Quantitative Macroeconomics w/ AI and ML
Lec. 4: Numerical Methods for Macroeconomists

Zhigang Feng

Dec., 2025

Floating-Point Format Structure

- A 32-bit single-precision number consists of:
  1. **Sign (S):** 1 bit (0 for $+$, 1 for $-$).
  2. **Exponent (E):** 8 bits (Bias $= 127$).
  3. **Mantissa (M):** 23 bits (Fractional part).
- **Normalized Value Formula ($1 \leq E \leq 254$):**

$$Value = (-1)^S \times (1 + M) \times 2^{(E-127)}$$

Maximum Representable Value

- **Conditions for Max Value:**
  - $E = 254$ (Max normalized exponent).
  - $M =$ All 1s (Implicit 1 + 23 fractional bits).

- **Calculation:**

$$Val_{max} = \underbrace{(2 - 2^{-23})}_{\substack{\text{Binary } 1.11\ldots1 \\ \approx 2}} \times 2^{(254-127)}$$

$$Val_{max} \approx 2 \times 2^{127} = 2^{128} \approx 3.4028 \times 10^{38}$$

- **Note:** $(2 - 2^{-23})$ represents the implicit 1 plus the sum of all 23 fractional bits.

Example: Converting Bits to Value

Binary Pattern
`0 10000001 10100000000000000000000`

1. **Sign (S):** $0 \rightarrow$ Positive.
2. **Exponent (E):** $10000001_2 = 129$.
   - True Exponent $= 129 - 127 = \mathbf{2}$.
3. **Mantissa (M):** `101...` corresponds to $1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$.
   - Fraction $= 0.5 + 0.125 = \mathbf{0.625}$.
   - Significand $= 1 + 0.625 = \mathbf{1.625}$.
4. **Result:**

$$Value = +1.625 \times 2^2 = 1.625 \times 4 = \mathbf{6.5}$$

Machine Epsilon (Precision)

- **Concept:** The "step size" between representable numbers is determined by the last bit, but scaled by the exponent.
- **Visualizing the Step (at 1.0):**
  - **Value 1.0:**
    `0 01111111 00000000000000000000000`
    $\hookrightarrow (-1)^0 \times \underbrace{1.0}_{\text{Implicit 1}} \times 2^{(127-127)} = 1 \times 1.0 \times 2^0 = \mathbf{1.0}$
  - **Next Value (Smallest Step Up):**
    `0 01111111 00000000000000000000001`
    $\hookrightarrow (-1)^0 \times (1 + \underbrace{2^{-23}}_{\text{Last bit}}) \times 2^{(127-127)} = \mathbf{1.0 + 2^{-23}}$
- **Machine Epsilon (ϵ_{mach}):** This gap is the base precision.

$$\epsilon_{mach} = 2^{-23} \approx 1.19 \times 10^{-7}$$

- **Scaling:** At larger magnitudes, this gap is multiplied by 2^E.
  Example: If $2^E = 2^{20}$, the gap becomes $2^{-23} \times 2^{20} = 2^{-3} = 0.125$.

Visualizing Floating-Point Spacing

- Floating-point numbers are **dense** near zero and **sparse** further out.
- As the Exponent (E) increases, the distance between ticks doubles.

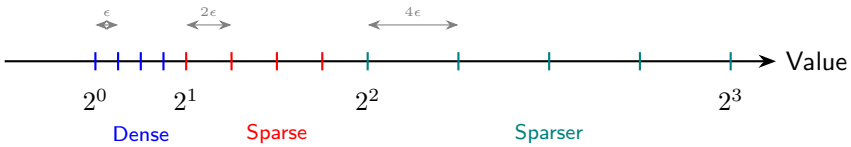


Figure 1: The "Gap" doubles at each power of 2

The "Floating" Point
The binary point "floats" to maintain relative precision, but the **absolute error** grows with magnitude.

Denormals and Absolute Minimum

- **The Gap at Zero:** Normalized numbers stop at 2^{-126}.
- **Denormals ($E = 0$):** Use fixed exponent -126 to fill the gap down to zero.

$$\text{Smallest Positive} = 2^{-149} \approx 1.4 \times 10^{-45}$$

Boundary	Value	Concept
Max Value	$\approx 3.4 \times 10^{38}$	Range Limit
Machine Epsilon	$\approx 1.2 \times 10^{-7}$	Precision Limit (at 1.0)
Min Normalized	$\approx 1.2 \times 10^{-38}$	Full Precision Floor
True Minimum	$\approx 1.4 \times 10^{-45}$	Absolute Floor (Denormal)

Implications for Quantitative Macroeconomics

- **The Scale Problem:** Macro models often combine variables of vastly different magnitudes.
  - *Large:* Aggregate Capital (K), GDP (Y) $\sim 10^{12}$ to 10^{14}.
  - *Small:* Interest rates (r), depreciation (δ), or individual shocks $\sim 10^{-2}$ to 10^{-4}.
- **The "Big + Small" Trap (Absorption):**
- Because the "gap" grows with the number, adding a small value to a large aggregate may result in **zero change**.
- **Example (Single Precision):**
  - U.S. GDP $\approx \$28$ Trillion ($\approx 2.8 \times 10^{13}$).
  - At this magnitude, the "gap" (precision) is:

$$\text{Gap} \approx 2.8 \times 10^{13} \times 10^{-7} \approx \mathbf{2,800,000}$$

  - *Consequence:* You cannot add a \$100,000 transaction to the U.S. GDP. The computer effectively calculates:

$$28,000,000,000,000 + 100,000 = 28,000,000,000,000$$

Recommendation for Macro Modeling

While ML often defaults to **Float32** (or Float16) for speed, macro-accounting identities usually require **Double Precision (Float64)** to prevent "vanishing" shocks.

Good Programming

Good Programming: Style & Readability

- **Meaningful Naming:** Use descriptive variable names.
  - *Bad:* `int n = 30;`
  - *Good:* `int numStudents = 30;`

- **Consistent Formatting:** Use indentation and spacing to show logical structure.

```
if (age >= 18) {
    System.out.println("Eligible to vote.");
} else {
    System.out.println("Not eligible.");
}
```

- **Documentation:** Comment complex logic or headers.

```
// Returns the average of an array of numbers
double calculateAverage(double[] numbers) { ... }
```

Good Programming: Development Workflow

- **Start Simple (Incrementalism):** Do not code the whole system at once. Start with the simplest case.
  - *Example:* Implement `Bubble Sort` to verify logic before optimizing with `QuickSort`.
- **Test Frequently:** Run your code after every new function or feature. This isolates errors immediately.
- **Progress Monitoring:** For long-running loops, print status updates.

```
for (int i = 0; i < data.length; i++) {
    // Process data...
    if (i % 1000 == 0) {
        System.out.println("Processed " + i + " items...");
    }
}
```

Good Programming: Numerical Verification

- **Theoretical Consistency Checks:** Verify results against known theoretical relationships.
  - *Example:* If finding roots of a quadratic, plug them back into $ax^2 + bx + c$ to see if the result is ≈ 0.
- **Convergence Testing:**
  As accuracy requirements increase (e.g., more integration intervals), the solution should stabilize.
- **Sensitivity Analysis:** Make small changes to parameters (mass, velocity, initial guess). Large jumps in the output may indicate instability.
  - *Tip:* Try different initial guesses for iterative solvers (like Newton's Method).

Good Programming: Leveraging AI

- **Decomposition is Key:** reak complex problems into manageable tasks.
  - *Step 1:* Define Input/Output.
  - *Step 2:* Outline the algorithm.
  - *Step 3:* Request implementation of specific functions.

- **Iterative Refinement:** Prioritize a "working" solution first, then prompt for optimization.
  - *Prompt 1:* "Write a code that sorts this list."
  - *Prompt 2:* "Now refactor it to handle null values safely."

Numerical Techniques

Optimization

- Derivatives based
- Do not use derivatives
- Mixed methods
- Simulation methods

Newton Method

- Most common optimization method in economics (either basic implementation or, more likely, with modifications).
- Works with univariate and multivariate optimization problems, but requires twice-differentiability of function.
- Named after Isaac Newton and Joseph Raphson.
- Intimately related with the Newton method designed to solve for root to equation $f(x) = 0$.
- Optimizes $f(x)$ by using successive quadratic approximations to it.

Newton Method

- Given an initial guess x_0, compute the second-order Taylor approximation of $f(x)$ around x_0:

$$\tilde{f}(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2$$

- The minimization of this approximation with respect to x has first-order conditions

$$f'(x_0) + \frac{1}{2}f''(x_0) \cdot 2(x - x_0) = 0 \tag{1}$$

  which gives: $x^* = x_0 - \frac{f'(x_0)}{f''(x_0)}$.

- This suggests the iteration

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

  which converges quadratically.

Newton Method

- $\frac{f'(x_n)}{f''(x_n)}$ is a measure of how far away the current guess x_n is from the point where the derivative is zero.
- If $f'(x_n)$ is large (either positive or negative), it suggests we are far from the extremum (maximum or minimum).
- If $f''(x_n)$ is large, it suggests the derivative is changing rapidly, so our guess might not be far off.
- We subtract this ratio from the current guess x_n because we want to move in the direction that brings the derivative closer to zero.
- If $f''(x_n)$ is positive, we are likely near a minimum, so we want to move in the direction that decreases the derivative. If $f''(x_n)$ is negative, we are likely near a maximum, so we want to move in the direction that increases the derivative.

Newton Method

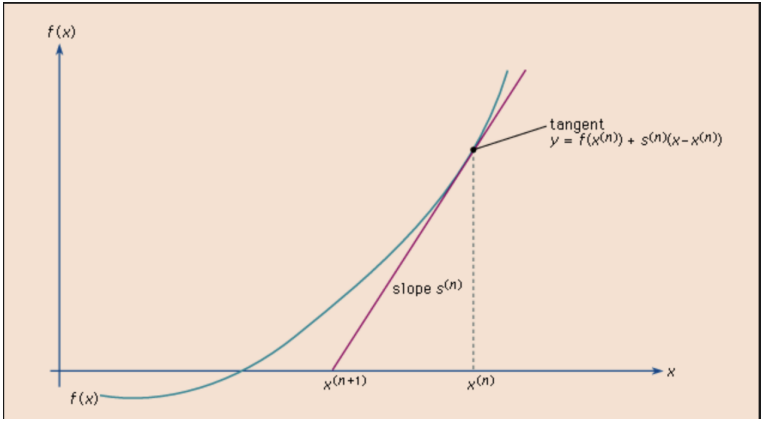


Figure 2: Newton's Iteration

Newton Method

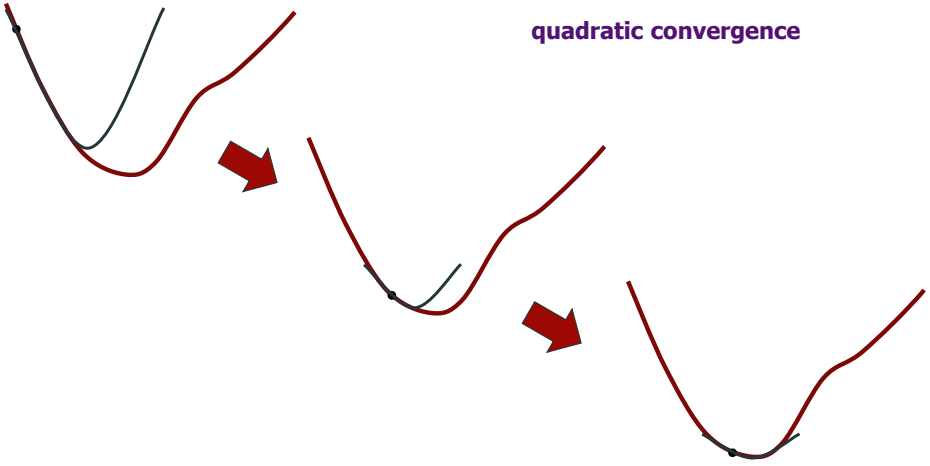


Figure 3: Newton's Iteration

Newton's Method: The Multivariate Generalization

- **1-Dimensional Case ($x \in \mathbb{R}$):**

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

- **N-Dimensional Case ($x \in \mathbb{R}^N$):**
  - The derivative $f'(x)$ becomes the **Gradient** $\nabla f(x)$ (Vector).
  - The curvature $f''(x)$ becomes the **Hessian** $H(x)$ (Matrix).
  - Division becomes **Matrix Inversion**.

$$x_{n+1} = x_n - \underbrace{H(x_n)^{-1}}_{\text{Inverse Hessian}} \times \underbrace{\nabla f(x_n)}_{\text{Gradient}}$$

Why Multivariate is Hard: The "Curse" of Dimensionality

- **The Bottleneck:** Storing and inverting the Hessian.
- Let's compare a scalar case to a medium-sized macro model ($N = 1000$ variables):

Operation	**1D ($N = 1$)**	**Multivariate ($N = 1,000$)**
Storage	1 number	$1,000^2 = \mathbf{1,000,000}$ entries
Compute Step	1 Division	Invert 1000×1000 Matrix
Complexity	$O(1)$	$O(N^3) \approx \mathbf{1,000,000,000}$ ops

Intuition

In 1D, checking the curvature is free. In high dimensions, simply *constructing* the map of curvature (the Hessian) can take longer than the entire optimization process.

Case Study: The Jump from 1 to 2 Variables

1 Variable $f(x)$

- **Curvature:** A single number (Scalar).

$$H = f''(x)$$

- **Update Cost:** Simple division.

$$\Delta x = -\frac{f'(x)}{f''(x)}$$

- **Condition:** Check if $f''(x) > 0$.

2 Variables $f(x, y)$

- **Curvature:** A 2×2 Matrix.

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$$

- **Update Cost:** Matrix Inversion.

$$\Delta \mathbf{x} = -H^{-1} \nabla f$$

- **Condition:** Check Eigenvalues (Positive Definite).

The Source of Complexity: Cross-Terms

The terms $\frac{\partial^2 f}{\partial x \partial y}$ represent **interactions**.

- In 1D, the slope changes only because you move.
- In 2D, moving in x changes the slope in y. The variables are **coupled**, so you cannot optimize them separately.

Geometric Complexity: The Saddle Point Problem

- **1D Intuition:** If the slope is zero ($\nabla f = 0$), you are usually at a minimum or maximum.
- **ND Reality:** In high dimensions, you are most likely at a **Saddle Point**.
  - *Example:* In a 100-dimensional landscape, to be a minimum, the curvature must be positive in **all 100 directions** simultaneously.
  - If just one direction curves down while 99 curve up, Newton's Method may get stuck or diverge.
- **Ill-Conditioning:**
  - If the "bowl" is very steep in one direction and very flat in another (like a taco shell), inverting the Hessian becomes numerically unstable (H^{-1} explodes).

Beyond Standard Newton: The Strategy

- **The Bottleneck:** Computing and inverting the Hessian H ($O(N^3)$) is too slow for large N.
- **Three Major Strategies to fix this:**
  1. **Quasi-Newton (e.g., BFGS):**
     - *Idea:* Don't compute H. Estimate it using the change in gradients over time.
     - *Cost:* $O(N^2)$ (Matrix-vector multiplication).
  2. **Conjugate Gradient:**
     - *Idea:* Use only gradient information but choose "smart" search directions that preserve previous progress.
     - *Cost:* $O(N)$ (Linear).
  3. **Stochastic Methods (e.g., SGD):**
     - *Idea:* Don't use all data. Estimate gradients using small "mini-batches."
     - *Cost:* Independent of data size M.

Quasi-Newton: The Intuition

- **The Cost Savings:**
  - *Newton:* Calculates the true Hessian H from scratch every step ($O(N^3)$ cost).
  - *Quasi-Newton:* Updates an *estimate* of H using information we already have ($O(N^2)$ cost).
- **The Logic (Secant Condition):** We treat the Gradient $\nabla f(x)$ as a function. Its "slope" is the Hessian H.

$$\underbrace{\nabla f(x_{n+1}) - \nabla f(x_n)}_{\text{Change in Gradient } (y_n)} \approx \underbrace{H}_{\text{Slope}} \cdot \underbrace{(x_{n+1} - x_n)}_{\text{Change in Position } (s_n)}$$

- Instead of computing derivatives, we force our approximation matrix B to satisfy $Bs_n = y_n$.

BFGS and the Sherman-Morrison Trick

- **Definitions:**
  - $s_n = x_{n+1} - x_n$ (Step taken).
  - $y_n = \nabla f(x_{n+1}) - \nabla f(x_n)$ (Gradient change).
- **The Challenge:** Even if we approximate H, we still need to invert it to find the step direction: $\Delta x = -H^{-1}\nabla f$.
- **The Solution (Sherman-Morrison Formula):**
  - An algebraic identity that allows us to update the **Inverse Matrix** (H^{-1}) directly.
  - *Intuition:* "If we change the matrix by a small rank-2 amount, we can update the inverse by a small additive amount."
- **Result:** Converts matrix inversion (Costly) into matrix multiplication (Cheap).

Baseline: Standard Newton's Method (The Expensive Way)

- **Scenario:** Minimize $f(x, y) = 0.5x^2 + y^2$ starting at $(x_0, y_0) = (2, 2)$.
  Notation: Let $\boldsymbol{\theta} = [x, y]^\top$ denote the parameter vector.
- **Step 1: Compute Exact Derivatives**
  - Gradient: $\nabla f = [x, 2y]^\top$.
  - Hessian: We must calculate second derivatives for every element.

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

- **Step 2: Invert and Update**

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta}_{\text{old}} - H^{-1} \nabla f$$

- **Critique:** Here H is constant. But for complex functions, re-calculating and inverting this matrix at every step is the bottleneck ($O(N^3)$).

BFGS Example Step 1: The First Step

- **Start:** $\boldsymbol{\theta}_0 = [2, 2]^\top$. Function $f(x, y) = 0.5x^2 + y^2$.
- **Gradient:** $g_0 = \nabla f(\boldsymbol{\theta}_0) = [2, 4]^\top$.
- **Direction:** We compute the search direction using

$$d_0 = -B_0^{-1} g_0 = -I \cdot [2, 4]^\top = [-2, -4]^\top$$

  (With initial guess $B_0 = I$, this reduces to steepest descent.)
- **Line Search for α_0:** Find α that minimizes $f(\boldsymbol{\theta}_0 + \alpha d_0)$.

$$\boldsymbol{\theta}_1 = \boldsymbol{\theta}_0 + \alpha_0 d_0 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0.5 \begin{bmatrix} -2 \\ -4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

- **Result:** We arrive at $\boldsymbol{\theta}_1 = [1, 0]^\top$ with gradient $g_1 = [1, 0]^\top$.

Will α appear in future steps?

Yes! At every iteration k, we compute $d_k = -B_k^{-1} g_k$ and perform a line search to find α_k. The step size α_k changes at each iteration.

Interlude: What is a Line Search?

- The algorithm gives a **direction** d, but not **how far** to go.
- **Why minimize** $f(\boldsymbol{\theta} + \alpha d)$**?**
  - d points "downhill" (descent direction), so moving along it reduces f
  - Too small α: wastes iterations, slow progress
  - Too large α: overshoots the minimum, may increase f
  - **Optimal** α: maximizes progress per iteration

The 1D Subproblem

Define $\phi(\alpha) = f(\boldsymbol{\theta}_k + \alpha \cdot d_k)$. This converts optimization over $\mathbb{R}^N$ into a 1D problem: find $\alpha^* = \arg\min_{\alpha \geq 0} \phi(\alpha)$.

- **In practice:** Use Wolfe conditions or backtracking (don't need exact minimum).
- **In our example:** We chose $\alpha_0 = 0.5$ for clean arithmetic.

BFGS Example Step 2: The Correction Vectors

We need to update our approximation B to reflect the curvature we just observed.

1. **Change in Position (s_0):**

$$s_0 = \boldsymbol{\theta}_1 - \boldsymbol{\theta}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ -2 \end{bmatrix}$$

2. **Change in Gradient (y_0):**

$$y_0 = g_1 - g_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} -1 \\ -4 \end{bmatrix}$$

The Insight

The gradient changed by -4 in the y-component but only -1 in the x-component. This tells the algorithm: **curvature is steeper in the y-direction** (which matches $H_{22} = 2 > H_{11} = 1$).

BFGS Example Step 3a: The Secant Condition

The Core Requirement: We want our new approximation B_1 to satisfy:

$$\boxed{B_1 \, s_0 = y_0}$$

Why this condition?

- For the true Hessian H, the mean value theorem implies:

$$\nabla f(x_1) - \nabla f(x_0) \approx H \cdot (x_1 - x_0)$$

- In our notation: $y_0 \approx H \cdot s_0$.
- We want B_1 to mimic this behavior: $B_1 s_0 = y_0$.

Verification with True Hessian

$$H \cdot s_0 = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} -1 \\ -2 \end{bmatrix} = \begin{bmatrix} -1 \\ -4 \end{bmatrix} = y_0 \quad \checkmark$$

The secant condition ensures B_1 captures the correct curvature information along direction s_0.

BFGS Example Step 3b: The Update Formula

The BFGS Update: Among all matrices satisfying $B_1 s_0 = y_0$, BFGS finds the one "closest" to B_0 while remaining symmetric positive definite:

$$B_1 = B_0 + \frac{y_0 y_0^\top}{y_0^\top s_0} - \frac{B_0 s_0 s_0^\top B_0}{s_0^\top B_0 s_0}$$

Intuition for each term:

- B_0: Start from our previous approximation.
- $+\frac{y_0 y_0^\top}{y_0^\top s_0}$: **Add curvature** in the direction we observed steep gradient change.
- $-\frac{B_0 s_0 s_0^\top B_0}{s_0^\top B_0 s_0}$: **Remove old curvature** estimate in direction s_0 (replace with new info).

Key Property

This is a **rank-2 update**: we only modify B along two directions, keeping computational cost at $O(N^2)$ instead of $O(N^3)$.

BFGS Example Step 3c: Computing B_1

Scalar computations:

$$y_0^\top s_0 = (-1)(-1) + (-4)(-2) = 9, \qquad s_0^\top B_0 s_0 = (-1)^2 + (-2)^2 = 5$$

Matrix computations:

$$\frac{y_0 y_0^\top}{y_0^\top s_0} = \frac{1}{9}\begin{bmatrix} 1 & 4 \\ 4 & 16 \end{bmatrix}, \qquad \frac{B_0 s_0 s_0^\top B_0}{s_0^\top B_0 s_0} = \frac{1}{5}\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

Result:

$$B_1 = I + \frac{1}{9}\begin{bmatrix} 1 & 4 \\ 4 & 16 \end{bmatrix} - \frac{1}{5}\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 0.91 & 0.04 \\ 0.04 & 1.98 \end{bmatrix}$$

Comparison with True Hessian

$$B_1 = \begin{bmatrix} 0.91 & 0.04 \\ 0.04 & 1.98 \end{bmatrix} \approx H = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

After one step, $B_1 \approx H$: diagonals $\approx 1, 2$; off-diagonals small. More iterations: $B_k \to H$.

BFGS Example Step 3d: Verifying the Secant Condition

Let's verify that our computed B_1 satisfies $B_1 s_0 = y_0$:

$$B_1 s_0 = \begin{bmatrix} 0.91 & 0.04 \\ 0.04 & 1.98 \end{bmatrix} \begin{bmatrix} -1 \\ -2 \end{bmatrix} = \begin{bmatrix} -0.91 - 0.08 \\ -0.04 - 3.96 \end{bmatrix} = \begin{bmatrix} -0.99 \\ -4.0 \end{bmatrix} \approx y_0 \checkmark$$

Summary

After just **one step**, BFGS has learned:

- The curvature in the y-direction is approximately $2\times$ that of the x-direction.
- This matches the true Hessian $H = \text{diag}(1, 2)$.

For quadratic functions, BFGS converges to the exact Hessian in at most N steps!

Relationship: Quasi-Newton vs. BFGS

- **BFGS as a special case of Quasi-Newton**
  - **Quasi-Newton** is the *family* of methods that use gradient updates to approximate the Hessian via the secant condition $B_{k+1}s_k = y_k$.
  - **BFGS** (Broyden–Fletcher–Goldfarb–Shanno) is the most popular *specific algorithm* in this family.
  - Other members include **SR1** (Symmetric Rank 1) and **DFP** (Davidon–Fletcher–Powell).
- **Why BFGS wins:** It is more numerically stable than DFP and maintains positive-definiteness (ensures we always move downhill).

Alternative: Conjugate Gradient (CG) – The Concept

- **Problem with Steepest Descent:** It zig-zags! Progress in one direction gets partially "undone" by the next step.
- **CG Insight:** Choose directions that **don't interfere** with each other.
- **Definition:** Two directions d_i and d_j are H-**conjugate** if:

$$d_i^\top H\, d_j = 0 \quad \text{for } i \neq j$$

- **What this means:** Once we minimize along d_0, moving along d_1 won't change the objective in the d_0 direction—no backtracking!

Key Advantages

- Memory: $O(N)$ — only store a few vectors, no matrix
- Convergence: Exactly N steps for N-dimensional quadratics
- Use case: Massive systems where even $O(N^2)$ storage is too much

CG Example Step 1: Setup and First Direction

Same problem: Minimize $f(x, y) = 0.5x^2 + y^2$, starting at $\boldsymbol{\theta}_0 = [2, 2]^\top$.

Step 1: Initialize with steepest descent direction

- Gradient: $g_0 = \nabla f(\boldsymbol{\theta}_0) = [2, 4]^\top$
- First search direction: $d_0 = -g_0 = [-2, -4]^\top$

Step 2: Line search along d_0 (same principle as before!)

We want $\alpha^* = \arg\min_\alpha f(\boldsymbol{\theta}_0 + \alpha d_0)$. For quadratics, setting $\frac{d}{d\alpha} f = 0$ gives:

$$\alpha_0 = \frac{g_0^\top g_0}{d_0^\top H d_0} = \frac{20}{36} = \frac{5}{9}$$

Update: $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_0 + \alpha_0 d_0 = [8/9, -2/9]^\top$

Connection to General Line Search

This is exactly "minimize $f(\boldsymbol{\theta} + \alpha d)$" from before! For quadratics, we get a closed form. For general f, use Wolfe conditions or backtracking.

CG Example Step 2: The Complete CG Iteration

At iteration k, the CG algorithm has three steps:

Step A: Compute Direction

$$d_k = -g_k + \beta_{k-1} d_{k-1}, \quad \text{where } \beta_{k-1} = \frac{g_k^\top g_k}{g_{k-1}^\top g_{k-1}}$$

Step B: Compute Step Size (Line Search)

$$\alpha_k = \frac{g_k^\top g_k}{d_k^\top H d_k} \quad \text{(for quadratics; otherwise use Wolfe conditions)}$$

Step C: Update Position

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha_k d_k$$

Then: Compute new gradient $g_{k+1} = \nabla f(\boldsymbol{\theta}_{k+1})$ and repeat.

CG Example Step 2: Numerical Calculation

Continuing our example: $\boldsymbol{\theta}_1 = [8/9, -2/9]^\top$, $g_1 = [8/9, -4/9]^\top$

Step A: Direction

$$\beta_0 = \frac{g_1^\top g_1}{g_0^\top g_0} = \frac{80/81}{20} = \frac{4}{81}$$

$$d_1 = -g_1 + \beta_0 d_0 = -\begin{bmatrix} 8/9 \\ -4/9 \end{bmatrix} + \frac{4}{81}\begin{bmatrix} -2 \\ -4 \end{bmatrix} = \begin{bmatrix} -80/81 \\ 40/81 \end{bmatrix}$$

Step B: Step size

$$\alpha_1 = \frac{g_1^\top g_1}{d_1^\top H d_1} = \frac{80/81}{d_1^\top H d_1} = \ldots = \frac{9}{10}$$

Step C: Update

$$\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1 + \alpha_1 d_1 = \begin{bmatrix} 8/9 \\ -2/9 \end{bmatrix} + \frac{9}{10}\begin{bmatrix} -80/81 \\ 40/81 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \boldsymbol{\theta}^* \checkmark$$

CG: Connection to Newton's Method

Recall Newton: $\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta}_{\text{old}} - H^{-1}\nabla f$

The key insight: CG **implicitly** computes $H^{-1}g$ without ever forming H^{-1}!

How CG Approximates Newton

- Newton: One step with exact H^{-1}
- CG: N steps that **together** achieve the same effect
- After N conjugate directions, CG has effectively applied H^{-1}

Why this works:

- Conjugate directions $\{d_0, d_1, \ldots, d_{N-1}\}$ form a basis for $\mathbb{R}^N$
- Each direction is "H-orthogonal" to all previous ones
- Minimizing along all N directions = solving $H^{-1}g$ in that basis

Trade-off: N cheap iterations vs. 1 expensive $O(N^3)$ inversion

CG Example Step 3: Convergence

For our 2D quadratic: CG converges in exactly 2 steps (at most N steps for N-dimensional quadratics).

After line search along d_1: $\boldsymbol{\theta}_2 = \boldsymbol{\theta}^* = [0, 0]^\top$

Steepest Descent:

- Zig-zags toward minimum
- Many iterations needed

Conjugate Gradient:

- Orthogonal progress (w.r.t. H)
- At most N steps for quadratics

Memory Comparison

Method	Memory	Per-iteration cost
Newton	$O(N^2)$	$O(N^3)$
BFGS	$O(N^2)$	$O(N^2)$
CG	$O(N)$	$O(N)$

SGD: A Different Problem Setting

- **The Pivot:** BFGS and CG assume we can compute $\nabla f(x)$ exactly. But what if the objective is a **sum over data**?
- **Machine Learning Setting:** Minimize empirical risk:

$$L(\theta) = \frac{1}{M} \sum_{i=1}^{M} \ell(\theta; z_i)$$

  where $\{z_1, \ldots, z_M\}$ is the training dataset.
- **Example:** Least squares regression with M data points:

$$L(\theta) = \frac{1}{M} \sum_{i=1}^{M} (y_i - \theta^\top x_i)^2$$

- **Problem:** When $M = 10^6$ or 10^9, computing the full gradient

$$\nabla L(\theta) = \frac{1}{M} \sum_{i=1}^{M} \nabla \ell(\theta; z_i)$$

  at every iteration is prohibitively expensive!

Stochastic Gradient Descent: The Algorithm

Key Idea: Replace the exact gradient with a **noisy but cheap** estimate.

SGD Update Rule

At iteration k:

1. Sample a mini-batch $\mathcal{B}_k \subset \{1, \ldots, M\}$ of size $|\mathcal{B}|$
2. Compute stochastic gradient: $g_k = \dfrac{1}{|\mathcal{B}_k|} \displaystyle\sum_{i \in \mathcal{B}_k} \nabla \ell(\theta_k; z_i)$
3. Update: $\theta_{k+1} = \theta_k - \alpha_k \, g_k$

Cost comparison per iteration:

- Full gradient: $O(M \cdot N)$ — process all M samples
- SGD: $O(|\mathcal{B}| \cdot N)$ — process only $|\mathcal{B}| \ll M$ samples

Trade-off: Noisy gradient $\Rightarrow$ more iterations, but each is $M/|\mathcal{B}|$ times faster!

SGD Example: Connecting to Our Function

Reinterpret our function as an empirical risk:

Suppose we have 2 data points contributing to:

$$f(\theta_1, \theta_2) = \underbrace{0.5\theta_1^2}_{\ell_1(\theta)} + \underbrace{\theta_2^2}_{\ell_2(\theta)}$$

Full gradient (batch):

$$\nabla f = \frac{1}{2}(\nabla \ell_1 + \nabla \ell_2) = \frac{1}{2}\left([\theta_1, 0]^\top + [0, 2\theta_2]^\top\right) = \begin{bmatrix} 0.5\theta_1 \\ \theta_2 \end{bmatrix}$$

SGD iteration (sample one term randomly):

- If we sample ℓ_1: $g = [\theta_1, 0]^\top$ (only updates θ_1)
- If we sample ℓ_2: $g = [0, 2\theta_2]^\top$ (only updates θ_2)

At $\theta = [2, 2]^\top$ with $\alpha = 0.5$:

- Sample ℓ_1: $\theta_{\text{new}} = [2, 2]^\top - 0.5[2, 0]^\top = [1, 2]^\top$
- Sample ℓ_2: $\theta_{\text{new}} = [2, 2]^\top - 0.5[0, 4]^\top = [2, 0]^\top$

SGD Example: The Noisy Path

Starting at $\theta_0 = [2, 2]^\top$, possible SGD trajectory:

Step	Sampled	Gradient estimate	New θ
0	ℓ_2	$[0, 4]^\top$	$[2, 0]^\top$
1	ℓ_1	$[2, 0]^\top$	$[1, 0]^\top$
2	ℓ_1	$[1, 0]^\top$	$[0.5, 0]^\top$
3	ℓ_2	$[0, 0]^\top$	$[0.5, 0]^\top$
4	ℓ_1	$[0.5, 0]^\top$	$[0.25, 0]^\top$

(Using $\alpha = 0.5$ throughout)

Observations

- Path is **stochastic**—different runs give different trajectories
- Progress is **uneven**—some steps help one coordinate only
- Overall trend: converging toward $[0, 0]^\top$

Why Does SGD Converge Despite the Noise?

- **1. Unbiased Estimator:** The expected value of the stochastic gradient equals the true gradient:

$$\mathbb{E}[g_{\mathcal{B}}] = \nabla L(\theta)$$

  Over many iterations, errors cancel out (Law of Large Numbers).
- **2. Data Redundancy:** In large datasets, many data points are similar. The gradient from a random 100 samples approximates the gradient from 10^6 samples well.
- **3. Implicit Regularization:** The noise helps escape sharp local minima and saddle points—finding flatter, more generalizable solutions.
- **4. Learning Rate Schedule:** Decreasing $\alpha_k \to 0$ (but $\sum \alpha_k = \infty$) ensures:
  - Early: large steps for fast progress
  - Late: small steps for precise convergence

Why High Dimensions Require Small α

- **The Math:** Suppose every parameter has a small gradient component of magnitude 1.
  - **1 Dimension ($N = 1$):**
    $$\|\nabla L\| = \sqrt{1^2} = 1$$
  - **Deep Learning ($N = 10^6$):**
    $$\|\nabla L\| = \sqrt{\sum_{i=1}^{10^6} 1^2} = \sqrt{10^6} = 1000$$
- **The Step Size Explosion:**
  $$\|\text{Step}\| = \alpha \times \|\nabla L\|$$
- If $\|\nabla L\| = 1000$, keeping α constant would result in a huge step that overshoots the minimum.
- **Conclusion:** To keep the physical step size safe, α must scale as $O(1/\sqrt{N})$ or be carefully tuned to counterbalance the high-dimensional gradient magnitude.

Summary: Choosing the Right Method

Method	Memory	Per-step	Best for
Newton	$O(N^2)$	$O(N^3)$	Small N, need precision
BFGS	$O(N^2)$	$O(N^2)$	Medium N ($< 10^4$)
L-BFGS	$O(mN)$	$O(N)$	Large N, limited memory
CG	$O(N)$	$O(N)$	Huge N, sparse systems
SGD	$O(N)$	$O(\|\mathcal{B}\|N)$	Huge M (data), ML

Key Insight

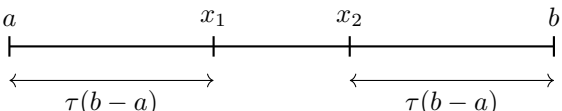
- **Classical optimization** (Newton, BFGS, CG): exact gradients, deterministic paths
- **Stochastic optimization** (SGD): noisy gradients, random paths, but scales to massive data

The choice depends on your problem's structure: dimension N, data size M, and required precision.

Golden Section Search: The 1D Line Search

Problem: Find minimum of unimodal $f(x)$ on interval $[a, b]$.

Strategy: Iteratively narrow the interval by evaluating f at two interior points.

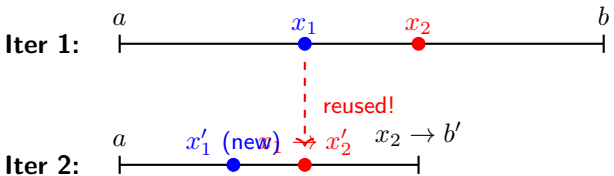


Decision rule:

- If $f(x_1) < f(x_2)$: minimum is in $[a, x_2] \Rightarrow$ discard $[x_2, b]$
- If $f(x_1) > f(x_2)$: minimum is in $[x_1, b] \Rightarrow$ discard $[a, x_1]$

The Efficiency Hack: Recycling Points

Key insight: Choose x_1, x_2 so that after discarding, one point can be **reused**!

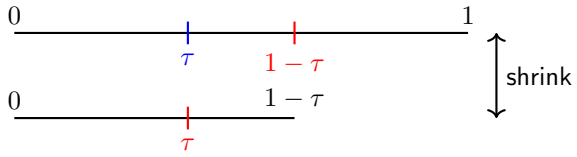


Benefit: Only **1 new evaluation** per iteration (not 2)!

Question: What ratio τ allows this recycling?

Why the "Golden" Ratio?

Requirement: After shrinking, the **old interior point** must be at the **correct position** in the new interval.



The constraint: Old x_1 at position τ must equal new x_2 at position $(1-\tau)$ of shrunken interval:

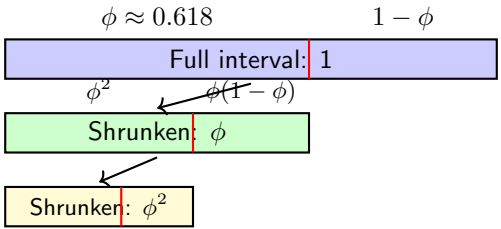
$$\tau = (1-\tau)\cdot(1-\tau) \quad \Rightarrow \quad \tau = (1-\tau)^2$$

$$\tau^2 - 3\tau + 1 = 0 \quad \Rightarrow \quad \boxed{\tau = \frac{3-\sqrt{5}}{2} \approx 0.382}$$

The ratio $\phi = 1 - \tau = \frac{\sqrt{5}-1}{2} \approx 0.618$ is the **Golden Ratio**!

Golden Ratio: The Self-Similar Property

The magic: The golden ratio is the *only* number with this self-similarity.



Each iteration:

- Interval shrinks by factor $\phi \approx 0.618$
- One point is reused $\Rightarrow$ only 1 new f evaluation
- After n iterations: interval size $= \phi^n$ (exponential convergence)

Random Walk Metropolis-Hastings (RWMH)

Goal: Sample from a probability distribution $\pi(\theta)$ (e.g., posterior in Bayesian inference).

Algorithm:

1. **Propose:** $\theta^* = \theta_{n-1} + \lambda\epsilon$, where $\epsilon \sim N(0, I)$
2. **Accept/Reject:** Compute acceptance probability

$$\alpha = \min\left\{1, \frac{\pi(\theta^*)}{\pi(\theta_{n-1})}\right\}$$

3. **Update:**

$$\theta_n = \begin{cases} \theta^* & \text{with probability } \alpha \\ \theta_{n-1} & \text{with probability } 1-\alpha \end{cases}$$

Key Property

Always accept "uphill" moves (higher π). Sometimes accept "downhill" moves—this allows escaping local modes!

Simulated Annealing: MCMC for Optimization

Goal: Find $\theta^* = \arg\min f(\theta)$ (optimization, not sampling).

Key modification: Add a **temperature** T that decreases over time.

Acceptance probability:

$$\alpha = \min\left\{1, \exp\left(-\frac{f(\theta^*) - f(\theta_{n-1})}{T_n}\right)\right\}$$

Temperature schedule: $T_n \to 0$ as $n \to \infty$

- **High T (early):** Accept most moves $\Rightarrow$ explore widely
- **Low T (late):** Only accept improvements $\Rightarrow$ exploit local minimum

Physical Analogy

Named after annealing in metallurgy: heat metal (high T, atoms move freely), then cool slowly (atoms settle into low-energy crystal structure).

RWMH vs. Simulated Annealing: Key Differences

	RWMH	Simulated Annealing
Goal	Sample from $\pi(\theta)$	Find $\arg\min f(\theta)$
Temperature	Fixed ($T = 1$)	Decreasing $T_n \to 0$
Output	Chain of samples	Single best point
Acceptance	$\min\{1, \pi^*/\pi\}$	$\min\{1, e^{-\Delta f/T}\}$
Convergence	To stationary dist. π	To global minimum

The Connection

- Both use **random proposals + probabilistic acceptance**
- SA is MCMC with $\pi(\theta) \propto e^{-f(\theta)/T}$ and shrinking T
- As $T \to 0$: π concentrates on $\arg\min f$ (Gibbs distribution)
- RWMH at fixed $T = 1$: samples from $\pi \propto e^{-f}$ indefinitely

MCMC vs. Simulated Annealing: When to Use?

Use RWMH (MCMC) when:

- You need the **full posterior distribution** (uncertainty quantification)
- Bayesian inference: parameter estimation with credible intervals
- You want to **average over** multiple modes, not find just one

Use Simulated Annealing when:

- You only need the **single best solution**
- Combinatorial optimization (TSP, scheduling)
- Landscape has many local minima (SA can escape them early on)

Modern Alternatives

- For sampling: Hamiltonian Monte Carlo (HMC), NUTS
- For optimization: Genetic algorithms, particle swarm, Bayesian optimization

Numerical Differentiation: Why?

The Problem: We often need derivatives but:

- Analytical derivatives may be unavailable or too complex
- Functions may be defined only through simulation or data
- Automatic differentiation may not be available

The Solution: Approximate $f'(x)$ using function evaluations only.

Key Trade-off

Smaller step size $h \Rightarrow$ better approximation of the limit, but larger floating-point errors (subtracting nearly equal numbers).

Numerical Differentiation: Two Approaches

Starting from the definition:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

Method 1: Forward Difference (one-sided)

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

Method 2: Central Difference (two-sided)

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

Which is Better?

Central difference has **smaller error** — the error terms cancel! Let's see why...

Numerical Differentiation: Error Analysis

Taylor expansion of $f(x_0 + h)$:

$$f(x_0 + h) = f(x_0) + f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + \frac{1}{6}f'''(x_0)h^3 + R_3(x_0 + h)$$

Forward Difference Error:

$$\frac{f(x_0 + h) - f(x_0)}{h} = f'(x_0) + \underbrace{\frac{1}{2}f''(x_0)h}_{O(h)} + \frac{1}{6}f'''(x_0)h^2 + \cdots$$

Central Difference Error: (expand both $f(x_0 + h)$ and $f(x_0 - h)$)

$$\frac{f(x_0 + h) - f(x_0 - h)}{2h} = f'(x_0) + \underbrace{\frac{1}{6}f'''(x_0)h^2}_{O(h^2)} + \cdots$$

Conclusion

Central difference: $O(h^2)$ error. Forward difference: $O(h)$ error.

Central difference is **second-order accurate**!

Numerical Differentiation: Derivation Details

Full Taylor expansions:

$$f(x_0 + h) = f(x_0) + f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + \frac{1}{6}f'''(x_0)h^3 + R_3(x_0 + h)$$

$$f(x_0 - h) = f(x_0) - f'(x_0)h + \frac{1}{2}f''(x_0)h^2 - \frac{1}{6}f'''(x_0)h^3 + R_3(x_0 - h)$$

Subtracting and dividing by $2h$:

$$\frac{f(x_0 + h) - f(x_0 - h)}{2h} = f'(x_0) + \frac{1}{6}f'''(x_0)h^2 + \frac{R_3(x_0 + h) - R_3(x_0 - h)}{2h}$$

Why the Cancellation?

The even-order terms (f'', f'''', ...) have the same sign in both expansions $\Rightarrow$ they cancel when we subtract!

Interpolation: Why?

The Problem: We know function values at discrete points $\{(x_i, y_i)\}_{i=1}^{N}$, but need values at arbitrary points.

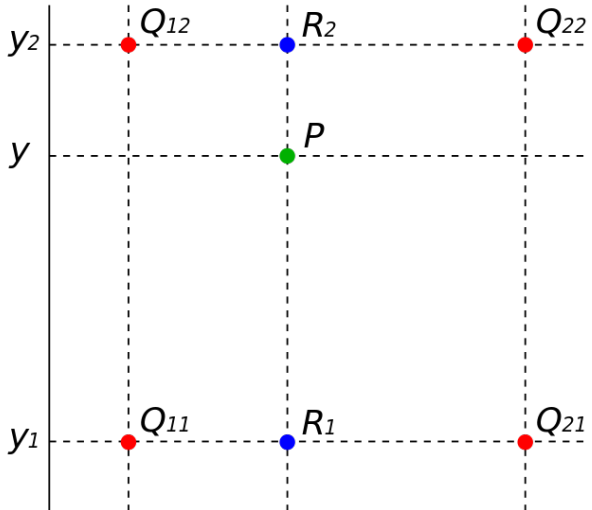
Applications in Economics:

- Value function iteration: evaluate $V(k')$ for any k'
- Policy functions: evaluate $c(k, z)$ off the grid
- Simulation: interpolate between grid points

Methods we'll cover:

1. Bilinear interpolation (2D rectangular grids)
2. Triangular interpolation (2D unstructured grids)
3. Cubic spline interpolation (1D, smooth)

Bilinear Interpolation: Setup



Bilinear Interpolation: Algorithm

Goal: Find $f(\hat{x}, \hat{y})$ given values at four corners.

Step 1: Define corner values

$$f_a = f(x_1, y_1), \quad f_b = f(x_1, y_2)$$
$$f_c = f(x_2, y_2), \quad f_d = f(x_2, y_1)$$

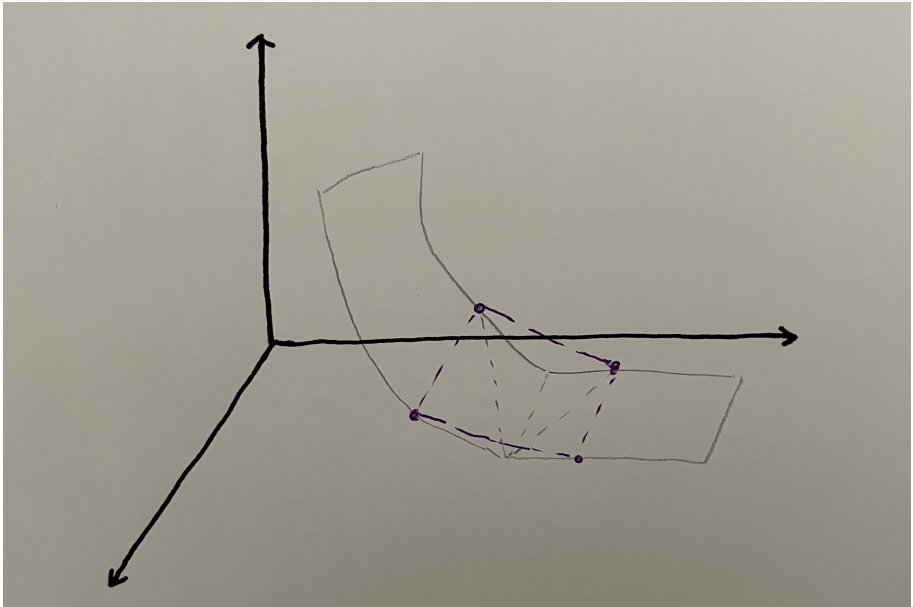
Step 2: Interpolate along y (vertical edges)

$$f_t = f_a + (f_b - f_a) \cdot \frac{\hat{y} - y_1}{y_2 - y_1} \quad \text{(left edge)}$$
$$f_u = f_d + (f_c - f_d) \cdot \frac{\hat{y} - y_1}{y_2 - y_1} \quad \text{(right edge)}$$

Step 3: Interpolate along x (between edges)

$$f(\hat{x}, \hat{y}) = f_t + (f_u - f_t) \cdot \frac{\hat{x} - x_1}{x_2 - x_1}$$

Triangular Interpolation: Setup



Triangular Interpolation: Algorithm

Goal: Find $f(x, y)$ inside triangle with vertices $(x_1, y_1), (x_2, y_2), (x_3, y_3)$.

Idea: Express (x, y) as weighted average of vertices (barycentric coordinates).

Step 1: Compute weights

$$w_1 = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)}$$

$$w_2 = \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)}$$

$$w_3 = 1 - w_1 - w_2$$

Step 2: Interpolate

$$f(x, y) = w_1 \cdot f(x_1, y_1) + w_2 \cdot f(x_2, y_2) + w_3 \cdot f(x_3, y_3)$$

Triangular Interpolation: Properties

- **Weights sum to 1:** $w_1 + w_2 + w_3 = 1$ (by construction)
- **At vertices:** If $(x, y) = (x_i, y_i)$, then $w_i = 1$ and $w_j = 0$ for $j \neq i$
- **Inside triangle:** All weights $\in (0, 1)$
- **Outside triangle:** Some weight < 0 (can use for extrapolation, but risky)

When to Use?

- Unstructured grids (adaptive mesh refinement)
- Irregular domains where rectangular grids don't fit
- Delaunay triangulation of scattered data

Cubic Spline Interpolation: Motivation

Problem with linear interpolation:

- Derivatives are discontinuous at grid points
- This causes problems in optimization (non-smooth objectives)

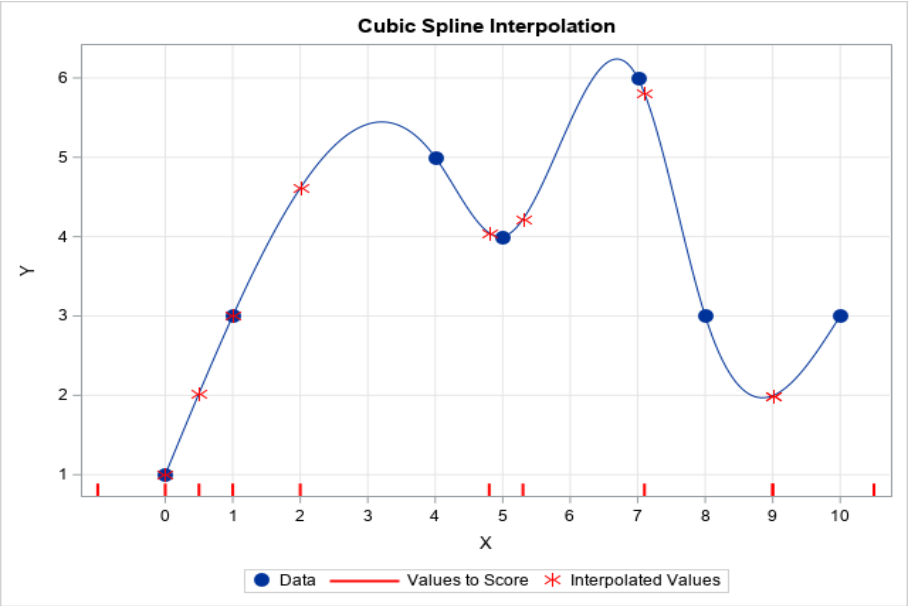
Solution: Cubic Splines

- Use piecewise cubic polynomials
- Enforce smoothness: continuous f, f', and f'' at all knots
- The term "spline" comes from flexible drafting strips used to draw smooth curves

Key Property

Cubic splines are the **smoothest** interpolant that passes through the data points (minimizes $\int [f''(x)]^2 dx$).

Cubic Spline Interpolation: Visualization



Cubic Spline: Mathematical Setup

Given: Data points $\{(x_i, y_i)\}_{i=1}^{N}$

Goal: Find piecewise cubic $f(x)$ such that:

$$f_{[x_i, x_{i+1}]}(x) = a_i + b_i x + c_i x^2 + d_i x^3, \quad i = 1, \ldots, N-1$$

Unknowns: $\{a_i, b_i, c_i, d_i\}_{i=1}^{N-1}$ — that's $4(N-1)$ parameters!

Constraints we need:

1. Interpolation: $f(x_i) = y_i$ for all i [$2(N-1)$ equations]
2. Continuity of f' at interior knots [$N-2$ equations]
3. Continuity of f'' at interior knots [$N-2$ equations]

Count: $2(N-1) + (N-2) + (N-2) = 4N - 6$ equations.

We have $4(N-1) = 4N - 4$ unknowns $\Rightarrow$ need 2 more conditions!

Cubic Spline: Boundary Conditions

The system:

$$\begin{cases} f_{[x_{i-1},x_i]}(x_i) = y_i, \quad f_{[x_i,x_{i+1}]}(x_i) = y_i \\ f'_{[x_{i-1},x_i]}(x_i) = f'_{[x_i,x_{i+1}]}(x_i), \quad i = 2, \ldots, N-1 \\ f''_{[x_{i-1},x_i]}(x_i) = f''_{[x_i,x_{i+1}]}(x_i), \quad i = 2, \ldots, N-1 \end{cases}$$

Common boundary conditions (pick one):

- **Natural spline:** $f''(x_1) = f''(x_N) = 0$
- **Clamped spline:** Specify $f'(x_1)$ and $f'(x_N)$
- **Hermite spline:** Use finite difference at boundaries:

$$f'(x_1) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}, \quad f'(x_N) = \frac{f(x_N) - f(x_{N-1})}{x_N - x_{N-1}}$$

Now we have $4(N-1)$ equations for $4(N-1)$ unknowns ✓

AR(1) Discretization: Why?

The Problem: Many economic models have continuous shocks:

$$z_{t+1} = \rho z_t + \varepsilon_{t+1}, \quad \varepsilon \sim N(0, \sigma_\varepsilon^2)$$

But: Value function iteration requires a **finite state space**.

Solution: Approximate the continuous AR(1) with a discrete Markov chain:

- Grid of states: $\{z_1, z_2, \ldots, z_n\}$
- Transition matrix: $\Pi = [P_{ij}]$ where $P_{ij} = \Pr(z_{t+1} = z_j \mid z_t = z_i)$

Key methods:

- Tauchen (1986) — what we'll cover
- Rouwenhorst (1995) — better for high persistence

AR(1) Process: Properties

The process:

$$y_t = \mu(1-\rho) + \rho \cdot y_{t-1} + \varepsilon_t, \quad \varepsilon_t \sim N(0, \sigma_\varepsilon^2)$$

Stationary distribution: (when $|\rho| < 1$)

$$y_t \sim N\left(\mu, \frac{\sigma_\varepsilon^2}{1-\rho^2}\right)$$

What we need to approximate:

1. Grid points $\{z_i\}_{i=1}^n$ covering the stationary distribution
2. Transition probabilities P_{ij} matching the AR(1) dynamics

Tauchen Method: Visual Intuition

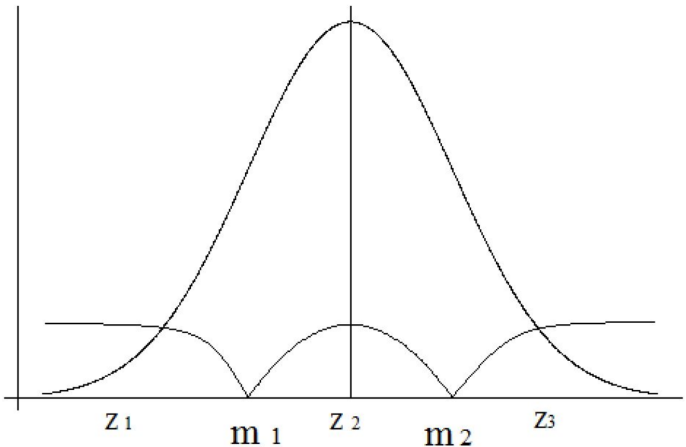


Figure 7: State Transition: From z_i to intervals around each z_j

Tauchen Method: Computing Transition Probabilities

Key insight: Given $z_t = z_i$, where does z_{t+1} land?

$$z_{t+1} = \rho z_i + \varepsilon_{t+1}, \quad \varepsilon_{t+1} \sim N(0, \sigma_\varepsilon^2)$$

So: $z_{t+1} \mid z_t = z_i \sim N(\rho z_i, \sigma_\varepsilon^2)$

Probability of transitioning to state j:

$$P_{ij} = \Pr\left(z_{t+1} \in I_j \mid z_t = z_i\right)$$

where $I_j = (m_{j-1}, m_j]$ is the interval around z_j.

Using the normal CDF Φ:

$$P_{ij} = \Phi\left(\frac{m_j - \rho z_i}{\sigma_\varepsilon}\right) - \Phi\left(\frac{m_{j-1} - \rho z_i}{\sigma_\varepsilon}\right)$$

Tauchen Method: Full Algorithm

Inputs: ρ, σ_ε, μ, number of states n, coverage λ

Step 1: Compute stationary distribution parameters

$$\mu_y = \mu, \quad \sigma_y = \frac{\sigma_\varepsilon}{\sqrt{1-\rho^2}}$$

Step 2: Set grid endpoints

$$\underline{y} = \mu_y - \lambda\sigma_y, \quad \bar{y} = \mu_y + \lambda\sigma_y$$

(Typical choice: $\lambda = 3$ covers 99.7% of stationary distribution)

Step 3: Create equally-spaced grid

$$z_i = \underline{y} + \frac{\bar{y} - \underline{y}}{n-1}(i-1), \quad i = 1, \ldots, n$$

Tauchen Method: Full Algorithm (cont.)

Step 4: Construct interval midpoints

$$m_i = \frac{z_i + z_{i+1}}{2}, \quad i = 1, \ldots, n-1$$

Step 5: Define intervals

$$I_1 = (-\infty, m_1], \quad I_j = (m_{j-1}, m_j], \quad I_n = (m_{n-1}, \infty)$$

Step 6: Compute transition probabilities

$$P_{ij} = \Phi\left(\frac{m_j - \mu(1-\rho) - \rho z_i}{\sigma_\varepsilon}\right) - \Phi\left(\frac{m_{j-1} - \mu(1-\rho) - \rho z_i}{\sigma_\varepsilon}\right)$$

Boundary cases:

$$P_{i1} = \Phi\left(\frac{m_1 - \mu(1-\rho) - \rho z_i}{\sigma_\varepsilon}\right)$$

$$P_{in} = 1 - \Phi\left(\frac{m_{n-1} - \mu(1-\rho) - \rho z_i}{\sigma_\varepsilon}\right)$$

Tauchen Method: Implementation Notes

Output:

- Grid: $Z = \{z_1, z_2, \ldots, z_n\}$
- Transition matrix: $\Pi = [P_{ij}]_{n \times n}$

Properties to verify:

- Each row sums to 1: $\sum_j P_{ij} = 1$
- All probabilities non-negative: $P_{ij} \geq 0$
- Stationary distribution matches AR(1)

Practical Tips

- Use $n = 5$ to 11 states (diminishing returns beyond)
- Use $\lambda = 3$ for most cases
- For high persistence ($\rho > 0.95$), consider Rouwenhorst method