ECO 612: Numerical Methods
Basics, Functional Approximation, Bellman Equations

January 31, 2026

Outline

- Basics
  - Linear systems, root finding, optimization
  - Numerical derivatives and quadrature
- Functional approximation
  - Function spaces, nodes, basis matrices, coefficient updates
  - Splines and Markov transitions
  - Multiple dimensions and tensor products
- Bellman equation
  - Coefficient VFI $\rightarrow$ Howard improvement $\rightarrow$ precomputed continuation

Basics

CompEcon

- You provide a function handle for a residual map, objective, or operator
- CompEcon provides stable implementations and sensible defaults for the numerics
- We'll talk about this in the context of dynamic programming
  - Re-parameterizations
  - Pre-computation
  - Howard Improvement

Linear Systems: $Ax = b$

- Many nonlinear routines repeatedly solve linear systems internally
- Stable default in MATLAB: solve by backslash (pivoted factorization)
- Conditioning: if columns are nearly collinear, you are effectively inverting noise

What you should check

- $\mathrm{cond}(A)$ or $\mathrm{condest}(A)$
- if A comes from approximation (Φ), change basis/nodes before changing solvers

Root Finding: Bisection

Solve $f(x) = 0$ when f is continuous and you have a bracket $[a, b]$ with a sign change.

- Requires: sgn $f(a) \neq$ sgn $f(b)$
- Update: midpoint $c = (a + b)/2$, keep the half interval that preserves the sign change
- Converges linearly with factor $1/2$: reliable, not fast
- **Parallelizes:** if you have many independent scalar equations $f_i(x_i) = 0$, you can solve them simultaneously (vectorized calls/independent brackets), since each bisection path depends only on its own $[a_i, b_i]$

CompEcon call

- **Call:** `x = bisect(f, a, b)`

Root Finding (Systems): Newton

Solve $f(x) = 0$ with $f : \mathbb{R}^n \to \mathbb{R}^n$ using local linearization.

$$x^{new} = x - J(x)^{-1} f(x), \qquad J(x) = \frac{\partial f(x)}{\partial x}.$$

- Extremely fast when you are close and J is accurate (quadratic local convergence)
- Can diverge if started poorly; sensitive to scaling and bad derivatives
- In practice: use as a "finishing method" after you are in the basin

CompEcon call

- **Call:** `x = newton(f, x0)`

Root Finding (Systems): Broyden

Broyden replaces repeated Jacobians with an updating approximation.

$$x^{new} = x - A^{-1} f(x), \qquad A \approx J(x).$$

- Avoids Jacobian evaluation; learns curvature gradually
- Often the best default when: medium dimension, Jacobians are costly/annoying
- Converges superlinearly under standard regularity conditions

CompEcon call

- **Call:** `x = broyden(f, x0)`

Optimization: Golden Section (1D)

Maximize a *unimodal* $f(x)$ on a bracket $[a, b]$: robust, derivative-free line search.

- Requires: a bracket $[a, b]$ containing the maximizer (typically unimodality on $[a, b]$)
- Update: evaluate at two interior points spaced by the golden ratio $\varphi = (1 + \sqrt{5})/2$
- Discard the subinterval that cannot contain the maximizer
- Converges linearly: interval length shrinks by $1/\varphi \approx 0.618$ each step (reliable, not fast)
- **Parallelizes:** if you need $\arg\max_x f_i(x)$ for many independent univariate objectives f_i, you can run golden-section searches simultaneously (vectorized evaluations/separate brackets), since each search uses only its own function values

CompEcon call

- **Call:** `xstar = golden(f, a, b)`

Optimization: Nelder–Mead (Simplex)

Derivative-free optimization in $\mathbb{R}^n$ using a moving simplex (geometric search).

- Maintains $n + 1$ vertices (a simplex); ranks them by objective value each iteration
- Update uses simplex moves: reflection $\rightarrow$ expansion/contraction, and shrink
- Practical for small/moderate n and rough/noisy objectives, but can stall
- Sensitive to scaling; weak global convergence guarantees

CompEcon call

- **Call:** `xstar = neldmead(f, x0)`

Optimization: Quasi-Newton (BFGS-type)

Fast local optimization for smooth problems by updating a curvature approximation.

$$x^{new} = x + \alpha\, p, \qquad p = -H\, \nabla f(x), \qquad H \approx \left[\nabla^2 f(x)\right]^{-1}.$$

- Requires (good) gradients; uses curvature updates rather than exact Hessians
- Typically very fast on smooth problems (often superlinear) with a line search
- Good default when f is well-behaved; performance degrades with noisy gradients

CompEcon call

- **Call:** `xstar = qnewton(f, x0)`

Numerical Differentiation: What You Call and What You Get

Central differences (scalar case):

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \qquad \text{error } O(h^2).$$

- Jacobian routines are for $f : \mathbb{R}^n \to \mathbb{R}^m$
- Hessian routines are for scalar objectives $g : \mathbb{R}^n \to \mathbb{R}$
- Step size is not innocent: too big gives truncation error, too small gives precision error

CompEcon calls

- **Call:** `J = fdjac(f, x)`
- **Call:** `H = fdhess(g, x)`

Quadrature: Nodes and Weights

Gaussian quadrature approximates an integral by a weighted sum:

$$\int f(x)\, w(x)\, dx \;\approx\; \sum_{i=1}^{n} \omega_i f(x_i).$$

- CompEcon returns $\{x_i, \omega_i\}$ so you can write expectations as dot products
- Multi-dimensional quadrature is typically a tensor product of univariate rules
- Costs scale as n^d: usable in low dimension, explosive otherwise

CompEcon calls (just examples, there are more)

- **Call:** `[x,w] = qnwnorm(n, mu, Sigma)`
- **Call:** `[x,w] = qnwcheb(n, a, b)`

Quadrature: Low-Order Alternatives

Sometimes you want simple composite rules instead of Gaussian nodes.

- Trapezoid and Simpson are straightforward and easy to debug
- They are not competitive for high accuracy in smooth problems
- They are useful when you want transparency more than speed

CompEcon calls

- **Call:** `[x,w] = qnwtrap(n, a, b)`
- **Call:** `[x,w] = qnwsimp(n, a, b)`

Functional approximation

Function Approximation: The Objects

We approximate a scalar function $f : [a, b] \to \mathbb{R}$ by a basis expansion

$$\hat{f}(x) = \sum_{q=1}^{Q} c_q \, \phi_q(x) \;=\; \phi(x)'c, \qquad \phi(x) \equiv \begin{bmatrix} \phi_1(x) & \cdots & \phi_Q(x) \end{bmatrix}', \quad c \in \mathbb{R}^Q.$$

- Choose nodes $x_1, \ldots, x_N$ (typically determined by the approximation space)
- Evaluate the target function at the nodes: $F_n = f(x_n)$
- Choose c so $\hat{f}$ matches f on the nodes (interpolation) or fits well (least squares)

The Basis Matrix Φ and the Linear System

Define the node-value vector and coefficient vector

$$F \equiv \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_N) \end{bmatrix} \in \mathbb{R}^N, \qquad c \equiv \begin{bmatrix} c_1 \\ \vdots \\ c_Q \end{bmatrix} \in \mathbb{R}^Q.$$

Define the basis (design) matrix $\Phi \in \mathbb{R}^{N \times Q}$ by $\Phi_{nq} = \phi_q(x_n)$, i.e.

$$\Phi \equiv \begin{bmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_Q(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_Q(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \cdots & \phi_Q(x_N) \end{bmatrix}.$$

- Interpolation is the linear algebra problem $\Phi\, c \approx F$
- In MATLAB we compute: $c = \Phi \backslash F$

What $\Phi \backslash F$ Means Here

- If $N = Q$ and Φ is nonsingular:

$$\Phi c = F \quad \Rightarrow \quad c = \Phi^{-1} F \quad \text{(exact interpolation)}.$$

- If $N > Q$:

$$c = \arg\min_{c \in \mathbb{R}^Q} \|\Phi\, c - F\|_2^2 \quad \text{(least squares fit)}.$$

- The normal equations are

$$(\Phi' \Phi)\, c = \Phi^\top F,$$

  but in practice you just use $\Phi \backslash F$ where we have SVD under the hood.

CompEcon Approximation: The Standard Pipeline

The mechanical pipeline is always the same: define a space → get nodes → build Φ → solve c.

- Choose a function space (Chebyshev, splines, mixed)
- Get nodes consistent with that space
- Build Φ, compute F, solve $c = \Phi \backslash F$
- Evaluate elsewhere using $\hat{f}(x) = \phi(x)'c$ (or `funeval`)

CompEcon calls

- **Call:** `fspace = fundef('cheb', n, a, b)`
- **Call:** `x = funnode(fspace)`
- **Call:** `Phi = funbas(fspace, x)`
- **Call:** `c = Phi\F`

Evaluate Elsewhere: $\hat{f}(x^\star) = \phi(x^\star)'c$

Once you have c, evaluation at any point $x^\star \in [a, b]$ is just

$$\hat{f}(x^\star) = \begin{bmatrix} \phi_1(x^\star) & \cdots & \phi_Q(x^\star) \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_Q \end{bmatrix} = \phi(x^\star)'c.$$

- Compute $\phi(x^\star)$ (a $Q \times 1$ vector) and take the dot product with c
- In batch form, for many points $x_1^\star, \ldots, x_M^\star$, stack them:

$$\hat{F}^\star = \Phi^\star c, \qquad \Phi^\star_{mq} = \phi_q(x^\star_m).$$

CompEcon evaluation call

- **Call:** `fhat = funeval(c, fspace, xstar);` (vectorized in `xstar`)

Choice of Basis: Why Monomials Are a Bad Default

If $\phi_q(x) = x^{q-1}$ (monomials), then Φ is the Vandermonde matrix

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{Q-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{Q-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^{Q-1} \end{bmatrix}.$$

- On any nontrivial interval $[a, b]$, powers become badly scaled as Q grows
- Columns become nearly collinear $\Rightarrow$ $\mathrm{cond}(\Phi)$ can explode
- Then $c = \Phi \backslash F$ becomes dominated by roundoff (even if f is smooth)

Chebyshev Polynomials: Stable Coordinates for Smooth Functions

Chebyshev polynomials $\{T_j\}$ are defined on $[-1, 1]$ by

$$T_0(x) = 1, \qquad T_1(x) = x, \qquad T_j(x) = 2x\,T_{j-1}(x) - T_{j-2}(x) \;\; (j \geq 2).$$

- Orthogonal under $w(x) = (1 - x^2)^{-1/2}$, which kills the "nearly collinear columns" problem
- Three-term recursion is numerically stable (no huge powers)
- For smooth f, Chebyshev approximation often gives very fast error decay

Why you should care

Switching from monomials to Chebyshev often turns an unstable Φ into a well-behaved Φ.

Chebyshev vs. Splines: When Each Is the Default

- Chebyshev: smooth f, want high accuracy with moderate order
- Splines: kinks, local curvature, constraints, or when you want local control
- In economics, splines are a workhorse for value functions with borrowing constraints

CompEcon space definitions

- **Call:** `fspace = fundef('cheb', n, a, b)`
- **Call:** `fspace = fundef({'spli', v, 0, k})` (breakpoints v, order k)

CompEcon Implementation: What You Should Actually Check

Once you form Φ and solve $c = \Phi \backslash F$, the diagnostics are mechanical:

- **Conditioning:** compute $\text{cond}(\Phi)$. If it's enormous, it's usually basis/nodes.
- **Out-of-sample check:** evaluate on a dense grid and look for boundary artifacts.
- **Order sensitivity:** increase Q moderately and see if the approximation stabilizes.

CompEcon calls you use for checks

- **Call:** `condPhi = cond(Phi);`
- **Call:** `xDense = linspace(a,b,2000)'; fHatDense = funeval(c,fspace,xDense);`

Bellman Equations

Standard Bellman for IFP

State (a, z). Income is AR(1) in logs:

$$z' = \rho\, z + \varepsilon' \quad \text{with} \quad \varepsilon' \sim \mathcal{N}(0, \sigma_\varepsilon^2).$$

Budget (prices r, w taken as given):

$$c + a' = (1 + r)\, a + w \exp(z) \quad \text{with} \quad c \geq 0 \;\text{ and }\; a' \geq \underline{a}.$$

Bellman

$$V(a, z) = \max_{c, a'} \Big\{ u(c) + \beta\, \mathbb{E}\big[V(a', z') \mid z\big] : \text{ subject to constraints } \Big\}$$

Cash-on-Hand and a Scalar Control

Define cash-on-hand:

$$m(a, z) \equiv (1 + r)\, a + w\, \exp(z).$$

Parameterize feasible choices by a share $x \in [0, 1]$:

$$a' = \underline{a} + x\,(m - \underline{a}), \qquad c = (1 - x)\,(m - \underline{a}).$$

Why this is a helpful parameterization for code

- feasibility is automatic (constraints enforced by construction)
- the control is one scalar $x \in [0, 1]$ at every state node
- scalar maximization routines (golden) don't require search for bracket

Remove Curvature: Certainty-Equivalent and a Monotone Transform

Work with a certainty-equivalent recursion. Define $v(a, z)$ by

$$v(a, z) = \left((1-\beta)^{\,1-\sigma} + \beta\, \mathbb{E}\Big[v(a', z')^{1-\sigma} \mid z \Big] \right)^{\frac{1}{1-\sigma}}.$$

Define $w(a, z) \equiv v(a, z)^{1-\sigma}$

$$w(a, z) = \max_{x \in [0,1]} \left\{ (1-\beta) \Big((1-x) \left(m(a, z) - \underline{a} \right) \Big)^{1-\sigma} + \beta\, \mathbb{E} \left[w \Big(\underline{a} + x \left(m(a, z) - \underline{a} \right), \rho\, z + \epsilon' \Big) \right] \right\}.$$

Equivalence and stability

- $v \mapsto w = v^{1-\sigma}$ is strictly monotone $\Rightarrow$ same optimal policy
- numerically avoids pathological scaling from very negative V when c is small

Nurmerical Integration in Bellman Equations

For convenience, let's set $\underline{a} = 0$. The, starting from

$$w(a,z) = \max_{x\in[0,1]} \left\{ (1-\beta)\Big((1-x)\,m(a,z)\Big)^{1-\sigma} + \beta\,\mathbb{E}\Big[w\Big(x\,m(a,z), \rho\,z + \epsilon'\Big)\Big] \right\}$$

we can replace the expectation by some quadrature with weight and nodes $\{w_\ell, \epsilon_\ell\}$

$$w(a,z) = \max_{x\in[0,1]} \left\{ (1-\beta)\Big((1-x)\,m(a,z)\Big)^{1-\sigma} + \beta \sum_{\ell=1}^{L} w_\ell\, w\Big(x\,m(a,z), \rho\,z + \epsilon_\ell\Big) \right\}.$$

Coefficient VFI: Approximation Setup

Approximate w by a basis expansion with coefficients $c \in \mathbb{R}^Q$:

$$w(s) \approx \hat{w}(a, z; c) = \phi(a, z)'c, \qquad \phi(s) = \begin{bmatrix} \phi_1(a, z) & \cdots & \phi_Q(a, z) \end{bmatrix}'.$$

Stack the N state nodes (a_n, z_n) and define the design matrix

$$\Phi \in \mathbb{R}^{N \times Q}, \qquad \Phi_{nq} = \phi_q(a_n, z_n).$$

Then values on the nodes are the vector

$$\hat{w}(c) \equiv \begin{bmatrix} \hat{w}(a_1, z_1; c) \\ \vdots \\ \hat{w}(a_N, z_N; c) \end{bmatrix} = \Phi c \ \in \mathbb{R}^N.$$

- Objects: $c \in \mathbb{R}^Q$, $\Phi \in \mathbb{R}^{N \times Q}$, node-value vector $w \in \mathbb{R}^N$
- "Evaluate on nodes" is one matrix–vector multiply: $w = \Phi c$

Approximating the Value Function

With this notation, we can now write

$$w(a,z) = \max_{x\in[0,1]} \left\{ (1-\beta)\Big((1-x)\,m(a,z)\Big)^{1-\sigma} + \beta \sum_{\ell=1}^{L} w_\ell \, w\Big(x\,m(a,z), \rho\, z + \epsilon_\ell\Big) \right\}.$$

as

$$\phi(a,z)'c = \max_{x\in[0,1]} \left\{ (1-\beta)\Big((1-x)\,m(a,z)\Big)^{1-\sigma} + \beta \sum_{\ell=1}^{L} w_\ell \, \phi\Big(x\,m(a,z), \rho\, z + \epsilon_\ell\Big)' c \right\}.$$

Instead of finding a functional fixed point w, we are now looking for a fixed point $c \in \mathbb{R}^Q$

Coefficient VFI: One Outer Iteration

With the approximation $w(a,z) \approx \hat{w}(a,z;c) = \phi(a,z)'c$, the *node-by-node update* is:

- For each state node (a_n, z_n) and a current guess $c^{(k)}$, define the updated value

$$w_n^{(k+1)} \;=\; \max_{x\in[0,1]} \left\{ (1-\beta)\Big((1-x)\,m(a_n,z_n)\Big)^{1-\sigma} + \beta \sum_{\ell=1}^{L} w_\ell\, \phi\Big(x\,m(a_n,z_n),\, \rho\, z_n + \epsilon_\ell\Big)' c^{(k)} \right\}.$$

- Input: current coefficients $c^{(k)}$
- Output: updated node values $w^{(k+1)} \in \mathbb{R}^N$ with entries $w_n^{(k+1)}$
- Numerically: solve the 1D maximization in $x \in [0,1]$ (e.g. golden section) at each node
- **Parallel across nodes:** each (a_n, z_n) maximization is independent given $c^{(k)}$

Coefficient VFI: Coefficient Update

Stack the updated node values and refit the coefficients so that $\hat{w}(\cdot;\, c)$ matches on the nodes.

- Let

$$w^{(k+1)} \equiv \begin{bmatrix} w_1^{(k+1)} \\ \vdots \\ w_N^{(k+1)} \end{bmatrix} \in \mathbb{R}^N, \qquad \Phi_{nq} = \phi_q(a_n, z_n), \qquad \hat{w}(c) = \Phi c.$$

- Then the coefficient update is the least-squares projection:

$$c^{(k+1)} = \arg\min_{c \in \mathbb{R}^Q} \|\Phi c - w^{(k+1)}\|_2^2.$$

- Normal equations: $(\Phi'\Phi)\, c^{(k+1)} = \Phi' w^{(k+1)}$.
- Solve in practice: $c^{(k+1)} = \Phi \backslash w^{(k+1)}$
- This closes the outer loop: $c^{(k)} \to w^{(k+1)} \to c^{(k+1)}$

Howard Improvement: The Idea

Baseline coefficient VFI repeats an expensive maximization at every node, every iteration.

- **Policy improvement (expensive):** for each node (a_n, z_n), solve 1D maximization
- **Policy evaluation (cheap):** hold the policy fixed and update the implied value function
- **Howard improvement:** do improvement only occasionally in between evaluations

Takeaway

- fewer golden searches $\Rightarrow$ large speedups in practice

Fixed Policy: Bellman Equation (No Maximization)

Fix a policy on nodes: $x_n \in [0,1]$ for each state node (a_n, z_n).

- For each node n

$$\phi(a_n, z_n)'c = (1-\beta)\Big((1-x_n)\, m(a_n, z_n)\Big)^{1-\sigma} + \beta \sum_{\ell=1}^{L} w_\ell \, \phi\Big(x_n \, m(a_n, z_n), \rho \, z_n + \epsilon_\ell\Big)' c.$$

  so that equivalently

$$\left[\phi(a_n, z_n)' - \beta \sum_{\ell=1}^{L} w_\ell \, \phi\Big(x_n \, m(a_n, z_n), \rho \, z_n + \epsilon_\ell\Big)'\right] c = (1-\beta)\Big((1-x_n)\, m(a_n, z_n)\Big)^{1-\sigma}$$

- With $\{x_n\}$ fixed, the update is **linear in** c

Howard Step: One Linear System in c

- Stack node equations and use $\Phi_{nq} = \phi_q(a_n, z_n)$ so $\hat{w}(c) = \Phi\, c$.
- Define

$$u_x \equiv \begin{bmatrix} (1-\beta)\Big((1-x_1)m(a_1,z_1)\Big)^{1-\sigma} \\ \vdots \\ (1-\beta)\Big((1-x_N)m(a_N,z_N)\Big)^{1-\sigma} \end{bmatrix}, \qquad (B_x)_{nq} \equiv \sum_{\ell=1}^{L} w_\ell\, \phi_q\Big(x_n\, m(a_n,z_n), \rho\, z_n + \epsilon_\ell\Big).$$

- Then policy evaluation solves

$$(\Phi - \beta B_x)\, c = u_x.$$

- No maximization, just linear algebra given $\{x_n\}$

Why Howard Helps

- **Contraction:** The fixed-policy mapping is still a β-contraction $\Rightarrow$ evaluation converges
- **Cost:** mMst iterations solve $(\Phi - \beta B_x)c = u_x$; only occasionally do optimization

- **Typical loop:**
  - Improvement: Update $\{x_n\}$ using current c
  - Evaluation: Just solve solve $(\Phi - \beta B_x)c = u_x$ for the next 10, 25, or perhaps 50 iterations
  - There is some problem-specific tuning; go by trial and error

Where Things are Still Expensive

when optimizing: maximizer typically needs many objective evaluations for each candiate x of at each node

$$(1-\beta)\,c^{1-\sigma} + \beta\,\mathbb{E}\big[\hat{w}(a', z') \mid z\big].$$

- even when just evaluating
- each objective evaluation needs an expectation over z'
- each expectation needs off-grid evaluation of $\hat{w}$ at (a', z'_ℓ)

The basic insight here

- Most of the work is repeated continuation evaluation.
- Precomputation is how you stop paying that price repeatedly.

Where Things are Still Expensive

Even with Howard, the bottleneck is the *inner-loop objective evaluation* during improvement. For a given node n and coefficients $c^{(k)}$, each candidate $x \in [0,1]$ requires many evaluations of

$$(1-\beta)\Big((1-x)\,m(a_n,z_n)\Big)^{1-\sigma} + \beta \sum_{\ell=1}^{L} w_\ell\, \phi\Big(x\,m(a_n,z_n), \rho\, z_n + \epsilon_\ell\Big)' c^{(k)}.$$

- Golden search calls this (as a function of x) many times per node n
- Each call recomputes the same conditional expectation structure: $\sum_{\ell=1}^{L} w_\ell\, \hat{w}(a', z'_\ell; c^{(k)})$
- Most work is repeated continuation evaluation at off-grid (a', z')

Key idea

Precompute the continuation term *as a function of* a' for each z.

Continuation Value: Pull Out the Expectation

Define the (quadrature) continuation value induced by c:

$$\Gamma(a', z;\, c) \;\equiv\; \sum_{\ell=1}^{L} w_\ell \, \phi\Big(a', \rho\, z + \epsilon_\ell\Big)' c.$$

Then the node objective becomes

$$(1-\beta)\Big((1-x)\, m(a_n, z_n)\Big)^{1-\sigma} + \beta\, \Gamma\Big(x\, m(a_n, z_n), z_n;\, c\Big).$$

- Conditioning on z_n, the entire expectation is a scalar function of a'
- In the maximization, the only way x enters is through $a' = x\, m(a_n, z_n)$
- So we can precompute $a' \mapsto \Gamma(a', z_n; c^{(k)})$ once, then reuse it for all trial x

Precompute $a' \mapsto \Gamma(a', z_n; c^{(k)})$ on an Asset Grid

- In code we typically use a tensor grid $\{a_i\}_{i=1}^{N_a} \times \{z_j\}_{j=1}^{N_z}$ (so $N = N_a N_z$).
- For each income node z_j, precompute the continuation values on the asset grid:

$$\Gamma_{ij}^{(k)} \;\equiv\; \Gamma(a_i, z_j;\, c^{(k)}) = \sum_{\ell=1}^{L} w_\ell \, \phi\Big(a_i, \rho\, z_j + \epsilon_\ell\Big)' c^{(k)}.$$

- Then build a fast evaluator in a' for each z_j:

$$\widehat{\Gamma}^{(k)}(\cdot, z_j) \quad : \quad a' \mapsto \text{interpolation of } \{\Gamma_{ij}^{(k)}\}_{i=1}^{N_a}.$$

- Pay the quadrature sum $\sum_{\ell=1}^{L}$ once per (a_i, z_j) at the start of an outer iteration
- Thereafter, evaluating $\widehat{\Gamma}^{(k)}(a', z_j)$ is just interpolation in a'

Maximization with a Precomputed Continuation Term

With $\widehat{\Gamma}^{(k)}$ in hand, the node update during policy improvement becomes

$$w_n^{(k+1)} = \max_{x \in [0,1]} \left\{ (1-\beta)\Big((1-x)\, m(a_n, z_n)\Big)^{1-\sigma} + \beta\, \widehat{\Gamma}^{(k)}\Big(x\, m(a_n, z_n), z_n\Big) \right\}.$$

- The maximizer still sees the same Bellman objective; we only reorganize computation
- Each trial x now calls $\widehat{\Gamma}^{(k)}(\cdot, z_n)$ once, instead of recomputing $\sum_{\ell=1}^{L}$
- Biggest gains occur exactly where you spend the most time