Quantitative Macroeconomics w/ AI and ML
Lec. 3-A: Coding with AI

Zhigang Feng

Dec., 2025

Introduction: The Research Architect Approach

The Core Message

You can implement sophisticated macro models with AI assistance, even if coding from scratch feels difficult. The key is mastering five foundational areas:

1. **Economic Theory**: Optimization problems, equilibrium concepts, optimality conditions
2. **Algorithm Design**: Value function iteration, Euler equation methods, policy iteration
3. **Numerical Techniques**: Optimization, interpolation, discretization, simulation
4. **Advanced Methods**: Perturbation, projection methods, and their trade-offs
5. **Code Literacy**: Reading code, understanding what it delivers, spotting issues

Once these foundations are solid, the actual coding can be **assisted or outsourced to AI** via tools like Cursor, Claude Code, and Claude Skills. You become the **research architect**—AI becomes your implementation partner.

What's Changing in the AI Era

AI tools can now write code from descriptions, debug errors, translate math into implementations, and refactor existing code. This shifts where you spend your time:

Traditional Approach

- Write every line yourself
- Debug syntax errors manually
- Hours on implementation details
- Bottleneck: *coding skill*

Research Architect

- Design algorithms precisely
- Direct AI implementation
- Validate economic correctness
- Bottleneck: *understanding*

Your comparative advantage lies in knowing *what question you want to answer*. AI can check whether code is correct, but only you know what problem you're trying to solve and whether the answer makes economic sense.

The Five Pillars of Knowledge

Pillar 1: Economic Theory

You must be able to **formulate the economic problem** precisely:

Household optimization:

- Write the objective: $\max \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t, n_t)$
- Specify constraints: budget, borrowing limits, information structure
- Derive optimality conditions: Euler equations, transversality

Equilibrium concepts:

- Define competitive equilibrium: optimality + market clearing
- Understand recursive competitive equilibrium formulation
- Know when to use partial vs. general equilibrium

Why this matters: AI can verify whether code implements an algorithm correctly, but only you know *what research question you want to answer*. You must specify the problem; AI helps solve it.

Pillar 2: Algorithm Design

You must know **how to translate theory into computation**:

Dynamic programming approach:

- Bellman equation formulation: $V(s) = \max_a \{u(s,a) + \beta \mathbb{E}[V(s')]\}$
- Value function iteration: iterate $T(V) \to V^*$ until convergence
- Policy function iteration: faster convergence, more complex per step

Euler equation methods:

- Time iteration on Euler equations directly
- Endogenous grid method (EGM): invert Euler equation for speed

Note: These are foundational examples to get you started. In practice, you'll learn many more methods (parameterized expectations, monotone operators, etc.). The key is understanding enough to choose appropriately and communicate effectively with AI.

Pillar 3: Numerical Techniques

You must understand **how computation works** and potential pitfalls:

Core techniques:

- **Optimization**: Grid search, Newton methods, choice of solver for FOCs
- **Interpolation**: Linear, cubic spline, Chebyshev—accuracy vs. speed trade-offs
- **Discretization**: Tauchen, Rouwenhorst for AR(1) processes; grid design
- **Simulation**: Monte Carlo for moments, ergodic distribution computation

Common issues you must recognize:

- Occasionally binding constraints (borrowing limits, irreversibility)
- Curse of dimensionality in high-dimensional state spaces
- Interpolation errors near kinks in policy functions

Why this matters: When AI-generated code produces wrong results, you diagnose whether it's a grid problem, interpolation error, or algorithm bug.

Pillar 4: Advanced Numerical Methods

For larger or more complex models, you need **advanced solution techniques**:

Perturbation methods:

- Linear approximation around steady state (1st order)
- Higher-order for welfare, risk premia (2nd, 3rd order)
- Fast but local—accuracy degrades far from steady state

Projection methods:

- Global approximation using basis functions (Chebyshev, finite elements)
- Handles nonlinearities and occasionally binding constraints
- Computationally intensive, requires careful convergence checks

Note: Again, these are illustrative. You should develop familiarity with multiple methods. Understanding trade-offs (speed vs. accuracy, local vs. global) lets you choose wisely and guide AI implementation.

Pillar 5: Code Literacy

You must be able to **read and understand code**, even if you don't write it from scratch:

What code literacy means:

- Read a VFI loop and verify it matches your algorithm design
- Identify where the Bellman equation is evaluated, how expectations are computed
- Spot potential issues: wrong indexing, missing constraints, numerical instability
- Understand data structures: arrays, grids, interpolation objects

Language familiarity (Python focus):

- NumPy arrays and vectorized operations
- Control flow: loops, conditionals, function definitions
- Common libraries: SciPy (optimization, interpolation), Numba (speed)

Why this matters: AI writes code; you verify it implements what you designed. Without code literacy, you cannot validate AI output or diagnose problems.

From Foundations to AI-Assisted Coding

You Provide (Foundations)	AI Provides (Implementation)
Research question and model	Syntactically correct code
Algorithm choice and design	Translation to Python/Julia/etc.
Numerical method selection	Optimization, interpolation routines
Convergence/accuracy criteria	Diagnostic outputs and plots
Economic interpretation of results	Debugging and refactoring

Once you master the five pillars, you can effectively direct AI to handle implementation. The division of labor is clear: **you are the architect**, AI is the builder. You design the structure and inspect the construction; AI handles the bricks and mortar.

AI Tools for Implementation

AI Tools: Overview

With foundations in place, you can leverage AI tools for implementation:

Type	Examples	Best For
Chatbox	Claude, ChatGPT	Algorithm design, theory questions, debugging with full context
IDE-Integrated	Cursor, Copilot	Real-time coding assistance, inline suggestions, file-aware edits
Autonomous Agent	Claude Code, Aider	Multi-file projects, large refactoring, autonomous iteration

Recommended workflow: Use Claude chatbox for algorithm design and deep debugging. Use Cursor for day-to-day implementation. Use Claude Code for larger autonomous tasks like converting legacy codebases.

Tool 1: Claude Chatbox (Design and Reasoning)

Claude's chatbox interface excels at algorithm design and deep reasoning:

Strengths:

- Large context window—can process papers, code, and detailed specifications together
- Strong mathematical and economic reasoning
- Can discuss trade-offs: "Should I use VFI or EGM for this problem?"

Best uses:

- **Algorithm design**: "Design a VFI algorithm for Aiyagari with these features..."
- **Paper implementation**: Upload PDF, ask "Explain and implement Section 3's algorithm"
- **Debugging**: Provide code + error + economic context for diagnosis

Note: AI capabilities evolve rapidly. Context window sizes, reasoning abilities, and tool integration improve frequently. The principle remains: use chatbox for design-level thinking.

Tool 2: Cursor (IDE Integration)

Cursor integrates AI directly into VS Code, providing real-time assistance as you implement:

Key features:

- **Inline completion** (Tab): AI suggests code as you type based on context
- **Chat** (Cmd/Ctrl+L): Ask questions with full file context—"Why is this loop slow?"
- **Composer** (Cmd/Ctrl+I): Describe changes in natural language, AI edits multiple files

Typical workflow:

1. Design your algorithm in Claude chatbox (get the logic right)
2. Open Cursor, describe the algorithm, let AI generate initial code
3. Use inline completion to fill in details; use Chat to debug
4. Run code, validate results, iterate

Best practice: Always provide economic context, not just code snippets.

Tool 3: Claude Code (Autonomous Agent)

Claude Code is a terminal-based agent that works autonomously on larger tasks:

What it does:

- Reads and writes multiple files across your project
- Runs code, observes errors, debugs and fixes autonomously
- Continues iterating until the task is complete (or asks for guidance)

Best uses:

- Converting a MATLAB codebase to Python
- Implementing a complete algorithm from a paper description
- Large-scale refactoring (e.g., restructuring a project)

Key difference from Cursor: Cursor assists your coding in real-time; Claude Code works autonomously while you supervise. You provide the specification and validation criteria; it handles the implementation loop.

The Iterative Development Workflow

The Iterative Workflow: Core Principle

Regardless of which AI tool you use, follow this systematic approach:

Design Algorithm → Implement Code → Validate → Extend → Repeat

Key distinction: Separate **algorithm design** from **code implementation**.

- First, design the algorithm on paper or in discussion with AI (state space, Bellman equation, solution method, convergence criterion)
- Then, implement the designed algorithm in code
- These are different activities requiring different thinking

Start simple: Begin with the simplest model version that has an analytical benchmark. Validate thoroughly. Add features one at a time, validating at each step. This catches bugs early and builds confidence incrementally.

The Five-Step Workflow

Step 1: Design Algorithm (before any coding)

- Specify state space, Bellman equation, solution method, convergence criterion
- **Check extensibility**: "Can this design accommodate the next feature?"

Step 2: Implement Code

- Translate algorithm to code; include diagnostics (convergence plots, Euler residuals)

Step 3: Validate Thoroughly

- Technical: convergence, no NaN, grid adequacy
- Economic: sensible results, correct comparative statics, limiting cases

Step 4: Extend Algorithm Design

- Add *one* feature; update algorithm design *before* coding

Step 5: Repeat

- Each validated version becomes the foundation for the next

Why This Workflow Works

1. Separating design from implementation catches conceptual errors early

Algorithm bugs are cheaper to fix on paper than in code. Design first, then implement.

2. Clear attribution of bugs

Adding one feature at a time means you know exactly what caused any new issue.

3. AI succeeds with structured, well-specified instructions

Good communication makes good collaboration. When you provide clear algorithm specifications with precise inputs, outputs, and success criteria, AI delivers better results. Vague requests produce vague code.

4. Analytical benchmarks catch errors early

The simplest version often has closed-form solutions. Validating against these catches fundamental errors before complexity obscures them.

Example: RBC Model

Example 1: RBC Model

Goal: Implement a stochastic RBC model with variable labor supply.

Don't start with the full model! Instead, build incrementally:

Version	Features	Benchmark
0	Deterministic, fixed labor	Closed-form steady state
1	Add stochastic productivity	Limiting case = Version 0
2	Add variable labor supply	Limiting case = Version 1

Version 0 specifics: Log utility, Cobb-Douglas production, VFI solution. Steady state is $k^* = \left(\frac{\alpha\beta}{1-\beta(1-\delta)}\right)^{\frac{1}{1-\alpha}}$. We use VFI even for the deterministic case because the algorithm structure extends naturally to the stochastic version.

RBC: Design Algorithm First

Step 1: Design algorithm (before any coding)

Your prompt to AI: "Design a VFI algorithm for deterministic RBC with: (1) Production $Y = K^\alpha$, depreciation δ; (2) Bellman equation $V(k) = \max_{k'} \log(K^\alpha + (1-\delta)K - k') + \beta V(k')$; (3) Grid search over k'."

Before implementing, check extensibility:

"Can this algorithm design extend to: (a) stochastic productivity with state (k, A)? (b) variable labor as additional choice? Show where these would enter the algorithm."

Why? If the algorithm structure can't accommodate shocks, you'll redesign later. Better to get the design right first. The algorithm should have clear places where you'd add the expectation operator and additional choice variables.

RBC: Implement and Validate

Step 2: Implement code with diagnostics

"Implement this algorithm in Python. Include: (1) Convergence plot: sup norm $|V^{n+1} - V^n|$ vs iteration; (2) Compare numerical k^* to analytical formula; (3) Plot policy $k'(k)$ with 45-degree line; (4) Euler equation residuals."

Step 3: Validate (your job, not AI's)

- Does $k'(k)$ cross the 45-degree line at analytical k^*?
- Are Euler residuals near machine precision ($< 10^{-8}$)?
- Do comparative statics work? (Higher $\beta \Rightarrow$ higher k^*)

Step 4-5: Extend and repeat

After Version 0 validates, design the stochastic extension (add AR(1) productivity, Tauchen discretization). Implement. Validate that setting $\sigma = 0$ recovers Version 0. Continue to variable labor.

Example: Aiyagari Model

Example 2: Aiyagari Model (Partial Equilibrium)

Goal: Solve the household problem in Aiyagari (1994) with idiosyncratic income risk.

Model: Households maximize $\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t)$ subject to $c + a' = (1 + r)a + wz$ and $a' \geq \underline{a}$, where z is idiosyncratic productivity following a Markov process.

Incremental build:

Version	Features	Benchmark
0	Deterministic income ($z = 1$)	Three analytical cases
1	Stochastic income	Limiting case = Version 0
2	Stationary distribution	Distribution sums to 1

Partial equilibrium: r and w taken as given. General equilibrium comes later.

Aiyagari: Version 0 and Extensions

Version 0: Deterministic savings

Three analytical benchmarks based on $\beta(1+r)$:

- $\beta(1+r) = 1$: Constant consumption; $a'(a) = a$
- $\beta(1+r) < 1$: Decumulate; $a \to \underline{a}$
- $\beta(1+r) > 1$: Accumulate; a grows

All three cases must match theory before proceeding.

Version 1: Add income uncertainty

Design algorithm with state (a, z), Tauchen discretization. Validate that $\sigma_z = 0$ recovers Version 0. Check precautionary savings: $\mathbb{E}[a]$ exceeds deterministic steady state.

Version 2: Stationary distribution

Compute $\mu(a, z)$ from policy function. Validate: sums to 1, mass at constraint, higher $\sigma_z \Rightarrow$ more inequality.

Validation Principles

Validation: Your Core Responsibility

AI writes code. **You validate economics.** At every iteration:

Technical Checks

- Convergence achieved?
- No NaN/Inf values?
- Policy not hitting grid boundaries?
- Euler residuals small?

Economic Checks

- Results economically sensible?
- Comparative statics correct?
- Limiting cases recover benchmarks?
- Matches analytical solutions?

Debugging with AI: Provide full context (code + error + expected behavior + economic intuition). AI suggests diagnostics; you run them and report back. AI catches indexing errors and numerical issues; you catch economic nonsense.

Claude Skills: Encoding Your Workflow

The Evolution of Coding Assistance

Consider how coding assistance has evolved:

Two decades ago (C++):

- Write basic functions yourself: interpolation, Brent optimization, etc.
- Debug issues like integer division ($1/3 = 0$, not 0.333)
- Low-level control, but time-consuming

A few years ago (Python):

- Libraries handle common tasks: `scipy.optimize`, `numpy.interp`
- Call functions instead of writing them; integer division handled ($1/3 = 0.333$)
- Higher-level, faster development

Now (AI assistance):

- AI writes the code that calls the libraries
- You specify *what* you want; AI handles *how*

What Are Claude Skills?

Skills extend this evolution one step further. Think of them as:

Analogy to programming concepts:

- Like **libraries** (SciPy, Intel MKL): pre-built functionality you call
- Like **macros** (in SAS or C): reusable templates that expand into detailed instructions
- Like **configuration files**: settings that customize behavior

What a skill actually is:

A skill is a **structured instruction file** that you create and upload to Claude. It contains your methodology, domain knowledge, and quality standards. When Claude encounters a relevant task, it follows your skill's instructions automatically.

The building blocks: Claude is already a powerful, knowledgeable AI. A skill adds *your specific workflow* on top of that foundation—your "start simple, validate, extend" methodology, your diagnostic requirements, your validation checklists.

How Skills Differ from Regular Chatbox Use

Without a skill (regular chatbox):

- Every conversation starts fresh
- You must re-explain your methodology each time
- "Remember to start simple... include diagnostics... validate before extending..."
- Inconsistent results across sessions

With a skill:

- Your methodology is encoded once, applied automatically
- Claude follows your workflow without re-prompting
- Consistent "start simple, validate, extend" discipline every time
- Like training an assistant once and having them remember forever

Key difference from a "detailed prompt": A prompt is one-time; a skill persists across all relevant conversations. You don't paste it each time—Claude recognizes when to apply it.

How Skills Work: The Mechanism

Step 1: You create a skill file (a folder with SKILL.md and optional references)

Step 2: You upload it to Claude (Settings → Skills, or add to a Project)

Step 3: Claude uses it automatically

Three-level loading system:

Level	Content	When Loaded
1	Name + Description	Always visible to Claude
2	SKILL.md body	When skill triggers
3	Reference files	On demand, as needed

How Skills Work: The Mechanism

Triggering: Claude matches your request against skill descriptions. When you say "implement an RBC model using VFI," Claude recognizes this matches your quant-macro skill and loads it.

Do you need to mention it? No. Once uploaded, Claude applies relevant skills automatically based on context. You just ask your question normally.

Skill File Structure

A skill is a folder with a required `SKILL.md` file and optional references:

```
quant-macro-solver/
  ├ SKILL.md              (required:  triggers + workflow)
  └ references/            (optional:  detailed guides)
      ├ rbc-guide.md      (RBC algorithm details)
      └ aiyagari-guide.md   (Aiyagari algorithm details)
```

SKILL.md has two parts:

1. **YAML frontmatter**: Name and description (determines when skill triggers)
2. **Markdown body**: Workflow instructions, validation criteria, prompting patterns

Reference files contain detailed model-specific algorithms. Claude loads them only when needed, keeping context efficient.

SKILL.md: The Frontmatter

The file starts with **YAML frontmatter**—a simple key-value format. (YAML = "YAML Ain't Markup Language," a human-readable data format.)

```
---
name: quant-macro-solver
description: "Iterative workflow for implementing quantitative
  macroeconomic models (RBC, Aiyagari, heterogeneous agents).
  Use when: (1) Building VFI or policy iteration solutions,
  (2) Implementing Bellman equations, (3) Computing stationary
  distributions, (4) Any dynamic programming for macro models."
---
```

The description is the trigger mechanism. Claude reads this to decide when to apply the skill. Be comprehensive: list use cases, key terms, model types. When your request matches, Claude loads the skill body.

SKILL.md: The Body

Below the frontmatter, the Markdown body specifies *how* to execute:

```
# Quantitative Macro Model Solver

## Core Workflow
For ANY macro model, follow these steps in order:

### Step 1: Design Algorithm (before coding)
Specify state space, Bellman, method. Check extensibility...

### Step 2: Implement Code with Diagnostics
REQUIRED: convergence plot, Euler residuals, benchmark comparison...

### Step 3: Validate
Technical + Economic checks. Do NOT proceed until all pass.

### Step 4: Extend Algorithm Design
Add ONE feature to algorithm, then implement.

### Step 5: Repeat
```

Use imperative form ("Check convergence," not "You should check...").

SKILL.md: Linking References

Keep SKILL.md lean; put details in reference files:

```
## Model-Specific Guides

### RBC Model
See [references/rbc-guide.md](references/rbc-guide.md) for:
- Version 0: Deterministic algorithm and steady state formula
- Version 1: Stochastic extension with Tauchen discretization
- Version 2: Variable labor with intratemporal FOC

### Aiyagari Model
See [references/aiyagari-guide.md](references/aiyagari-guide.md) for:
- Version 0: Deterministic savings and three benchmark cases
- Version 1: Income risk extension
- Version 2: Stationary distribution computation
```

Claude loads reference files only when needed. When you ask about RBC, it reads rbc-guide.md; it doesn't load aiyagari-guide.md unnecessarily.

Using a Skill

Three ways to use a skill you've created:

Method 1: Upload to Claude.ai

- Settings → Skills → Upload your skill folder
- Skill available in all future conversations; triggers automatically

Method 2: Add to a Claude Project

- Create a Project for your research (e.g., "Macro Model Development")
- Add skill files to project knowledge base
- All conversations in that project automatically have access

Method 3: Paste into conversation

- Copy SKILL.md content: "Follow this workflow: [content]"
- Good for one-off use or testing before formal upload

The Analogy Summarized

Programming Concept	Skill Analogy
Writing functions from scratch	Chatbox without guidance
Calling library functions (SciPy)	Claude's built-in knowledge
Custom macros / templates	Skill file (reusable workflow)
Configuration files	YAML frontmatter (when to trigger)
Documentation / style guides	Reference files (how to execute)

Just as libraries saved you from rewriting interpolation routines, skills save you from re-explaining your methodology. You encode it once; Claude applies it consistently across all relevant tasks.

Summary

The Complete Picture

Master the five pillars:

1. Economic theory: formulate problems, define equilibria
2. Algorithm design: VFI, EGM, policy iteration—and many more
3. Numerical techniques: optimization, interpolation, their pitfalls
4. Advanced methods: perturbation, projection, trade-offs
5. Code literacy: read code, verify it matches your design

Then leverage AI tools:

- Claude chatbox for algorithm design and debugging
- Cursor for day-to-day implementation
- Claude Code for autonomous larger tasks
- Claude Skills to encode your methodology for consistent reuse

Always follow the workflow: Design algorithm $\rightarrow$ Implement $\rightarrow$ Validate $\rightarrow$ Extend $\rightarrow$ Repeat.

Key Takeaway

You don't need to be an expert coder.

You need to understand theory, algorithms, and numerics well enough to **specify problems clearly**, **direct AI effectively**, and **validate results rigorously**.

AI democratizes implementation. Your unique value is knowing *what* to compute, *why* it matters economically, and *whether* the results are correct. Master the foundations, communicate clearly with AI, and it handles the rest.