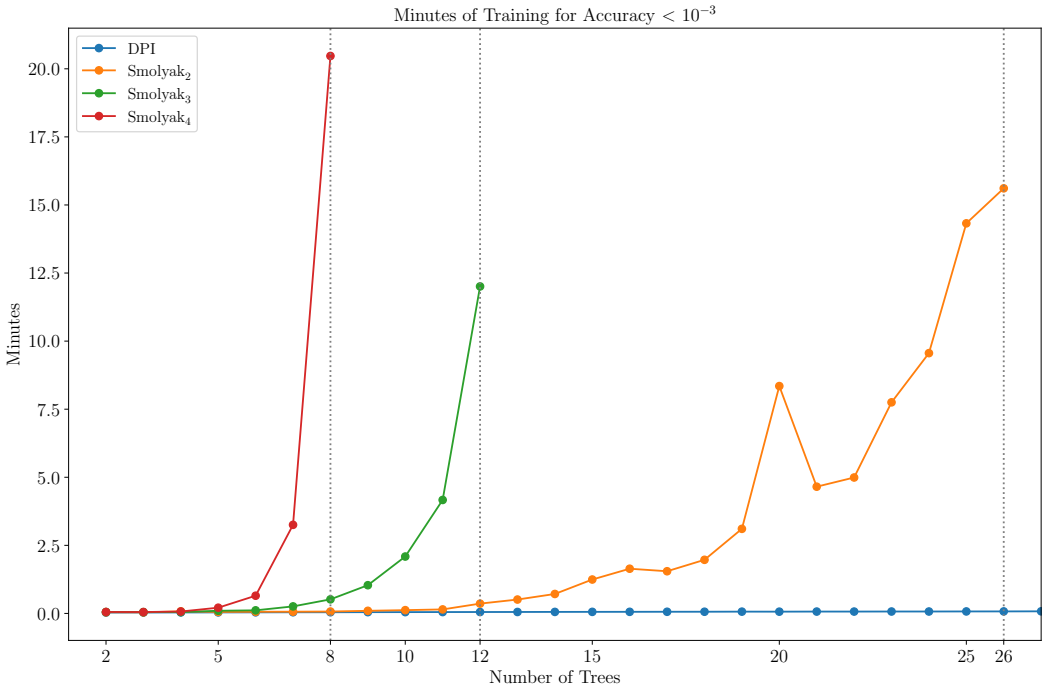
Panel (a) of Figure 5.5 reports the results. The DPI method achieves accurate solutions extremely quickly, even in high-dimensional settings. Increasing the number of trees or tightening the accuracy criterion has only a modest effect on computational cost. For

instance, in an economy with 100 trees, the DPI algorithm reaches an MSE of 10^{-8} in less than one minute.

Figure 5.5: Accuracy and Time-to-Solution in a Lucas Orchard Economy



(a) Time to solution.



(b) Smolyak methods and DPI algorithm MSEs.

Notes. Panel (a) shows the time-to-solution of the DPI algorithm, measured by the number of minutes required for the MSE or 90th-percentile squared error to fall below 10^{-8}. Panel (b) compares the time-to-solution of the DPI method and the Smolyak methods of orders 2, 3, and 4. The tolerance is set to 10^{-3}, the highest accuracy threshold reached by all Smolyak variants. The parameter values are $\rho = 0.04$, $\gamma = 1$, $\varrho = 0.0$, $\mu = 0.015$, and $\sigma = 0.1$. The HJB residuals are computed on a random sample of 2^{13} points from the state space.

Panel (b) of Figure 5.5 compares the time-to-solution of the DPI algorithm with the Smolyak methods of orders 2, 3, and 4. The Smolyak grids are solved using a conjugate-gradient method with a tolerance of 10^{-3}. The computational burden of the Smolyak approach rises sharply with both dimensionality and approximation order, and memory limits are reached for $N = 8$, 12, and 26 trees at orders 2, 3, and 4, respectively. In contrast, the DPI method maintains high accuracy and short solution times even for economies with more than 100 trees. This illustrates how DPI effectively overcomes the curse of dimensionality that constrains traditional sparse-grid methods.

5.4.3 Corporate Finance: Hennessy and Whited (2007)

We now apply the DPI algorithm to a corporate finance problem, a simplified version of the model in Hennessy and Whited (2007). This problem illustrates how the method can handle dynamic optimization with kinks in the value function and inaction regions in the optimal policy.

Model setup. Consider a firm with operating profits $\pi(k_t, z_t) = e^{z_t} k_t^{\alpha}$, where k_t denotes the capital stock and z_t the firm's log productivity. Log productivity follows an Ornstein–Uhlenbeck process:

$$dz_t = -\theta(z_t - \bar{z})\, dt + \sigma\, dB_t, \qquad \theta, \sigma > 0. \tag{5.30}$$

Given an investment rate i_t and depreciation rate δ, capital evolves as

$$dk_t = (i_t - \delta)\, k_t\, dt. \tag{5.31}$$

The state vector $\mathbf{s}_t = (k_t, z_t)^{\top}$ thus follows

$$d\mathbf{s}_t = \boldsymbol{\mu}_s(\mathbf{s}_t, i_t)\, dt + \boldsymbol{\sigma}_s(\mathbf{s}_t, i_t)\, dB_t, \tag{5.32}$$

with drift and diffusion

$$\boldsymbol{\mu}_s(\mathbf{s}_t, i_t) = \begin{bmatrix} (i_t - \delta) k_t \\ -\theta(z_t - \bar{z}) \end{bmatrix}, \qquad \boldsymbol{\sigma}_s(\mathbf{s}_t, i_t) = \begin{bmatrix} 0 \\ \sigma \end{bmatrix}.$$

The firm faces quadratic adjustment costs $\Lambda(k_t, i_t) = \frac{1}{2}\chi k_t (i_t - \delta)^2$ and linear equity issuance costs $\lambda > 0$. Operating profits net of adjustment costs are

$$D^*(k, z, i) = e^z k^{\alpha} - \left(i + \tfrac{1}{2}\chi(i - \delta)^2\right) k. \tag{5.33}$$

If net operating profits are negative, the firm issues equity to cover the shortfall, incurring the cost λ. Dividends are therefore given by

$$D(k, z, i) = \begin{cases} D^*(k, z, i), & D^*(k, z, i) \geq 0, \\ D^*(k, z, i)(1 + \lambda), & D^*(k, z, i) < 0. \end{cases} \tag{5.34}$$

The firm chooses the investment rate i_t to maximize the expected discounted value of future dividends:

$$v(\mathbf{s}_0) = \max_{\{i_t\}_{t \geq 0}} \mathbb{E}\left[\int_0^{\infty} e^{-\rho s} D(k_s, z_s, i_s)\, ds\right], \tag{5.35}$$

subject to (5.32).

The HJB equation. Equation (5.35) gives the sequential formulation of the firm's problem. In its recursive form, the value function $v(\mathbf{s})$ and policy function $i(\mathbf{s})$ satisfy

$$0 = \max_i \text{HJB}(\mathbf{s}, i, v(\mathbf{s})), \tag{5.36}$$

where

$$\text{HJB}(\mathbf{s}, i, v) = D(k, z, i) + \nabla v^\top \boldsymbol{\mu}_s(\mathbf{s}, i) + \tfrac{1}{2}\boldsymbol{\sigma}_s(\mathbf{s}, i)^\top \mathbf{H} v \, \boldsymbol{\sigma}_s(\mathbf{s}, i) - \rho v. \tag{5.37}$$

The first-order condition for the optimal investment rate is

$$\frac{\partial \text{HJB}}{\partial i} = -\big(1 + \lambda \mathbf{1}_{D^*(k,z,i)<0}\big)\big[1 + \chi(i - \delta)\big]k + v_k(\mathbf{s})k = 0. \tag{5.38}$$

When the shadow value of capital $v_k(\mathbf{s})$ equals the marginal adjustment cost, the firm is indifferent to adjusting capital. Because of the issuance cost λ, the value function exhibits a kink at this point, producing an inaction region where the firm finds it optimal neither to pay dividends nor to issue equity. In Appendix A.4, we provide an explicit characterization of the optimal investment policy for this version of the Hennessy and Whited (2007) model in terms of the derivatives of the value function. Below, we show how the DPI algorithm can recover the solution directly, even when solving for the policy variable analytically is infeasible or cumbersome.

A special case. To gain intuition about the model's behavior, it is useful to begin with a simple special case. We assume that productivity is constant, so $\theta = \sigma = 0$, and that investment adjustment costs are prohibitively high, so $\chi \to \infty$. This implies that it is optimal to keep the capital stock constant, $i(k, z) = \delta$. From the HJB equation, we obtain the value function:

$$v(k, z) = \frac{D(k, z, \delta)}{\rho} = \begin{cases} \frac{e^z k^\alpha - \delta k}{\rho}, & \text{if } k \le k_{\max}(z), \\ \frac{e^z k^\alpha - \delta k}{\rho}(1 + \lambda), & \text{if } k > k_{\max}(z), \end{cases} \tag{5.39}$$

where $k_{\max}(z) = \left(\frac{e^z}{\delta}\right)^{\frac{1}{1-\alpha}}$.

At $k = k_{\max}(z)$, the firm switches from internal to external financing, so the equity issuance cost λ becomes binding. This creates a kink in the value function $v(k, z)$, corresponding to the boundary between the internal- and external-finance regions. Equation (5.39) therefore defines a smooth function of the state variables, except at $k = k_{\max}(z)$, where its derivative is discontinuous.

These properties are important when choosing the neural network architecture. A standard ReLU activation function can approximate the kink well but produces piecewise-linear derivatives, making it difficult to maintain smoothness elsewhere. Conversely, a GELU activation yields smooth derivatives but tends to blur sharp discontinuities. To capture both behaviors, we introduce a *shifted ReLU* activation function:

$$\text{ShiftedReLU}(x; c) = \max(0,\, x - c), \tag{5.40}$$

where c is a trainable parameter that shifts the activation threshold. By learning the shift parameter c, the neural network can capture the discontinuity in the derivative at $k = k_{\max}(z)$ while remaining smooth elsewhere.

Listing 5.15 shows how to implement the shifted ReLU activation function in `Lux.jl`.

```julia
# Shifted ReLU activation function
struct ShiftedReLU <: Lux.AbstractLuxLayer end
Lux.initialparameters(::Random.AbstractRNG, ::ShiftedReLU) =
    (c = [0.0f0],)
function (m::ShiftedReLU)(x, ps, st)
    c = ps.c[1]
    return max.(0, x .- c), st
end

v_core = Chain(
    Dense(2, 24, Lux.gelu),
    Dense(24, 12, Lux.gelu),
    ShiftedReLU(),
    Dense(12, 1)
)

# Enforce boundary condition: V(0, z) = 0
v_net(s, θᵥ) =  s[1:1, :].^m.α .* v_core(s, θᵥ, stᵥ)[1]
```

Listing 5.15: Shifted ReLU activation function.

We first create a struct to store the layer parameters and define the forward pass of the activation. Having defined the layer, we can include it in the neural network as a standard `Lux` component. Listing 5.15 also shows how to enforce the boundary condition $v(0, z) = 0$ by defining a wrapper function that multiplies the network output by a term that vanishes at $k = 0$. A convenient choice is k^α, which not only enforces the boundary condition but also captures the asymptotic behavior of the value function as $k \to 0$, where $v_k(k, z)$ becomes unbounded.

Policy evaluation. In this special case, the policy function is known and constant, $i(k, z) = \delta$, so there is no policy improvement step. We therefore solve for the value function directly using the DPI algorithm.

We start by defining the model struct. It contains the model default parameters and the functions for the drift and diffusion of the state vector.

```julia
# Model parameters
@kwdef struct HennessyWhited
    α::Float64      = 0.55
    θ::Float64      = 0.26
    z̄ ::Float64      = -2.2976
    σz::Float64     = 0.123
    δ::Float64      = 0.1
    χ::Float64      = 20.0
    λ::Float64      = 0.059
```

```
    ρ::Float64      = 0.04
    kmax::Float64   = 2.5
    μₛ::Function = (s,i) -> vcat((i .-δ) .* s[1,:]',  # μk
                                 -θ.*(s[2,:] .- z̄)')  # μz
    σₛ::Function = (s,i) -> vcat(zeros(1,size(s,2)),  # σk
                                 σz*ones(1,size(s,2))) # σz
end;
```

Listing 5.16: Model struct.

Listing 5.17 uses the hyper-dual approach to Itô's lemma to compute the drift of the value function. It also defines the target value function and the loss function for this task.

```
# Hyper-dual approach to Ito's lemma
function drift_hyper(s::AbstractMatrix, m::HennessyWhited, θᵥ)
    ī       = m.δ * ones(1, size(s,2))
    μₛ, σₛ  = m.μₛ(s,ī), m.σₛ(s,ī)
    F(ϵ)    = v_net(s .+ σₛ .* (ϵ / sqrt(2f0)) .+
                        μₛ .* (ϵ^2 / (2f0)), θᵥ)
  return ForwardDiff.derivative(ϵ ->
            ForwardDiff.derivative(F,ϵ),0.0f0)
end

function dividends(s, m::HennessyWhited)
    k = @view s[1,:]; z = @view s[2,:]
    π = (exp.(z) .* k.^m.α)'
    D_star = π .-  m.δ * k' # i = δ
    return D_star .* (1 .+ m.λ * (D_star .< 0))
end

function hjb_residual(s, m::HennessyWhited, θᵥ)
    D, drift = dividends(s, m), drift_hyper(s, m, θᵥ)
    return D .+ drift .- m.ρ .* v_net(s, θᵥ)
end

function target(s, m, θᵥ; Δtᵥ = 0.2f0)
    hjb      = hjb_residual(s, m, θᵥ)
    tgtᵥ     = v_net(s, θᵥ) + hjb * Δtᵥ
    return tgtᵥ, mean(abs2, hjb)
end

function loss_fn(dnn, θ, s, target)
    return mean(abs2, dnn(s, θ) - target)
end
```

Listing 5.17: Target value function for the special case.

We then train the neural network using the loop in Listing 5.18. The model parameters and neural network architecture follow the same setup as in the previous section. We first define the hyperparameters and specify the sampling of the state space. Each iteration of the training loop consists of sampling a mini-batch of states, computing the target value using the HJB equation, and updating the neural network parameters using the Adam optimizer.

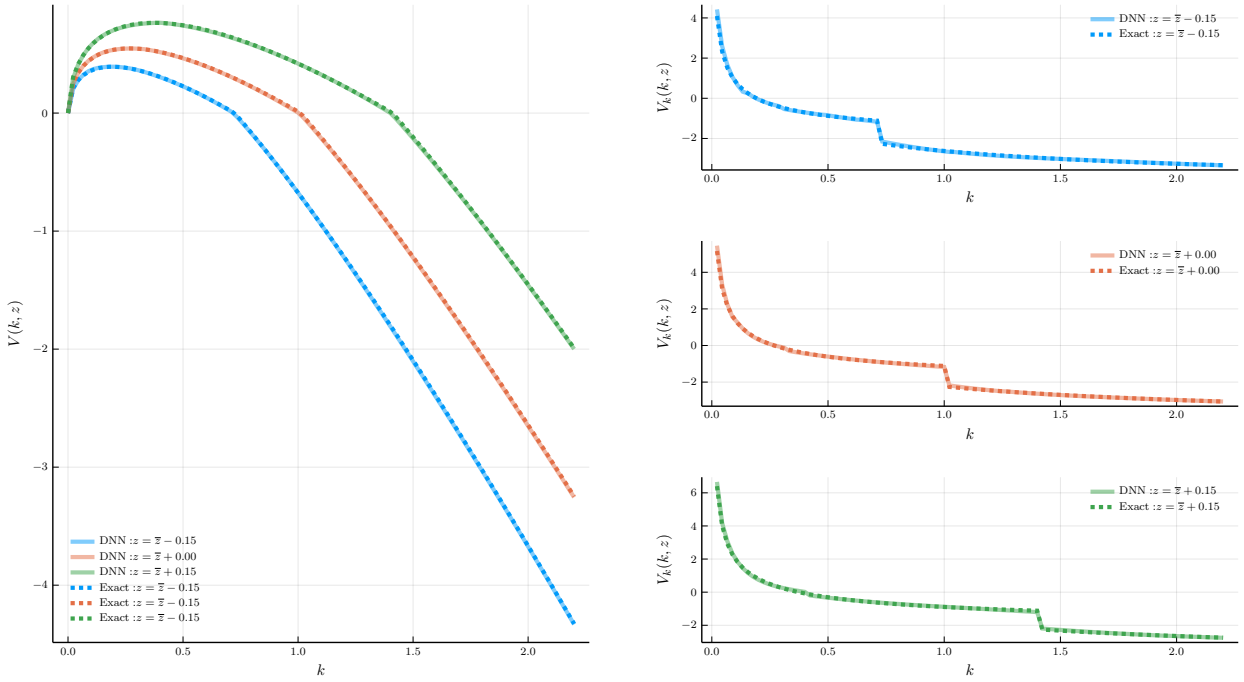
```
# Initializaion:
rng        = Xoshiro(1234)
θᵥ, stᵥ    = Lux.setup(rng, v_core)
optᵥ       = Optimisers.Adam(1e-3, (0.9, 0.98))
osᵥ        = Optimisers.setup(optᵥ, θᵥ)

max_iter, Δtᵥ  = 200_000, 0.1
d_k, d_z       = Uniform(0.01, m.kmax), Normal(m.z̄ , m.σz/sqrt(2m.θ))

p = Progress(max_iter; desc = "Training...", dt = 1.0)
for it in 1:max_iter
    # Sample states
    k_batch = rand(rng, d_k, 100)'
    z_batch = rand(rng, d_z, 100)'
    s_batch = vcat(k_batch, z_batch)
    # Computing targets and policy evaluation step
    tgt, hjb_res = target(s_batch, m, θᵥ, Δtᵥ = Δtᵥ)
    loss, back = Zygote.pullback(p->loss_fn(v_net,p,s_batch,tgt),θᵥ)
    grad       = first(back(1.0))
    osᵥ, θᵥ    = Optimisers.update(osᵥ,θᵥ, grad)
    # Progress bar
    next!(p, showvalues = [(:iter, it),("Loss_v", lossᵥ),
                           ("HJB residual", hjb_res)])
end
```

Listing 5.18: Training the neural network for the special case.

Tip

The Ornstein-Uhlenbeck process. When sampling for log productivity, we draw from the *unconditional distribution* for z_t, that is, its long-run distribution. To derive this distribution, notice we can write z_t in terms of the following stochastic integral:

$$z_t = \bar{z} + (z_0 - \bar{z})e^{-\theta t} + \sigma \int_0^t e^{-\theta(t-s)} dB_s,$$



(a) Value function.

(b) Value function derivative.

Figure 5.6: Value function and its derivative for the special case.

Using Itô's isometry, we can show that the distribution of z_t is

$$z_t \sim \mathcal{N}\left(\bar{z} + (z_0 - \bar{z})e^{-\theta t}, \frac{\sigma^2}{2\theta}(1 - e^{-2\theta t})\right).$$

Taking the limit as $t \to \infty$, we get the unconditional distribution: $\mathcal{N}(\bar{z}, \frac{\sigma^2}{2\theta})$.

Figure 5.6 presents the results. Panel (a) shows the value function, while Panel (b) plots its derivative with respect to capital. The neural network's predictions are compared with the analytical benchmark from Equation (5.39). The value function is smooth except for a kink at $k = k_{\max}(z)$, where dividends switch from internal to external financing. The neural network reproduces this feature precisely, matching the analytical solution almost perfectly in both levels and derivatives. This confirms that the shifted ReLU activation provides an effective representation for problems involving kinks.

This exercise illustrates how the policy evaluation component of the DPI algorithm can recover the value function when the control is fixed. Next, we allow investment to be an endogenous choice variable and solve the full Hennessy–Whited model.

Solving the full model. We now solve the full Hennessy–Whited model using the DPI algorithm. We start by defining the neural network for the value function and for the investment policy function, as shown in Listing 5.19.

```julia
# Value function
v_core = Chain(
    Dense(2, 32, Lux.gelu),
    Dense(32, 32, Lux.gelu),
    Dense(32, 12, Lux.gelu),
    ShiftedReLU(),
    Dense(12, 1)
)
v_net(s, θᵥ) =  s[1:1, :].^m.α .* v_core(s, θᵥ, stᵥ)[1]

# Investment policy function
i_core = Chain(
    Dense(2, 32, Lux.gelu),
    Dense(32, 32, Lux.gelu),
    Dense(32, 12, Lux.gelu),
    ShiftedReLU(),
    Dense(12, 1)
)
i_net(s, θᵢ) = i_core(s, θᵢ, stᵢ)[1]
```

Listing 5.19: Neural network for the value function and the investment policy function.

Listing 5.20 shows the computation of the HJB residual and the corresponding loss function. These calculations are nearly identical to the ones for the special case, except that they depend on the parameters of the investment network $\boldsymbol{\theta}_i$. The dividends function computes both D and D^*, the latter of which is used to compute the FOC residual later on.

```julia
function drift_hyper(s::AbstractMatrix, m::HennessyWhited, θᵥ, θᵢ)
    ī       = i_net(s, θᵢ)
    μₛ, σₛ  = m.μₛ(s,ī ), m.σₛ(s,ī )
    F(ϵ)    = v_net(s .+ σₛ .* (ϵ / sqrt(2)) .+
                        μₛ .* (ϵ^2 / 2.0), θᵥ)
  return ForwardDiff.derivative(ϵ ->
            ForwardDiff.derivative(F, ϵ), 0.0)
end

function dividends(s, m::HennessyWhited, θᵢ)
    k = @view s[1,:]; z = @view s[2,:]
    ī  = i_net(s, θᵢ)
    π = (exp.(z) .* k.^m.α)'
    D_star = π .- (ī  + 0.5f0 * m.χ * (ī  .- m.δ).^2).* k'
    D      = D_star .* (1 .+ m.λ * (D_star .< 0))
    return D, D_star
end
```

```julia

function hjb_residual(s, m::HennessyWhited, θᵥ, θᵢ)
    D, drift = dividends(s, m, θᵢ)[1], drift_hyper(s, m, θᵥ, θᵢ)
    return D .+ drift .- m.ρ .* v_net(s, θᵥ)
end

function loss_fn(dnn, θ, s, target)
    return mean(abs2, dnn(s, θ) - target)
end
```

Listing 5.20: Computing the HJB residual and the loss function.

Listing 5.21 shows the computation of the FOC residual and the targets used for training. We start by computing $v_k(\mathbf{s})$ using the forward-mode differentiation of the value function network. As the `ForwardDiff.jl` package works with scalar inputs, we first define a function that computes the derivative when taking a single value for k and z, and then define a batched version that takes a $2 \times B$ matrix of states. Given $v_k(\mathbf{s})$, the FOC residual is computed as the difference between the marginal adjustment cost and the value of capital. Finally, we define the targets for the value function and the investment policy function.

```julia
## Computing vk for the FOC residual
# Scalar version
function vk_scalar(k::T, z::T, ps_v) where {T}
    ForwardDiff.derivative(K -> begin
        x = @SVector [K, z]     # static vector
        v_net(x, ps_v)[1]
    end, k)
end

# Batched version
function vk(s::AbstractMatrix, ps_v)
    B = size(s, 2)
    vals = [vk_scalar(s[1,b], s[2,b], ps_v) for b in 1:B]
    return reshape(vals, 1, B)  # 1×B
end

function foc_residual(s, m::HennessyWhited, θᵥ, θᵢ)
    ī        = i_net(s, θᵢ)
    D_star   = dividends(s, m, θᵢ)[2]
    return @. vk(s, θᵥ) .- (1.0 + m.χ * (ī  .- m.δ)) * (1.0 +
                m.λ * (D_star .< 0.0))
end

function targets(s, m, θᵥ, θᵢ; Δtᵥ = 0.2, Δtᵢ = 0.2)
    hjb      = hjb_residual(s, m, θᵥ, θᵢ)
```

```
    foc      = foc_residual(s, m, θᵥ, θᵢ)
    tgtᵥ     = v_net(s, θᵥ) + hjb * Δtᵥ
    tgtᵢ     = i_net(s, θᵢ) + foc * Δtᵢ
    return tgtᵥ, tgtᵢ, mean(abs2, hjb), mean(abs2, foc)
end
```

Listing 5.21: Computing the FOC residual and the loss function.

We then train the neural network using the loop in Listing **??**. The model parameters and neural network architecture follow the same setup as in the previous section. We first define the hyperparameters and specify the sampling of the state space. Each iteration of the training loop consists of sampling a mini-batch of states, computing the target value using the HJB equation, and updating the neural network parameters using the Adam optimizer.

Appendix A

Appendix

A.1 The multivariate version of Itô's lemma

In Chapter 5, we used the multivariate version of Itô's lemma to derive the HJB equation. In this section, we provide a derivation of the multivariate version of Itô's lemma, which builds on the multivariate Taylor expansion and the rules of Itô's calculus. For a comprehensive treatment of matrix differential calculus, see Magnus and Neudecker (1999).

A.1.1 The multivariate Taylor expansion

First-order approximation. Let $f(\mathbf{s}) \in \mathbb{R}$ be a scalar-valued function of the state vector $\mathbf{s} \in \mathbb{R}^n$. The derivative of $f(\mathbf{s})$ is a linear operator that maps a small perturbation $\mathbf{h} \in \mathbb{R}^n$ in $\mathbf{s}$ to a small perturbation in $f(\mathbf{s})$, that is, the *differential* of $f(\mathbf{s})$ is given by:

$$df(\mathbf{s}; \mathbf{h}) = Df(\mathbf{s})\mathbf{h}. \tag{A.1}$$

Equivalently, this gives the first-order approximation of $f(\mathbf{s})$ around a given point $\mathbf{s}_0 \in \mathbb{R}^n$:

$$f(\mathbf{s} + \mathbf{h}) = f(\mathbf{s}) + Df(\mathbf{s})\mathbf{h} + o(\|\mathbf{h}\|). \tag{A.2}$$

As $\mathbf{s}$ is a column vector, $Df(\mathbf{s}_0)$ must be a row vector, such that the product $Df(\mathbf{s}_0)(\mathbf{s} - \mathbf{s}_0)$ is a scalar. Formally, the derivative is given by:

$$Df(\mathbf{s}_0) = \frac{\partial f}{\partial \mathbf{s}^\top}\bigg|_{\mathbf{s}=\mathbf{s}_0} = \left(\frac{\partial f}{\partial \mathbf{s}_1}, \frac{\partial f}{\partial \mathbf{s}_2}, \ldots, \frac{\partial f}{\partial \mathbf{s}_n}\right). \tag{A.3}$$

Notice that we denote the derivative as $\frac{\partial f}{\partial \mathbf{s}^\top}$, where the transpose in $\mathbf{s}^\top$ acts as a reminder that the derivative is a row vector.

It is often useful to work with the gradient of $f(\mathbf{s})$ instead of the derivative. In our setting, the gradient is a column vector, given by the transpose of the derivative:

$$\nabla f(\mathbf{s}) = Df(\mathbf{s})^\top = \left(\frac{\partial f}{\partial \mathbf{s}_1}, \frac{\partial f}{\partial \mathbf{s}_2}, \ldots, \frac{\partial f}{\partial \mathbf{s}_n}\right)^\top. \tag{A.4}$$

We can then write the first-order approximation of $f(\mathbf{s})$ around a given point $\mathbf{s}_0 \in \mathbb{R}^n$ as:

$$f(\mathbf{s} + \mathbf{h}) = f(\mathbf{s}) + \nabla f(\mathbf{s})^\top \mathbf{h} + o(\|\mathbf{h}\|). \tag{A.5}$$

Second-order approximation. We can define the second differential of $f(\mathbf{s})$, which is simply the differential of the first differential:

$$d^2 f(\mathbf{s};\mathbf{h}) = d(df(\mathbf{s};\mathbf{h});\mathbf{h}). \tag{A.6}$$

Using the expression for the first differential, we can write the second differential as:

$$d^2 f(\mathbf{s};\mathbf{h}) = d(\nabla f(\mathbf{s})^\top \mathbf{h}) = d(\nabla f(\mathbf{s}))^\top \mathbf{h} = (D\nabla f(\mathbf{s})\mathbf{h})^\top \mathbf{h} = \mathbf{h}^\top (D\nabla f(\mathbf{s}))^\top \mathbf{h}, \tag{A.7}$$

where $D\nabla f(\mathbf{s}) \in \mathbb{R}^{n\times n}$ is *Jacobian* matrix of $\nabla f(\mathbf{s})$.

The matrix inside the quadratic form is the *Hessian* matrix of $f(\mathbf{s})$, given by:

$$\mathbf{H}f(\mathbf{s}) = D\nabla f(\mathbf{s})^\top = \frac{\partial^2 f}{\partial \mathbf{s}\partial \mathbf{s}^\top} = \begin{pmatrix} \frac{\partial^2 f}{\partial \mathbf{s}_1 \partial \mathbf{s}_1} & \frac{\partial^2 f}{\partial \mathbf{s}_1 \partial \mathbf{s}_2} & \cdots & \frac{\partial^2 f}{\partial \mathbf{s}_1 \partial \mathbf{s}_n} \\ \frac{\partial^2 f}{\partial \mathbf{s}_2 \partial \mathbf{s}_1} & \frac{\partial^2 f}{\partial \mathbf{s}_2 \partial \mathbf{s}_2} & \cdots & \frac{\partial^2 f}{\partial \mathbf{s}_2 \partial \mathbf{s}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial \mathbf{s}_n \partial \mathbf{s}_1} & \frac{\partial^2 f}{\partial \mathbf{s}_n \partial \mathbf{s}_2} & \cdots & \frac{\partial^2 f}{\partial \mathbf{s}_n \partial \mathbf{s}_n} \end{pmatrix}. \tag{A.8}$$

Given the second differential, we can obtain a second-order approximation of $f(\mathbf{s})$:

$$f(\mathbf{s}+\mathbf{h}) = f(\mathbf{s}) + df(\mathbf{s};\mathbf{h}) + \frac{1}{2} d^2 f(\mathbf{s};\mathbf{h}) + o(\|\mathbf{h}\|^2). \tag{A.9}$$

Using the expression for the first and second differential, we obtain the second-order Taylor expansion of $f(\mathbf{s})$ around a given point $\mathbf{s}_0 \in \mathbb{R}^n$ as:

$$f(\mathbf{s}+\mathbf{h}) = f(\mathbf{s}) + \nabla f(\mathbf{s})^\top \mathbf{h} + \frac{1}{2}\mathbf{h}^\top \mathbf{H}f(\mathbf{s})\mathbf{h} + o(\|\mathbf{h}\|^2). \tag{A.10}$$

A.1.2 Itô's lemma in higher dimensions

Suppose now that $\mathbf{s}$ follows a diffusion process given by:

$$d\mathbf{s} = \mathbf{f}(\mathbf{s},\mathbf{c})dt + \mathbf{g}(\mathbf{s},\mathbf{c})d\mathbf{B}, \tag{A.11}$$

where $\mathbf{f}(\mathbf{s},\mathbf{c}) \in \mathbb{R}^n$ is the drift and $\mathbf{g}(\mathbf{s},\mathbf{c}) \in \mathbb{R}^{n\times m}$ is the diffusion matrix. We can think of $d\mathbf{s}$ as a small perturbation in $\mathbf{s}$, so using the second-order Taylor expansion of $f(\mathbf{s})$ around $\mathbf{s}$, we obtain:

$$df(\mathbf{s}) = \nabla f(\mathbf{s})^\top d\mathbf{s} + \frac{1}{2} d\mathbf{s}^\top \mathbf{H}f(\mathbf{s}) d\mathbf{s} + o(\|d\mathbf{s}\|^2). \tag{A.12}$$

Now, using the Itô's product rules, $d\mathbf{B}d\mathbf{B}^\top = I_m dt$ and $d\mathbf{B}dt = dt^2 = 0$, and taking expectations, we obtain:

$$df(\mathbf{s}) = \nabla f(\mathbf{s})^\top \mathbf{f}(\mathbf{s},\mathbf{c})dt + \frac{1}{2}\mathbb{E}\left[d\mathbf{B}^\top \mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}f(\mathbf{s})\mathbf{g}(\mathbf{s},\mathbf{c})d\mathbf{B}\right] + o(dt^2), \tag{A.13}$$

using the fact that $\mathbb{E}[\|d\mathbf{s}\|^2] = \mathcal{O}(dt^2)$.

Notice that the second term in the expression above is a scalar, so it is equal to its trace. Using the cyclic property of the trace ($\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$), we obtain:

$$\mathbb{E}\left[d\mathbf{B}^\top \mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}f(\mathbf{s})\mathbf{g}(\mathbf{s},\mathbf{c})d\mathbf{B}\right] = \mathrm{Tr}\left[\mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H}f(\mathbf{s})\mathbf{g}(\mathbf{s},\mathbf{c})\mathbb{E}\left[d\mathbf{B}d\mathbf{B}^\top\right]\right]. \tag{A.14}$$

Using the fact that $\mathbb{E}[d\mathbf{B}d\mathbf{B}^\top] = I_m dt$, we obtain the multivariate version of Itô's lemma:

$$df(\mathbf{s}) = \mathcal{D}f(\mathbf{s})dt, \tag{A.15}$$

where

$$\mathcal{D}f(\mathbf{s}) = \nabla f(\mathbf{s})^\top \mathbf{f}(\mathbf{s},\mathbf{c}) + \frac{1}{2}\operatorname{Tr}\left[\mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H} f(\mathbf{s}) \mathbf{g}(\mathbf{s},\mathbf{c})\right]. \tag{A.16}$$

A.2 The hyper-dual approach to Itô's lemma

A.2.1 Proof of Proposition 5.1

In this section, we provide a proof of Proposition 5.1 in Chapter 5. We start from the multivariate version of Itô's lemma for a function $V(\mathbf{s})$:

$$dV(\mathbf{s}) = \mathcal{D}V(\mathbf{s})dt + \nabla V(\mathbf{s})^\top \mathbf{g}(\mathbf{s},\mathbf{c})d\mathbf{B}, \tag{A.17}$$

where

$$\mathcal{D}V(\mathbf{s}) = \nabla V(\mathbf{s})^\top \mathbf{f}(\mathbf{s},\mathbf{c}) + \frac{1}{2}\operatorname{Tr}\left[\mathbf{g}(\mathbf{s},\mathbf{c})^\top \mathbf{H} V(\mathbf{s}) \mathbf{g}(\mathbf{s},\mathbf{c})\right], \tag{A.18}$$

and we drop the dependence on the control $\mathbf{c}$ for simplicity.

The matrix inside the trace is the Hessian matrix of $V(\mathbf{s})$, given by:

$$\mathbf{H}V(\mathbf{s}) = \frac{\partial^2 V}{\partial \mathbf{s} \partial \mathbf{s}^\top} = \begin{pmatrix} \frac{\partial^2 V}{\partial \mathbf{s}_1 \partial \mathbf{s}_1} & \frac{\partial^2 V}{\partial \mathbf{s}_1 \partial \mathbf{s}_2} & \cdots & \frac{\partial^2 V}{\partial \mathbf{s}_1 \partial \mathbf{s}_n} \\ \frac{\partial^2 V}{\partial \mathbf{s}_2 \partial \mathbf{s}_1} & \frac{\partial^2 V}{\partial \mathbf{s}_2 \partial \mathbf{s}_2} & \cdots & \frac{\partial^2 V}{\partial \mathbf{s}_2 \partial \mathbf{s}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 V}{\partial \mathbf{s}_n \partial \mathbf{s}_1} & \frac{\partial^2 V}{\partial \mathbf{s}_n \partial \mathbf{s}_2} & \cdots & \frac{\partial^2 V}{\partial \mathbf{s}_n \partial \mathbf{s}_n} \end{pmatrix}. \tag{A.19}$$

Therefore, the trace of the matrix above is given by:

$$\operatorname{Tr}\left[\mathbf{g}(\mathbf{s})^\top \mathbf{H} V(\mathbf{s}) \mathbf{g}(\mathbf{s})\right] = \sum_{i=1}^{m} \mathbf{g}_i(\mathbf{s})^\top \mathbf{H} V(\mathbf{s}) \mathbf{g}_i(\mathbf{s}), \tag{A.20}$$

and we can write the drift of $V(\mathbf{s})$ as:

$$\mathcal{D}V(\mathbf{s}) = \nabla V(\mathbf{s})^\top \mathbf{f}(\mathbf{s}) + \frac{1}{2}\sum_{i=1}^{m} \mathbf{g}_i(\mathbf{s})^\top \mathbf{H} V(\mathbf{s}) \mathbf{g}_i(\mathbf{s}). \tag{A.21}$$

Auxiliary functions. Next, define the auxiliary functions $F_i : \mathbb{R} \to \mathbb{R}$ as

$$F_i(\epsilon) = V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right). \tag{A.22}$$

The derivative of $F_i(\epsilon)$ is given by:

$$F_i'(\epsilon) = \nabla V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right)^\top \left(\frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}} + \frac{\mathbf{f}(\mathbf{s})}{m}\epsilon\right). \tag{A.23}$$

Evaluating at $\epsilon = 0$, we obtain:

$$F_i'(0) = \nabla V(\mathbf{s})^\top \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}. \tag{A.24}$$

This implies that the diffusion matrix of $V(\mathbf{s})$ is given by:

$$\nabla V(\mathbf{s})^\top \mathbf{g}(\mathbf{s}) = \sqrt{2}[F_1'(0), F_2'(0), \cdots, F_m'(0)]. \tag{A.25}$$

Drift. The second derivative of $F_i(\epsilon)$ is given by:

$$F_i''(\epsilon) = \left[\frac{d}{d\epsilon}\nabla V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right)\right]^\top \left(\frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}} + \frac{\mathbf{f}(\mathbf{s})}{m}\epsilon\right) + \nabla V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right)^\top \frac{\mathbf{f}(\mathbf{s})}{m}. \tag{A.26}$$

Notice that the term inside the brackets is given by:

$$\frac{d}{d\epsilon}\nabla V\left(\mathbf{s} + \frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}}\epsilon + \frac{\mathbf{f}(\mathbf{s})}{2m}\epsilon^2\right) = \frac{\partial \nabla V}{\partial \mathbf{s}^\top}\left(\frac{\mathbf{g}_i(\mathbf{s})}{\sqrt{2}} + \frac{\mathbf{f}(\mathbf{s})}{m}\epsilon\right). \tag{A.27}$$

Evaluating at $\epsilon = 0$, we obtain:

$$F_i''(0) = \frac{1}{2}\mathbf{g}_i(\mathbf{s})^\top \left(\frac{\partial \nabla V(\mathbf{s})}{\partial \mathbf{s}}\right)^\top \mathbf{g}_i(\mathbf{s}) + \nabla V(\mathbf{s})^\top \frac{\mathbf{f}(\mathbf{s})}{m}. \tag{A.28}$$

Defining the function $F(\epsilon) = \sum_{i=1}^m F_i(\epsilon)$, we obtain:

$$F''(0) = \sum_{i=1}^m F_i''(0) = \frac{1}{2}\sum_{i=1}^m \mathbf{g}_i(\mathbf{s})^\top \mathbf{H}V(\mathbf{s})\mathbf{g}_i(\mathbf{s}) + \nabla V(\mathbf{s})^\top \mathbf{f}(\mathbf{s}), \tag{A.29}$$

using the fact that $\mathbf{H}V(\mathbf{s}) = \frac{\partial^2 V(\mathbf{s})}{\partial \mathbf{s}\partial \mathbf{s}^\top} = \left[\frac{\partial \nabla V(\mathbf{s})}{\partial \mathbf{s}^\top}\right]^\top$. Hence, $F''(0) = \mathcal{D}V(\mathbf{s})$. This concludes the proof.

A.2.2 A hyper-dual implementation

We can implement the computation of the drift of $V(\mathbf{s})$ using a hyper-dual number. Let's start by defining hyper-dual number for a scalar-valued function as a triplet containing the value of the function, the drift vector, and the matrix of risk exposures:

$$s = s_0 + \mathbf{s}_\sigma \mathbf{i} + s_\mu \mathbf{j}, \tag{A.30}$$

where $s_0 \in \mathbb{R}$ corresponds to the primal value, $\mathbf{s}_\sigma \in \mathbb{R}^{1\times m}$ corresponds to the diffusion matrix, and $s_\mu \in \mathbb{R}$ corresponds to the drift.

For a pair of hyper-dual numbers, s and t, we can define the following operations:

1. Multiplication by scalar: $\alpha \cdot s \equiv (\alpha \cdot s_0) + (\alpha \cdot \mathbf{s}_\sigma)\,\mathbf{i} + (\alpha \cdot s_\mu)\,\mathbf{j}$.
2. Addition: $s + t \equiv (s_0 + t_0) + (\mathbf{s}_\sigma + \mathbf{t}_\sigma)\,\mathbf{i} + (s_\mu + t_\mu)\,\mathbf{j}$.

3. Multiplication: $s \cdot t \equiv (s_0 \cdot t_0) + (s_0 \cdot \mathbf{t}_\sigma + \mathbf{s}_\sigma \cdot t_0)\,\mathbf{i} + (s_0 \cdot t_\mu + s_\mu \cdot t_0 + \frac{1}{2}\mathbf{s}_\sigma^\top \mathbf{t}_\sigma)\,\mathbf{j}$.

4. For an arbitrary unary function $f : \mathbb{R} \to \mathbb{R}$, we have

$$f(s) \equiv f(s_0) + (\mathbf{s}_\sigma \cdot f'(s_0))\,\mathbf{i} + (s_\mu \cdot f'(s_0) + \frac{1}{2} f''(s_0)\mathbf{s}_\sigma^\top \mathbf{s}_\sigma)\,\mathbf{j}. \tag{A.31}$$

Given the rules of propagation for a scalar-valued hyper-dual number, it is straightforward to extend to the case of a vector-valued hyper-dual number: $\mathbf{s} = (s_1, \ldots, s_n)^\top$, where $s_i = s_{i0} + \mathbf{s}_{i\sigma}\mathbf{i} + s_{i\mu}\mathbf{j}$ is a scalar-valued hyper-dual number.

A.3 Derivations for the Lucas orchard model

Consider a Lucas orchard model with $N+1$ trees. Dividends for the i-th tree are given by:

$$\frac{dD_{i,t}}{D_{i,t}} = \mu_i dt + \sigma_i dB_{i,t}, \qquad i = 1, 1, \ldots, N, \tag{A.32}$$

where $B_{i,t}$ is a Brownian motion

As in the two-trees model, it is useful to work with the dividend share for the i-th tree, $s_i = D_{i,t}/C_t$, where aggregate consumption is given by $C_t = \sum_{i=1}^N D_{i,t}$.

The process for consumption is given by:

$$\frac{dC_t}{C_t} = \sum_{i=1}^N \frac{D_{i,t}}{C_t}\frac{dD_{i,t}}{D_{i,t}} = \mu_c(\mathbf{s}_t)dt + \sigma_c(\mathbf{s}_t)^\top d\mathbf{B}, \tag{A.33}$$

where $\mu_c(\mathbf{s}_t) = \sum_{i=1}^N s_i\mu_i$ and $\sigma_c(\mathbf{s}_t) = [s_1\sigma_1, s_2\sigma_2, \ldots, s_N\sigma_N]^\top$.

The dividend share process is given by:

$$\frac{ds_i}{s_i} = \frac{dD_{i,t}}{D_{i,t}} - \frac{dC_t}{C_t} - \frac{dD_{i,t}}{D_{i,t}}\frac{dC_t}{C_t} + \left(\frac{dC_t}{C_t}\right)^2 \Rightarrow ds_{i,t} = \mu_{s_i}(\mathbf{s}_t)dt + \sigma_{s_i}(\mathbf{s}_t)^\top d\mathbf{B}, \tag{A.34}$$

where

$$\mu_{s_i}(\mathbf{s}_t) = s_i\left[\mu_i - \mu_c(\mathbf{s}_t) - s_i\sigma_i^2 + \sigma_c(\mathbf{s}_t)^\top\sigma_c(\mathbf{s}_t)\right], \tag{A.35}$$

$$\sigma_{s_i}(\mathbf{s}_t) = s_i\left[\sigma_i e_i - \sigma_c(\mathbf{s}_t)\right], \tag{A.36}$$

and e_i is the i-th unit vector. Notice that $\mu_{s_i}(\mathbf{s}) = 0$ and $\sigma_{s_i}(\mathbf{s}) = \mathbf{0}_{N\times 1}$ for $s_i = 0$ and $s_i = 1$. We can then write the law of motion of the vector of dividend shares as:

$$d\mathbf{s}_t = \mu_s(\mathbf{s}_t)dt + \sigma_s(\mathbf{s}_t)d\mathbf{B}, \tag{A.37}$$

where $\mu_s(\mathbf{s}_t) = [\mu_{s_1}(\mathbf{s}_t), \mu_{s_2}(\mathbf{s}_t), \ldots, \mu_{s_N}(\mathbf{s}_t)]^\top$ and $\sigma_s(\mathbf{s}_t) = [\sigma_{s_1}(\mathbf{s}_t), \sigma_{s_2}(\mathbf{s}_t), \ldots, \sigma_{s_N}(\mathbf{s}_t)]^\top$. Notice that the $n \times n$ diffusion matrix is singular, as $\mathbf{1}_N^\top\sigma_s(\mathbf{s}_t) = \mathbf{0}_{1\times N}$.

The price-consumption ratio $v_{i,t} \equiv P_{i,t}/C_t$ for the i-th tree is given by:

$$v_{i,t} = \mathbb{E}_t\left[\int_0^\infty e^{-\rho s} s_{i,t+s} ds\right]. \tag{A.38}$$

The stationary HJB equation for $v_{i,t} = v_i(\mathbf{s}_t)$ is given by:

$$\rho v_i(\mathbf{s}) = s_i + \nabla v_i^\top \mu_s(\mathbf{s}) + \frac{1}{2}\,\mathrm{Tr}\left[\sigma_s(\mathbf{s})^\top \mathbf{H} v_i(\mathbf{s})\sigma_s(\mathbf{s})\right]. \tag{A.39}$$

with boundary conditions $v_{i,t}(0) = 0$ and $v_{i,t}(1) = 1/\rho$.

A.4 Derivations for the Hennessy and Whited (2007) model

In this section, we provide a chracterization of the optimal investment policy for our version of the Hennessy and Whited (2007) model.

Optimal investment policy. The firm's investment behavior depends on whether the firm is incurring the equity issuance costs or not. We have three possible cases. First, if the firm pays positive dividends, then the optimal investment policy is given by:

$$i_+(k,z) = \delta + \frac{v_k(\mathbf{s}) - 1}{\chi}, \tag{A.40}$$

Second, if firm decides to issue equity, so $D_t < 0$, then the optimal investment policy is given by:

$$i_-(k,z) = \delta + \frac{v_k(\mathbf{s})/(1+\lambda) - 1}{\chi}, \tag{A.41}$$

There is a region of the state space where the firm chooses not to issue equity and not to pay dividends. We refer to this region as the *inaction region*. When the firm is in the inaction region, dividends are zero, so the optimal investment policy is given by:

$$e^z k^{\alpha-1} = i + 0.5\chi(i-\delta)^2 \Rightarrow \chi i^2 + 2(1-\chi\delta)i - \left(2e^z k^{\alpha-1} - \chi\delta^2\right) = 0. \tag{A.42}$$

The quadratic equation will have two real solutions if the condition is satisfied:

$$(1-\chi\delta)^2 + 2\chi e^z k^{\alpha-1} - \chi^2\delta^2 \geq 0 \iff 1 + 2\chi e^z k^{\alpha-1} \geq 2\chi\delta \tag{A.43}$$

Dividends will be negative if $i > i_{\max}$ or $i < i_{\min}$, where

$$i_{\max}(k,z) = \frac{\sqrt{1+2\chi(e^z k^{\alpha-1} - \delta)} - (1-\chi\delta)}{\chi}, \quad i_{\min}(k,z) = \frac{-\sqrt{1+2\chi(e^z k^{\alpha-1} - \delta)} - (1-\chi\delta)}{\chi}. \tag{A.44}$$

Therefore, the optimal investment policy is given by:

$$i(k,z) = \begin{cases} i_+(k,z) & \text{if } |v_k(\mathbf{s})| < \sqrt{1+2\chi e^z k^{\alpha-1}}, \\ i_-(k,z) & \text{if } |v_k(\mathbf{s})| > (1+\lambda)\sqrt{1+2\chi e^z k^{\alpha-1}}, \\ i_0(k,z) & \text{if } \sqrt{1+2\chi e^z k^{\alpha-1}} \leq |v_k(\mathbf{s})| \leq (1+\lambda)\sqrt{1+2\chi e^z k^{\alpha-1}}. \end{cases} \tag{A.45}$$

where $i_0(k,z) = i_{\max}(k,z)$ if $v_k(x,z) > 0$ and $i_0(k,z) = i_{\min}(k,z)$ if $v_k(x,z) < 0$.

A.4.1 Finite-differences solution

For simplicity, consider the case where $\theta = \sigma_z = 0$, so z is constant. In this case, we can treat z as a parameter and the problem reduces to a single state variable. Fix a grid for k,

$\{k_1, \ldots, k_N\}$, and let $v_j^n \equiv v^n(k_j)$ denote the value function evaluated at k_j. We can write the discretized HJB equation as:

$$\frac{v_j^{n+1} - v_i^n}{\Delta t} + \rho v_i^n = e^z k_j^{\alpha-1} - (i_j^n + 0.5\chi(i_j^n - \delta)^2)k_j \tag{A.46}$$

$$+ \frac{v_{j+1}^n - v_j^n}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n > \delta} + \frac{v_j^n - v_{j-1}^n}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n < \delta}, \tag{A.47}$$

We can rearrange the expression above to obtain:

$$v_j^{n+1} = u_j^n + \ell_j^n v_{j-1}^n + s_j^n v_j^n + r_j^n v_{j+1}^n, \tag{A.48}$$

where

$$\ell_j^n \equiv \frac{\Delta t}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n > \delta} \tag{A.49}$$

$$r_j^b \equiv -\frac{\Delta t}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n < \delta} \tag{A.50}$$

$$s_j^n \equiv 1 - \rho\Delta t - \frac{\Delta t}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n > \delta} + \frac{\Delta t}{\Delta k}(i_j^n - \delta)k_j \mathbf{1}_{i_j^n < \delta} \tag{A.51}$$

The first-order condition for the optimal investment policy is given by:

$$i_j^n = \begin{cases} \delta + \frac{v_j^n - v_{j-1}^n}{2\Delta k} & \text{if } i_j^n > \delta \\ \delta + \frac{v_{j+1}^n - v_j^n}{2\Delta k} & \text{if } i_j^n < \delta \\ \delta + \frac{v_{j+1}^n - v_{j-1}^n}{2\Delta k} & \text{if } i_j^n = \delta \end{cases} \tag{A.52}$$

The boundary conditions are $v_1^n = 0$ and $v_{N+1}^n = 1/\rho$.
We can solve this system of ODEs using a finite-difference method.

Bibliography

Achdou, Yves, Jiequn Han, Jean-Michel Lasry, Pierre-Louis Lions, and Benjamin Moll, "Income and wealth distribution in macroeconomics: A continuous-time approach," *The review of economic studies*, 2022, *89* (1), 45–86.

Barles, Guy and Panagiotis E. Souganidis, "Convergence of Approximation Schemes for Fully Nonlinear Second Order Equations," *Asymptotic Analysis*, 1991, *4*, 271–283.

Black, Fischer and Myron Scholes, "The Pricing of Options and Corporate Liabilities," *Journal of Political Economy*, 1973, *81* (3), 637–654.

Bonnans, J. Frédéric, Élisabeth Ottenwaelter, and Housnaa Zidani, "A fast algorithm for the two dimensional HJB equation of stochastic control," *ESAIM: Modélisation mathématique et analyse numérique*, 2004, *38* (4), 723–735.

Boyd, John P., *Chebyshev and Fourier Spectral Methods*, Mineola, NY: Dover Publications, 2001.

Brenner, Susanne C. and L. Ridgway Scott, *The Mathematical Theory of Finite Element Methods*, 3 ed., Vol. 15 of *Texts in Applied Mathematics*, New York: Springer, 2008.

Brumm, Johannes and Simon Scheidegger, "Using Adaptive Sparse Grids to Solve High-Dimensional Dynamic Models," *Econometrica*, 2017, *85* (5), 1575–1612.

Carroll, Christopher D., "The Method of Endogenous Gridpoints for Solving Dynamic Stochastic Optimization Problems," *Economics Letters*, 2006, *91* (3), 312–320.

_ **and Miles S. Kimball**, "On the Concavity of the Consumption Function," *Econometrica*, 1996, *64* (4), 981–992.

Cochrane, John H., Francis A. Longstaff, and Pedro Santa-Clara, "Two Trees," *The Review of Financial Studies*, 2008, *21* (1), 347–385.

Cybenko, G, "Approximation by superposition of sigmoidal functions," *Mathematics of Control, Signals and Systems*, 1989, *2* (4), 303–314.

d'Avernas, Adrien, Damon Petersen, and Quentin Vandeweyer, "A Solution Method for Continuous-Time Models," 2024. Working paper. Available at https://faculty.chicagobooth.edu/quentin-vandeweyer/research.

Duchi, John, Elad Hazan, and Yoram Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, 2011, *12*, 2121–2159.

Fella, Giulio, "Endogenous grid method," 2025.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2 ed., New York, NY: Springer, 2009.

Hendrycks, Dan and Kevin Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

Hennessy, Christopher A and Toni M Whited, "How costly is external financing? Evidence from a structural estimation," *Journal of Finance*, 2007, *62* (4), 1705–1745.

Hornik, Kurt, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, 1991, *4* (2), 251 – 257.

Judd, Kenneth L., Lilia Maliar, Serguei Maliar, and Rafael Valero, "Smolyak Method for Solving Dynamic Economic Models: Lagrange Interpolation, Anisotropic Grid and Adaptive Domain," *Journal of Economic Dynamics and Control*, 2014, *44*, 92–123.

Kingma, Diederik P. and Jimmy Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds., *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., 2012, pp. 1097–1105.

Kushner, Harold J. and Paul G. Dupuis, *Numerical Methods for Stochastic Control Problems in Continuous Time* number 24. In 'Applications of Mathematics.', 2 ed., New York: Springer, 2001.

Leshno, Moshe, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural networks*, 1993, *6* (6), 861–867.

Loshchilov, Ilya and Frank Hutter, "Decoupled Weight Decay Regularization," in "International Conference on Learning Representations" 2019.

Lu, Zhou, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang, "The Expressive Power of Neural Networks: A View from the Width," in "Advances in Neural Information Processing Systems," Vol. 30 2017, pp. 6232–6240.

Ludwig, Alexander and Matthias Schoen, "Endogenous Grids in Higher Dimensions: Delaunay Interpolation and Hybrid Methods," *Computational Economics*, 2018, *51* (3), 607–636.

Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in "Proceedings of the 30th International Conference on Machine Learning (ICML 2013), Workshop on Deep Learning for Audio, Speech and Language Processing" 2013.

Magnus, Jan R. and Heinz Neudecker, *Matrix Differential Calculus with Applications in Statistics and Econometrics*, revised ed., Chichester, UK: John Wiley & Sons, 1999.

Martin, Ian, "The Lucas Orchard," *Econometrica*, 2013, *81* (1), 55–111.

McGrattan, Ellen R., "Solving the Stochastic Growth Model with a Finite Element Method," *Journal of Economic Dynamics and Control*, 1996, *20* (1-3), 19–42.

Merton, Robert C., "Theory of Rational Option Pricing," *The Bell Journal of Economics and Management Science*, 1973, *4* (1), 141–183.

Phelan, Thomas and Keyvan Eslami, "Applications of Markov Chain Approximation Methods to Optimal Control Problems in Economics," *Journal of Economic Dynamics and Control*, 2022, *143*, 104437.

Prince, Simon J. D., *Understanding Deep Learning*, Cambridge, UK: Cambridge University Press, 2023.

Ross, Stephen A, "Options and efficiency," *Quarterly Journal of Economics*, 1976, *90* (1), 75–89.

Rouwenhorst, K. Geert, "Asset Pricing Implications of Equilibrium Business Cycle Models," in Thomas F. Cooley, ed., *Frontiers of Business Cycle Research*, Princeton, NJ: Princeton University Press, 1995, chapter 10, pp. 294–330.

Schaab, Andreas and Allen Zhang, "Dynamic Programming in Continuous Time with Adaptive Sparse Grids," *SSRN Electronic Journal*, May 2022, pp. 1–57.

Smolyak, Sergei Abramovich, "Quadrature and interpolation formulas for tensor products of certain classes of functions," in "Doklady Akademii Nauk," Vol. 148 Russian Academy of Sciences 1963, pp. 1042–1045.

Sutton, Richard S. and Andrew G. Barto, *Reinforcement Learning: An Introduction*, 2 ed., Cambridge, MA: MIT Press, 2018.

Tauchen, George, "Finite State Markov-Chain Approximations to Univariate and Vector Autoregressions," *Economics Letters*, 1986, *20* (2), 177–181.

_ **and Robert Hussey**, "Quadrature-Based Methods for Obtaining Approximate Solutions to Nonlinear Asset Pricing Models," *Econometrica*, 1991, *59* (2), 371–396.

White, Matthew N., "The Method of Endogenous Gridpoints in Theory and Practice," *Journal of Economic Dynamics and Control*, 2015, *60*, 26–41.