Assignment: Projection Methods for Dynamic Programming

This assignment explores how to solve continuous-state dynamic programming problems numerically. When the state space is continuous (or simply very large), we cannot store the value function at every state. Instead, we must **approximate** it using a finite-dimensional representation. The core question is: how do we choose good approximations, and how do we solve for them? You will implement two discretization strategies (**collocation** and **Galerkin**) and discover that classic algorithms like fitted value iteration emerge naturally from this framework.

Problem Setting: Timber Harvesting

Imagine you own a forest plot. Each year, the biomass s (measured in tons of harvestable timber) grows according to a simple ecological model. You face a binary decision: cut the timber now and replant, or wait another year and let it grow. Cutting generates revenue proportional to biomass minus a replanting cost; waiting means the biomass evolves to next year's level. Your goal is to maximize the discounted sum of revenues over an infinite horizon.

The Model

State: $s \in [0, K]$, where s is current biomass (tons) and K is the carrying capacity (maximum sustainable biomass).

Actions: - **Cut** ($a = 1$): Harvest the timber, receive revenue $\text{price} \cdot s - C$, and reset to $s' = 0$ - **Wait** ($a = 0$): No revenue now, biomass grows to $s' = K + e^{-\alpha}(s - K)$

The growth function models logistic-like dynamics: biomass approaches K over time at rate determined by α.

Discount factor: $\gamma \in (0, 1)$ (we use $\gamma = 0.95$).

Parameters: - $K = 0.5$ (carrying capacity) - price = 1.0 (revenue per ton) - $C = 0.2$ (replanting cost) - $\alpha = 0.1$ (growth rate)

The Bellman Equation

The optimal value function $v^*(s)$ satisfies

$$v^*(s) = \max\left\{ \underbrace{\text{price} \cdot s - C + \gamma v^*(0)}_{\text{cut}}, \quad \underbrace{\gamma v^*(K + e^{-\alpha}(s - K))}_{\text{wait}} \right\}.$$

This is a **functional equation**: the unknown is a function $v : [0, K] \to \mathbb{R}$, not a finite-dimensional vector. In principle, we need to satisfy this equation at every point $s \in [0, K]$—infinitely many conditions!

Function Approximation

We cannot store $v(s)$ for all s. We must approximate:

$$v(s) \approx \hat{v}(s) = \sum_{i=1}^{n} a_i \varphi_i(s),$$

where $\{\varphi_1, \dots, \varphi_n\}$ are **basis functions** (e.g., piecewise linear "hat" functions—triangular functions that peak at grid points—or Chebyshev polynomials), and $a \in \mathbb{R}^n$ are coefficients we must find. See the Basis Functions section below for details.

It's useful to write the Bellman equation more abstractly as an **operator equation**. Define the **Bellman operator** L that acts on functions:

$$Lv(s) = \max_{a \in \{0,1\}} \{r(s,a) + \gamma v(s'(s,a))\},$$

where $r(s,a)$ is the immediate reward and $s'(s,a)$ is the next state. For our timber problem, $r(s,1) = \text{price} \cdot s - C$ (cut), $r(s,0) = 0$ (wait), $s'(s,1) = 0$, and $s'(s,0) = g(s) = K + e^{-\alpha}(s - K)$.

The Bellman equation is then simply $Lv = v$: we seek a fixed point of L. Banach's fixed-point theorem guarantees a unique solution if L is a γ-contraction.

Substituting our approximation $\hat{v}$ into this operator equation, we get a **residual function**:

$$R(s;a) = L\hat{v}(s) - \hat{v}(s).$$

For a true solution, $R(s;a) = 0$ for all s. For our approximation, we cannot enforce this everywhere—we have only n degrees of freedom. Instead, we require R to be "small" in a precise sense.

Projection Methods: Two Approaches

A **projection method** chooses a way to measure "smallness" of the residual. We pick **test functions** $\{p_1, \dots, p_n\}$ and require the residual to be orthogonal to each test function:

$$\langle R(\cdot;a), p_i \rangle = 0, \quad i = 1, \dots, n,$$

where $\langle f, g \rangle = \int f(s)g(s)\,ds$ is an inner product. This projection condition gives us n equations for the n unknown coefficients a. Different choices of test functions give different methods.

Method	Test functions	Meaning	Mathematical Form	Integration?
Collocation	$\delta(s - s_i)$ (Dirac deltas)	Residual zero at nodes s_i	$L\hat{v}(s_i) = \hat{v}(s_i)$	No: just evaluate at nodes
Galerkin	$\varphi_i(s)$ (same as basis)	Residual orthogonal to basis	$\int_0^K [L\hat{v}(s) - \hat{v}(s)]\, \varphi_i(s)\, ds = 0$	Yes: need quadrature

Collocation says: "The Bellman equation holds exactly at n chosen points $\{s_1, \dots, s_n\}$." Since $\langle R, \delta(\cdot - s_i)\rangle = R(s_i; a)$, this reduces to pointwise evaluation: $R(s_i; a) = 0$ for each node. Fast, no integrals, but only enforces the equation at discrete points.

Galerkin says: "The residual is orthogonal to the entire span of our basis." This gives the L^2-optimal approximation within the subspace $V_n = \text{span}\{\varphi_1, \dots, \varphi_n\}$. Requires numerical integration (typically Gauss-Legendre quadrature) to compute the integrals, but theoretically superior.

Basis Functions

We will be experimenting with different kinds of basis functions for this assignment (think of them as feature maps in machine learning if you want).

Linear splines (Hat Functions): Hat functions $\varphi_i(s)$ are called "hat" functions because they look like triangular hats when plotted. For nodes $s_1 < s_2 < \dots < s_n$, the i-th hat function is defined as:

$$\varphi_i(s) = \begin{cases} \frac{s - s_{i-1}}{s_i - s_{i-1}} & \text{if } s_{i-1} \leq s \leq s_i \\ \frac{s_{i+1} - s}{s_{i+1} - s_i} & \text{if } s_i \leq s \leq s_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Some of their interesting properties: - $\varphi_i(s_j) = 1$ if $i = j$, and $\varphi_i(s_j) = 0$ if $i \neq j$ (they form a **nodal basis**) - Each function "peaks" at one node and linearly drops to zero at neighboring nodes - Between any two nodes, the approximation $\hat{v}(s) = \sum_i a_i \varphi_i(s)$ is just a straight line - They sum to 1 everywhere: $\sum_{i=1}^n \varphi_i(s) = 1$ for all s (partition of unity) - The coefficients a_i are simply the function values at nodes: $\hat{v}(s_i) = a_i$

These functions create piecewise linear interpolation and preserve monotonicity and concavity (shape-preserving).

Chebyshev polynomials: A family of smooth basis functions $T_i(s)$ that are orthogonal over a rescaled interval (typically $[-1, 1]$). They are often used to approximate smooth functions with high accuracy. However, because their values swing rapidly between nodes near the edges of the interval, they can produce oscillations in the approximation.

You will compare both.

Solving the Discretized System

Once we apply collocation or Galerkin, we have n equations for n unknowns a. But these equations are **nonlinear** (because of the max in the Bellman operator). How do we solve them?

Two Different Strategies

Strategy 1: Newton's Method (Direct Root-Finding) View the projection conditions as a nonlinear system $F(a) = 0$. Use Newton's method (via `scipy.optimize.fsolve`):

$$a^{(k+1)} = a^{(k)} - [J_F(a^{(k)})]^{-1} F(a^{(k)}),$$

where J_F is the Jacobian. This is a standard numerical approach: solve the system directly. Works for both collocation and Galerkin. Does **not** exploit any special structure of the Bellman equation.

Strategy 2: Fixed-Point Iteration (Parametric Value Iteration) View the Bellman equation as a fixed-point problem. Instead of solving $F(a) = 0$ directly, exploit the contraction property of the Bellman operator. For collocation, the algorithm is:

1. Start with coefficients $a^{(k)}$ (which define $\hat{v}^{(k)} = \sum_j a_j^{(k)} \varphi_j$)
2. Apply the Bellman operator at each collocation node: $v_i^{(k+1)} = L\hat{v}^{(k)}(s_i)$ for $i = 1, \dots, n$
3. Fit new coefficients $a^{(k+1)}$ so that $\hat{v}^{(k+1)}(s_i) = v_i^{(k+1)}$
4. Repeat until $\|a^{(k+1)} - a^{(k)}\| < \epsilon$

Mathematically, let $\Phi \in \mathbb{R}^{n \times n}$ be the basis matrix with $\Phi_{ij} = \varphi_j(s_i)$. Step 3 solves the linear system:

$$\Phi a^{(k+1)} = v^{(k+1)},$$

which defines a map $\hat{T} : \mathbb{R}^n \to \mathbb{R}^n$ via $a^{(k+1)} = \hat{T}(a^{(k)})$. This is a discrete version of the Bellman operator acting in coefficient space rather than function space.

This is **exactly** what the RL community calls **fitted value iteration** or **parametric value iteration**! In reinforcement learning, you start with an approximate value function $\hat{v}^{(k)}$, apply the Bellman operator to get target values at sampled states, then fit a new approximation $\hat{v}^{(k+1)}$ (e.g., via neural network regression or least squares). Collocation is the special case where the "fit" step is interpolation through the collocation nodes.

Convergence: If the discretized operator $\hat{T}$ is a contraction, iterating it converges to a unique fixed point. Whether $\hat{T}$ is a contraction depends on both the discount factor γ and the basis functions: - **Linear interpolation** is non-expansive (an "averager") and typically preserves the contraction property, so $\hat{T}$ inherits the γ-contraction from L. Parametric value iteration converges reliably. - **Polynomial interpolation** can amplify differences (Runge phenomenon), potentially giving $\|\hat{T}a_1 - \hat{T}a_2\| > \gamma\|a_1 - a_2\|$. Parametric value iteration may diverge.

Newton's method does not require contraction but needs a good initial guess. For collocation with linear splines, parametric value iteration usually works well. For Galerkin or polynomial bases, Newton's method is safer.

What You Will Implement

The assignment template has **six strategic blanks** across three files. Each blank forces you to engage with a key concept. Below is what you need to implement and why each part matters.

Part 1: Projection Framework (`projection_framework.py`)

Task 1.1: Implement `CollocationTest.apply()` **What to do:** Given a residual function and collocation nodes, evaluate the residual at each node.

Collocation uses Dirac delta test functions $p_i = \delta(x - x_i)$. The inner product $\langle R, \delta(x-x_i)\rangle$ simply equals $R(x_i)$: pointwise evaluation. No integration needed!

Hint: Call `residual_func(self.nodes, coeffs)`.

Task 1.2: Implement `GalerkinTest.apply()` **What to do:** Compute $\langle R, \varphi_i\rangle = \int_a^b R(x)\varphi_i(x)dx$ for each basis function using **Gauss-Legendre quadrature**.

Galerkin requires numerical integration. Gaussian quadrature approximates integrals as weighted sums:

$$\int_a^b f(x)dx \approx \sum_{j=1}^{m} w_j f(x_j)$$

where $\{x_j\}$ are optimally chosen points (roots of Legendre polynomials) and $\{w_j\}$ are weights. This is exact for polynomials up to degree $2m - 1$.

Steps: 1. Get Gauss-Legendre nodes and weights on $[-1, 1]$: `np.polynomial.legendre.leggauss(n_quad)` 2. Transform to your domain $[a, b]$: - $x_{\text{quad}} = 0.5(b - a)x_{\text{std}} + 0.5(a + b)$ - $w_{\text{quad}} = w_{\text{std}} \cdot 0.5(b - a)$ 3. Evaluate residual at quadrature points 4. For each basis function i: compute $\sum_j w_j \cdot R(x_j) \cdot \varphi_i(x_j)$

Part 2: Timber Application (`timber_projection.py`)

Task 2.1: Implement `BellmanEquation.residual()` What to do: Compute the residual $R(s; a) = L\hat{v}(s) - \hat{v}(s)$.

The residual measures how far your approximation is from satisfying the Bellman equation. For a true solution, $R \equiv 0$ everywhere.

Steps: 1. Create a callable value function: `v_func(s) = basis.evaluate_approximation(s, coeffs)` 2. Apply the Bellman operator: `Lv = self.bellman_operator(s, v_func)` 3. Evaluate the approximation: `v_hat = v_func(s)` 4. Return the residual: `Lv - v_hat`

Task 2.2: Implement `BellmanProjectionMethod._solve_iterate()` What to do: Implement parametric/fitted value iteration: value iteration in coefficient space.

Instead of iterating $v^{(k+1)}(s) = Lv^{(k)}(s)$ for all s (impossible with continuous state), we: 1. Represent v using coefficients a 2. Apply L at collocation nodes to get target values 3. Fit new coefficients to match those targets 4. Repeat

Algorithm:

```
Initialize: a^(0) = 0
For k = 0, 1, 2, ... until convergence:
    1. Create v_func from a^(k) via basis.evaluate_approximation
    2. Apply Bellman at nodes: v_new[i] = L(v_func)(nodes[i])
    3. Fit new coefficients: solve Phi @ a^(k+1) = v_new
       where Phi[i,j] = phi_j(nodes[i])
    4. Check: if ||a^(k+1) - a^(k)|| < tol, stop
```

Part 3: Contraction Investigation (`contraction_investigation.py`)

Task 3.1: Implement `apply_discretized_bellman()` What to do: Apply ONE step of the discretized Bellman operator: $a \mapsto \widehat{T}(a)$.

This is the composition: (1) convert coefficients to function, (2) apply Bellman operator, (3) refit coefficients. It's the same as one iteration of parametric value iteration!

Steps: 1. Convert coefficients to function: `v_func = self.coeffs_to_function(coeffs)` 2. Apply Bellman at nodes: `v_new = self.bellman.bellman_operator(self.nodes, v_func)` 3. Fit new coefficients by solving $\Phi \cdot a' = v_{\text{new}}$

Task 3.2: Implement `estimate_lipschitz_constant()` What to do: Empirically estimate the Lipschitz constant $L = \sup_{v_1 \neq v_2} \frac{\|\hat{T}(v_1) - \hat{T}(v_2)\|}{\|v_1 - v_2\|}$.

The true Bellman operator is a γ-contraction under the sup-norm. But is the discretized operator also a contraction? We can test this by sampling random pairs and measuring the worst-case amplification.

Algorithm:

```
For i = 1 to n_samples:
    1. Sample random coefficients a1, a2
    2. Compute input distance: d_in = ||v1 - v2||
    3. Apply operator: Ta1 = T_hat(a1), Ta2 = T_hat(a2)
    4. Compute output distance: d_out = ||T(v1) - T(v2)||
    5. Compute ratio: d_out / d_in
    6. Track maximum ratio

Return: max(ratios) as Lipschitz estimate
```

Test: If $L \leq \gamma$, contraction is preserved! If $L > \gamma$, it's destroyed.

You'll empirically verify Gordon's theory (1995) that linear interpolation preserves contraction while polynomials may not. This explains why practical DP codes use simple linear interpolation!

References

- Judd, K. L. (1998). *Numerical Methods in Economics*, MIT Press. Chapter 11 (projection methods), Chapter 12 (fitted value iteration).
- Miranda, M. J., & Fackler, P. L. (2002). *Applied Computational Economics and Finance*, MIT Press. Chapter 6.
- Gordon, G. J. (1995). "Stable Function Approximation in Dynamic Programming," *ICML*.